Programmer's Reference Guide

Zend Framework 1.10.x

Programmer's Reference Guide: Zend Framework 1.10.x

Publication date 01/27/2010 by mikaelkael (SVN 20684)

Table of Contents

Part I. Introduction to Zend Framework

Table of Contents

Overview

Zend Framework is an open source framework for developing web applications and services with PHP 5. Zend Framework is implemented using 100% object-oriented code. The component structure of Zend Framework is somewhat unique; each component is designed with few dependencies on other components. This loosely coupled architecture allows developers to use components individually. We often call this a "use-at-will" design.

While they can be used separately, Zend Framework components in the standard library form a powerful and extensible web application framework when combined. Zend Framework offers a robust, high performance MVC implementation, a database abstraction that is simple to use, and a forms component that implements HTML form rendering, validation, and filtering so that developers can consolidate all of these operations using one easy-to-use, object oriented interface. Other components, such as `Zend_Auth` and `Zend_Acl`, provide user authentication and authorization against all common credential stores. Still others implement client libraries to simply access to the most popular web services available. Whatever your application needs are, you're likely to find a Zend Framework component that can be used to dramatically reduce development time with a thoroughly tested foundation.

The principal sponsor of the project 'Zend Framework' is Zend Technologies, but many companies have contributed components or significant features to the framework. Companies such as Google, Microsoft, and StrikeIron have partnered with Zend to provide interfaces to web services and other technologies that they wish to make available to Zend Framework developers.

Zend Framework could not deliver and support all of these features without the help of the vibrant Zend Framework community. Community members, including contributors, make themselves available on mailing lists, IRC channels, and other forums. Whatever question you have about Zend Framework, the community is always available to address it.

Installation

See the requirements appendix for a detailed list of requirements for Zend Framework.

Installing Zend Framework is extremely simple. Once you have downloaded and extracted the framework, you should add the `/library` folder in the distribution to the beginning of your include path. You may also want to move the library folder to another – possibly shared – location on your file system.

- Download the latest stable release. This version, available in both `.zip` and `.tar.gz` formats, is a good choice for those who are new to Zend Framework.
- Download the latest nightly snapshot. For those who would brave the cutting edge, the nightly snapshots represent the latest progress of Zend Framework development. Snapshots are bundled with documentation either in English only or in all available languages. If you anticipate working with the latest Zend Framework developments, consider using a Subversion (SVN) client.
- Using a Subversion (SVN) client. Zend Framework is open source software, and the Subversion repository used for its development is publicly available. Consider using SVN to get Zend Framework if you already use SVN for your application development, want to contribute back to the framework, or need to upgrade your framework version more often than releases occur.

  Exporting is useful if you want to get a particular framework revision without the `.svn` directories as created in a working copy.

  Check out a working copy if you want contribute to Zend Framework, a working copy can be updated any time with svn update and changes can be commited to our SVN repository with the svn commit command.

  An externals definition is quite convenient for developers already using SVN to manage their application's working copies.

  The URL for the trunk of Zend Framework's SVN repository is: http://framework.zend.com/svn/framework/standard/trunk

Once you have a copy of Zend Framework available, your application needs to be able to access the framework classes. Though there are several ways to achieve this, your PHP include_path needs to contain the path to Zend Framework's library.

Zend provides a QuickStart to get you up and running as quickly as possible. This is an excellent way to begin learning about the framework with an emphasis on real world examples that you can build upon.

Since Zend Framework components are loosely coupled, you may use a somewhat unique combination of them in your own applications. The following chapters provide a comprehensive reference to Zend Framework on a component-by-component basis.

Part II. Learning Zend Framework

Table of Contents

Zend Framework Quick Start

1. Zend Framework & MVC Introduction

1.1. Zend Framework

Zend Framework is an open source, object oriented web application framework for PHP 5. Zend Framework is often called a 'component library', because it has many loosely coupled components that you can use more or less independently. But Zend Framework also provides an advanced Model-View-Controller (MVC) implementation that can be used to establish a basic structure for your Zend Framework applications. A full list of Zend Framework components along with short descriptions may be found in the components overview. This QuickStart will introduce you to some of Zend Framework's most commonly used components, including `Zend_Controller`, `Zend_Layout`, `Zend_Config`, `Zend_Db`, `Zend_Db_Table`, `Zend_Registry`, along with a few view helpers.

Using these components, we will build a simple database-driven guest book application within minutes. The complete source code for this application is available in the following archives:

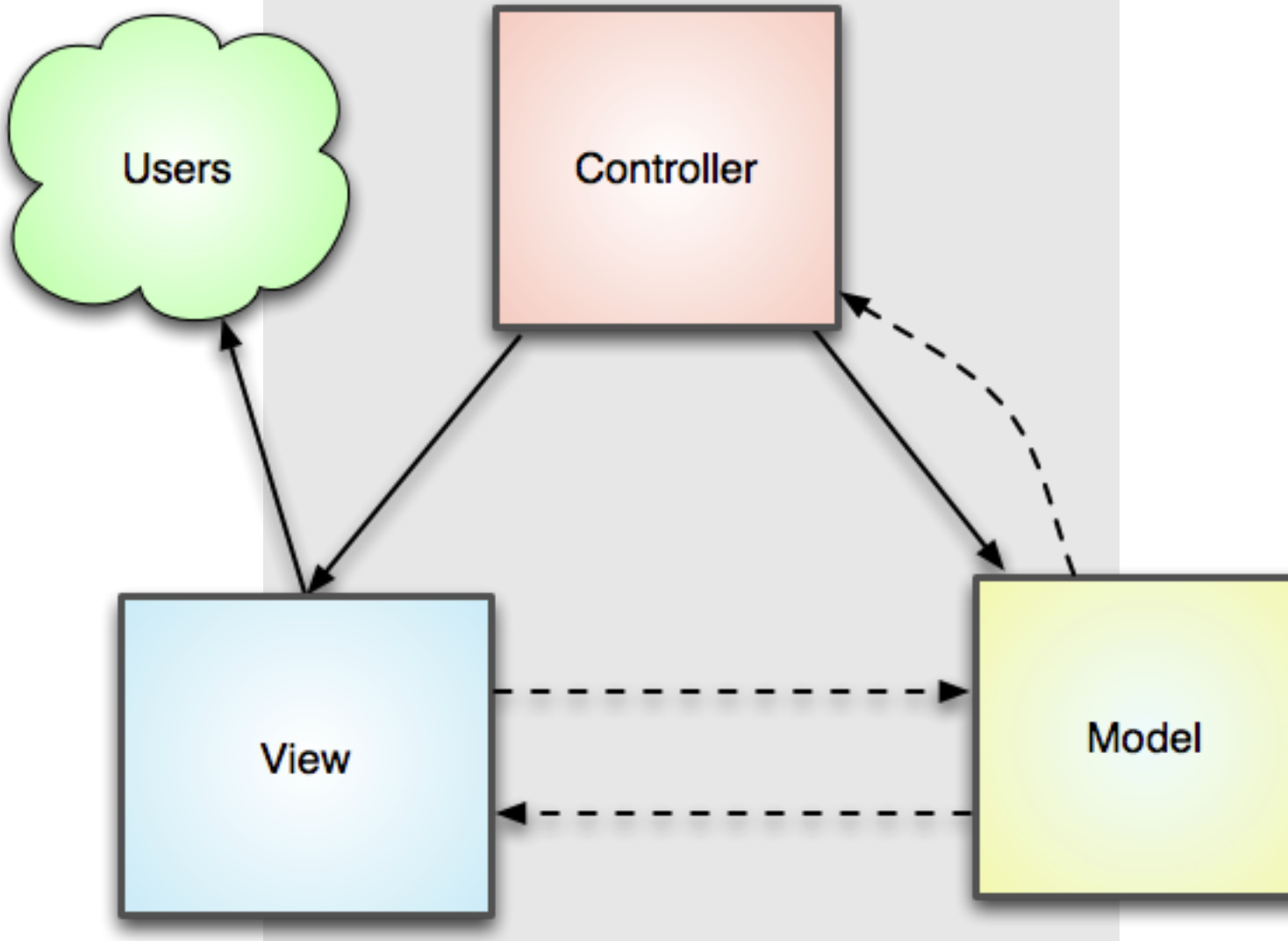
- zip
- tar.gz

1.2. Model-View-Controller

So what exactly is this MVC pattern everyone keeps talking about, and why should you care? MVC is much more than just a three-letter acronym (TLA) that you can whip out anytime you want to sound smart; it has become something of a standard in the design of modern web applications. And for good reason. Most web application code falls under one of the following three categories: presentation, business logic, and data access. The MVC pattern models this separation of concerns well. The end result is that your presentation code can be consolidated in one part of your application with your business logic in another and your data access code in yet another. Many developers have found this well-defined separation indispensable for keeping their code organized, especially when more than one developer is working on the same application.

More Information

Let's break down the pattern and take a look at the individual pieces:



- *Model* - This is the part of your application that defines its basic functionality behind a set of abstractions. Data access routines and some business logic can be defined in the model.
- *View* - Views define exactly what is presented to the user. Usually controllers pass data to each view to render in some format. Views will often collect data from the user, as well. This is where you're likely to find HTML markup in your MVC applications.
- *Controller* - Controllers bind the whole pattern together. They manipulate models, decide which view to display based on the user's request and other factors, pass along the data that each view will need, or hand off control to another controller entirely. Most MVC experts recommend keeping controllers as skinny as possible.

Of course there is more to be said about this critical pattern, but this should give you enough background to understand the guestbook application we'll be building.

2. Create Your Project

In order to create your project, you must first download and extract Zend Framework.

2.1. Install Zend Framework

The easiest way to get Zend Framework along with a complete PHP stack is by installing Zend Server. Zend Server has native installers for Mac OSX, Windows, Fedora Core, and Ubuntu, as well as a universal installation package compatible with most Linux distributions.

After you have installed Zend Server, the Framework files may be found under `/usr/local/zend/share/ZendFramework` on Mac OSX and Linux, and `C:\Program Files\Zend\ZendServer\share\ZendFramework` on Windows. The `include_path` will already be configured to include Zend Framework.

Alternately, you can Download the latest version of Zend Framework and extract the contents; make a note of where you have done so.

Optionally, you can add the path to the `library/` subdirectory of the archive to your `php.ini`'s `include_path` setting.

That's it! Zend Framework is now installed and ready to use.

2.2. Create Your Project

zf Command Line Tool

In your Zend Framework installation is a `bin/` subdirectory, containing the scripts `zf.sh` and `zf.bat` for Unix-based and Windows-based users, respectively. Make a note of the absolute path to this script.

Wherever you see references to `zf.sh` or `zf.bat`, please substitute the absolute path to the script. On Unix-like systems, you may want to use your shell's alias functionality: **alias zf.sh=path/to/ZendFramework/bin/zf.sh**.

If you have problems setting up the **zf** command-line tool, please refer to the manual.

Open a terminal (in Windows, **Start -> Run**, and then use **cmd**). Navigate to a directory where you would like to start a project. Then, use the path to the appropriate script, and execute one of the following:

```
# Unix:
% zf.sh create project quickstart

# DOS/Windows:
C:> zf.bat create project quickstart
```

Running this command will create your basic site structure, including your initial controllers and views. The tree looks like the following:

```
quickstart
|-- application
|   |-- Bootstrap.php
|   |-- configs
|   |   `-- application.ini
|   |-- controllers
|   |   |-- ErrorController.php
|   |   `-- IndexController.php
|   |-- models
|   `-- views
|       |-- helpers
|       `-- scripts
|           |-- error
|           |   `-- error.phtml
|           `-- index
|               `-- index.phtml
|-- library
|-- public
|   `-- index.php
`-- tests
    |-- application
    |   `-- bootstrap.php
    |-- library
    |   `-- bootstrap.php
    `-- phpunit.xml
```

At this point, if you haven't added Zend Framework to your `include_path`, we recommend either copying or symlinking it into your `library/` directory. In either case, you'll want to either recursively copy or symlink the `library/Zend/` directory of your Zend Framework installation into the `library/` directory of your project. On unix-like systems, that would look like one of the following:

```
# Symlink:
% cd library; ln -s path/to/ZendFramework/library/Zend .

# Copy:
% cd library; cp -r path/to/ZendFramework/library/Zend .
```

On Windows systems, it may be easiest to do this from the Explorer.

Now that the project is created, the main artifacts to begin understanding are the bootstrap, configuration, action controllers, and views.

2.3. The Bootstrap

Your `Bootstrap` class defines what resources and components to initialize. By default, Zend Framework's Front Controller is initialized, and it uses the `application/controllers/` as the default directory in which to look for action controllers (more on that later). The class looks like the following:

```
// application/Bootstrap.php

class Bootstrap extends Zend_Application_Bootstrap_Bootstrap
{
}
```

As you can see, not much is necessary to begin with.

2.4. Configuration

While Zend Framework is itself configurationless, you often need to configure your application. The default configuration is placed in `application/configs/application.ini`, and contains some basic directives for setting your PHP environment (for instance, turning error reporting on and off), indicating the path to your bootstrap class (as well as its class name), and the path to your action controllers. It looks as follows:

```
; application/configs/application.ini

[production]
phpSettings.display_startup_errors = 0
phpSettings.display_errors = 0
includePaths.library = APPLICATION_PATH "/../library"
bootstrap.path = APPLICATION_PATH "/Bootstrap.php"
bootstrap.class = "Bootstrap"
resources.frontController.controllerDirectory = APPLICATION_PATH "/controllers"

[staging : production]

[testing : production]
phpSettings.display_startup_errors = 1
phpSettings.display_errors = 1

[development : production]
phpSettings.display_startup_errors = 1
phpSettings.display_errors = 1
```

Several things about this file should be noted. First, when using INI-style configuration, you can reference constants directly and expand them; `APPLICATION_PATH` is actually a constant. Additionally note that there are several sections defined: production, staging, testing, and development. The latter three inherit settings from the "production" environment. This is a useful way to organize configuration to ensure that appropriate settings are available in each stage of application development.

2.5. Action Controllers

Your application's *action controllers* contain your application workflow, and do the work of mapping your requests to the appropriate models and views.

An action controller should have one or more methods ending in "Action"; these methods may then be requested via the web. By default, Zend Framework URLs follow the schema `/controller/action`, where "controller" maps to the action controller name (minus the "Controller" suffix) and "action" maps to an action method (minus the "Action" suffix).

Typically, you always need an `IndexController`, which is a fallback controller and which also serves the home page of the site, and an `ErrorController`, which is used to indicate things such as HTTP 404 errors (controller or action not found) and HTTP 500 errors (application errors).

The default `IndexController` is as follows:

```
// application/controllers/IndexController.php

class IndexController extends Zend_Controller_Action
{

    public function init()
```

```
    {
        /* Initialize action controller here */
    }

    public function indexAction()
    {
        // action body
    }
}
```

And the default `ErrorController` is as follows:

```
// application/controllers/ErrorController.php

class ErrorController extends Zend_Controller_Action
{

    public function errorAction()
    {
        $errors = $this->_getParam('error_handler');

        switch ($errors->type) {
            case Zend_Controller_Plugin_ErrorHandler::EXCEPTION_NO_ROUTE:
            case Zend_Controller_Plugin_ErrorHandler::EXCEPTION_NO_CONTROLLER:
            case Zend_Controller_Plugin_ErrorHandler::EXCEPTION_NO_ACTION:

                // 404 error -- controller or action not found
                $this->getResponse()->setHttpResponseCode(404);
                $this->view->message = 'Page not found';
                break;
            default:
                // application error
                $this->getResponse()->setHttpResponseCode(500);
                $this->view->message = 'Application error';
                break;
        }

        $this->view->exception = $errors->exception;
        $this->view->request   = $errors->request;
    }
}
```

You'll note that (1) the `IndexController` contains no real code, and (2) the `ErrorController` makes reference to a "view" property. That leads nicely into our next subject.

2.6. Views

Views in Zend Framework are written in plain old PHP. View scripts are placed in `application/views/scripts/`, where they are further categorized using the controller names. In our case, we have an `IndexController` and an `ErrorController`, and thus we have corresponding `index/` and `error/` subdirectories within our view scripts directory. Within these subdirectories, you will then find and create view scripts that correspond to each controller action exposed; in the default case, we thus have the view scripts `index/index.phtml` and `error/error.phtml`.

View scripts may contain any markup you want, and use the `<?php` opening tag and `?>` closing tag to insert PHP directives.

The following is what we install by default for the `index/index.phtml` view script:

```
<!-- application/views/scripts/index/index.phtml -->
<style>

    a:link,
    a:visited
    {
        color: #0398CA;
    }

    span#zf-name
    {
        color: #91BE3F;
    }

    div#welcome
    {
        color: #FFFFFF;
        background-image: url(http://framework.zend.com/images/bkg_header.jpg);
        width:  600px;
        height: 400px;
        border: 2px solid #444444;
        overflow: hidden;
        text-align: center;
    }

    div#more-information
    {
        background-image: url(http://framework.zend.com/images/bkg_body-bottom.gif);
        height: 100%;
    }

</style>
<div id="welcome">
    <h1>Welcome to the <span id="zf-name">Zend Framework!</span><h1 />
    <h3>This is your project's main page<h3 />
    <div id="more-information">
        <p>
            <img src="http://framework.zend.com/images/PoweredBy_ZF_4LightBG.png" />
        </p>

        <p>
            Helpful Links: <br />
            <a href="http://framework.zend.com/">Zend Framework Website</a> |
            <a href="http://framework.zend.com/manual/en/">Zend Framework
                Manual</a>
        </p>
    </div>
</div>
```

The `error/error.phtml` view script is slightly more interesting as it uses some PHP conditionals:

```
<!-- application/views/scripts/error/error.phtml -->
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN";
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
  <title>Zend Framework Default Application</title>
```

```
</head>
<body>
  <h1>An error occurred</h1>
  <h2><?php echo $this->message ?></h2>

  <?php if ('development' == $this->env): ?>
  <h3>Exception information:</h3>
  <p>
      <b>Message:</b> <?php echo $this->exception->getMessage() ?>
  </p>

  <h3>Stack trace:</h3>
  <pre><?php echo $this->exception->getTraceAsString() ?>
  </pre>

  <h3>Request Parameters:</h3>
  <pre><?php echo var_export($this->request->getParams(), 1) ?>
  </pre>
  <?php endif ?>
</body>
</html>
```

2.7. Checkpoint

At this point, you should be able to fire up your initial Zend Framework application. Create a virtual host in your web server, and point its document root to your application's `public/` subdirectory. Make sure your host's name is in your DNS or hosts file, and then point your browser to it. You should be able to see a welcome page at this point.

3. Create A Layout

You may have noticed that the view scripts in the previous sections were HTML fragments- not complete pages. This is by design; we want our actions to return content only related to the action itself, not the application as a whole.

Now we must compose that generated content into a full HTML page. We'd also like to have a consistent look and feel for the application. We will use a global site layout to accomplish both of these tasks.

There are two design patterns that Zend Framework uses to implement layouts: Two Step View and Composite View. *Two Step View* is usually associated with the Transform View pattern; the basic idea is that your application view creates a representation that is then injected into the master view for final transformation. The *Composite View* pattern deals with a view made of one or more atomic, application views.

In Zend Framework, Zend_Layout combines the ideas behind these patterns. Instead of each action view script needing to include site-wide artifacts, they can simply focus on their own responsibilities.

Occasionally, however, you may need application-specific information in your site-wide view script. Fortunately, Zend Framework provides a variety of view *placeholders* to allow you to provide such information from your action view scripts.

To get started using Zend_Layout, first we need to inform our bootstrap to use the `Layout` resource. This can be done by adding the following line to your `application/configs/application.ini` file, within the `production` section:

```
; application/configs/application.ini

; Add to [production] section:
resources.layout.layoutPath = APPLICATION_PATH "/layouts/scripts"
```

The final INI file should look as follows:

```
; application/configs/application.ini

[production]
; PHP settings we want to initialize
phpSettings.display_startup_errors = 0
phpSettings.display_errors = 0
includePaths.library = APPLICATION_PATH "/../library"
bootstrap.path = APPLICATION_PATH "/Bootstrap.php"
bootstrap.class = "Bootstrap"
resources.frontController.controllerDirectory = APPLICATION_PATH "/controllers"
resources.layout.layoutPath = APPLICATION_PATH "/layouts/scripts"

[staging : production]

[testing : production]
phpSettings.display_startup_errors = 1
phpSettings.display_errors = 1

[development : production]
phpSettings.display_startup_errors = 1
phpSettings.display_errors = 1
```

This directive tells your application to look for layout view scripts in `application/layouts/scripts`. Create those directories now.

We also want to ensure we have an XHTML DocType declaration for our application. To enable this, we need to add a resource to our bootstrap.

The simplest way to add a bootstrap resource is to simply create a protected method beginning with the phrase `_init`. In this case, we want to initialize the doctype, so we'll create an `_initDoctype()` method within our bootstrap class:

```
// application/Bootstrap.php

class Bootstrap extends Zend_Application_Bootstrap_Bootstrap
{
    protected function _initDoctype()
    {
    }
}
```

Within that method, we need to hint to the view to use the appropriate doctype. But where will the view object come from? The easy solution is to initialize the `View` resource; once we have, we can pull the view object from the bootstrap and use it.

To initialize the view resource, add the following line to your `application/configs/application.ini` file, in the section marked `production`:

```
; application/configs/application.ini
```

```
; Add to [production] section:
resources.view[] =
```

This tells us to initialize the view with no options (the '[]' indicates that the "view" key is an array, and we pass nothing to it).

Now that we have a view, let's flesh out our `_initDoctype()` method. In it, we will first ensure the `View` resource has run, fetch the view object, and then configure it:

```
// application/Bootstrap.php

class Bootstrap extends Zend_Application_Bootstrap_Bootstrap
{
    protected function _initDoctype()
    {
        $this->bootstrap('view');
        $view = $this->getResource('view');
        $view->doctype('XHTML1_STRICT');
    }
}
```

Now that we've initialized `Zend_Layout` and set the Doctype, let's create our site-wide layout:

```
// application/layouts/scripts/layout.phtml

echo $this->doctype() ?>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
  <title>Zend Framework Quickstart Application</title>
  <?php echo $this->headLink()->appendStylesheet('/css/global.css') ?>
</head>
<body>
<div id="header" style="background-color: #EEEEEE; height: 30px;">
    <div id="header-logo" style="float: left">
        <b>ZF Quickstart Application</b>
    </div>
    <div id="header-navigation" style="float: right">
        <a href="<?php echo $this->url(
            array('controller'=>'guestbook'),
            'default',
            true) ?>">Guestbook</a>
    </div>
</div>

<?php echo $this->layout()->content ?>
</body>
</html>
```

We grab our application content using the `layout()` view helper, and accessing the "content" key. You may render to other response segments if you wish to, but in most cases, this is all that's necessary.

Note also the use of the `headLink()` placeholder. This is an easy way to generate the HTML for <link> elements, as well as to keep track of them throughout your application. If you need to add additional CSS sheets to support a single action, you can do so, and be assured it will be present in the final rendered page.

Checkpoint

Now go to "http://localhost" and check out the source. You should see your XHTML header, head, title, and body sections.

4. Create a Model and Database Table

Before we get started, let's consider something: where will these classes live, and how will we find them? The default project we created instantiates an autoloader. We can attach other autoloaders to it so that it knows where to find different classes. Typically, we want our various MVC classes grouped under the same tree -- in this case, `application/` -- and most often using a common prefix.

`Zend_Controller_Front` has a notion of "modules", which are individual mini-applications. Modules mimic the directory structure that the **zf** tool sets up under `application/`, and all classes inside them are assumed to begin with a common prefix, the module name. `application/` is itself a module -- the "default" or "application" module. As such, we'll want to setup autoloading for resources within this directory.

`Zend_Application_Module_Autoloader` provides the functionality needed to map the various resources under a module to the appropriate directories, and provides a standard naming mechanism as well. An instance of the class is created by default during initialization of the bootstrap object; your application bootstrap will be default use the module prefix "Application". As such, our models, forms, and table classes will all begin with the class prefix "Application_".

Now, let's consider what makes up a guestbook. Typically, they are simply a list of entries with a *comment*, *timestamp*, and, often, *email address*. Assuming we store them in a database, we may also want a *unique identifier* for each entry. We'll likely want to be able to save an entry, fetch individual entries, and retrieve all entries. As such, a simple guestbook model API might look something like this:

```
// application/models/Guestbook.php

class Application_Model_Guestbook
{
    protected $_comment;
    protected $_created;
    protected $_email;
    protected $_id;

    public function __set($name, $value);
    public function __get($name);

    public function setComment($text);
    public function getComment();

    public function setEmail($email);
    public function getEmail();

    public function setCreated($ts);
    public function getCreated();

    public function setId($id);
    public function getId();

    public function save();
    public function find($id);
```

```
    public function fetchAll();
}
```

`__get()` and `__set()` will provide a convenience mechanism for us to access the individual entry properties, and proxy to the other getters and setters. They also will help ensure that only properties we whitelist will be available in the object.

`find()` and `fetchAll()` provide the ability to fetch a single entry or all entries.

Now from here, we can start thinking about setting up our database.

First we need to initialize our `Db` resource. As with the `Layout` and `View` resource, we can provide configuration for the `Db` resource. In your `application/configs/application.ini` file, add the following lines in the appropriate sections.

```
; application/configs/application.ini

; Add these lines to the appropriate sections:
[production]
resources.db.adapter       = "PDO_SQLITE"
resources.db.params.dbname = APPLICATION_PATH "/../data/db/guestbook.db"

[testing : production]
resources.db.params.dbname = APPLICATION_PATH "/../data/db/guestbook-testing.db"

[development : production]
resources.db.params.dbname = APPLICATION_PATH "/../data/db/guestbook-dev.db"
```

Your final configuration file should look like the following:

```
; application/configs/application.ini

[production]
phpSettings.display_startup_errors = 0
phpSettings.display_errors = 0
bootstrap.path = APPLICATION_PATH "/Bootstrap.php"
bootstrap.class = "Bootstrap"
resources.frontController.controllerDirectory = APPLICATION_PATH "/controllers"
resources.layout.layoutPath = APPLICATION_PATH "/layouts/scripts"
resources.view[] =
resources.db.adapter = "PDO_SQLITE"
resources.db.params.dbname = APPLICATION_PATH "/../data/db/guestbook.db"

[staging : production]

[testing : production]
phpSettings.display_startup_errors = 1
phpSettings.display_errors = 1
resources.db.params.dbname = APPLICATION_PATH "/../data/db/guestbook-testing.db"

[development : production]
phpSettings.display_startup_errors = 1
phpSettings.display_errors = 1
resources.db.params.dbname = APPLICATION_PATH "/../data/db/guestbook-dev.db"
```

Note that the database(s) will be stored in `data/db/`. Create those directories, and make them world-writeable. On unix-like systems, you can do that as follows:

```
% mkdir -p data/db; chmod -R a+rwX data
```

On Windows, you will need to create the directories in Explorer and set the permissions to allow anyone to write to the directory.

At this point we have a connection to a database; in our case, its a connection to a Sqlite database located inside our `application/data/` directory. So, let's design a simple table that will hold our guestbook entries.

```
-- scripts/schema.sqlite.sql
--
-- You will need load your database schema with this SQL.

CREATE TABLE guestbook (
    id INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT,
    email VARCHAR(32) NOT NULL DEFAULT 'noemail@test.com',
    comment TEXT NULL,
    created DATETIME NOT NULL
);

CREATE INDEX "id" ON "guestbook" ("id");
```

And, so that we can have some working data out of the box, lets create a few rows of information to make our application interesting.

```
-- scripts/data.sqlite.sql
--
-- You can begin populating the database with the following SQL statements.

INSERT INTO guestbook (email, comment, created) VALUES
    ('ralph.schindler@zend.com',
    'Hello! Hope you enjoy this sample zf application!',
    DATETIME('NOW'));
INSERT INTO guestbook (email, comment, created) VALUES
    ('foo@bar.com',
    'Baz baz baz, baz baz Baz baz baz - baz baz baz.',
    DATETIME('NOW'));
```

Now that we have both the schema and some data defined. Lets get a script together that we can now execute to build this database. Naturally, this is not needed in production, but this script will help developers build out the database requirements locally so they can have the fully working application. Create the script as `scripts/load.sqlite.php` with the following contents:

```
// scripts/load.sqlite.php

/**
 * Script for creating and loading database
 */

// Initialize the application path and autoloading
defined('APPLICATION_PATH')
    || define('APPLICATION_PATH', realpath(dirname(__FILE__) . '/../application'));
set_include_path(implode(PATH_SEPARATOR, array(
    APPLICATION_PATH . '/../library',
    get_include_path(),
)));
require_once 'Zend/Loader/Autoloader.php';
Zend_Loader_Autoloader::getInstance();

// Define some CLI options
```

```
$getopt = new Zend_Console_Getopt(array(
    'withdata|w' => 'Load database with sample data',
    'env|e-s'    => 'Application environment for which to create database (defaults to deve
    'help|h'     => 'Help -- usage message',
));
try {
    $getopt->parse();
} catch (Zend_Console_Getopt_Exception $e) {
    // Bad options passed: report usage
    echo $e->getUsageMessage();
    return false;
}

// If help requested, report usage message
if ($getopt->getOption('h')) {
    echo $getopt->getUsageMessage();
    return true;
}

// Initialize values based on presence or absence of CLI options
$withData = $getopt->getOption('w');
$env      = $getopt->getOption('e');
defined('APPLICATION_ENV')
    || define('APPLICATION_ENV', (null === $env) ? 'development' : $env);

// Initialize Zend_Application
$application = new Zend_Application(
    APPLICATION_ENV,
    APPLICATION_PATH . '/configs/application.ini'
);

// Initialize and retrieve DB resource
$bootstrap = $application->getBootstrap();
$bootstrap->bootstrap('db');
$dbAdapter = $bootstrap->getResource('db');

// let the user know whats going on (we are actually creating a
// database here)
if ('testing' != APPLICATION_ENV) {
    echo 'Writing Database Guestbook in (control-c to cancel): ' . PHP_EOL;
    for ($x = 5; $x > 0; $x--) {
        echo $x . "\r"; sleep(1);
    }
}

// Check to see if we have a database file already
$options = $bootstrap->getOption('resources');
$dbFile  = $options['db']['params']['dbname'];
if (file_exists($dbFile)) {
    unlink($dbFile);
}

// this block executes the actual statements that were loaded from
// the schema file.
try {
    $schemaSql = file_get_contents(dirname(__FILE__) . '/schema.sqlite.sql');
    // use the connection directly to load sql in batches
    $dbAdapter->getConnection()->exec($schemaSql);
    chmod($dbFile, 0666);
```

```
    if ('testing' != APPLICATION_ENV) {
        echo PHP_EOL;
        echo 'Database Created';
        echo PHP_EOL;
    }

    if ($withData) {
        $dataSql = file_get_contents(dirname(__FILE__) . '/data.sqlite.sql');
        // use the connection directly to load sql in batches
        $dbAdapter->getConnection()->exec($dataSql);
        if ('testing' != APPLICATION_ENV) {
            echo 'Data Loaded.';
            echo PHP_EOL;
        }
    }

} catch (Exception $e) {
    echo 'AN ERROR HAS OCCURED:' . PHP_EOL;
    echo $e->getMessage() . PHP_EOL;
    return false;
}

// generally speaking, this script will be run from the command line
return true;
```

Now, let's execute this script. From a terminal or the DOS command line, do the following:

```
% php scripts/load.sqlite.php --withdata
```

You should see output like the following:

```
path/to/ZendFrameworkQuickstart/scripts$ php load.sqlite.php --withdata
Writing Database Guestbook in (control-c to cancel):
1
Database Created
Data Loaded.
```

Now we have a fully working database and table for our guestbook application. Our next few steps are to build out our application code. This includes building a data source (in our case, we will use `Zend_Db_Table`), and a data mapper to connect that data source to our domain model. Finally we'll also create the controller that will interact with this model to both display existing entries and process new entries.

We'll use a Table Data Gateway to connect to our data source; `Zend_Db_Table` provides this functionality. To get started, lets create a `Zend_Db_Table`-based table class. First, create the directory `application/models/DbTable/`. Then create and edit a file `Guestbook.php` within it, and add the following contents:

```
// application/models/DbTable/Guestbook.php

/**
 * This is the DbTable class for the guestbook table.
 */
class Application_Model_DbTable_Guestbook extends Zend_Db_Table_Abstract
{
    /** Table name */
    protected $_name    = 'guestbook';
}
```

Note the class prefix: `Application_Model_DbTable`. The class prefix for our module, "Application", is the first segment, and then we have the component, "Model_DbTable"; the latter is mapped to the `models/DbTable/` directory of the module.

All that is truly necessary when extending `Zend_Db_Table` is to provide a table name and optionally the primary key (if it is not "id").

Now let's create a Data Mapper. A *Data Mapper* maps a domain object to the database. In our case, it will map our model, `Application_Model_Guestbook`, to our data source, `Application_Model_DbTable_Guestbook`. A typical API for a data mapper is as follows:

```
// application/models/GuestbookMapper.php

class Application_Model_GuestbookMapper
{
    public function save($model);
    public function find($id, $model);
    public function fetchAll();
}
```

In addition to these methods, we'll add methods for setting and retrieving the Table Data Gateway. The final class, located in `application/models/GuestbookMapper.php`, looks like this:

```
// application/models/GuestbookMapper.php

class Application_Model_GuestbookMapper
{
    protected $_dbTable;

    public function setDbTable($dbTable)
    {
        if (is_string($dbTable)) {
            $dbTable = new $dbTable();
        }
        if (!$dbTable instanceof Zend_Db_Table_Abstract) {
            throw new Exception('Invalid table data gateway provided');
        }
        $this->_dbTable = $dbTable;
        return $this;
    }

    public function getDbTable()
    {
        if (null === $this->_dbTable) {
            $this->setDbTable('Application_Model_DbTable_Guestbook');
        }
        return $this->_dbTable;
    }

    public function save(Application_Model_Guestbook $guestbook)
    {
        $data = array(
            'email'   => $guestbook->getEmail(),
            'comment' => $guestbook->getComment(),
            'created' => date('Y-m-d H:i:s'),
        );
```

```
        if (null === ($id = $guestbook->getId())) {
            unset($data['id']);
            $this->getDbTable()->insert($data);
        } else {
            $this->getDbTable()->update($data, array('id = ?' => $id));
        }
    }

    public function find($id, Application_Model_Guestbook $guestbook)
    {
        $result = $this->getDbTable()->find($id);
        if (0 == count($result)) {
            return;
        }
        $row = $result->current();
        $guestbook->setId($row->id)
                  ->setEmail($row->email)
                  ->setComment($row->comment)
                  ->setCreated($row->created);
    }

    public function fetchAll()
    {
        $resultSet = $this->getDbTable()->fetchAll();
        $entries   = array();
        foreach ($resultSet as $row) {
            $entry = new Application_Model_Guestbook();
            $entry->setId($row->id)
                  ->setEmail($row->email)
                  ->setComment($row->comment)
                  ->setCreated($row->created)
                  ->setMapper($this);
            $entries[] = $entry;
        }
        return $entries;
    }
}
```

Now it's time to update our model class slightly, to accomodate the data mapper. Just like the data mapper contains a reference to the data source, the model contains a reference to the data mapper. Additionally, we'll make it easy to populate the model by passing an array of data either to the constructor or a `setOptions()` method. The final model class, located in `application/models/Guestbook.php`, looks like this:

```
// application/models/Guestbook.php

class Application_Model_Guestbook
{
    protected $_comment;
    protected $_created;
    protected $_email;
    protected $_id;
    protected $_mapper;

    public function __construct(array $options = null)
    {
        if (is_array($options)) {
            $this->setOptions($options);
        }
```

```
    }

    public function __set($name, $value)
    {
        $method = 'set' . $name;
        if (('mapper' == $name) || !method_exists($this, $method)) {
            throw new Exception('Invalid guestbook property');
        }
        $this->$method($value);
    }

    public function __get($name)
    {
        $method = 'get' . $name;
        if (('mapper' == $name) || !method_exists($this, $method)) {
            throw new Exception('Invalid guestbook property');
        }
        return $this->$method();
    }

    public function setOptions(array $options)
    {
        $methods = get_class_methods($this);
        foreach ($options as $key => $value) {
            $method = 'set' . ucfirst($key);
            if (in_array($method, $methods)) {
                $this->$method($value);
            }
        }
        return $this;
    }

    public function setComment($text)
    {
        $this->_comment = (string) $text;
        return $this;
    }

    public function getComment()
    {
        return $this->_comment;
    }

    public function setEmail($email)
    {
        $this->_email = (string) $email;
        return $this;
    }

    public function getEmail()
    {
        return $this->_email;
    }

    public function setCreated($ts)
    {
        $this->_created = $ts;
        return $this;
    }
```

```
    public function getCreated()
    {
        return $this->_created;
    }

    public function setId($id)
    {
        $this->_id = (int) $id;
        return $this;
    }

    public function getId()
    {
        return $this->_id;
    }

    public function setMapper($mapper)
    {
        $this->_mapper = $mapper;
        return $this;
    }

    public function getMapper()
    {
        if (null === $this->_mapper) {
            $this->setMapper(new Application_Model_GuestbookMapper());
        }
        return $this->_mapper;
    }

    public function save()
    {
        $this->getMapper()->save($this);
    }

    public function find($id)
    {
        $this->getMapper()->find($id, $this);
        return $this;
    }

    public function fetchAll()
    {
        return $this->getMapper()->fetchAll();
    }
}
```

Lastly, to connect these elements all together, lets create a guestbook controller that will both list the entries that are currently inside the database.

To create a new controller, open a terminal or DOS console, navigate to your project directory, and enter the following:

```
# Unix-like systems:
% zf.sh create controller guestbook

# DOS/Windows:
C:> zf.bat create controller guestbook
```

This will create a new controller, `GuestbookController`, in `application/controllers/GuestbookController.php`, with a single action method, `indexAction()`. It will also create a view script directory for the controller, `application/views/scripts/guestbook/`, with a view script for the index action.

We'll use the "index" action as a landing page to view all guestbook entries.

Now, let's flesh out the basic application logic. On a hit to `indexAction()`, we'll display all guestbook entries. This would look like the following:

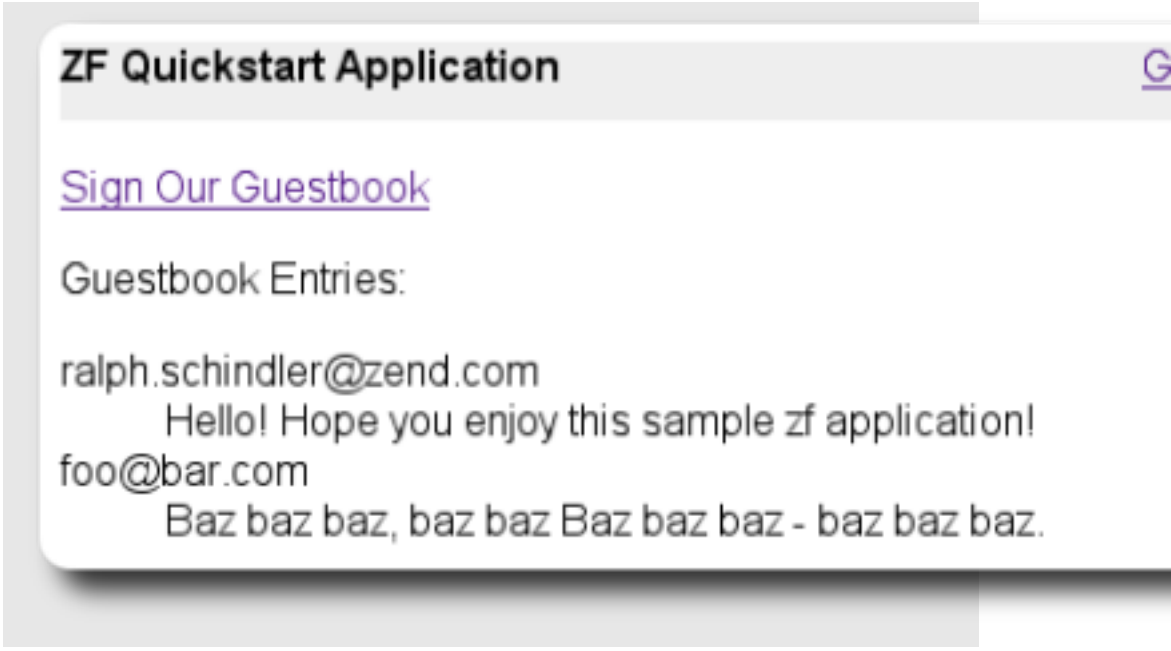
```
// application/controllers/GuestbookController.php

class GuestbookController extends Zend_Controller_Action
{
    public function indexAction()
    {
        $guestbook = new Application_Model_Guestbook();
        $this->view->entries = $guestbook->fetchAll();
    }
}
```

And, of course, we need a view script to go along with that. Edit `application/views/scripts/guestbook/index.phtml` to read as follows:

```
<!-- application/views/scripts/guestbook/index.phtml -->

<p><a href="<?php echo $this->url(
    array(
        'controller' => 'guestbook',
        'action'     => 'sign'
    ),
    'default',
    true) ?>">Sign Our Guestbook</a></p>

Guestbook Entries: <br />
<dl>
    <?php foreach ($this->entries as $entry): ?>
    <dt><?php echo $this->escape($entry->email) ?></dt>
    <dd><?php echo $this->escape($entry->comment) ?></dd>
    <?php endforeach ?>
</dl>
```

Checkpoint

Now browse to "http://localhost/guestbook". You should see the following in your browser:



Using the data loader script

The data loader script introduced in this section (`scripts/load.sqlite.php`) can be used to create the database for each environment you have defined, as well as to load it with sample data. Internally, it utilizes `Zend_Console_Getopt`, which allows it to provide a number of command line switches. If you pass the "-h" or "--help" switch, it will give you the available options:

```
Usage: load.sqlite.php [ options ]
--withdata|-w         Load database with sample data
--env|-e [  ]         Application environment for which to create database
                      (defaults to development)
--help|-h             Help -- usage message)]]
```

The "-e" switch allows you to specify the value to use for the constant `APPLICATION_ENV` -- which in turn allows you to create a SQLite database for each environment you define. Be sure to run the script for the environment you choose for your application when deploying.

5. Create A Form

For our guestbook to be useful, we need a form for submitting new entries.

Our first order of business is to create the actual form class. First, create the directory `application/forms/`. This directory will contain form classes for the application. Next, we'll create a form class in `application/forms/Guestbook.php`:

```
// application/forms/Guestbook.php

class Application_Form_Guestbook extends Zend_Form
{
    public function init()
    {
```

```
        // Set the method for the display form to POST
        $this->setMethod('post');

        // Add an email element
        $this->addElement('text', 'email', array(
            'label'      => 'Your email address:',
            'required'   => true,
            'filters'    => array('StringTrim'),
            'validators' => array(
                'EmailAddress',
            )
        ));

        // Add the comment element
        $this->addElement('textarea', 'comment', array(
            'label'      => 'Please Comment:',
            'required'   => true,
            'validators' => array(
                array('validator' => 'StringLength', 'options' => array(0, 20))
                )
        ));

        // Add a captcha
        $this->addElement('captcha', 'captcha', array(
            'label'      => 'Please enter the 5 letters displayed below:',
            'required'   => true,
            'captcha'    => array(
                'captcha' => 'Figlet',
                'wordLen' => 5,
                'timeout' => 300
            )
        ));

        // Add the submit button
        $this->addElement('submit', 'submit', array(
            'ignore'   => true,
            'label'    => 'Sign Guestbook',
        ));

        // And finally add some CSRF protection
        $this->addElement('hash', 'csrf', array(
            'ignore' => true,
        ));
    }
}
```

The above form defines five elements: an email address field, a comment field, a CAPTCHA for preventing spam submissions, a submit button, and a CSRF protection token.

Next, we will add a `signAction()` to our `GuestbookController` which will process the form upon submission. To create the action and related view script, execute the following:

```
# Unix-like systems:
% zf.sh create action sign guestbook

# DOS/Windows:
C:> zf.bat create action sign guestbook
```

This will create a `signAction()` method in our controller, as well as the appropriate view script.

Let's add some logic into our guestbook controller's sign action. We need to first check if we're getting a POST or a GET request; in the latter case, we'll simply display the form. However, if we get a POST request, we'll want to validate the posted data against our form, and, if valid, create a new entry and save it. The logic might look like this:

```
// application/controllers/GuestbookController.php

class GuestbookController extends Zend_Controller_Action
{
    // snipping indexAction()...

    public function signAction()
    {
        $request = $this->getRequest();
        $form    = new Application_Form_Guestbook();

        if ($this->getRequest()->isPost()) {
            if ($form->isValid($request->getPost())) {
                $model = new Application_Model_Guestbook($form->getValues());
                $model->save();
                return $this->_helper->redirector('index');
            }
        }

        $this->view->form = $form;
    }
}
```

Of course, we also need to edit the view script; edit `application/views/scripts/guestbook/sign.phtml` to read:

```
<!-- application/views/scripts/guestbook/sign.phtml -->

Please use the form below to sign our guestbook!

<?php
$this->form->setAction($this->url());
echo $this->form;
```

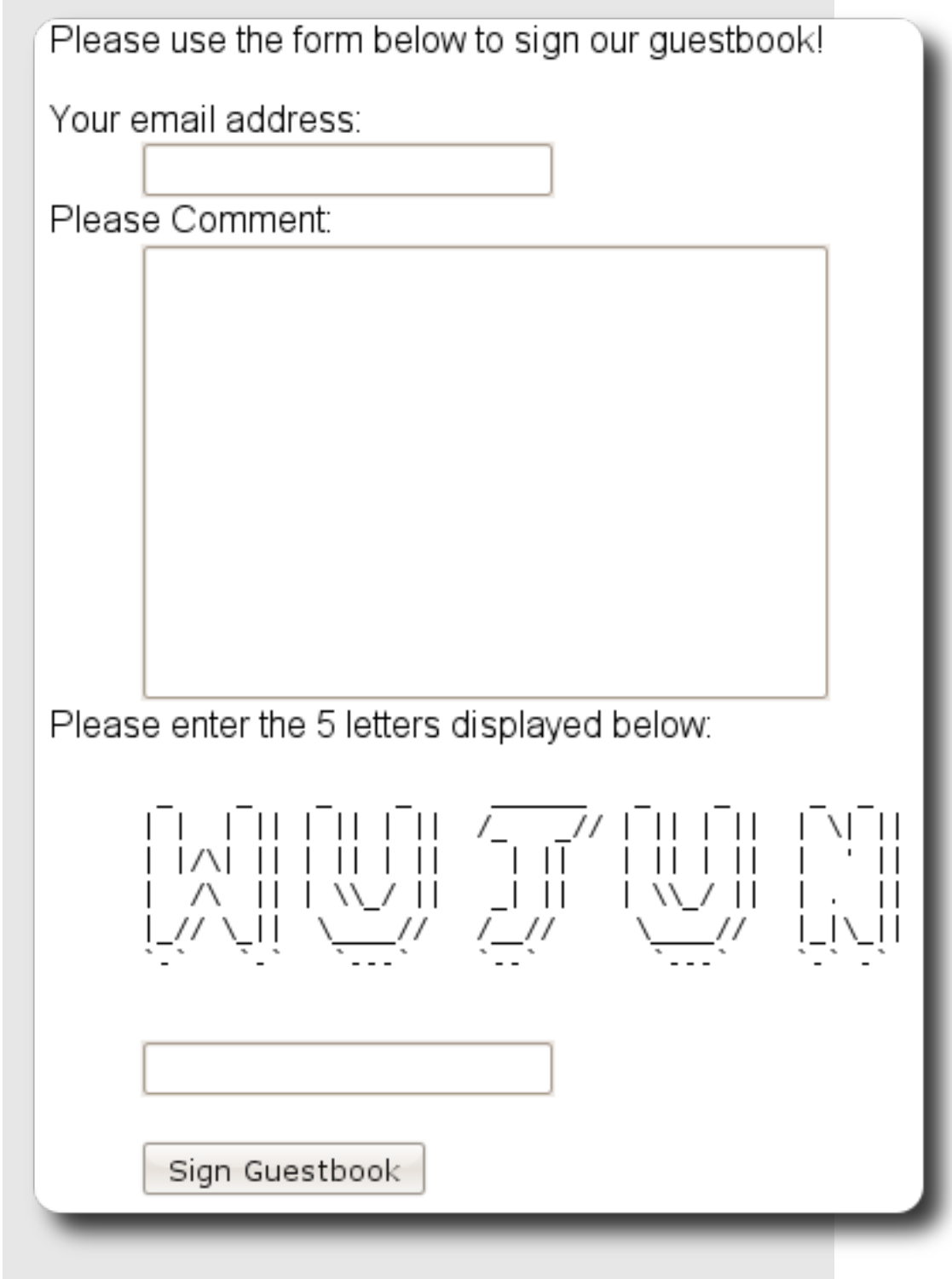
Better Looking Forms

No one will be waxing poetic about the beauty of this form anytime soon. No matter - form appearance is fully customizable! See the decorators section in the reference guide for details.

Additionally, you may be interested in this series of posts on decorators.

Checkpoint

Now browse to "http://localhost/guestbook/sign". You should see the following in your browser:



6. Congratulations!

You have now built a very simple application using some of the most commonly used Zend Framework components. Zend Framework makes many components available to you which address most common requirements in web applications, including web services, search, PDF

reading and writing, authentication, authorization, and much more. The Reference Guide is a great place to find out more about the components you've used in this QuickStart as well as other components. We hope you find Zend Framework useful and - more importantly - fun!

Autoloading in Zend Framework

1. Introduction

Autoloading is a mechanism that eliminates the need to manually require dependencies within your PHP code. Per the PHP autoload manual, once an autoloader has been defined, it "is automatically called in case you are trying to use a class or an interface which hasn't been defined yet."

Using autoloading, you do not need to worry about *where* a class exists in your project. With well-defined autoloaders, you do not need to worry about where a class file is relative to the current class file; you simply use the class, and the autoloader will perform the file lookup.

Additionally, autoloading, because it defers loading to the last possible moment and ensures that a match only has to occur once, can be a huge performance boost -- particularly if you take the time to strip out `require_once()` calls before you move to deployment.

Zend Framework encourages the use of autoloading, and provides several tools to provide autoloading of both library code as well as application code. This tutorial covers these tools, as well as how to use them effectively.

2. Goals and Design

2.1. Class Naming Conventions

To understand autoloading in Zend Framework, first you need to understand the relationship between class names and class files.

Zend Framework has borrowed an idea from PEAR, whereby class names have a 1:1 relationship with the filesystem. Simply put, the underscore character ("_") is replaced by a directory separator in order to resolve the path to the file, and then the suffix "`.php`" is added. For example, the class "`Foo_Bar_Baz`" would correspond to "`Foo/Bar/Baz.php`" on the filesystem. The assumption is also that the classes may be resolved via PHP's include_path setting, which allows both `include()` and `require()` to find the filename via a relative path lookup on the include_path.

Additionally, per PEAR as well as the PHP project, we use and recommend using a vendor or project prefix for your code. What this means is that all classes you write will share a common class prefix; for example, all code in Zend Framework has the prefix "Zend_". This naming convention helps prevent naming collisions. Within Zend Framework, we often refer to this as the "namespace" prefix; be careful not to confuse it with PHP's native namespace implementation.

Zend Framework follows these simple rules internally, and our coding standards encourage that you do so as well for all library code.

2.2. Autoloader Conventions and Design

Zend Framework's autoloading support, provided primarily via `Zend_Loader_Autoloader`, has the following goals and design elements:

- *Provide namespace matching*. If the class namespace prefix is not in a list of registered namespaces, return `FALSE` immediately. This allows for more optimistic matching, as well as fallback to other autoloaders.

- *Allow the autoloader to act as a fallback autoloader*. In the case where a team may be widely distributed, or using an undetermined set of namespace prefixes, the autoloader should still be configurable such that it will attempt to match any namespace prefix. It will be noted, however, that this practice is not recommended, as it can lead to unnecessary lookups.
- *Allow toggling error suppression*. We feel -- and the greater PHP community does as well -- that error suppression is a bad idea. It's expensive, and it masks very real application problems. So, by default, it should be off. However, if a developer *insists* that it be on, we allow toggling it on.
- *Allow specifying custom callbacks for autoloading*. Some developers don't want to use `Zend_Loader::loadClass()` for autoloading, but still want to make use of Zend Framework's mechanisms. `Zend_Loader_Autoloader` allows specyfing an alternate callback for autoloading.
- *Allow manipulation of the SPL autoload callback chain*. The purpose of this is to allow specifying additional autoloaders -- for instance, resource loaders for classes that don't have a 1:1 mapping to the filesystem -- to be registered before or after the primary Zend Framework autoloader.

3. Basic Autoloader Usage

Now that we have an understanding of what autoloading is and the goals and design of Zend Framework's autoloading solution, let's look at how to use `Zend_Loader_Autoloader`.

In the simplest case, you would simply require the class, and then instantiate it. Since `Zend_Loader_Autoloader` is a singleton (due to the fact that the SPL autoloader is a single resource), we use `getInstance()` to retrieve an instance.

```
require_once 'Zend/Loader/Autoloader.php';
Zend_Loader_Autoloader::getInstance();
```

By default, this will allow loading any classes with the class namespace prefixes of "Zend_" or "ZendX_", as long as they are on your include_path.

What happens if you have other namespace prefixes you wish to use? The best, and simplest, way is to call the `registerNamespace()` method on the instance. You can pass a single namespace prefix, or an array of them:

```
require_once 'Zend/Loader/Autoloader.php';
$loader = Zend_Loader_Autoloader::getInstance();
$loader->registerNamespace('Foo_');
$loader->registerNamespace(array('Foo_', 'Bar_'));
```

Alternately, you can tell `Zend_Loader_Autoloader` to act as a "fallback" autoloader. This means that it will try to resolve any class regardless of namespace prefix.

```
$loader->setFallbackAutoloader(true);
```

Do not use as a fallback autoloader

While it's tempting to use `Zend_Loader_Autoloader` as a fallback autoloader, we do not recommend the practice.

Internally, `Zend_Loader_Autoloader` uses `Zend_Loader::loadClass()` to load classes. That method uses `include()` to attempt to load the given class

file. `include()` will return a boolean FALSE if not successful -- but also issues a PHP warning. This latter fact can lead to some issues:

- If display_errors is enabled, the warning will be included in output.
- Depending on the error_reporting level you have chosen, it could also clutter your logs.

You can suppress the error messages (the `Zend_Loader_Autoloader` documentation details this), but note that the suppression is only relevant when display_errors is enabled; the error log will always display the messages. For these reasons, we recommend always configuring the namespace prefixes the autoloader should be aware of

Namespace Prefixes vs PHP Namespaces

At the time this is written, PHP 5.3 has been released. With that version, PHP now has official namespace support.

However, Zend Framework predates PHP 5.3, and thus namespaces. Within Zend Framework, when we refer to "namespaces", we are referring to a practice whereby classes are prefixed with a vender "namespace". As an example, all Zend Framework class names are prefixed with "Zend_" -- that is our vendor "namespace".

Zend Framework plans to offer native PHP namespace support to the autoloader in future revisions, and its own library will utilize namespaces starting with version 2.0.0.

If you have a custom autoloader you wish to use with Zend Framework -- perhaps an autoloader from a third-party library you are also using -- you can manage it with `Zend_Loader_Autoloader`'s `pushAutoloader()` and `unshiftAutoloader()` methods. These methods will append or prepend, respectively, autoloaders to a chain that is called prior to executing Zend Framework's internal autoloading mechanism. This approach offers the following benefits:

- Each method takes an optional second argument, a class namespace prefix. This can be used to indicate that the given autoloader should only be used when looking up classes with that given class prefix. If the class being resolved does not have that prefix, the autoloader will be skipped -- which can lead to performance improvements.
- If you need to manipulate `spl_autoload()`'s registry, any autoloaders that are callbacks pointing to instance methods can pose issues, as `spl_autoload_functions()` does not return the exact same callbacks. `Zend_Loader_Autoloader` has no such limitation.

Autoloaders managed this way may be any valid PHP callback.

```
// Append function 'my_autoloader' to the stack,
// to manage classes with the prefix 'My_':
$loader->pushAutoloader('my_autoloader', 'My_');

// Prepend static method Foo_Loader::autoload() to the stack,
// to manage classes with the prefix 'Foo_':
$loader->unshiftAutoloader(array('Foo_Loader', 'autoload'), 'Foo_');
```

4. Resource Autoloading

Often, when developing an application, it's either difficult to package classes in the 1:1 classname:filename standard Zend Framework recommends, or it's advantageous for purposes of packaging not to do so. However, this means you class files will not be found by the autoloader.

If you read through the design goals for the autoloader, the last point in that section indicated that the solution should cover this situation. Zend Framework does so with `Zend_Loader_Autoloader_Resource`.

A resource is just a name that corresponds to a component namespace (which is appended to the autoloader's namespace) and a path (which is relative to the autoloader's base path). In action, you'd do something like this:

```
$loader = new Zend_Application_Module_Autoloader(array(
    'namespace' => 'Blog',
    'basePath'  => APPLICATION_PATH . '/modules/blog',
));
```

Once you have the loader in place, you then need to inform it of the various resource types it's aware of. These resource types are simply pairs of subtree and prefix.

As an example, consider the following tree:

```
path/to/some/resources/
|-- forms/
|   `-- Guestbook.php        // Foo_Form_Guestbook
|-- models/
|   |-- DbTable/
|   |   `-- Guestbook.php    // Foo_Model_DbTable_Guestbook
|   |-- Guestbook.php        // Foo_Model_Guestbook
|   `-- GuestbookMapper.php  // Foo_Model_GuestbookMapper
```

Our first step is creating the resource loader:

```
$loader = new Zend_Loader_Autoloader_Resource(array(
    'basePath'  => 'path/to/some/resources/',
    'namespace' => 'Foo',
));
```

Next, we need to define some resource types. `Zend_Loader_Autoloader_Resourse::addResourceType()` has three arguments: the "type" of resource (an arbitrary string), the path under the base path in which the resource type may be found, and the component prefix to use for the resource type. In the above tree, we have three resource types: form (in the subdirectory "forms", with a component prefix of "Form"), model (in the subdirectory "models", with a component prefix of "Model"), and dbtable (in the subdirectory "`models/DbTable`", with a component prefix of "`Model_DbTable`"). We'd define them as follows:

```
$loader->addResourceType('form', 'forms', 'Form')
       ->addResourceType('model', 'models', 'Model')
       ->addResourceType('dbtable', 'models/DbTable', 'Model_DbTable');
```

Once defined, we can simply use these classes:

```
$form      = new Foo_Form_Guestbook();
$guestbook = new Foo_Model_Guestbook();
```

Module Resource Autoloading

Zend Framework's MVC layer encourages the use of "modules", which are self-contained applications within your site. Modules typically have a number of resource types by default, and Zend Framework even recommends a standard directory layout for modules. Resource autoloaders are therefore quite useful in this paradigm -- so useful that they are enabled by default when you create a bootstrap class for your module that extends `Zend_Application_Module_Bootstrap`. For more information, read the Zend_Loader_Autoloader_Module documentation.

5. Conclusion

Zend Framework encourages the use of autoloading, and even initializes it by default in `Zend_Application`. Hopefully this tutorial provides you with the information you need to use `Zend_Loader_Autoloader` to its best advantage, as well as extend its capabilities by attaching custom autoloaders or resource autoloaders.

For more information on its usage, read the Zend_Loader_Autoloader and Zend_Loader_Autoloader_Resource manual pages.

Plugins in Zend Framework

1. Introduction

Zend Framework makes heavy use of plugin architectures. Plugins allow for easy extensibility and customization of the framework while keeping your code separate from Zend Framework's code.

Typically, plugins in Zend Framework work as follows:

- Plugins are classes. The actual class definition will vary based on the component -- you may need to extend an abstract class or implement an interface, but the fact remains that the plugin is itself a class.

- Related plugins will share a common class prefix. For instance, if you have created a number of view helpers, they might all share the class prefix "`Foo_View_Helper_`".

- Everything after the common prefix will be considered the *plugin name* or *short name* (versus the "long name", which is the full classname). For example, if the plugin prefix is "`Foo_View_Helper_`", and the class name is "`Foo_View_Helper_Bar`", the plugin name will be simply "`Bar`".

- Plugin names are typically case sensitive. The one caveat is that the initial letter can often be either lower or uppercase; in our previous example, both "bar" and "Bar" would refer to the same plugin.

Now let's turn to using plugins.

2. Using Plugins

Components that make use of plugins typically use `Zend_Loader_PluginLoader` to do their work. This class has you register plugins by specifying one or more "prefix paths". The component will then call the PluginLoader's `load()` method, passing the plugin's short name to it. The PluginLoader will then query each prefix path to see if a class matching that short name exists. Prefix paths are searched in LIFO (last in, first out) order, so it will match those prefix paths registered last first -- allowing you to override existing plugins.

Some examples will make all of this more clear.

Example 1. Basic Plugin Example: Adding a single prefix path

In this example, we will assume some validators have been written and placed in the directory `foo/plugins/validators/`, and that all these classes share the class prefix "Foo_Validate_"; these two bits of information form our "prefix path". Furthermore, let's assume we have two validators, one named "Even" (ensuring a number to be validated is even), and another named "Dozens" (ensuring the number is a multiple of 12). The tree might look like this:

```
foo/
|-- plugins/
|   |-- validators/
|   |   |-- Even.php
|   |   |-- Dozens.php
```

Now, we'll inform a `Zend_Form_Element` instance of this prefix path. `Zend_Form_Element`'s `addPrefixPath()` method expects a third argument that indicates the type of plugin for which the path is being registered; in this case, it's a "validate" plugin.

```
$element->addPrefixPath('Foo_Validate', 'foo/plugins/validators/', 'validate');
```

Now we can simply tell the element the short name of the validators we want to use. In the following example, we're using a mix of standard validators ("NotEmpty", "Int") and custom validators ("Even", "Dozens"):

```
$element->addValidator('NotEmpty')
        ->addValidator('Int')
        ->addValidator('Even')
        ->addValidator('Dozens');
```

When the element needs to validate, it will then request the plugin class from the PluginLoader. The first two validators will resolve to `Zend_Validate_NotEmpty` and `Zend_Validate_Int`, respectively; the next two will resolve to `Foo_Validate_Even` and `Foo_Validate_Dozens`, respectively.

What happens if a plugin is not found?

What happens if a plugin is requested, but the PluginLoader is unable to find a class matching it? For instance, in the above example, if we registered the plugin "Bar" with the element, what would happen?

The plugin loader will look through each prefix path, checking to see if a file matching the plugin name is found on that path. If the file is not found, it then moves on to the next prefix path.

Once the stack of prefix paths has been exhausted, if no matching file has been found, it will throw a `Zend_Loader_PluginLoader_Exception`.

Example 2. Intermediate Plugin Usage: Overriding existing plugins

One strength of the PluginLoader is that its use of a LIFO stack allows you to override existing plugins by creating your own versions locally with a different prefix path, and registering that prefix path later in the stack.

For example, let's consider `Zend_View_Helper_FormButton` (view helpers are one form of plugin). This view helper accepts three arguments, an element name (also used as the element's DOM identifier), a value (used as the button label), and an optional array of attributes. The helper then generates HTML markup for a form input element.

Let's say you want the helper to instead generate a true HTML `button` element; don't want the helper to generate a DOM identifier, but instead use the value for a CSS class selector; and that you have no interest in handling arbitrary attributes. You could accomplish this in a couple of ways. In both cases, you'd create your own view helper class that implements the behavior you want; the difference is in how you would name and invoke them.

Our first example will be to name the element with a unique name: `Foo_View_Helper_CssButton`, which implies the plugin name "CssButton". While this certainly is a viable approach, it poses several issues: if you've already used the Button view helper in your code, you now have to refactor; alternately, if another developer starts writing code for your application, they may inadvertently use the Button view helper instead of your new view helper.

So, the better example is to use the plugin name "Button", giving us the class name `Foo_View_Helper_Button`. We then register the prefix path with the view:

```
// Zend_View::addHelperPath() utilizes the PluginLoader; however, it inverts
// the arguments, as it provides a default value of "Zend_View_Helper" for the
// plugin prefix.
//
// The below assumes your class is in the directory 'foo/view/helpers/'.
$view->addHelperPath('foo/view/helpers', 'Foo_View_Helper');
```

Once done, anywhere you now use the "Button" helper will delegate to your custom `Foo_View_Helper_Button` class!

3. Conclusion

Understanding the concept of prefix paths and overriding existing plugins will help you with your understanding of many components within the framework. Plugins are used in a variety of places:

- `Zend_Application`: resources.
- `Zend_Controller_Action`: action helpers.
- `Zend_Feed_Reader`: plugins.
- `Zend_Form`: elements, filters, validators, and decorators.
- `Zend_View`: view helpers.

And several more places, besides. Learn the concepts early so you can leverage this important extension point in Zend Framework.

Caveat

We'll note here that `Zend_Controller_Front` has a plugin system - but it does not adhere to any of the guidelines offerred in this tutorial. The plugins registered with the front controller must be instantiated directly and registered individually with it. The reason for this is that this system predates any other plugin system in the framework, and changes to it must be carefully weighed to ensure existing plugins written by developers continue to work with it.

Getting Started with Zend_Layout

1. Introduction

When building a website using Zend Framework MVC layers, your view scripts will typically be just snippets of HTML pertinent to the requested action. For instance, if you had the action "`/user/list`", you might create a view script that iterates through the users and presents an unordered list:

```
<h2>Users</h2>
<ul>
    <?php if (!count($this->users)): ?>
    <li>No users found</li>
    <?php else: ?>
    <?php foreach ($this->users as $user): ?>
    <li>
        <?php echo $this->escape($user->fullname) ?>
        (<?php echo $this->escape($user->email) ?>)
    </li>
    <?php endforeach ?>
    <?php endif ?>
</ul>
```

Since this is just a snippet of HTML, it's not a valid page; it's missing a DOCTYPE declaration, and the opening HTML and BODY tags. So, the question is, where will these be created?

In early versions of Zend Framework, developers often created "header" and "footer" view scripts that had these artifacts, and then in each view script they would render them. While this methodology worked, it also made it difficult to refactor later, or to build composite content by calling multiple actions.

The Two Step View design pattern answers many of the issues presented. In this pattern, the "application" view is created first, and then injected into the "page" view, which is then presented to the client. The page view can be thought of as your site-wide template or layout, and would have common elements used across various pages.

Within Zend Framework, `Zend_Layout` implements the Two Step View pattern.

2. Using Zend_Layout

Basic usage of `Zend_Layout` is fairly trivial. Assuming you're using `Zend_Application` already, you can simply provide some configuration options and create a layout view script.

2.1. Layout Configuration

The recommended location of layouts is in a "`layouts/scripts/`" subdirectory of your application:

```
application
|-- Bootstrap.php
|-- configs
|   `-- application.ini
|-- controllers
```

```
|-- layouts
|   `-- scripts
|       |-- layout.phtml
```

To initialize `Zend_Layout`, add the following to your configuration file ("`application/configs/application.ini`"):

```
resources.layout.layoutPath = APPLICATION_PATH "/layouts/scripts"
resources.layout.layout = "layout"
```

The first line indicates where to look for layout scripts; the second line gives the name of the layout to use, minus the view script extension (which is assumed to be "`.phtml`" by default).

2.2. Create a Layout Script

Now that you have your configuration in place, you need to create your layout script. First, make sure that you've created the "`application/layouts/scripts`" directory; then, open an editor, and create the markup for your layout. Layout scripts are simply view scripts, with some slight differences.

```
<html>
<head>
    <title>My Site</title>
</head>
<body>
    <?php echo $this->layout()->content ?>
</body>
</html>
```

In the example above, you'll note the call to a `layout()` view helper. When you register the `Zend_Layout` resource, you also gain access to both an action and view helper that allow you access to the `Zend_Layout` instance; you can then call operations on the layout object. In this case, we're retrieving a named variable, `$content`, and echoing it. By default, the `$content` variable is populated for you from the application view script rendered. Otherwise, anything you'd normally do in a view script is perfectly valid -- call any helpers or view methods you desire.

At this point, you have a working layout script, and your application is informed of its location and knows to render it.

2.3. Accessing the Layout Object

On occasion, you may need direct access to the layout object. There are three ways you can do this:

- *Within view scripts:* use the `layout()` view helper, which returns the `Zend_Layout` instance registered with the front controller plugin.

  ```
  <?php $layout = $this->layout(); ?>
  ```

  Since it returns the layout instance, you can also simply call methods on it, rather than assigning it to a variable.

- *Within action controllers:* use the `layout()` action helper, which acts just like the view helper.

  ```
  // Calling helper as a method of the helper broker:
  ```

```
$layout = $this->_helper->layout();

// Or, more verbosely:
$helper = $this->_helper->getHelper('Layout');
$layout = $helper->getLayoutInstance();
```

As with the view helper, since the action helper returns the layout instance, you can also simply call methods on it, rather than assigning it to a variable.

- *Elsewhere:* use the static method `getMvcInstance()`. This will return the layout instance registered by the bootstrap resource.

```
$layout = Zend_Layout::getMvcInstance();
```

- *Via the bootstrap:* retrieve the layout resource, which will be the `Zend_Layout` instance.

```
$layout = $bootstrap->getResource('Layout');
```

Anywhere you have access to the bootstrap object, this method is preferred over using the static `getMvcInstance()` method.

2.4. Other Operations

In most cases, the above configuration and layout script (with modifications) will get you what you need. However, some other functionality exists you will likely use sooner or later. In all of the following examples, you may use one of the methods listed above for retrieving the layout object.

- *Setting layout variables*. `Zend_Layout` keeps its own registry of layout-specific view variables that you can access; the `$content` key noted in the initial layout script sample is one such example. You can assign and retrieve these using normal property access, or via the `assign()` method.

```
// Setting content:
$layout->somekey = "foo"

// Echoing that same content:
echo $layout->somekey; // 'foo'

// Using the assign() method:
$layout->assign('someotherkey', 'bar');

// Access to assign()'d variables remains the same:
echo $layout->someotherkey; // 'bar'
```

- `disableLayout()`. Occasionally, you may want to disable layouts; for example, when answering an Ajax request, or providing a RESTful representation of a resource. In these cases, you can call the `disableLayout()` method on your layout object.

```
$layout->disableLayout();
```

The opposite of this method is, of course, `enableLayout()`, which can be called at any time to re-enable layouts for the requested action.

- *Selecting an alternate layout*: If you have multiple layouts for your site or application, you can select the layout to use at any time by simply calling the `setLayout()` method. Call it by specifying the name of the layout script without the file suffix.

```
// Use the layout script "alternate.phtml":
$layout->setLayout('alternate');
```

The layout script should reside in the `$layoutPath` directory specified in your configuration. `Zend_Layout` will then use this new layout when rendering.

3. Zend_Layout: Conclusions

`Zend_Layout` is a very simple wrapper around `Zend_View` which provides you immediately with the benefits of a Two Step View, giving you the flexibility to create a site-wide design into which you may inject your application content.

If you're looking closely at the examples, however, you may come away with the idea that the functionality is relatively limited: how do you alter the page title, inject an optional script tag, or even create an optional sidebar? These questions are addressed with the concept of a "Composite View" -- and are the subject of the next chapter in the tutorial, which covers view "placeholders."

Getting Started Zend_View Placeholders

1. Introduction

In the previous chapter, we looked at primarily the Two Step View pattern, which allows you to embed individual application views within a sitewide layout. At the end of that chapter, however, we discussed some limitations:

- How do you alter the page title?
- How would you inject conditional scripts or stylesheets into the sitewide layout?
- How would you create and render an optional sidebar? What if there was some content that was unconditional, and other content that was conditional for the sidebar?

These questions are addressed in the Composite View design pattern. One approach to that pattern is to provide "hints" or content to the sitewide layout. In Zend Framework, this is achieved through specialized view helpers called "placeholders." Placeholders allow you to aggregate content, and then render that aggregate content elsewhere.

2. Basic Placeholder Usage

Zend Framework defines a generic `placeholder()` view helper that you may use for as many custom placeholders you need. It also provides a variety of specific placeholder implementations for often-needed functionality, such as specifying the *DocType* declaration, document title, and more.

All placeholders operate in roughly the same way. They are containers, and thus allow you to operate on them as collections. With them you can:

- *Append* or *prepend* items to the collection.
- *Replace* the entire collection with a single value.
- Specify a string with which to *prepend output* of the collection when rendering.
- Specify a string with which to *append output* of the collection when rendering.
- Specify a string with which to *separate items* of the collection when rendering.
- *Capture content* into the collection.
- *Render* the aggregated content.

Typically, you will call the helper with no arguments, which will return a container on which you may operate. You will then either echo this container to render it, or call methods on it to configure or populate it. If the container is empty, rendering it will simply return an empty string; otherwise, the content will be aggregated according to the rules by which you configure it.

As an example, let's create a sidebar that consists of a number of "blocks" of content. You'll likely know up-front the structure of each block; let's assume for this example that it might look like this:

```
<div class="sidebar">
    <div class="block">
        <p>
            Lorem ipsum dolor sit amet, consectetur adipiscing elit. Vivamus
            consectetur aliquet odio ac consectetur. Nulla quis eleifend
            tortor. Pellentesque varius, odio quis bibendum consequat, diam
            lectus porttitor quam, et aliquet mauris orci eu augue.
        </p>
    </div>
    <div class="block">
        <ul>
            <li><a href="/some/target">Link</a></li>
            <li><a href="/some/target">Link</a></li>
        </ul>
    </div>
</div>
```

The content will vary based on the controller and action, but the structure will be the same. Let's first setup the sidebar in a resource method of our bootstrap:

```
class Bootstrap extends Zend_Application_Bootstrap_Bootstrap
{
    // ...

    protected function _initSidebar()
    {
        $this->bootstrap('View');
        $view = $this->getResource('View');

        $view->placeholder('sidebar')
             // "prefix" -> markup to emit once before all items in collection
             ->setPrefix("<div class=\"sidebar\">\n    <div class=\"block\">\n")
             // "separator" -> markup to emit between items in a collection
             ->setSeparator("</div>\n    <div class=\"block\">\n")
             // "postfix" -> markup to emit once after all items in a collection
             ->setPostfix("</div>\n</div>");
    }

    // ...
}
```

The above defines a placeholder, "sidebar", that has no items. It configures the basic markup structure of that placeholder, however, per our requirements.

Now, let's assume for the "user" controller that for all actions we'll want a block at the top containing some information. We could accomplish this in two ways: (a) we could add the content to the placeholder directly in the controller's `preDispatch()` method, or (b) we could render a view script from within the `preDispatch()` method. We'll use (b), as it follows a more proper separation of concerns (leaving view-related logic and functionality within a view script).

We'll name the view script "`user/_sidebar.phtml`", and populate it as follows:

```
<?php $this->placeholder('sidebar')->captureStart() ?>
<h4>User Administration</h4>
<ul>
    <li><a href="<?php $this->url(array('action' => 'list')) ?>">
        List</a></li>
    <li><a href="<?php $this->url(array('action' => 'create')) ?>">
```

```
        Create</a></a></li>
</ul>
<?php $this->placeholder('sidebar')->captureEnd() ?>
```

The above example makes use of the content capturing feature of placeholders. By default, content is appended as a new item in the container, allowing us to aggregate content. This example makes use of view helpers and static HTML in order to generate markup, and the content is then captured and appended into the placeholder itself.

To invoke the above view script, we would write the following in our `preDispatch()` method:

```
class UserController extends Zend_Controller_Action
{
    // ...

    public function preDispatch()
    {
        // ...

        $this->view->render('user/_sidebar.phtml');

        // ...
    }

    // ...
}
```

Note that we're not capturing the rendered value; there's no need, as the entierty of that view is being captured into a placeholder.

Now, let's assume our "view" action in that same controller needs to present some information. Within the "`user/view.phtml`" view script, we might have the following snippet of content:

```
$this->placeholder('sidebar')
     ->append('<p>User: ' . $this->escape($this->username) .  '</p>');
```

This example makes use of the `append()` method, and passes it some simple markup to aggregate.

Finally, let's modify our layout view script, and have it render the placeholder.

```
<html>
<head>
    <title>My Site</title>
</head>
<body>
    <div class="content">
        <?php echo $this->layout()->content ?>
    </div>
    <?php echo $this->placeholder('sidebar') ?>
</body>
</html>
```

For controllers and actions that do not populate the "sidebar" placeholder, no content will be rendered; for those that do, however, echoing the placeholder will render the content according to the rules we created in our bootstrap, and the content we aggregated throughout the application.

In the case of the "`/user/view`" action, and assuming a username of "matthew", we would get content for the sidebar as follows (formatted for readability):

```
<div class="sidebar">
    <div class="block">
        <h4>User Administration</h4>
        <ul>
            <li><a href="/user/list">List</a></li>
            <li><a href="/user/create">Create</a></a></li>
        </ul>
    </div>
    <div class="block">
        <p>User: matthew</p>
    </div>
</div>
```

There are a large number of things you can do by combining placeholders and layout scripts; experiment with them, and read the relevant manual sections for more information.

3. Standard Placeholders

In the previous section, we learned about the `placeholder()` view helper, and how it can be used to aggregate custom content. In this section, we'll look at some of the concrete placeholders shipped with Zend Framework, and how you can use them to your advantage when creating complex composite layouts.

Most of the shipped placeholders are for generating content for the `<head>` section of your layout content -- an area you typically cannot manipulate directly via your application view scripts, but one you may want to influence. As examples: you may want your title to contain certain content on every page, but specific content based on the controller and/or action; you may want to specify CSS files to load based on what section of the application you're in; you may need specific JavaScript scripts loaded at different times; or you may want to set the DocType declaration.

Zend Framework ships with placeholder implementations for each of these situations, and several more.

3.1. Setting the DocType

DocType declarations are troublesome to memorize, and often essential to include in your document to ensure the browser properly renders your content. The `doctype()` view helper allows you to use simple string mnemonics to specify the desired DocType; additionally, other helpers will query the `doctype()` helper to ensure the output generated conforms with the requested DocType.

As an example, if you want to use the XHTML1 Strict DTD, you can simply specify:

```
$this->doctype('XHTML1_STRICT');
```

Among the other available mnemonics, you'll find these common types:

XHTML1_STRICT	XHTML 1.0 Strict
XHTML1_TRANSITIONAL	XHTML 1.0 Transitional
HTML4_STRICT	HTML 4.01 Strict

HTML4_Loose	HTML 4.01 Loose
HTML5	HTML 5

You can assign the type and render the declaration in a single call:

```
echo $this->doctype('XHTML1_STRICT');
```

However, the better approach is to assign the type in your bootstrap, and then render it in your layout. Try adding the following to your bootstrap class:

```
class Bootstrap extends Zend_Application_Bootstrap_Bootstrap
{
    protected function _initDocType()
    {
        $this->bootstrap('View');
        $view = $this->getResource('View');
        $view->doctype('XHTML1_STRICT');
    }
}
```

Then, in your layout script, simply `echo` the helper at the top of the file:

```
<?php echo $this->doctype() ?>
<html>
    <!-- ... -->
```

This will ensure that your DocType-aware view helpers render the appropriate markup, ensure that the type is set well before the layout is rendered, and provide a single location to change the DocType.

3.2. Specifying the Page Title

Often, a site will include the site or business name as part of the page title, and then add additional information based on the location within the site. As an example, the zend.com website includes the string "Zend.com" on all pages, and the prepends information based on the page: "Zend Server - Zend.com". Within Zend Framework, the `headTitle()` view helper can help simplify this task.

At its simplest, the `headTitle()` helper allows you to aggregate content for the `<title>` tag; when you echo it, it then assembles it based on the order in which segments are added. You can control the order using `prepend()` and `append()`, and provide a separator to use between segments using the `setSeparator()` method.

Typically, you should specify any segments common to all pages in your bootstrap, similar to how we define the doctype. In this case, we'll define a `_initPlaceholders()` method for operating on all the various placeholders, and specify an initial title as well as a separator.

```
class Bootstrap extends Zend_Application_Bootstrap_Bootstrap
{
    // ...

    protected function _initPlaceholders()
    {
        $this->bootstrap('View');
        $view = $this->getResource('View');
```

```
        $view->doctype('XHTML1_STRICT');

        // Set the initial title and separator:
        $view->headTitle('My Site')
             ->setSeparator(' :: ');
    }

    // ...
}
```

Within a view script, we might want to add another segment:

```
<?php $this->headTitle()->append('Some Page'); // place after other segments ?>
<?php $this->headTitle()->prepend('Some Page'); // place before ?>
```

In our layout, we will simply echo the `headTitle()` helper:

```
<?php echo $this->doctype() ?>
<html>
    <?php echo $this->headTitle() ?>
    <!-- ... -->
```

This will generate the following output:

```
<!-- If append() was used: -->
<title>My Site :: Some Page</title>

<!-- If prepend() was used: -->
<title>Some Page :: My Site</title>
```

3.3. Specifying Stylesheets with HeadLink

Good CSS developers will often create a general stylesheet for sitewide styles, and individual stylesheets for specific sections or pages of the website, and load these latter conditionally so as to decrease the amount of data needing to be transferred on each request. The `headLink()` placeholder makes such conditional aggregation of stylesheets trivial within your application.

To accomplish this, `headLink()` defines a number of "virtual" methods (via overloading) to make the process trivial. The ones we will be concerned with are `appendStylesheet()` and `prependStylesheet()`. Each takes up to four arguments, `$href` (the relative path to the stylesheet), `$media` (the MIME type, which defaults to "text/css"), `$conditionalStylesheet` (which can be used to specify a "condition" under which the stylesheet will be evaluated), and `$extras` (an associative array of key and value pairs, commonly used to specify a key for "media"). In most cases, you will only need to specify the first argument, the relative path to the stylesheet.

In our example, we'll assume that all pages need to load the stylesheet located in "`/styles/site.css`" (relative to the document root); we'll specify this in our `_initPlaceholders()` bootstrap method.

```
class Bootstrap extends Zend_Application_Bootstrap_Bootstrap
{
    // ...

    protected function _initPlaceholders()
    {
```

```
        $this->bootstrap('View');
        $view = $this->getResource('View');
        $view->doctype('XHTML1_STRICT');

        // Set the initial title and separator:
        $view->headTitle('My Site')
             ->setSeparator(' :: ');

        // Set the initial stylesheet:
        $view->headLink()->prependStylesheet('/styles/site.css');
    }

    // ...
}
```

Later, in a controller or action-specific view script, we can add more stylesheets:

```
<?php $this->headLink()->appendStylesheet('/styles/user-list.css') ?>
```

Within our layout view script, once again, we simply echo the placeholder:

```
<?php echo $this->doctype() ?>
<html>
    <?php echo $this->headTitle() ?>
    <?php echo $this->headLink() ?>
    <!-- ... -->
```

This will generate the following output:

```
<link rel="stylesheet" type="text/css" href="/styles/site.css" />
<link rel="stylesheet" type="text/css" href="/styles/user-list.css" />
```

3.4. Aggregating Scripts Using HeadScript

Another common tactic to prevent long page load times is to only load JavaScript when necessary. That said, you may need several layers of scripts: perhaps one for progressively enhancing menus on the site, and another for page-specific content. In these situations, the `headScript()` helper presents a solution.

Similar to the `headLink()` helper, `headScript()` provides the ability to append or prepend scripts to the collection, and then echo the entire set. It provides the flexibility to specify either script files themselves to load, or explicit JavaScript. You also have the option of capturing JavaScript via `captureStart()`/`captureEnd()`, which allows you to simply inline the JavaScript instead of requiring an additional call to your server.

Also like `headLink()`, `headScript()` provides "virtual" methods via overloading as a convenience when specifying items to aggregate; common methods include `prependFile()`, `appendFile()`, `prependScript()`, and `appendScript()`. The first two allow you to specify files that will be referenced in a `<script>` tag's `$src` attribute; the latter two will take the content provided and render it as literal JavaScript within a `<script>` tag.

In this example, we'll specify that a script, "`/js/site.js`" needs to be loaded on every page; we'll update our `_initPlaceholders()` bootstrap method to do this.

```
class Bootstrap extends Zend_Application_Bootstrap_Bootstrap
{
```

```
    // ...

    protected function _initPlaceholders()
    {
        $this->bootstrap('View');
        $view = $this->getResource('View');
        $view->doctype('XHTML1_STRICT');

        // Set the initial title and separator:
        $view->headTitle('My Site')
             ->setSeparator(' :: ');

        // Set the initial stylesheet:
        $view->headLink()->prependStylesheet('/styles/site.css');

        // Set the initial JS to load:
        $view->headScript()->prependFile('/js/site.js');
    }

    // ...
}
```

Within a view script, we might then add an extra script file to source, or capture some JavaScript to include in our document.

```
<?php $this->headScript()->appendFile('/js/user-list.js') ?>
<?php $this->headScript()->captureStart() ?>
site = {
    baseUrl: "<?php echo $this->baseUrl() ?>"
};
<?php $this->headScript()->captureEnd() ?>
```

Within our layout script, we then simply echo the placeholder, just as we have all the others:

```
<?php echo $this->doctype() ?>
<html>
    <?php echo $this->headTitle() ?>
    <?php echo $this->headLink() ?>
    <?php echo $this->headScript() ?>
    <!-- ... -->
```

This will generate the following output:

```
<script type="text/javascript" src="/js/site.js"></script>
<script type="text/javascript" src="/js/user-list.js"></script>
<script type="text/javascript">
site = {
    baseUrl: "<?php echo $this->baseUrl() ?>"
};
</script>
```

InlineScript Variant

Many browsers will often block display of a page until all scripts and stylesheets referenced in the <head> section have loaded. If you have a number of such directives, this can impact how soon somebody can start actually viewing the page.

One way around this is to emit your `<script>` tags just prior to closing the `<body>` of your document. (This is a practice specifically recommend by the Y! Slow project.)

Zend Framework supports this in two different ways:

- You can render your `headScript()` tag whereever you like in your layout script; just because the title references "head" does not mean it needs to be rendered in that location.
- Alternately, you may use the `inlineScript()` helper, which is simply a variant on `headScript()`, and retains the same behavior, but uses a separate registry.

4. View Placeholders: Conclusion

View placeholders are a simple and powerful method for creating rich layouts for your application. You can use a variety of standard placeholders, such as those discussed (`doctype()`, `headTitle()`, `headLink()`, and `headScript()`), or use the generic `placeholder()` helper to aggregate content and render it in custom ways. Experiment with their exposed functionality, and visit the appropriate sections in the reference guide to find out about the additional features they offer -- and how you may leverage those features to create rich content for your readers.

Understanding and Using Zend Form Decorators

1. Introduction

Zend_Form utilizes the *decorator* pattern in order to render elements and forms. Unlike the classic decorator pattern, in which you pass an object to a wrapping class, decorators in `Zend_Form` implement a strategy pattern, and utilize the metadata contained in an element or form in order to create a representation of it

Don't let the terminology scare you away, however; at heart, decorators in `Zend_Form` are not terribly difficult, and the mini-tutorials that follow should help you along the way. They will guide you through the basics of decoration, all the way to creating decorators for composite elements.

2. Decorator Basics

2.1. Overview of the Decorator Pattern

To begin, we'll cover some background on the Decorator design pattern. One common technique is to define a common interface that both your originating object and decorator will implement; your decorator than accepts the originating object as a dependency, and will either proxy to it or override its methods. Let's put that into code to make it more easily understood:

```
interface Window
{
    public function isOpen();
    public function open();
    public function close();
}

class StandardWindow implements Window
{
    protected $_open = false;

    public function isOpen()
    {
        return $this->_open;
    }

    public function open()
    {
        if (!$this->_open) {
            $this->_open = true;
        }
    }

    public function close()
    {
        if ($this->_open) {
            $this->_open = false;
        }
    }
}
```

```
class LockedWindow implements Window
{
    protected $_window;

    public function __construct(Window $window)
    {
        $this->_window = $window;
        $this->_window->close();
    }

    public function isOpen()
    {
        return false;
    }

    public function open()
    {
        throw new Exception('Cannot open locked windows');
    }

    public function close()
    {
        $this->_window->close();
    }
}
```

You then create an object of type `StandardWindow`, pass it to the constructor of `LockedWindow`, and your window instance now has different behavior. The beauty is that you don't have to implement any sort of "locking" functionality on your standard window class -- the decorator takes care of that for you. In the meantime, you can pass your locked window around as if it were just another window.

One particular place where the decorator pattern is useful is for creating textual representations of objects. As an example, you might have a "Person" object that, by itself, has no textual representation. By using the Decorator pattern, you can create an object that will act as if it were a Person, but also provide the ability to render that Person textually.

In this particular example, we're going to use duck typing instead of an explicit interface. This allows our implementation to be a bit more flexible, while still allowing the decorator object to act exactly as if it were a Person object.

```
class Person
{
    public function setFirstName($name) {}
    public function getFirstName() {}
    public function setLastName($name) {}
    public function getLastName() {}
    public function setTitle($title) {}
    public function getTitle() {}
}

class TextPerson
{
    protected $_person;

    public function __construct(Person $person)
    {
```

```
        $this->_person = $person;
    }

    public function __call($method, $args)
    {
        if (!method_exists($this->_person, $method)) {
            throw new Exception('Invalid method called on HtmlPerson: '
                .  $method);
        }
        return call_user_func_array(array($this->_person, $method), $args);
    }

    public function __toString()
    {
        return $this->_person->getTitle() . ' '
            . $this->_person->getFirstName() . ' '
            . $this->_person->getLastName();
    }
}
```

In this example, you pass your `Person` instance to the `TextPerson` constructor. By using method overloading, you are able to continue to call all the methods of `Person` -- to set the first name, last name, or title -- but you also now gain a string representation via the `__toString()` method.

This latter example is getting close to how `Zend_Form` decorators work. The key difference is that instead of a decorator wrapping the element, the element has one or more decorators attached to it that it then injects itself into in order to render. The decorator then can access the element's methods and properties in order to create a representation of the element -- or a subset of it.

2.2. Creating Your First Decorator

`Zend_Form` decorators all implement a common interface, `Zend_Form_Decorator_Interface`. That interface provides the ability to set decorator-specific options, register and retrieve the element, and render. A base decorator, `Zend_Form_Decorator_Abstract`, provides most of the functionality you will ever need, with the exception of the rendering logic.

Let's consider a situation where we simply want to render an element as a standard form text input with a label. We won't worry about error handling or whether or not the element should be wrapped within other tags for now -- just the basics. Such a decorator might look like this:

```
class My_Decorator_SimpleInput extends Zend_Form_Decorator_Abstract
{
    protected $_format = '<label for="%s">%s</label>'
                       . '<input id="%s" name="%s" type="text" value="%s"/>';

    public function render($content)
    {
        $element = $this->getElement();
        $name    = htmlentities($element->getFullyQualifiedName());
        $label   = htmlentities($element->getLabel());
        $id      = htmlentities($element->getId());
        $value   = htmlentities($element->getValue());

        $markup  = sprintf($this->_format, $name, $label, $id, $name, $value);
        return $markup;
    }
```

```
}
```

Let's create an element that uses this decorator:

```
$decorator = new My_Decorator_SimpleInput();
$element   = new Zend_Form_Element('foo', array(
    'label'      => 'Foo',
    'belongsTo'  => 'bar',
    'value'      => 'test',
    'decorators' => array($decorator),
));
```

Rendering this element results in the following markup:

```
<label for="bar[foo]">Foo</label>
<input id="bar-foo" name="bar[foo]" type="text" value="test"/>
```

You could also put this class in your library somewhere, inform your element of that path, and refer to the decorator as simply "SimpleInput" as well:

```
$element = new Zend_Form_Element('foo', array(
    'label'      => 'Foo',
    'belongsTo'  => 'bar',
    'value'      => 'test',
    'prefixPath' => array('decorator' => array(
        'My_Decorator' => 'path/to/decorators/',
    )),
    'decorators' => array('SimpleInput'),
));
```

This gives you the benefit of re-use in other projects, and also opens the door for providing alternate implementations of that decorator later.

In the next section, we'll look at how to combine decorators in order to create composite output.

3. Layering Decorators

If you were following closely in the previous section, you may have noticed that a decorator's `render()` method takes a single argument, `$content`. This is expected to be a string. `render()` will then take this string and decide to either replace it, append to it, or prepend it. This allows you to have a chain of decorators -- which allows you to create decorators that render only a subset of the element's metadata, and then layer these decorators to build the full markup for the element.

Let's look at how this works in practice.

For most form element types, the following decorators are used:

- `ViewHelper` (render the form input using one of the standard form view helpers).
- `Errors` (render validation errors via an unordered list).
- `Description` (render any description attached to the element; often used for tooltips).
- `HtmlTag` (wrap all of the above in a <*dd*> tag.
- `Label` (render the label preceding the above, wrapped in a <*dt*> tag.

You'll notice that each of these decorators does just one thing, and operates on one specific piece of metadata stored in the form element: the `Errors` decorator pulls validation errors and renders them; the `Label` decorator pulls just the label and renders it. This allows the individual decorators to be very succinct, repeatable, and, more importantly, testable.

It's also where that `$content` argument comes into play: each decorator's `render()` method is designed to accept content, and then either replace it (usually by wrapping it), prepend to it, or append to it.

So, it's best to think of the process of decoration as one of building an onion from the inside out.

To simplify the process, we'll take a look at the example from the previous section. Recall:

```
class My_Decorator_SimpleInput extends Zend_Form_Decorator_Abstract
{
    protected $_format = '<label for="%s">%s</label>'
                       . '<input id="%s" name="%s" type="text" value="%s"/>';

    public function render($content)
    {
        $element = $this->getElement();
        $name    = htmlentities($element->getFullyQualifiedName());
        $label   = htmlentities($element->getLabel());
        $id      = htmlentities($element->getId());
        $value   = htmlentities($element->getValue());

        $markup  = sprintf($this->_format, $id, $label, $id, $name, $value);
        return $markup;
    }
}
```

Let's now remove the label functionality, and build a separate decorator for that.

```
class My_Decorator_SimpleInput extends Zend_Form_Decorator_Abstract
{
    protected $_format = '<input id="%s" name="%s" type="text" value="%s"/>';

    public function render($content)
    {
        $element = $this->getElement();
        $name    = htmlentities($element->getFullyQualifiedName());
        $id      = htmlentities($element->getId());
        $value   = htmlentities($element->getValue());

        $markup  = sprintf($this->_format, $id, $name, $value);
        return $markup;
    }
}

class My_Decorator_SimpleLabel extends Zend_Form_Decorator_Abstract
{
    protected $_format = '<label for="%s">%s</label>';

    public function render($content)
    {
        $element = $this->getElement();
        $id      = htmlentities($element->getId());
        $label   = htmlentities($element->getLabel());
```

```
        $markup = sprintf($this->_format, $id, $label);
        return $markup;
    }
}
```

Now, this may look all well and good, but here's the problem: as written currently, the last decorator to run wins, and overwrites everything. You'll end up with just the input, or just the label, depending on which you register last.

To overcome this, simply concatenate the passed in `$content` with the markup somehow:

```
return $content . $markup;
```

The problem with the above approach comes when you want to programmatically choose whether the original content should precede or append the new markup. Fortunately, there's a standard mechanism for this already; `Zend_Form_Decorator_Abstract` has a concept of placement and defines some constants for matching it. Additionally, it allows specifying a separator to place between the two. Let's make use of those:

```
class My_Decorator_SimpleInput extends Zend_Form_Decorator_Abstract
{
    protected $_format = '<input id="%s" name="%s" type="text" value="%s"/>';

    public function render($content)
    {
        $element = $this->getElement();
        $name    = htmlentities($element->getFullyQualifiedName());
        $id      = htmlentities($element->getId());
        $value   = htmlentities($element->getValue());

        $markup  = sprintf($this->_format, $id, $name, $value);

        $placement = $this->getPlacement();
        $separator = $this->getSeparator();
        switch ($placement) {
            case self::PREPEND:
                return $markup . $separator . $content;
            case self::APPEND:
            default:
                return $content . $separator . $markup;
        }
    }
}

class My_Decorator_SimpleLabel extends Zend_Form_Decorator_Abstract
{
    protected $_format = '<label for="%s">%s</label>';

    public function render($content)
    {
        $element = $this->getElement();
        $id      = htmlentities($element->getId());
        $label   = htmlentities($element->getLabel());

        $markup = sprint($this->_format, $id, $label);

        $placement = $this->getPlacement();
        $separator = $this->getSeparator();
```

```
        switch ($placement) {
            case self::APPEND:
                return $markup . $separator . $content;
            case self::PREPEND:
            default:
                return $content . $separator . $markup;
        }
    }
}
```

Notice in the above that I'm switching the default case for each; the assumption will be that labels prepend content, and input appends.

Now, let's create a form element that uses these:

```
$element = new Zend_Form_Element('foo', array(
    'label'      => 'Foo',
    'belongsTo'  => 'bar',
    'value'      => 'test',
    'prefixPath' => array('decorator' => array(
        'My_Decorator' => 'path/to/decorators/',
    )),
    'decorators' => array(
        'SimpleInput',
        'SimpleLabel',
    ),
));
```

How will this work? When we call `render()`, the element will iterate through the various attached decorators, calling `render()` on each. It will pass an empty string to the very first, and then whatever content is created will be passed to the next, and so on:

- Initial content is an empty string: ''.
- '' is passed to the `SimpleInput` decorator, which then generates a form input that it appends to the empty string: *<input id="bar-foo" name="bar[foo]" type="text" value="test"/>*.
- The input is then passed as content to the `SimpleLabel` decorator, which generates a label and prepends it to the original content; the default separator is a `PHP_EOL` character, giving us this: *<label for="bar-foo">\n<input id="bar-foo" name="bar[foo]" type="text" value="test"/>*.

But wait a second! What if you wanted the label to come after the input for some reason? Remember that "placement" flag? You can pass it as an option to the decorator. The easiest way to do this is to pass an array of options with the decorator during element creation:

```
$element = new Zend_Form_Element('foo', array(
    'label'      => 'Foo',
    'belongsTo'  => 'bar',
    'value'      => 'test',
    'prefixPath' => array('decorator' => array(
        'My_Decorator' => 'path/to/decorators/',
    )),
    'decorators' => array(
        'SimpleInput'
        array('SimpleLabel', array('placement' => 'append')),
    ),
));
```

Notice that when passing options, you must wrap the decorator within an array; this hints to the constructor that options are available. The decorator name is the first element of the array, and options are passed in an array to the second element of the array.

The above results in the markup *<input id="bar-foo" name="bar[foo]" type="text" value="test"/> \n<label for="bar-foo">*.

Using this technique, you can have decorators that target specific metadata of the element or form and create only the markup relevant to that metadata; by using mulitiple decorators, you can then build up the complete element markup. Our onion is the result.

There are pros and cons to this approach. First, the cons:

- More complex to implement. You have to pay careful attention to the decorators you use and what placement you utilize in order to build up the markup in the correct sequence.
- More resource intensive. More decorators means more objects; multiply this by the number of elements you have in a form, and you may end up with some serious resource usage. Caching can help here.

The advantages are compelling, though:

- Reusable decorators. You can create truly re-usable decorators with this technique, as you don't have to worry about the complete markup, but only markup for one or a few pieces of element or form metadata.
- Ultimate flexibility. You can theoretically generate any markup combination you want from a small number of decorators.

While the above examples are the intended usage of decorators within `Zend_Form`, it's often hard to wrap your head around how the decorators interact with one another to build the final markup. For this reason, some flexibility was added in the 1.7 series to make rendering individual decorators possible -- which gives some Rails-like simplicity to rendering forms. We'll look at that in the next section.

4. Rendering Individual Decorators

In the previous section, we looked at how you can combine decorators to create complex output. We noted that while you have a ton of flexibility with this approach, it also adds some complexity and overhead. In this section, we will examine how to render decorators individually in order to create custom markup for forms and/or individual elements.

Once you have registered your decorators, you can later retrieve them by name from the element. Let's review the previous example:

```
$element = new Zend_Form_Element('foo', array(
    'label'      => 'Foo',
    'belongsTo'  => 'bar',
    'value'      => 'test',
    'prefixPath' => array('decorator' => array(
        'My_Decorator' => 'path/to/decorators/',
    )),
    'decorators' => array(
        'SimpleInput'
        array('SimpleLabel', array('placement' => 'append')),
    ),
));
```

If we wanted to pull and render just the `SimpleInput` decorator, we can do so using the `getDecorator()` method:

```
$decorator = $element->getDecorator('SimpleInput');
echo $decorator->render('');
```

This is pretty easy, but it can be made even easier; let's do it in a single line:

```
echo $element->getDecorator('SimpleInput')->render('');
```

Not too bad, but still a little complex. To make this easier, a shorthand notation was introduced into `Zend_Form` in 1.7: you can render any registered decorator by calling a method of the format `renderDecoratorName()`. This will effectively perform what you see above, but makes the `$content` argument optional and simplifies the usage:

```
echo $element->renderSimpleInput();
```

This is a neat trick, but how and why would you use it?

Many developers and designers have very precise markup needs for their forms. They would rather have full control over the output than rely on a more automated solution which may or may not conform to their design. In other cases, the form layout may require a lot of specialized markup -- grouping arbitrary elements, making some invisible unless a particular link is selected, etc.

Let's utilize the ability to render individual decorators to create some specialized markup.

First, let's define a form. Our form will capture a user's demographic details. The markup will be highly customized, and in some cases use view helpers directly instead of form elements in order to achieve its goals. Here is the basic form definition:

```
class My_Form_UserDemographics extends Zend_Form
{
    public function init()
    {
        // Add a path for my own decorators
        $this->addElementPrefixPaths(array(
            'decorator' => array('My_Decorator' => 'My/Decorator'),
        ));

        $this->addElement('text', 'firstName', array(
            'label' => 'First name: ',
        ));
        $this->addElement('text', 'lastName', array(
            'label' => 'Last name: ',
        ));
        $this->addElement('text', 'title', array(
            'label' => 'Title: ',
        ));
        $this->addElement('text', 'dateOfBirth', array(
            'label' => 'Date of Birth (DD/MM/YYYY): ',
        ));
        $this->addElement('text', 'email', array(
            'label' => 'Your email address: ',
        ));
        $this->addElement('password', 'password', array(
            'label' => 'Password: ',
        ));
        $this->addElement('password', 'passwordConfirmation', array(
```

```
            'label' => 'Confirm Password: ',
        ));
    }
}
```

We're not defining any validators or filters at this time, as they are not relevant to the discussion of decoration. In a real-world scenario, you should define them.

With that out of the way, let's consider how we might want to display this form. One common idiom with first/last names is to display them on a single line; when a title is provided, that is often on the same line as well. Dates, when not using a JavaScript date chooser, will often be separated into three fields displayed side by side.

Let's use the ability to render an element's decorators one by one to accomplish this. First, let's note that no explicit decorators were defined for the given elements. As a refresher, the default decorators for (most) elements are:

- `ViewHelper`: utilize a view helper to render a form input
- `Errors`: utilize the `FormErrors` view helper to render validation errors
- `Description`: utilize the `FormNote` view helper to render the element description (if any)
- `HtmlTag`: wrap the above three items in a <dd> tag
- `Label`: render the element label using the `FormLabel` view helper (and wrap it in a <dt> tag)

Also, as a refresher, you can access any element of a form as if it were a class property; simply reference the element by the name you assigned it.

Our view script might then look like this:

```
<?php
$form = $this->form;
// Remove <dt> from label generation
foreach ($form->getElements() as $element) {
    $element->getDecorator('label')->setOption('tag', null);
}
?>
<form method="<?php echo $form->getMethod() ?>" action="<?php echo
    $form->getAction()?>">
    <div class="element">
        <?php echo $form->title->renderLabel()
              . $form->title->renderViewHelper() ?>
        <?php echo $form->firstName->renderLabel()
              . $form->firstName->renderViewHelper() ?>
        <?php echo $form->lastName->renderLabel()
              . $form->lastName->renderViewHelper() ?>
    </div>
    <div class="element">
        <?php echo $form->dateOfBirth->renderLabel() ?>
        <?php echo $this->formText('dateOfBirth[day]', '', array(
            'size' => 2, 'maxlength' => 2)) ?>
        /
        <?php echo $this->formText('dateOfBirth[month]', '', array(
            'size' => 2, 'maxlength' => 2)) ?>
        /
        <?php echo $this->formText('dateOfBirth[year]', '', array(
```

```
            'size' => 4, 'maxlength' => 4)) ?>
    </div>
    <div class="element">
        <?php echo $form->password->renderLabel()
              . $form->password->renderViewHelper() ?>
    </div>
    <div class="element">
        <?php echo $form->passwordConfirmation->renderLabel()
              . $form->passwordConfirmation->renderViewHelper() ?>
    </div>
    <?php echo $this->formSubmit('submit', 'Save') ?>
</form>
```

If you use the above view script, you'll get approximately the following HTML (approximate, as the HTML below is formatted):

```
<form method="post" action="">
    <div class="element">
        <label for="title" tag="" class="optional">Title:</label>
        <input type="text" name="title" id="title" value=""/>

        <label for="firstName" tag="" class="optional">First name:</label>
        <input type="text" name="firstName" id="firstName" value=""/>

        <label for="lastName" tag="" class="optional">Last name:</label>
        <input type="text" name="lastName" id="lastName" value=""/>
    </div>

    <div class="element">
        <label for="dateOfBirth" tag="" class="optional">Date of Birth
            (DD/MM/YYYY):</label>
        <input type="text" name="dateOfBirth[day]" id="dateOfBirth-day"
            value="" size="2" maxlength="2"/>
        /
        <input type="text" name="dateOfBirth[month]" id="dateOfBirth-month"
            value="" size="2" maxlength="2"/>
        /
        <input type="text" name="dateOfBirth[year]" id="dateOfBirth-year"
            value="" size="4" maxlength="4"/>
    </div>

    <div class="element">
        <label for="password" tag="" class="optional">Password:</label>
        <input type="password" name="password" id="password" value=""/>
    </div>

    <div class="element">
        <label for="passwordConfirmation" tag="" class="" id="submit"
            value="Save"/>
</form>
```

It may not be truly pretty, but with some CSS, it could be made to look exactly how you might want to see it. The main point, however, is that this form was generated using almost entirely custom markup, while still leveraging decorators for the most common markup (and to ensure things like escaping with htmlentities and value injection occur).

By this point in the tutorial, you should be getting fairly comfortable with the markup possibilities using `Zend_Form`'s decorators. In the next section, we'll revisit the date element from above, and demonstrate how to create a custom element and decorator for composite elements.

5. Creating and Rendering Composite Elements

In the last section, we had an example showing a "date of birth element":

```
<div class="element">
    <?php echo $form->dateOfBirth->renderLabel() ?>
    <?php echo $this->formText('dateOfBirth[day]', '', array(
        'size' => 2, 'maxlength' => 2)) ?>
    /
    <?php echo $this->formText('dateOfBirth[month]', '', array(
        'size' => 2, 'maxlength' => 2)) ?>
    /
    <?php echo $this->formText('dateOfBirth[year]', '', array(
        'size' => 4, 'maxlength' => 4)) ?>
</div>
```

How might you represent this element as a `Zend_Form_Element`? How might you write a decorator to render it?

5.1. The Element

The questions about how the element would work include:

- How would you set and retrieve the value?
- How would you validate the value?
- Regardless, how would you then allow for discrete form inputs for the three segments (day, month, year)?

The first two questions center around the form element itself: how would `setValue()` and `getValue()` work? There's actually another question implied by the question about the decorator: how would you retrieve the discrete date segments from the element and/or set them?

The solution is to override the `setValue()` method of your element to provide some custom logic. In this particular case, our element should have three discrete behaviors:

- If an integer timestamp is provided, it should be used to determine and store the day, month, and year.
- If a textual string is provided, it should be cast to a timestamp, and then that value used to determine and store the day, month, and year.
- If an array containing keys for date, month, and year is provided, those values should be stored.

Internally, the day, month, and year will be stored discretely. When the value of the element is retrieved, it will be done so in a normalized string format. We'll override `getValue()` as well to assemble the discrete date segments into a final string.

Here's what the class would look like:

```
class My_Form_Element_Date extends Zend_Form_Element_Xhtml
{
    protected $_dateFormat = '%year%-%month%-%day%';
    protected $_day;
    protected $_month;
    protected $_year;
```

```
    public function setDay($value)
    {
        $this->_day = (int) $value;
        return $this;
    }

    public function getDay()
    {
        return $this->_day;
    }

    public function setMonth($value)
    {
        $this->_month = (int) $value;
        return $this;
    }

    public function getMonth()
    {
        return $this->_month;
    }

    public function setYear($value)
    {
        $this->_year = (int) $value;
        return $this;
    }

    public function getYear()
    {
        return $this->_year;
    }

    public function setValue($value)
    {
        if (is_int($value)) {
            $this->setDay(date('d', $value))
                 ->setMonth(date('m', $value))
                 ->setYear(date('Y', $value));
        } elseif (is_string($value)) {
            $date = strtotime($value);
            $this->setDay(date('d', $date))
                 ->setMonth(date('m', $date))
                 ->setYear(date('Y', $date));
        } elseif (is_array($value)
                  && (isset($value['day'])
                      && isset($value['month'])
                      && isset($value['year'])
                  )
        ) {
            $this->setDay($value['day'])
                 ->setMonth($value['month'])
                 ->setYear($value['year']);
        } else {
            throw new Exception('Invalid date value provided');
        }

        return $this;
    }
```

```
    public function getValue()
    {
        return str_replace(
            array('%year%', '%month%', '%day%'),
            array($this->getYear(), $this->getMonth(), $this->getDay()),
            $this->_dateFormat
        );
    }
}
```

This class gives some nice flexibility -- we can set default values from our database, and be certain that the value will be stored and represented correctly. Additionally, we can allow for the value to be set from an array passed via form input. Finally, we have discrete accessors for each date segment, which we can now use in a decorator to create a composite element.

5.2. The Decorator

Revisiting the example from the last section, let's assume that we want users to input each of the year, month, and day separately. PHP fortunately allows us to use array notation when creating elements, so it's still possible to capture these three entities into a single value -- and we've now created a `Zend_Form` element that can handle such an array value.

The decorator is relatively simple: it will grab the day, month, and year from the element, and pass each to a discrete view helper to render individual form inputs; these will then be aggregated to form the final markup.

```
class My_Form_Decorator_Date extends Zend_Form_Decorator_Abstract
{
    public function render($content)
    {
        $element = $this->getElement();
        if (!$element instanceof My_Form_Element_Date) {
            // only want to render Date elements
            return $content;
        }

        $view = $element->getView();
        if (!$view instanceof Zend_View_Interface) {
            // using view helpers, so do nothing if no view present
            return $content;
        }

        $day   = $element->getDay();
        $month = $element->getMonth();
        $year  = $element->getYear();
        $name  = $element->getFullyQualifiedName();

        $params = array(
            'size'      => 2,
            'maxlength' => 2,
        );
        $yearParams = array(
            'size'      => 4,
            'maxlength' => 4,
        );

        $markup = $view->formText($name . '[day]', $day, $params)
```

```
                . ' / ' . $view->formText($name . '[month]', $month, $params)
                . ' / ' . $view->formText($name . '[year]', $year, $yearParams);

        switch ($this->getPlacement()) {
            case self::PREPEND:
                return $markup . $this->getSeparator() . $content;
            case self::APPEND:
            default:
                return $content . $this->getSeparator() . $markup;
        }
    }
}
```

We now have to do a minor tweak to our form element, and tell it that we want to use the above decorator as a default. That takes two steps. First, we need to inform the element of the decorator path. We can do that in the constructor:

```
class My_Form_Element_Date extends Zend_Form_Element_Xhtml
{
    // ...

    public function __construct($spec, $options = null)
    {
        $this->addPrefixPath(
            'My_Form_Decorator',
            'My/Form/Decorator',
            'decorator'
        );
        parent::__construct($spec, $options);
    }

    // ...
}
```

Note that this is being done in the constructor and not in `init()`. This is for two reasons. First, it allows extending the element later to add logic in `init` without needing to worry about calling `parent::init()`. Second, it allows passing additional plugin paths via configuration or within an `init` method that will then allow overriding the default `Date` decorator with my own replacement.

Next, we need to override the `loadDefaultDecorators()` method to use our new `Date` decorator:

```
class My_Form_Element_Date extends Zend_Form_Element_Xhtml
{
    // ...

    public function loadDefaultDecorators()
    {
        if ($this->loadDefaultDecoratorsIsDisabled()) {
            return;
        }

        $decorators = $this->getDecorators();
        if (empty($decorators)) {
            $this->addDecorator('Date')
                 ->addDecorator('Errors')
                 ->addDecorator('Description', array(
```

```
                    'tag'   => 'p',
                    'class' => 'description'
                ))
                ->addDecorator('HtmlTag', array(
                    'tag' => 'dd',
                    'id'  => $this->getName() . '-element'
                ))
                ->addDecorator('Label', array('tag' => 'dt'));
        }
    }

    // ...
}
```

What does the final output look like? Let's consider the following element:

```
$d = new My_Form_Element_Date('dateOfBirth');
$d->setLabel('Date of Birth: ')
  ->setView(new Zend_View());

// These are equivalent:
$d->setValue('20 April 2009');
$d->setValue(array('year' => '2009', 'month' => '04', 'day' => '20'));
```

If you then echo this element, you get the following markup (with some slight whitespace modifications for readability):

```
<dt id="dateOfBirth-label"><label for="dateOfBirth" class="optional">
    Date of Birth:
</label></dt>
<dd id="dateOfBirth-element">
    <input type="text" name="dateOfBirth[day]" id="dateOfBirth-day"
        value="20" size="2" maxlength="2"> /
    <input type="text" name="dateOfBirth[month]" id="dateOfBirth-month"
        value="4" size="2" maxlength="2"> /
    <input type="text" name="dateOfBirth[year]" id="dateOfBirth-year"
        value="2009" size="4" maxlength="4">
</dd>
```

5.3. Conclusion

We now have an element that can render multiple related form input fields, and then handle the aggregated fields as a single entity -- the `dateOfBirth` element will be passed as an array to the element, and the element will then, as we noted earlier, create the appropriate date segments and return a value we can use for most backends.

Additionally, we can use different decorators with the element. If we wanted to use a Dojo `DateTextBox` dijit decorator -- which accepts and returns string values -- we can, with no modifications to the element itself.

In the end, you get a uniform element API you can use to describe an element representing a composite value.

6. Conclusion

Form decorators are a system that takes some time to learn. At first, they will likely feel cumbersome and overly complex. Hopefully the various topics covered in this chapter will help

you to understand both how they work, as well as strategies for using them effectively in your forms.

Getting Started with Zend_Session, Zend_Auth, and Zend_Acl

1. Building Multi-User Applications With Zend Framework

1.1. Zend Framework

When the original "web" was created, it was designed to be a publishing platform for predominantly static content. As demand for content on the web grew, as did the number of consumers on the internet for web content, the demand for using the web as an application platform also grew. Since the web is inherently good at delivering a simultaneous experience to many consumers from a single location, it makes it an ideal environment for building dynamically driven, multi-user, and more commonly today, social systems.

HTTP is the protocol of the web: a stateless, typically short lived, request and response protocol. This protocol was designed this way because the original intent of the web was to serve or publish static content. It is this very design that has made the web as immensely successful as it is. It is also exactly this design that brings new concerns to developers who wish to use the web as an application platform.

These concerns and responsibilities can effectively be summed up by three questions:

- How do you distinguish one application consumer from another?
- How do you identify a consumer as authentic?
- How do you control what a consumer has access to?

Consumer Vs. User

Notice we use the term "consumer" instead of person. Increasingly, web applications are becoming service driven. This means that not only are real people ("users") with real web browsers consuming and using your application, but also other web applications through machine service technologies such as REST, SOAP, and XML-RPC. In this respect, people, as well as other consuming applications, should all be treated in same with regard to the concerns outlined above.

In the following chapters, we'll take a look at these common problems relating to authentication and authorization in detail. We will discover how 3 main components: `Zend_Session`, `Zend_Auth`, and `Zend_Acl`; provide an out-of-the-box solution as well as the extension points each have that will cater to a more customized solution.

2. Managing User Sessions In ZF

2.1. Introduction to Sessions

The success of the web is deeply rooted in the protocol that drives the web: HTTP. HTTP over TCP is by its very nature stateless, which means that inherently the web is also stateless. While this very aspect is one of the dominating factors for why the web has become such a popular

medium, it also causes an interesting problem for developers that want to use the web as an application platform.

The act of interacting with a web application is typically defined by the sum of all requests sent to a web server. Since there can be many consumers being served simultaneously, the application must decide which requests belong to which consumer. These requests are typically known as a "session".

In PHP, the session problem is solved by the session extension which utilizes some state tracking, typically cookies, and some form of local storage which is exposed via the $_SESSION superglobal. In Zend Framework, the component Zend_Session adds value to the php session extension making it easier to use and depend on inside object-oriented applications.

2.2. Basic Usage of Zend_Session

The Zend_Session component is both a session manager as well as an API for storing data into a session object for long-term persistence. The Zend_Session API is for managing the options and behavior of a session, like options, starting and stopping a session, whereas Zend_Session_Namespace is the actual object used to store data.

While its generally good practice to start a session inside a bootstrap process, this is generally not necessary as all sessions will be automatically started upon the first creation of a Zend_Session_Namespace object.

Zend_Application is capable of configuring Zend_Session for you as part of the Zend_Application_Resource system. To use this, assuming your project uses Zend_Application to bootstrap, you would add the following code to your application.ini file:

```
resources.session.save_path = APPLICATION_PATH "/../data/session"
resources.session.use_only_cookies = true
resources.session.remember_me_seconds = 864000
```

As you can see, the options passed in are the same options that you'd expect to find in the ext/session extension in PHP. Those options setup the path to the session files where data will be stored within the project. Since ini files can additionally use constants, the above will use the APPLICATION_PATH constant and relatively point to a data session directory.

Most Zend Framework components that use sessions need nothing more to use Zend_Session. At this point, you an either use a component that consumes Zend_Session, or start storing your own data inside a session with Zend_Session_Namespace.

Zend_Session_Namespace is a simple class that proxies data via an easy to use API into the Zend_Session managed $_SESSION superglobal. The reason it is called Zend_Session_Namespace is that it effectively namespaces the data inside $_SESSION, thus allowing multiple components and objects to safely store and retrieve data. In the following code, we'll explore how to build a simple session incrementing counter, starting at 1000 and resetting itself after 1999.

```
$mysession = Zend_Session_Namespace('mysession');

if (!isset($mysession->counter)) {
    $mysession->counter = 1000;
} else {
    $mysession->counter++;
}
```

```
if ($mysession->counter > 1999) {
    unset($mysession->counter);
}
```

As you can see above, the session namespace object uses the magic __get, __set, __isset, and __unset to allow you to seemlessly and fluently interact with the session. The information stored in the above example is stored at $_SESSION['mysession']['counter'].

2.3. Advanced Usage of Zend_Session

Addionally, if you wanted to use the DbTable save handler for Zend_Session, you'd add the following code to your application.ini:

```
resources.session.saveHandler.class = "Zend_Session_SaveHandler_DbTable"
resources.session.saveHandler.options.name = "session"
resources.session.saveHandler.options.primary.session_id = "session_id"
resources.session.saveHandler.options.primary.save_path = "save_path"
resources.session.saveHandler.options.primary.name = "name"
resources.session.saveHandler.options.primaryAssignment.sessionId = "sessionId"
resources.session.saveHandler.options.primaryAssignment.sessionSavePath = "sessionSavePath"
resources.session.saveHandler.options.primaryAssignment.sessionName = "sessionName"
resources.session.saveHandler.options.modifiedColumn = "modified"
resources.session.saveHandler.options.dataColumn = "session_data"
resources.session.saveHandler.options.lifetimeColumn = "lifetime"
```

3. Authenticating Users in Zend Framework

3.1. Introduction to Authentication

Once a web application has been able to distinguish one user from another by establishing a session, web applications typically want to validate the identity of a user. The process of validating a consumer as being authentic is "authentication." Authentication is made up of two distinctive parts: an identity and a set of credentials. It takes some variation of both presented to the application for processing so that it may authenticate a user.

While the most common pattern of authentication revolves around usernames and passwords, it should be stated that this is not always the case. Identities are not limited to usernames. In fact, any public identifier can be used: an assigned number, social security number, or residence address. Likewise, credentials are not limited to passwords. Credentials can come in the form of protected private information: fingerprint, eye retinal scan, passphrase, or any other obscure personal information.

3.2. Basic Usage of Zend_Auth

In the following example, we will be using `Zend_Auth` to complete what is probably the most prolific form of authentication: username and password from a database table. This example assumes that you have already setup your application using `Zend_Application`, and that inside that application you have configured a database connection.

The job of the `Zend_Auth` class is twofold. First, it should be able to accept an authentication adapter to use to authenticate a user. Secondly, after a successful authentication of a user, it should persist throughout each and every request that might need to know if the current user has indeed been authenticated. To persist this data, `Zend_Auth` consumes `Zend_Session_Namespace`, but you will generally never need to interact with this session object.

Lets assume we have the following database table setup:

```
CREATE TABLE users (
    id INTEGER  NOT NULL PRIMARY KEY,
    username VARCHAR(50) UNIQUE NOT NULL,
    password VARCHAR(32) NULL,
    password_salt VARCHAR(32) NULL,
    real_name VARCHAR(150) NULL
)
```

The above demonstrates a user table that includes a username, password, and also a password salt column. This salt column is used as part of a technique called salting that would improve the security of your database of information against brute force attacks targeting the algorithm of your password hashing. More information on salting.

For this implementation, we must first make a simple form that we can utilized as the "login form". We will use `Zend_Form` to accomplish this.

```
// located at application/forms/Auth/Login.php

class Default_Form_Auth_Login extends Zend_Form
{
    public function init()
    {
        $this->setMethod('post');

        $this->addElement(
            'text', 'username', array(
                'label' => 'Username:',
                'required' => true,
                'filters'    => array('StringTrim'),
            ));

        $this->addElement('password', 'password', array(
            'label' => 'Password:',
            'required' => true,
            ));

        $this->addElement('submit', 'submit', array(
            'ignore'   => true,
            'label'    => 'Login',
            ));

    }
}
```

With the above form, we can now go about creating our login action for our authentication controller. This controller will be called "`AuthController`", and will be located at `application/controllers/AuthController.php`. It will have a single method called "`loginAction()`" which will serve as the self-posting action. In other words, regardless of the url was POSTed to or GETed to, this method will handle the logic.

The following code will demonstrate how to construct the proper adapter, integrate it with the form:

```
class AuthController extends Zend_Controller_Action
{
```

```
    public function loginAction()
    {
        $db = $this->_getParam('db');

        $loginForm = new Default_Form_Auth_Login($_POST);

        if ($loginForm->isValid()) {

            $adapter = new Zend_Auth_Adapter_DbTable(
                $db,
                'users',
                'username',
                'password',
                'MD5(CONCAT(?, password_salt))'
                );

            $adapter->setIdentity($loginForm->getValue('username'));
            $adapter->setCredential($loginForm->getValue('password'));

            $result = $auth->authenticate($adapter);

            if ($result->isValid()) {
                $this->_helper->FlashMessenger('Successful Login');
                $this->redirect('/');
                return;
            }

        }

        $this->view->loginForm = $loginForm;

    }

}
```

The corresponding view script is quite simple for this action. It will set the current url since this form is self processing, and it will display the form. This view script is located at `application/views/scripts/auth/login.phtml`:

```
$this->form->setAction($this->url());
echo $this->form;
```

There you have it. With these basics you can expand the general concepts to include more complex authentication scenarios. For more information on other `Zend_Auth` adapters, have a look in the reference guide.

4. Building an Authorization System in Zend Framework

4.1. Introduction to Authorization

After a user has been identified as being authentic, an application can go about its business of providing some useful and desirable resources to a consumer. In many cases, applications might contain different resource types, with some resources having stricter rules regarding access. This process of determining who has access to which resources is the process of "authorization". Authorization in its simplest form is the composition of these elements:

- the identity whom wishes to be granted access

- the resource the identity is asking permission to consume
- and optionally, what the identity is privileged to do with the resource

In Zend Framework, the `Zend_Acl` component handles the task of building a tree of roles, resources and privileges to manage and query authorization requests against.

4.2. Basic Usage of Zend_Acl

When using `Zend_Acl`, any models can serve as roles or resources by simply implementing the proper interface. To be used in a role capacity, the class must implement the `Zend_Acl_Role_Interface`, which requires only `getRoleId()`. To be used in a resource capacity, a class must implement the `Zend_Acl_Resource_Interface` which similarly requires the class implement the `getResourceId()` method.

Demonstrated below is a simple user model. This model can take part in our ACL system simply by implementing the `Zend_Acl_Role_Interface`. The method `getRoleId()` will return the id "guest" when an ID is not known, or it will return the role ID that was assigned to this actual user object. This value can effectively come from anywhere, a static definition or perhaps dynamically from the users database role itself.

```
class Default_Model_User implements Zend_Acl_Role_Interface
{
    protected $_aclRoleId = null;

    public function getRoleId()
    {
        if ($this->_aclRoleId == null) {
            return 'guest';
        }

        return $this->_aclRoleId;
    }
}
```

While the concept of a user as a role is pretty straight forward, your application might choose to have any other models in your system as a potential "resource" to be consumed in this ACL system. For simplicity, we'll use the example of a blog post. Since the type of the resource is tied to the type of the object, this class will only return 'blogPost' as the resource ID in this system. Naturally, this value can be dynamic if your system requires it to be so.

```
class Default_Model_BlogPost implements Zend_Acl_Resource_Interface
{
    public function getResourceId()
    {
        return 'blogPost';
    }
}
```

Now that we have at least a role and a resource, we can go about defining the rules of the ACL system. These rules will be consulted when the system receives a query about what is possible given a certain role, resources, and optionally a privilege.

Lets assume the following rules:

```
$acl = new Zend_Acl();
```

```
// setup the various roles in our system
$acl->addRole('guest');
// owner inherits all of the rules of guest
$acl->addRole('owner', 'guest');

// add the resources
$acl->addResource('blogPost');

// add privileges to roles and resource combinations
$acl->allow('guest', 'blogPost', 'view');
$acl->allow('owner', 'blogPost', 'post');
$acl->allow('owner', 'blogPost', 'publish');
```

The above rules are quite simple: a guest role and an owner role exist; as does a blogPost type resource. Guests are allowed to view blog posts, and owners are allowed to post and publish blog posts. To query this system one might do any of the following:

```
// assume the user model is of type guest resource
$guestUser = new Default_Model_User();
$ownerUser = new Default_Model_Owner('OwnersUsername');

$post = new Default_Model_BlogPost();

$acl->isAllowed($guestUser, $post, 'view'); // true
$acl->isAllowed($ownerUser, $post, 'view'); // true
$acl->isAllowed($guestUser, $post, 'post'); // false
$acl->isAllowed($ownerUser, $post, 'post'); // true
```

As you can see, the above rules exercise whether owners and guests can view posts, which they can, or post new posts, which owners can and guests cannot. But as you might expect this type of system might not be as dynamic as we wish it to be. What if we want to ensure a specific owner actual owns a very specific blog post before allowing him to publish it? In other words, we want to ensure that only post owners have the ability to publish their own posts.

This is where assertions come in. Assertions are methods that will be called out to when the static rule checking is simply not enough. When registering an assertion object this object will be consulted to determine, typically dynamically, if some roles has access to some resource, with some optional privlidge that can only be answered by the logic within the assertion. For this example, we'll use the following assertion:

```
class OwnerCanPublishBlogPostAssertion implements Zend_Acl_Assert_Interface
{
    /**
     * This assertion should receive the actual User and BlogPost objects.
     *
     * @param Zend_Acl $acl
     * @param Zend_Acl_Role_Interface $user
     * @param Zend_Acl_Resource_Interface $blogPost
     * @param $privilege
     * @return bool
     */
    public function assert(Zend_Acl $acl,
                           Zend_Acl_Role_Interface $user = null,
                           Zend_Acl_Resource_Interface $blogPost = null,
                           $privilege = null)
    {
        if (!$user instanceof Default_Model_User) {
```

```
            throw new Exception(__CLASS__
                                . '::'
                                . __METHOD__
                                . ' expects the role to be'
                                . ' an instance of User');
        }

        if (!$blogPost instanceof Default_Model_BlogPost) {
            throw new Exception(__CLASS__
                                . '::'
                                . __METHOD__
                                . ' expects the resource to be'
                                . ' an instance of BlogPost');
        }

        // if role is publisher, he can always modify a post
        if ($user->getRoleId() == 'publisher') {
            return true;
        }

        // check to ensure that everyone else is only modifying their own post
        if ($user->id != null && $blogPost->ownerUserId == $user->id) {
            return true;
        } else {
            return false;
        }
    }
}
```

To hook this into our ACL system, we would do the following:

```
// replace this:
//   $acl->allow('owner', 'blogPost', 'publish');
// with this:
$acl->allow('owner',
            'blogPost',
            'publish',
            new OwnerCanPublishBlogPostAssertion());

// lets also add the role of a "publisher" who has access to everything
$acl->allow('publisher', 'blogPost', 'publish');
```

Now, anytime the ACL is consulted about whether or not an owner can publish a specific blog post, this assertion will be run. This assertion will ensure that unless the role type is 'publisher' the owner role must be logically tied to the blog post in question. In this example, we check to see that the ownerUserId property of the blog post matches the id of the owner passed in.

Getting Started with Zend_Search_Lucene

1. Zend_Search_Lucene Introduction

The Zend_Search_Lucene component is intended to provide a ready-for-use full-text search solution. It doesn't require any PHP extensions[1] or additional software to be installed, and can be used immediately after Zend Framework installation.

Zend_Search_Lucene is a pure PHP port of the popular open source full-text search engine known as Apache Lucene. See http://lucene.apache.org/ for the details.

Information must be indexed to be available for searching. Zend_Search_Lucene and Java Lucene use a document concept known as an "atomic indexing item."

Each document is a set of fields: <name, value> pairs where name and value are UTF-8 strings[2]. Any subset of the document fields may be marked as "indexed" to include field data in the text indexing process.

Field values may or may not be tokenized while indexing. If a field is not tokenized, then the field value is stored as one term; otherwise, the current analyzer is used for tokenization.

Several analyzers are provided within the Zend_Search_Lucene package. The default analyzer works with ASCII text (since the UTF-8 analyzer needs the *mbstring* extension to be turned on). It is case insensitive, and it skips numbers. Use other analyzers or create your own analyzer if you need to change this behavior.

Using analyzers during indexing and searching

Important note! Search queries are also tokenized using the "current analyzer", so the same analyzer must be set as the default during both the indexing and searching process. This will guarantee that source and searched text will be transformed into terms in the same way.

Field values are optionally stored within an index. This allows the original field data to be retrieved from the index while searching. This is the only way to associate search results with the original data (internal document IDs may be changed after index optimization or auto-optimization).

The thing that should be remembered is that a Lucene index is not a database. It doesn't provide index backup mechanisms except backup of the file system directory. It doesn't provide transactional mechanisms though concurrent index update as well as concurrent update and read are supported. It doesn't compare with databases in data retrieving speed.

So it's good idea:

- *Not* to use Lucene index as a storage since it may dramatically decrease search hit retrieving performance. Store only unique document identifiers (doc paths, URLs, database unique IDs) and associated data within an index. E.g. title, annotation, category, language info, avatar. (Note: a field may be included in indexing, but not stored, or stored, but not indexed).

[1] Though some UTF-8 processing functionality requires the *mbstring* extension to be turned on
[2] Binary strings are also allowed to be used as field values

- To write functionality that can rebuild an index completely if it's corrupted for any reason.

Individual documents in the index may have completely different sets of fields. The same fields in different documents don't need to have the same attributes. E.g. a field may be indexed for one document and skipped from indexing for another. The same applies for storing, tokenizing, or treating field value as a binary string.

2. Lucene Index Structure

In order to fully utilize `Zend_Search_Lucene`'s capabilities with maximum performance, you need to understand it's internal index structure.

An *index* is stored as a set of files within a single directory.

An *index* consists of any number of independent *segments* which store information about a subset of indexed documents. Each *segment* has its own *terms dictionary*, terms dictionary index, and document storage (stored field values) [3]. All segment data is stored in `_xxxxx.cfs` files, where *xxxxx* is a segment name.

Once an index segment file is created, it can't be updated. New documents are added to new segments. Deleted documents are only marked as deleted in an optional `<segmentname>.del` file.

Document updating is performed as separate delete and add operations, even though it's done using an `update()` API call [4]. This simplifies adding new documents, and allows updating concurrently with search operations.

On the other hand, using several segments (one document per segment as a borderline case) increases search time:

- retrieving a term from a dictionary is performed for each segment;
- the terms dictionary index is pre-loaded for each segment (this process takes the most search time for simple queries, and it also requires additional memory).

If the terms dictionary reaches a saturation point, then search through one segment is *N* times faster than search through *N* segments in most cases.

Index optimization merges two or more segments into a single new one. A new segment is added to the index segments list, and old segments are excluded.

Segment list updates are performed as an atomic operation. This gives the ability of concurrently adding new documents, performing index optimization, and searching through the index.

Index auto-optimization is performed after each new segment generation. It merges sets of the smallest segments into larger segments, and larger segments into even larger segments, if we have enough segments to merge.

Index auto-optimization is controlled by three options:

- *MaxBufferedDocs* (the minimal number of documents required before the buffered in-memory documents are written into a new segment);

[3] Starting with Lucene 2.3, document storage files can be shared between segments; however, `Zend_Search_Lucene` doesn't use this capability

[4] This call is provided only by Java Lucene now, but it's planned to extend the `Zend_Search_Lucene` API with similar functionality

- *MaxMergeDocs* (the largest number of documents ever merged by an optimization operation); and
- *MergeFactor* (which determines how often segment indices are merged by auto-optimization operations).

If we add one document per script execution, then *MaxBufferedDocs* is actually not used (only one new segment with only one document is created at the end of script execution, at which time the auto-optimization process starts).

3. Index Opening and Creation

All index operations (e.g., creating a new index, adding a document to the index, deleting a document, searching through the index) need an index object. One can be obtained using one of the following two methods.

Example 3. Lucene Index Creation

```
$index = Zend_Search_Lucene::create($indexPath);
```

Example 4. Lucene Index Opening

```
$index = Zend_Search_Lucene::open($indexPath);
```

4. Indexing

Indexing is performed by adding a new document to an existing or new index:

```
$index->addDocument($doc);
```

There are two ways to create document object. The first is to do it manually.

Example 5. Manual Document Construction

```
$doc = new Zend_Search_Lucene_Document();
$doc->addField(Zend_Search_Lucene_Field::Text('url', $docUrl));
$doc->addField(Zend_Search_Lucene_Field::Text('title', $docTitle));
$doc->addField(Zend_Search_Lucene_Field::unStored('contents', $docBody));
$doc->addField(Zend_Search_Lucene_Field::binary('avatar', $avatarData));
```

The second method is to load it from HTML or Microsoft Office 2007 files:

Example 6. Document loading

```
$doc = Zend_Search_Lucene_Document_Html::loadHTML($htmlString);
$doc = Zend_Search_Lucene_Document_Docx::loadDocxFile($path);
$doc = Zend_Search_Lucene_Document_Pptx::loadPptFile($path);
$doc = Zend_Search_Lucene_Document_Xlsx::loadXlsxFile($path);
```

If a document is loaded from one of the supported formats, it still can be extended manually with new user defined fields.

4.1. Indexing Policy

You should define indexing policy within your application architectural design.

You may need an on-demand indexing configuration (something like OLTP system). In such systems, you usually add one document per user request. As such, the *MaxBufferedDocs* option will not affect the system. On the other hand, *MaxMergeDocs* is really helpful as it allows you to limit maximum script execution time. *MergeFactor* should be set to a value that keeps balance between the average indexing time (it's also affected by average auto-optimization time) and search performance (index optimization level is dependent on the number of segments).

If you will be primarily performing batch index updates, your configuration should use a *MaxBufferedDocs* option set to the maximum value supported by the available amount of memory. *MaxMergeDocs* and *MergeFactor* have to be set to values reducing auto-optimization involvement as much as possible [5]. Full index optimization should be applied after indexing.

Example 7. Index optimization

```
$index->optimize();
```

In some configurations, it's more effective to serialize index updates by organizing update requests into a queue and processing several update requests in a single script execution. This reduces index opening overhead, and allows utilizing index document buffering.

5. Searching

Searching is performed by using the `find()` method:

Example 8. Searching through the index

```
$hits = $index->find($query);

foreach ($hits as $hit) {
    printf("%d %f %s\n", $hit->id, $hit->score, $hit->title);
}
```

This example demonstrates the usage of two special search hit properties - id and score.

id is an internal document identifier used within a Lucene index. It may be used for a variety of operations, including deleting a document from the index:

Example 9. Deleting an Indexed Document

```
$index->delete($id);
```

Or retrieving the document from the index:

Example 10. Retrieving an Indexed Document

```
$doc = $index->getDocument($id);
```

Internal Document Identifiers

Important note! Internal document identifiers may be changed by index optimization or the auto-optimization process, but it's never changed within a single script's execution unless the `addDocument()` (which may involve an auto-optimization procedure) or `optimize()` methods are called.

[5] An additional limit is the maximum file handlers supported by the operation system for concurrent open operations

The score field is a hit score. Search results are ordered by score by default (best results returned first).

It's also possible to order result sets by specific field values. See the `Zend_Search_Lucene` documentation for more details about this possibility.

The example also demonstrates an ability to access stored fields (e.g., **$hit->title**). At the first access to any hit property other than id or score, document stored fields are loaded, and the corresponding field value is returned.

This causes an ambiguity for documents having their own id or score fields; as a result, it's not recommended to use these field names within stored documents. Nevertheless, they still can be accessed via the `getDocument()` method:

Example 11. Accessing the original document's "id" and "score" fields

```
$id    = $hit->getDocument()->id;
$score = $hit->getDocument()->score;
```

6. Supported queries

`Zend_Search_Lucene` and Java Lucene support a powerful query language. It allows searching for individual terms, phrases, ranges of terms; using wildcards and fuzzy search; combining queries using boolean operators; and so on.

A detailed query language description can be found in the Zend_Search_Lucene component documentation.

What follows are examples of some common query types and strategies.

Example 12. Querying for a single word

```
hello
```

Searches for the word "hello" through all document fields.

Default search field

Important note! Java Lucene searches only through the "contents" field by default, but `Zend_Search_Lucene` searches through *all* fields. This behavior can be modified using the `Zend_Search_Lucene::setDefaultSearchField($fieldName)` method.

Example 13. Querying for multiple words

```
hello dolly
```

Searches for two words. Both words are optional; at least one of them must be present in the result.

Example 14. Requiring words in a query

```
+hello dolly
```

Searches for two words; "hello" is required, "dolly" is optional.

Example 15. Prohibiting words in queried documents

```
+hello -dolly
```

Searches for two words; "hello" is required, 'dolly' is prohibited. In other words, if the document matches "hello", but contains the word "dolly", it will not be returned in the set of matches.

Example 16. Querying for phrases

```
"hello dolly"
```

Searches for the phrase "hello dolly"; a document only matches if that exact string is present.

Example 17. Querying against specific fields

```
title:"The Right Way" AND text:go
```

Searches for the phrase "The Right Way" within the title field and the word "go" within the text field.

Example 18. Querying against specific fields as well as the entire document

```
title:"The Right Way" AND  go
```

Searches for the phrase "The Right Way" within the title field and the word "go" word appearing in any field of the document.

Example 19. Querying against specific fields as well as the entire document (alternate)

```
title:Do it right
```

Searches for the word "Do" within the title field and the words "it" and "right" words through all fields; any single one matching will result in a document match.

Example 20. Querying with the wildcard "?"

```
te?t
```

Search for words matching the pattern "te?t", where "?" is any single character.

Example 21. Querying with the wildcard "*"

```
test*
```

Search for words matching the pattern "test*", where "*" is any sequence of zero or more characters.

Example 22. Querying for an inclusive range of terms

```
mod_date:[20020101 TO 20030101]
```

Search for the range of terms (inclusive).

Example 23. Querying for an exclusive range of terms

```
title:{Aida to Carmen}
```

Search for the range of terms (exclusive).

Example 24. Fuzzy searches

```
roam~
```

Fuzzy search for the word "roam".

Example 25. Boolean searches

```
(framework OR library) AND php
```

Boolean query.

All supported queries can be constructed through `Zend_Search_Lucene`'s query construction API. Moreover, query parsing and query constructing may be combined:

Example 26. Combining parsed and constructed queries

```
$userQuery = Zend_Search_Lucene_Search_QueryParser::parse($queryStr);

$query = new Zend_Search_Lucene_Search_Query_Boolean();
$query->addSubquery($userQuery, true  /* required */);
$query->addSubquery($constructedQuery, true  /* required */);
```

7. Search result pagination

As mentioned above, search result hit objects use lazy loading for stored document fields. When any stored field is accessed, the complete document is loaded.

Do not retrieve all documents if you actually need to work only with some portion of them. Go through the search results and store document IDs (and optionally the score) somewhere to retrive documents from the index during the next script execution.

Example 27. Search result pagination example

```
$cacheId = md5($query);

if (!$resultSet = $cache->load($cacheId)) {
    $hits = $index->find($query);
    $resultSet = array();
    foreach ($hits as $hit) {
        $resultSetEntry          = array();
        $resultSetEntry['id']    = $hit->id;
        $resultSetEntry['score'] = $hit->score;

        $resultSet[] = $resultSetEntry;
    }

    $cache->save($resultSet, $cacheId);
}

$publishedResultSet = array();
for ($resultId = $startId; $resultId < $endId; $resultId++) {
    $publishedResultSet[$resultId] = array(
        'id'    => $resultSet[$resultId]['id'],
        'score' => $resultSet[$resultId]['score'],
        'doc'   => $index->getDocument($resultSet[$resultId]['id']),
    );
}
```

Getting Started with Zend_Paginator

1. Introduction

Let's say you're creating a blogging application that will be home to your vast collection of blog posts. There is a good chance that you do not want all of your blog posts to appear on one single page when someone visits your blog. An obvious solution would be to only display a small number of blog posts on the screen at a time, and allow the user to browse through the different pages, much like your favorite search engine shows you the result of your search query. `Zend_Paginator` is designed to help you achieve the goal of dividing collections of data in smaller, more manageable sets more easily, with more consistency, and with less duplicate code.

`Zend_Paginator` uses Adapters to support various data sources and ScrollingStyles to support various methods of showing the user which pages are available. In later sections of this text we will have a closer look at what these things are and how they can help you to make the most out of `Zend_Paginator`.

Before going in-depth, we will have a look at some simple examples first. After these simple examples, we will see how `Zend_Paginator` supports the most common use-case; paginating database results.

This introduction has given you a quick overview of `Zend_Paginator`. To get started and to have a look at some code snippets, let's have a look at some simple examples.

2. Simple Examples

In this first example we won't do anything spectacular, but hopefully it will give you a good idea of what `Zend_Paginator` is designed to do. Let's say we have an array called $data with the numbers 1 to 100 in it, which we want to divide over a number of pages. We can use the static `factory()` method in the `Zend_Paginator` class to get a `Zend_Paginator` object with our array in it.

```
// Create an array with numbers 1 to 100
$data = range(1, 100);

// Get a Paginator object using Zend_Paginator's built-in factory.
$paginator = Zend_Paginator::factory($data);
```

We're already almost done! The $paginator variable now contains a reference to the Paginator object. By default it is setup to display 10 items per page. To display the items for the currently active page, all you need to do is iterate over the Paginator object with a foreach loop. The currently active page defaults to the first page if it's not explicitly specified. We will see how you can select a specific page later on. The snippet below will display an unordered list containing the numbers 1 to 10, which are the numbers on the first page.

```
// Create an array with numbers 1 to 100
$data = range(1, 100);

// Get a Paginator object using Zend_Paginator's built-in factory.
$paginator = Zend_Paginator::factory($data);

?><ul><?php
```

```
// Render each item for the current page in a list-item
foreach ($paginator as $item) {
    echo '<li>' . $item . '</li>';
}

?></ul>
```

Now let's try and render the items on the second page. You can use the `setCurrentPageNumber()` method to select which page you want to view.

```
// Create an array with numbers 1 to 100
$data = range(1, 100);

// Get a Paginator object using Zend_Paginator's built-in factory.
$paginator = Zend_Paginator::factory($data);

// Select the second page
$paginator->setCurrentPageNumber(2);

?><ul><?php

// Render each item for the current page in a list-item
foreach ($paginator as $item) {
    echo '<li>' . $item . '</li>';
}

?></ul>
```

As expected, this little snippet will render an unordered list with the numbers 11 to 20 in it.

These simple examples demonstrate a small portion of what can be achieved with `Zend_Paginator`. However, a real application rarely reads its data from a plain array, so the next section is dedicated to showing you how you can use Paginator to paginate the results of a database query. Before reading on, make sure you're familiar with the way `Zend_Db_Select` works!

In the database examples we will look at a table with blog posts called 'posts'. The 'posts' table has four columns: id, title, body, date_created. Let's dive right in and have a look at a simple example.

```
// Create a select query. $db is a Zend_Db_Adapter object, which we assume
// already exists in your script.
$select = $db->select()->from('posts')->order('date_created DESC');

// Get a Paginator object using Zend_Paginator's built-in factory.
$paginator = Zend_Paginator::factory($select);

// Select the second page
$paginator->setCurrentPageNumber(2);

?><ul><?php

// Render each the title of each post for the current page in a list-item
foreach ($paginator as $item) {
    echo '<li>' . $item->title . '</li>';
}
```

```
?></ul>
```

As you can see, this example is not that different from the previous one. The only difference is that you pass a `Zend_Db_Select` object to the Paginator's `factory()` method, rather than an array. For more details on how the database adapter makes sure that your query is being executed efficiently, see the `Zend_Paginator` chapter in the reference manual on the DbSelect and DbTableSelect adapters.

3. Pagination Control and ScrollingStyles

Rendering the items for a page on the screen has been a good start. In the code snippets in previous section we have also seen the `setCurrentPageNumber()` method to set the active page number. The next step is to navigate through your pages. To do this, Paginator provides you with two important tools: the ability to render the Paginator with help of a View Partial, and support for so-called ScrollingStyles.

The View Partial is a small view script that renders the Pagination controls, such as buttons to go to the next or previous page. Which pagination controls are rendered depends on the contents of the view partial. Working with the view partial requires that you have set up Zend_View. To get started with the pagination control, create a new view script somewhere in your view scripts path. You can name it anything you want, but we'll call it "controls.phtml" in this text. The reference manual contains various examples of what might go in the view script. Here is one example.

```
<?php if ($this->pageCount): ?>
<!-- First page link -->
<?php if (isset($this->previous)): ?>
  <a href="<?php echo $this->url(array('page' => $this->first)); ?>">
    First
  </a> |
<?php else: ?>
  <span class="disabled">First</span> |
<?php endif; ?>
<!-- Previous page link -->
<?php if (isset($this->previous)): ?>
  <a href="<?php echo $this->url(array('page' => $this->previous)); ?>">
     Previous
  </a> |
<?php else: ?>
  <span class="disabled"> Previous</span> |
<?php endif; ?>
<!-- Next page link -->
<?php if (isset($this->next)): ?>
  <a href="<?php echo $this->url(array('page' => $this->next)); ?>">
    Next
  </a> |
<?php else: ?>
  <span class="disabled">Next </span> |
<?php endif; ?>
<!-- Last page link -->
<?php if (isset($this->next)): ?>
  <a href="<?php echo $this->url(array('page' => $this->last)); ?>">
    Last
  </a>
<?php else: ?>
  <span class="disabled">Last</span>
<?php endif; ?>
</div>
```

```
<?php endif; ?>
```

The next step is to tell Zend_Paginator which view partial can be used to render the navigation controls. Put the following line in your application's bootstrap file.

```
Zend_View_Helper_PaginationControl::setDefaultViewPartial('controls.phtml');
```

The last step is probably the easiest. Make sure you have assigned your Paginator object to the a script (NOT the 'controls.phtml' script!). The only thing left to do is echo the Paginator in the view script. This will automatically render the Paginator using the PaginationControl view helper. In this next example, the Paginator object has been assigned to the 'paginator' view variable. Don't worry if you don't fully get how it all works yet. The next section will feature a complete example.

```
<?php echo $this->paginator; ?>
```

`Zend_Paginator`, together with the 'controls.phtml' view script you wrote, makes sure your Paginator navigation is rendered properly. In order to decide which page numbers need to be displayed on screen, Paginator uses so-called ScrollingStyles. The default style is called "Sliding", which is similar to the way Yahoo's search result navigation works. To mimick Google's ScrollingStyle, use the Elastic style. You can set a default ScrollingStyle with the static `setDefaultScrollingStyle()` method, or you can specify a ScrollingStyle dynamically when rendering the Paginator in your view script. This requires manual invocation of the view helper in your view script.

```
// $this->paginator is a Paginator object
<?php echo $this->paginationControl($this->paginator, 'Elastic', 'controls.phtml'); ?>
```

For a list of all available ScrollingStyles, see the reference manual.

4. Putting it all Together

You have seen how to create a Paginator object, how to render the items on the current page, and how to render a navigation element to browse through your pages. In this section you will see how Paginator fits in with the rest of your MVC application.

In the following examples we will ignore the best practice implementation of using a Service Layer to keep the example simple and easier to understand. Once you get familiar with using Service Layers, it should be easy to see how Paginator can fit in with the best practice approach.

Lets start with the controller. The sample application is simple, and we'll just put everything in the IndexController and the IndexAction. Again, this is for demonstration purposes only. A real application should not use controllers in this manner.

```
class IndexController extends Zend_Controller_Action
{
    public function indexAction()
    {
        // Setup pagination control view script. See the pagation control tutorial page
        // for more information about this view script.
        Zend_View_Helper_PaginationControl::setDefaultViewPartial('controls.phtml');

        // Fetch an already instantiated database connection from the registry
        $db = Zend_Registry::get('db');

        // Create a select object which fetches blog posts, sorted decending by date of cre
```

```
        $select = $db->select()->from('posts')->sort('date_created DESC');

        // Create a Paginator for the blog posts query
        $paginator = Zend_Paginator::factory($select);

        // Read the current page number from the request. Default to 1 if no explicit page
        $paginator->setCurrentPageNumber($this->_getParam('page', 1));

        // Assign the Paginator object to the view
        $this->view->paginator = $paginator;
    }
}
```

The following view script is the index.phtml view script for the IndexController's indexAction. The view script can be kept simple. We're assuming the use of the default ScrollingStyle.

```
<ul>
<?php
// Render each the title of each post for the current page in a list-item
foreach ($this->paginator as $item) {
    echo '<li>' . $item->title . '</li>';
}
?>
</ul>
<?php echo $this->paginator; ?>
```

Now navigate to your project's index and see Paginator in action. What we have discussed in this tutorial is just the tip of the iceberg. The reference manual and API documentation can tell you more about what you can do with Zend_Paginator.

Part III. Zend Framework Reference

Table of Contents

Zend_Acl

1. Introduction

`Zend_Acl` provides a lightweight and flexible access control list (ACL) implementation for privileges management. In general, an application may utilize such ACL's to control access to certain protected objects by other requesting objects.

For the purposes of this documentation:

- a *resource* is an object to which access is controlled.
- a *role* is an object that may request access to a Resource.

Put simply, *roles request access to resources*. For example, if a parking attendant requests access to a car, then the parking attendant is the requesting role, and the car is the resource, since access to the car may not be granted to everyone.

Through the specification and use of an ACL, an application may control how roles are granted access to resources.

1.1. Resources

Creating a resource in `Zend_Acl` is very simple. `Zend_Acl` provides the resource, `Zend_Acl_Resource_Interface`, to facilitate creating resources in an application. A class need only implement this interface, which consists of a single method, `getResourceId()`, for `Zend_Acl` to recognize the object as a resource. Additionally, `Zend_Acl_Resource` is provided by `Zend_Acl` as a basic resource implementation for developers to extend as needed.

`Zend_Acl` provides a tree structure to which multiple resources can be added. Since resources are stored in such a tree structure, they can be organized from the general (toward the tree root) to the specific (toward the tree leaves). Queries on a specific resource will automatically search the resource's hierarchy for rules assigned to ancestor resources, allowing for simple inheritance of rules. For example, if a default rule is to be applied to each building in a city, one would simply assign the rule to the city, instead of assigning the same rule to each building. Some buildings may require exceptions to such a rule, however, and this can be achieved in `Zend_Acl` by assigning such exception rules to each building that requires such an exception. A resource may inherit from only one parent resource, though this parent resource can have its own parent resource, etc.

`Zend_Acl` also supports privileges on resources (e.g., "create", "read", "update", "delete"), so the developer can assign rules that affect all privileges or specific privileges on one or more resources.

1.2. Roles

As with resources, creating a role is also very simple. All roles must implement `Zend_Acl_Role_Interface`. This interface consists of a single method, `getRoleId()`, Additionally, `Zend_Acl_Role` is provided by `Zend_Acl` as a basic role implementation for developers to extend as needed.

In `Zend_Acl`, a role may inherit from one or more roles. This is to support inheritance of rules among roles. For example, a user role, such as "sally", may belong to one or more parent

roles, such as "editor" and "administrator". The developer can assign rules to "editor" and "administrator" separately, and "sally" would inherit such rules from both, without having to assign rules directly to "sally".

Though the ability to inherit from multiple roles is very useful, multiple inheritance also introduces some degree of complexity. The following example illustrates the ambiguity condition and how `Zend_Acl` solves it.

Example 28. Multiple Inheritance among Roles

The following code defines three base roles - "guest", "member", and "admin" - from which other roles may inherit. Then, a role identified by "someUser" is established and inherits from the three other roles. The order in which these roles appear in the `$parents` array is important. When necessary, `Zend_Acl` searches for access rules defined not only for the queried role (herein, "someUser"), but also upon the roles from which the queried role inherits (herein, "guest", "member", and "admin"):

```
$acl = new Zend_Acl();

$acl->addRole(new Zend_Acl_Role('guest'))
    ->addRole(new Zend_Acl_Role('member'))
    ->addRole(new Zend_Acl_Role('admin'));

$parents = array('guest', 'member', 'admin');
$acl->addRole(new Zend_Acl_Role('someUser'), $parents);

$acl->add(new Zend_Acl_Resource('someResource'));

$acl->deny('guest', 'someResource');
$acl->allow('member', 'someResource');

echo $acl->isAllowed('someUser', 'someResource') ? 'allowed' : 'denied';
```

Since there is no rule specifically defined for the "someUser" role and "someResource", `Zend_Acl` must search for rules that may be defined for roles that "someUser" inherits. First, the "admin" role is visited, and there is no access rule defined for it. Next, the "member" role is visited, and `Zend_Acl` finds that there is a rule specifying that "member" is allowed access to "someResource".

If `Zend_Acl` were to continue examining the rules defined for other parent roles, however, it would find that "guest" is denied access to "someResource". This fact introduces an ambiguity because now "someUser" is both denied and allowed access to "someResource", by reason of having inherited conflicting rules from different parent roles.

`Zend_Acl` resolves this ambiguity by completing a query when it finds the first rule that is directly applicable to the query. In this case, since the "member" role is examined before the "guest" role, the example code would print "allowed".

When specifying multiple parents for a role, keep in mind that the last parent listed is the first one searched for rules applicable to an authorization query.

1.3. Creating the Access Control List

An Access Control List (ACL) can represent any set of physical or virtual objects that you wish. For the purposes of demonstration, however, we will create a basic Content Management System

(CMS) ACL that maintains several tiers of groups over a wide variety of areas. To create a new ACL object, we instantiate the ACL with no parameters:

```
$acl = new Zend_Acl();
```

Until a developer specifies an "allow" rule, `Zend_Acl` denies access to every privilege upon every resource by every role.

1.4. Registering Roles

CMS's will nearly always require a hierarchy of permissions to determine the authoring capabilities of its users. There may be a 'Guest' group to allow limited access for demonstrations, a 'Staff' group for the majority of CMS users who perform most of the day-to-day operations, an 'Editor' group for those responsible for publishing, reviewing, archiving and deleting content, and finally an 'Administrator' group whose tasks may include all of those of the other groups as well as maintenance of sensitive information, user management, back-end configuration data, backup and export. This set of permissions can be represented in a role registry, allowing each group to inherit privileges from 'parent' groups, as well as providing distinct privileges for their unique group only. The permissions may be expressed as follows:

Table 1. Access Controls for an Example CMS

Name	Unique Permissions	Inherit Permissions From
Guest	View	N/A
Staff	Edit, Submit, Revise	Guest
Editor	Publish, Archive, Delete	Staff
Administrator	(Granted all access)	N/A

For this example, `Zend_Acl_Role` is used, but any object that implements `Zend_Acl_Role_Interface` is acceptable. These groups can be added to the role registry as follows:

```
$acl = new Zend_Acl();

// Add groups to the Role registry using Zend_Acl_Role
// Guest does not inherit access controls
$roleGuest = new Zend_Acl_Role('guest');
$acl->addRole($roleGuest);

// Staff inherits from guest
$acl->addRole(new Zend_Acl_Role('staff'), $roleGuest);

/*
Alternatively, the above could be written:
$acl->addRole(new Zend_Acl_Role('staff'), 'guest');
*/

// Editor inherits from staff
$acl->addRole(new Zend_Acl_Role('editor'), 'staff');

// Administrator does not inherit access controls
$acl->addRole(new Zend_Acl_Role('administrator'));
```

1.5. Defining Access Controls

Now that the ACL contains the relevant roles, rules can be established that define how resources may be accessed by roles. You may have noticed that we have not defined any particular resources for this example, which is simplified to illustrate that the rules apply to all resources. `Zend_Acl` provides an implementation whereby rules need only be assigned from general to specific, minimizing the number of rules needed, because resources and roles inherit rules that are defined upon their ancestors.

In general, `Zend_Acl` obeys a given rule if and only if a more specific rule does not apply.

Consequently, we can define a reasonably complex set of rules with a minimum amount of code. To apply the base permissions as defined above:

```
$acl = new Zend_Acl();

$roleGuest = new Zend_Acl_Role('guest');
$acl->addRole($roleGuest);
$acl->addRole(new Zend_Acl_Role('staff'), $roleGuest);
$acl->addRole(new Zend_Acl_Role('editor'), 'staff');
$acl->addRole(new Zend_Acl_Role('administrator'));

// Guest may only view content
$acl->allow($roleGuest, null, 'view');

/*
Alternatively, the above could be written:
$acl->allow('guest', null, 'view');
//*/

// Staff inherits view privilege from guest, but also needs additional
// privileges
$acl->allow('staff', null, array('edit', 'submit', 'revise'));

// Editor inherits view, edit, submit, and revise privileges from
// staff, but also needs additional privileges
$acl->allow('editor', null, array('publish', 'archive', 'delete'));

// Administrator inherits nothing, but is allowed all privileges
$acl->allow('administrator');
```

The `NULL` values in the above `allow()` calls are used to indicate that the allow rules apply to all resources.

1.6. Querying an ACL

We now have a flexible ACL that can be used to determine whether requesters have permission to perform functions throughout the web application. Performing queries is quite simple using the `isAllowed()` method:

```
echo $acl->isAllowed('guest', null, 'view') ?
     "allowed" : "denied";
// allowed

echo $acl->isAllowed('staff', null, 'publish') ?
```

```
     "allowed" : "denied";
// denied

echo $acl->isAllowed('staff', null, 'revise') ?
     "allowed" : "denied";
// allowed

echo $acl->isAllowed('editor', null, 'view') ?
     "allowed" : "denied";
// allowed because of inheritance from guest

echo $acl->isAllowed('editor', null, 'update') ?
     "allowed" : "denied";
// denied because no allow rule for 'update'

echo $acl->isAllowed('administrator', null, 'view') ?
     "allowed" : "denied";
// allowed because administrator is allowed all privileges

echo $acl->isAllowed('administrator') ?
     "allowed" : "denied";
// allowed because administrator is allowed all privileges

echo $acl->isAllowed('administrator', null, 'update') ?
     "allowed" : "denied";
// allowed because administrator is allowed all privileges
```

2. Refining Access Controls

2.1. Precise Access Controls

The basic ACL as defined in the previous section shows how various privileges may be allowed upon the entire ACL (all resources). In practice, however, access controls tend to have exceptions and varying degrees of complexity. `Zend_Acl` allows to you accomplish these refinements in a straightforward and flexible manner.

For the example CMS, it has been determined that whilst the 'staff' group covers the needs of the vast majority of users, there is a need for a new 'marketing' group that requires access to the newsletter and latest news in the CMS. The group is fairly self-sufficient and will have the ability to publish and archive both newsletters and the latest news.

In addition, it has also been requested that the 'staff' group be allowed to view news stories but not to revise the latest news. Finally, it should be impossible for anyone (administrators included) to archive any 'announcement' news stories since they only have a lifespan of 1-2 days.

First we revise the role registry to reflect these changes. We have determined that the 'marketing' group has the same basic permissions as 'staff', so we define 'marketing' in such a way that it inherits permissions from 'staff':

```
// The new marketing group inherits permissions from staff
$acl->addRole(new Zend_Acl_Role('marketing'), 'staff');
```

Next, note that the above access controls refer to specific resources (e.g., "newsletter", "latest news", "announcement news"). Now we add these resources:

```
// Create Resources for the rules

```

```
// newsletter
$acl->addResource(new Zend_Acl_Resource('newsletter'));

// news
$acl->addResource(new Zend_Acl_Resource('news'));

// latest news
$acl->addResource(new Zend_Acl_Resource('latest'), 'news');

// announcement news
$acl->addResource(new Zend_Acl_Resource('announcement'), 'news');
```

Then it is simply a matter of defining these more specific rules on the target areas of the ACL:

```
// Marketing must be able to publish and archive newsletters and the
// latest news
$acl->allow('marketing',
            array('newsletter', 'latest'),
            array('publish', 'archive'));

// Staff (and marketing, by inheritance), are denied permission to
// revise the latest news
$acl->deny('staff', 'latest', 'revise');

// Everyone (including administrators) are denied permission to
// archive news announcements
$acl->deny(null, 'announcement', 'archive');
```

We can now query the ACL with respect to the latest changes:

```
echo $acl->isAllowed('staff', 'newsletter', 'publish') ?
     "allowed" : "denied";
// denied

echo $acl->isAllowed('marketing', 'newsletter', 'publish') ?
     "allowed" : "denied";
// allowed

echo $acl->isAllowed('staff', 'latest', 'publish') ?
     "allowed" : "denied";
// denied

echo $acl->isAllowed('marketing', 'latest', 'publish') ?
     "allowed" : "denied";
// allowed

echo $acl->isAllowed('marketing', 'latest', 'archive') ?
     "allowed" : "denied";
// allowed

echo $acl->isAllowed('marketing', 'latest', 'revise') ?
     "allowed" : "denied";
// denied

echo $acl->isAllowed('editor', 'announcement', 'archive') ?
     "allowed" : "denied";
// denied

echo $acl->isAllowed('administrator', 'announcement', 'archive') ?
```

```
     "allowed" : "denied";
// denied
```

2.2. Removing Access Controls

To remove one or more access rules from the ACL, simply use the available `removeAllow()` or `removeDeny()` methods. As with `allow()` and `deny()`, you may provide a `NULL` value to indicate application to all roles, resources, and/or privileges:

```
// Remove the denial of revising latest news to staff (and marketing,
// by inheritance)
$acl->removeDeny('staff', 'latest', 'revise');

echo $acl->isAllowed('marketing', 'latest', 'revise') ?
     "allowed" : "denied";
// allowed

// Remove the allowance of publishing and archiving newsletters to
// marketing
$acl->removeAllow('marketing',
                  'newsletter',
                  array('publish', 'archive'));

echo $acl->isAllowed('marketing', 'newsletter', 'publish') ?
     "allowed" : "denied";
// denied

echo $acl->isAllowed('marketing', 'newsletter', 'archive') ?
     "allowed" : "denied";
// denied
```

Privileges may be modified incrementally as indicated above, but a `NULL` value for the privileges overrides such incremental changes:

```
// Allow marketing all permissions upon the latest news
$acl->allow('marketing', 'latest');

echo $acl->isAllowed('marketing', 'latest', 'publish') ?
     "allowed" : "denied";
// allowed

echo $acl->isAllowed('marketing', 'latest', 'archive') ?
     "allowed" : "denied";
// allowed

echo $acl->isAllowed('marketing', 'latest', 'anything') ?
     "allowed" : "denied";
// allowed
```

3. Advanced Usage

3.1. Storing ACL Data for Persistence

`Zend_Acl` was designed in such a way that it does not require any particular backend technology such as a database or cache server for storage of the ACL data. Its complete PHP implementation enables customized administration tools to be built upon `Zend_Acl` with relative ease and

flexibility. Many situations require some form of interactive maintenance of the ACL, and `Zend_Acl` provides methods for setting up, and querying against, the access controls of an application.

Storage of ACL data is therefore left as a task for the developer, since use cases are expected to vary widely for various situations. Because `Zend_Acl` is serializable, ACL objects may be serialized with PHP's `serialize()` function, and the results may be stored anywhere the developer should desire, such as a file, database, or caching mechanism.

3.2. Writing Conditional ACL Rules with Assertions

Sometimes a rule for allowing or denying a role access to a resource should not be absolute but dependent upon various criteria. For example, suppose that certain access should be allowed, but only between the hours of 8:00am and 5:00pm. Another example would be denying access because a request comes from an IP address that has been flagged as a source of abuse. `Zend_Acl` has built-in support for implementing rules based on whatever conditions the developer needs.

`Zend_Acl` provides support for conditional rules with `Zend_Acl_Assert_Interface`. In order to use the rule assertion interface, a developer writes a class that implements the `assert()` method of the interface:

```
class CleanIPAssertion implements Zend_Acl_Assert_Interface
{
    public function assert(Zend_Acl $acl,
                           Zend_Acl_Role_Interface $role = null,
                           Zend_Acl_Resource_Interface $resource = null,
                           $privilege = null)
    {
        return $this->_isCleanIP($_SERVER['REMOTE_ADDR']);
    }

    protected function _isCleanIP($ip)
    {
        // ...
    }
}
```

Once an assertion class is available, the developer must supply an instance of the assertion class when assigning conditional rules. A rule that is created with an assertion only applies when the assertion method returns TRUE.

```
$acl = new Zend_Acl();
$acl->allow(null, null, null, new CleanIPAssertion());
```

The above code creates a conditional allow rule that allows access to all privileges on everything by everyone, except when the requesting IP is "blacklisted." If a request comes in from an IP that is not considered "clean," then the allow rule does not apply. Since the rule applies to all roles, all resources, and all privileges, an "unclean" IP would result in a denial of access. This is a special case, however, and it should be understood that in all other cases (i.e., where a specific role, resource, or privilege is specified for the rule), a failed assertion results in the rule not applying, and other rules would be used to determine whether access is allowed or denied.

The `assert()` method of an assertion object is passed the ACL, role, resource, and privilege to which the authorization query (i.e., `isAllowed()`) applies, in order to provide a context for the assertion class to determine its conditions where needed.

Zend_Amf

1. Introduction

`Zend_Amf` provides support for Adobe's Action Message Format (AMF), to allow communication between Adobe's Flash Player and PHP. Specifically, it provides a gateway server implementation for handling requests sent from the Flash Player to the server and mapping these requests to object and class methods and arbitrary callbacks.

The AMF3 specification is freely available, and serves as a reference for what types of messages may be sent between the Flash Player and server.

2. Zend_Amf_Server

`Zend_Amf_Server` provides an RPC-style server for handling requests made from the Adobe Flash Player using the AMF protocol. Like all Zend Framework server classes, it follows the SoapServer API, providing an easy to remember interface for creating servers.

Example 29. Basic AMF Server

Let's assume that you have created a class `Foo` with a variety of public methods. You may create an AMF server using the following code:

```
$server = new Zend_Amf_Server();
$server->setClass('Foo');
$response = $server->handle();
echo $response;
```

Alternately, you may choose to attach a simple function as a callback instead:

```
$server = new Zend_Amf_Server();
$server->addFunction('myUberCoolFunction');
$response = $server->handle();
echo $response;
```

You could also mix and match multiple classes and functions. When doing so, we suggest namespacing each to ensure that no method name collisions occur; this can be done by simply passing a second string argument to either `addFunction()` or `setClass()`:

```
$server = new Zend_Amf_Server();
$server->addFunction('myUberCoolFunction', 'my')
       ->setClass('Foo', 'foo')
       ->setClass('Bar', 'bar');
$response = $server->handle();
echo $response;
```

The `Zend_Amf_Server` also allows services to be dynamically loaded based on a supplied directory path. You may add as many directories as you wish to the server. The order that you add the directories to the server will be the order that the LIFO search will be performed on the directories to match the class. Adding directories is completed with the `addDirectory()` method.

```
$server->addDirectory(dirname(__FILE__) .'/../services/');
$server->addDirectory(dirname(__FILE__) .'/../package/');
```

When calling remote services your source name can have underscore ("_") and dot (".") directory delimiters. When an underscore is used PEAR and Zend Framework class naming conventions will be respected. This means that if you call the service com_Foo_Bar the server will look for the file `Bar.php` in the each of the included paths at `com/Foo/Bar.php`. If the dot notation is used for your remote service such as `com.Foo.Bar` each included path will have `com/Foo/Bar.php` append to the end to autoload `Bar.php`

All AMF requests sent to the script will then be handled by the server, and an AMF response will be returned.

All Attached Methods and Functions Need Docblocks

Like all other server components in Zend Framework, you must document your class methods using PHP docblocks. At the minimum, you need to provide annotations for each required argument as well as the return value. As examples:

```
// Function to attach:

/**
```

```
 * @param  string $name
 * @param  string $greeting
 * @return string
 */
function helloWorld($name, $greeting = 'Hello')
{
    return $greeting . ', ' . $name;
}
```

```
// Attached class

class World
{
    /**
     * @param  string $name
     * @param  string $greeting
     * @return string
     */
    public function hello($name, $greeting = 'Hello')
    {
        return $greeting . ', ' . $name;
    }
}
```

Other annotations may be used, but will be ignored.

2.1. Connecting to the Server from Flex

Connecting to your `Zend_Amf_Server` from your Flex project is quite simple; you simply need to point your endpoint URI to your `Zend_Amf_Server` script.

Say, for instance, you have created your server and placed it in the `server.php` file in your application root, and thus the URI is `http://example.com/server.php`. In this case, you would modify your `services-config.xml` file to set the channel endpoint uri attribute to this value.

If you have never created a `service-config.xml` file you can do so by opening your project in your Navigator window. Right click on the project name and select 'properties'. In the Project properties dialog go into 'Flex Build Path' menu, 'Library path' tab and be sure the '`rpc.swc`' file is added to your projects path and Press Ok to close the window.

You will also need to tell the compiler to use the `service-config.xml` to find the RemoteObject endpoint. To do this open your project properties panel again by right clicking on the project folder from your Navigator and selecting properties. From the properties popup select 'Flex Compiler' and add the string: **-services "services-config.xml"**. Press Apply then OK to return to update the option. What you have just done is told the Flex compiler to look to the `services-config.xml` file for runtime variables that will be used by the RemotingObject class.

We now need to tell Flex which services configuration file to use for connecting to our remote methods. For this reason create a new '`services-config.xml`' file into your Flex project src folder. To do this right click on the project folder and select 'new' 'File' which will popup a new window. Select the project folder and then name the file '`services-config.xml`' and press finish.

Flex has created the new `services-config.xml` and has it open. Use the following example text for your `services-config.xml` file. Make sure that you update your endpoint to match that of your testing server. Make sure you save the file.

```
<?xml version="1.0" encoding="UTF-8"?>
<services-config>
    <services>
        <service id="zend-service"
            class="flex.messaging.services.RemotingService"
            messageTypes="flex.messaging.messages.RemotingMessage">
            <destination id="zend">
                <channels>
                    <channel ref="zend-endpoint"/>
                </channels>
                <properties>
                    <source>*</source>
                </properties>
            </destination>
        </service>
    </services>
    <channels>
        <channel-definition id="zend-endpoint"
            class="mx.messaging.channels.AMFChannel">
            <endpoint uri="http://example.com/server.php"
                class="flex.messaging.endpoints.AMFEndpoint"/>
        </channel-definition>
    </channels>
</services-config>
```

There are two key points in the example. First, but last in the listing, we create an AMF channel, and specify the endpoint as the URL to our `Zend_Amf_Server`:

```
<channel-definition id="zend-endpoint"
    <endpoint uri="http://example.com/server.php"
        class="flex.messaging.endpoints.AMFEndpoint"/>
</channel-definition>
```

Notice that we've given this channel an identifier, "zend-endpoint". The example create a service destination that refers to this channel, assigning it an ID as well -- in this case "zend".

Within our Flex MXML files, we need to bind a RemoteObject to the service. In MXML, this might be done as follows:

```
<mx:RemoteObject id="myservice"
    fault="faultHandler(event)"
    showBusyCursor="true"
    destination="zend">
```

Here, we've defined a new remote object identified by "myservice" bound to the service destination "zend" we defined in the `services-config.xml` file. We then call methods on it in our ActionScript by simply calling "myservice.<method>". As an example:

```
myservice.hello("Wade");
```

When namespacing, you would use "myservice.<namespace>.<method>":

```
myservice.world.hello("Wade");
```

For more information on Flex RemoteObject invocation, visit the Adobe Flex 3 Help site.

2.2. Error Handling

By default, all exceptions thrown in your attached classes or functions will be caught and returned as AMF ErrorMessages. However, the content of these ErrorMessage objects will vary based on whether or not the server is in "production" mode (the default state).

When in production mode, only the exception code will be returned. If you disable production mode -- something that should be done for testing only -- most exception details will be returned: the exception message, line, and backtrace will all be attached.

To disable production mode, do the following:

```
$server->setProduction(false);
```

To re-enable it, pass a TRUE boolean value instead:

```
$server->setProduction(true);
```

Disable production mode sparingly!

We recommend disabling production mode only when in development. Exception messages and backtraces can contain sensitive system information that you may not wish for outside parties to access. Even though AMF is a binary format, the specification is now open, meaning anybody can potentially deserialize the payload.

One area to be especially careful with is PHP errors themselves. When the display_errors INI directive is enabled, any PHP errors for the current error reporting level are rendered directly in the output -- potentially disrupting the AMF response payload. We suggest turning off the display_errors directive in production to prevent such problems

2.3. AMF Responses

Occasionally you may desire to manipulate the response object slightly, typically to return extra message headers. The `handle()` method of the server returns the response object, allowing you to do so.

Example 30. Adding Message Headers to the AMF Response

In this example, we add a 'foo' MessageHeader with the value 'bar' to the response prior to returning it.

```
$response = $server->handle();
$response->addAmfHeader(new Zend_Amf_Value_MessageHeader('foo', true, 'bar'))
echo $response;
```

2.4. Typed Objects

Similar to SOAP, AMF allows passing objects between the client and server. This allows a great amount of flexibility and coherence between the two environments.

`Zend_Amf` provides three methods for mapping ActionScript and PHP objects.

- First, you may create explicit bindings at the server level, using the `setClassMap()` method. The first argument is the ActionScript class name, the second the PHP class name it maps to:

```
// Map the ActionScript class 'ContactVO' to the PHP class 'Contact':
$server->setClassMap('ContactVO', 'Contact');
```

- Second, you can set the public property `$_explicitType` in your PHP class, with the value representing the ActionScript class to map to:

```
class Contact
{
    public $_explicitType = 'ContactVO';
}
```

- Third, in a similar vein, you may define the public method `getASClassName()` in your PHP class; this method should return the appropriate ActionScript class:

```
class Contact
{
    public function getASClassName()
    {
        return 'ContactVO';
    }
}
```

Although we have created the ContactVO on the server we now need to make its corresponding class in AS3 for the server object to be mapped to.

Right click on the src folder of the Flex project and select New -> ActionScript File. Name the file ContactVO and press finish to see the new file. Copy the following code into the file to finish creating the class.

```
package
{
    [Bindable]
    [RemoteClass(alias="ContactVO")]
    public class ContactVO
    {
        public var id:int;
        public var firstname:String;
        public var lastname:String;
        public var email:String;
        public var mobile:String;
        public function ProductVO():void {
        }
    }
}
```

The class is syntactically equivalent to the PHP of the same name. The variable names are exactly the same and need to be in the same case to work properly. There are two unique AS3 meta tags in this class. The first is bindable which makes fire a change event when it is updated. The second tag is the RemoteClass tag which defines that this class can have a remote object mapped with the alias name in this case *ContactVO*. It is mandatory that this tag the value that was set is the PHP class are strictly equivalent.

```
[Bindable]
private var myContact:ContactVO;

private function getContactHandler(event:ResultEvent):void {
```

```
    myContact = ContactVO(event.result);
}
```

The following result event from the service call is cast instantly onto the Flex ContactVO. Anything that is bound to myContact will be updated with the returned ContactVO data.

2.5. Resources

`Zend_Amf` provides tools for mapping resource types returned by service classes into data consumable by ActionScript.

In order to handle specific resource type, the user needs to create a plugin class named after the resource name, with words capitalized and spaces removed (so, resource type "mysql result" becomes MysqlResult), with some prefix, e.g. `My_MysqlResult`. This class should implement one method, `parse()`, receiving one argument - the resource - and returning the value that should be sent to ActionScript. The class should be located in the file named after the last component of the name, e.g. `MysqlResult.php`.

The directory containing the resource handling plugins should be registered with `Zend_Amf` type loader:

```
Zend_Amf_Parse_TypeLoader::addResourceDirectory(
    "My",
    "application/library/resources/My"
);
```

For detailed discussion of loading plugins, please see the plugin loader section.

Default directory for `Zend_Amf` resources is registered automatically and currently contains handlers for "mysql result" and "stream" resources.

```
// Example class implementing handling resources of type mysql result
class Zend_Amf_Parse_Resource_MysqlResult
{
    /**
     * Parse resource into array
     *
     * @param resource $resource
     * @return array
     */
    public function parse($resource) {
        $result = array();
        while($row = mysql_fetch_assoc($resource)) {
            $result[] = $row;
        }
        return $result;
    }
}
```

Trying to return unknown resource type (i.e., one for which no handler plugin exists) will result in an exception.

2.6. Connecting to the Server from Flash

Connecting to your `Zend_Amf_Server` from your Flash project is slightly different than from Flex. However once the connection Flash functions with `Zend_Amf_Server` the same way is

flex. The following example can also be used from a Flex AS3 file. We will reuse the same `Zend_Amf_Server` configuration along with the World class for our connection.

Open Flash CS and create and new Flash File (ActionScript 3). Name the document `ZendExample.fla` and save the document into a folder that you will use for this example. Create a new AS3 file in the same directory and call the file `Main.as`. Have both files open in your editor. We are now going to connect the two files via the document class. Select ZendExample and click on the stage. From the stage properties panel change the Document class to Main. This links the `Main.as` ActionScript file with the user interface in `ZendExample.fla`. When you run the Flash file ZendExample the `Main.as` class will now be run. Next we will add ActionScript to make the AMF call.

We now are going to make a Main class so that we can send the data to the server and display the result. Copy the following code into your `Main.as` file and then we will walk through the code to describe what each element's role is.

```
package {
  import flash.display.MovieClip;
  import flash.events.*;
  import flash.net.NetConnection;
  import flash.net.Responder;

  public class Main extends MovieClip {
    private var gateway:String = "http://example.com/server.php";
    private var connection:NetConnection;
    private var responder:Responder;

    public function Main() {
      responder = new Responder(onResult, onFault);
      connection = new NetConnection;
      connection.connect(gateway);
    }

    public function onComplete( e:Event ):void{
      var params = "Sent to Server";
      connection.call("World.hello", responder, params);
    }

    private function onResult(result:Object):void {
      // Display the returned data
      trace(String(result));
    }
    private function onFault(fault:Object):void {
      trace(String(fault.description));
    }
  }
}
```

We first need to import two ActionScript libraries that perform the bulk of the work. The first is NetConnection which acts like a by directional pipe between the client and the server. The second is a Responder object which handles the return values from the server related to the success or failure of the call.

```
import flash.net.NetConnection;
import flash.net.Responder;
```

In the class we need three variables to represent the NetConnection, Responder, and the gateway URL to our `Zend_Amf_Server` installation.

```
private var gateway:String = "http://example.com/server.php";
private var connection:NetConnection;
private var responder:Responder;
```

In the Main constructor we create a responder and a new connection to the `Zend_Amf_Server` endpoint. The responder defines two different methods for handling the response from the server. For simplicity I have called these onResult and onFault.

```
responder = new Responder(onResult, onFault);
connection = new NetConnection;
connection.connect(gateway);
```

In the onComplete function which is run as soon as the construct has completed we send the data to the server. We need to add one more line that makes a call to the `Zend_Amf_Server` World->hello function.

```
connection.call("World.hello", responder, params);
```

When we created the responder variable we defined an onResult and onFault function to handle the response from the server. We added this function for the successful result from the server. A successful event handler is run every time the connection is handled properly to the server.

```
private function onResult(result:Object):void {
    // Display the returned data
    trace(String(result));
}
```

The onFault function, is called if there was an invalid response from the server. This happens when there is an error on the server, the URL to the server is invalid, the remote service or method does not exist, and any other connection related issues.

```
private function onFault(fault:Object):void {
    trace(String(fault.description));
}
```

Adding in the ActionScript to make the remoting connection is now complete. Running the ZendExample file now makes a connection to Zend Amf. In review you have added the required variables to open a connection to the remote server, defined what methods should be used when your application receives a response from the server, and finally displayed the returned data to output via `trace()`.

2.7. Authentication

`Zend_Amf_Server` allows you to specify authentication and authorization hooks to control access to the services. It is using the infrastructure provided by `Zend_Auth` and `Zend_Acl` components.

In order to define authentication, the user provides authentication adapter extening `Zend_Amf_Auth_Abstract` abstract class. The adapter should implement the `authenticate()` method just like regular authentication adapter.

The adapter should use properties *_username* and *_password* from the parent `Zend_Amf_Auth_Abstract` class in order to authenticate. These values are set by the server using `setCredentials()` method before call to `authenticate()` if the credentials are received in the AMF request headers.

The identity returned by the adapter should be an object containing property role for the ACL access control to work.

If the authentication result is not successful, the request is not proceseed further and failure message is returned with the reasons for failure taken from the result.

The adapter is connected to the server using `setAuth()` method:

```
$server->setAuth(new My_Amf_Auth());
```

Access control is performed by using `Zend_Acl` object set by `setAcl()` method:

```
$acl = new Zend_Acl();
createPermissions($acl); // create permission structure
$server->setAcl($acl);
```

If the ACL object is set, and the class being called defines `initAcl()` method, this method will be called with the ACL object as an argument. The class then can create additional ACL rules and return TRUE, or return FALSE if no access control is required for this class.

After ACL have been set up, the server will check if access is allowed with role set by the authentication, resource being the class name (or `NULL` for function calls) and privilege being the function name. If no authentication was provided, then if the *anonymous* role was defined, it will be used, otherwise the access will be denied.

```
if($this->_acl->isAllowed($role, $class, $function)) {
    return true;
} else {
    require_once 'Zend/Amf/Server/Exception.php';
    throw new Zend_Amf_Server_Exception("Access not allowed");
}
```

Zend_Application

1. Introduction

`Zend_Application` provides a bootstrapping facility for applications which provides reusable resources, common- and module-based bootstrap classes and dependency checking. It also takes care of setting up the PHP environment and introduces autoloading by default.

2. Zend_Application Quick Start

There are two paths to getting started with `Zend_Application`, and they depend on how you start your project. In each case, you always start with creating a `Bootstrap` class, and a related configuration file.

If you plan on using `Zend_Tool` to create your project, continue reading below. If you will be adding `Zend_Application` to an existing project, you'll want to skip ahead.

2.1. Using Zend_Tool

The quickest way to start using `Zend_Application` is to use `Zend_Tool` to generate your project. This will also create your `Bootstrap` class and file.

To create a project, execute the **zf** command (on *nix systems):

```
% zf create project newproject
```

Or the Windows `zf.bat` command:

```
C:> zf.bat create project newproject
```

Both will create a project structure that looks like the following:

```
newproject
|-- application
|   |-- Bootstrap.php
|   |-- configs
|   |   `-- application.ini
|   |-- controllers
|   |   |-- ErrorController.php
|   |   `-- IndexController.php
|   |-- models
|   `-- views
|       |-- helpers
|       `-- scripts
|           |-- error
|           |   `-- error.phtml
|           `-- index
|               `-- index.phtml
|-- library
|-- public
|   `-- index.php
`-- tests
    |-- application
    |   `-- bootstrap.php
```

```
    |-- library
    |   `-- bootstrap.php
    `-- phpunit.xml
```

In the above diagram, your bootstrap is in `newproject/application/Bootstrap.php`, and looks like the following at first:

```
class Bootstrap extends Zend_Application_Bootstrap_Bootstrap
{
}
```

You'll also note that a configuration file, `newproject/application/configs/application.ini`, is created. It has the following contents:

```
[production]
phpSettings.display_startup_errors = 0
phpSettings.display_errors = 0
includePaths.library = APPLICATION_PATH "/../library"
bootstrap.path = APPLICATION_PATH "/Bootstrap.php"
bootstrap.class = "Bootstrap"
resources.frontController.controllerDirectory = APPLICATION_PATH "/controllers"

[staging : production]

[testing : production]
phpSettings.display_startup_errors = 1
phpSettings.display_errors = 1

[development : production]
phpSettings.display_startup_errors = 1
phpSettings.display_errors = 1
```

All settings in this configuration file are for use with `Zend_Application` and your bootstrap.

Another file of interest is the `newproject/public/index.php` file, which invokes `Zend_Application` and dispatches it.

```
// Define path to application directory
defined('APPLICATION_PATH')
    || define('APPLICATION_PATH',
              realpath(dirname(__FILE__) . '/../application'));

// Define application environment
defined('APPLICATION_ENV')
    || define('APPLICATION_ENV',
              (getenv('APPLICATION_ENV') ? getenv('APPLICATION_ENV')
                                         : 'production'));

/** Zend_Application */
require_once 'Zend/Application.php';

// Create application, bootstrap, and run
$application = new Zend_Application(
    APPLICATION_ENV,
    APPLICATION_PATH . '/configs/application.ini'
);
$application->bootstrap()
            ->run();
```

To continue the quick start, please skip to the Resources section.

2.2. Adding Zend_Application to your application

The basics of `Zend_Application` are fairly simple:

- Create an `application/Bootstrap.php` file, with the class `Bootstrap`.
- Create an `application/configs/application.ini` configuration file with the base configuration necessary for `Zend_Application`.
- Modify your `public/index.php` to utilize `Zend_Application`.

First, create your `Bootstrap` class. Create a file, `application/Bootstrap.php`, with the following contents:

```
class Bootstrap extends Zend_Application_Bootstrap_Bootstrap
{
}
```

Now, create your configuration. For this tutorial, we will use an INI style configuration; you may, of course, use an XML or PHP configuration file as well. Create the file `application/configs/application.ini`, and provide the following contents:

```
[production]
phpSettings.display_startup_errors = 0
phpSettings.display_errors = 0
includePaths.library = APPLICATION_PATH "/../library"
bootstrap.path = APPLICATION_PATH "/Bootstrap.php"
bootstrap.class = "Bootstrap"
resources.frontController.controllerDirectory = APPLICATION_PATH "/controllers"

[staging : production]

[testing : production]
phpSettings.display_startup_errors = 1
phpSettings.display_errors = 1

[development : production]
phpSettings.display_startup_errors = 1
phpSettings.display_errors = 1
```

Now, let's modify your gateway script, `public/index.php`. If the file does not exist, create it; otherwise, replace it with the following contents:

```
// Define path to application directory
defined('APPLICATION_PATH')
    || define('APPLICATION_PATH',
              realpath(dirname(__FILE__) . '/../application'));

// Define application environment
defined('APPLICATION_ENV')
    || define('APPLICATION_ENV',
              (getenv('APPLICATION_ENV') ? getenv('APPLICATION_ENV')
                                         : 'production'));

// Typically, you will also want to add your library/ directory
// to the include_path, particularly if it contains your ZF installed
```

```
set_include_path(implode(PATH_SEPARATOR, array(
    dirname(dirname(__FILE__)) . '/library',
    get_include_path(),
)));

/** Zend_Application */
require_once 'Zend/Application.php';

// Create application, bootstrap, and run
$application = new Zend_Application(
    APPLICATION_ENV,
    APPLICATION_PATH . '/configs/application.ini'
);
$application->bootstrap()
            ->run();
```

You may note that the application environment constant value looks for an environment variable "APPLICATION_ENV". We recommend setting this in your web server environment. In Apache, you can set this either in your vhost definition, or in your `.htaccess` file. We recommend the following contents for your `public/.htaccess` file:

```
SetEnv APPLICATION_ENV development

RewriteEngine On
RewriteCond %{REQUEST_FILENAME} -s [OR]
RewriteCond %{REQUEST_FILENAME} -l [OR]
RewriteCond %{REQUEST_FILENAME} -d
RewriteRule ^.*$ - [NC,L]
RewriteRule ^.*$ index.php [NC,L]
```

Learn about mod_rewrite

The above rewrite rules allow access to any file under your virtual host's document root. If there are files you do not want exposed in this way, you may want to be more restrictive in your rules. Go to the Apache website to learn more about mod_rewrite.

At this point, you're all set to start taking advantage of `Zend_Application`.

2.3. Adding and creating resources

If you followed the directions above, then your bootstrap class will be utilizing a front controller, and when it is run, it will dispatch the front controller. However, in all likelihood, you'll need a little more configuration than this.

In this section, we'll look at adding two resources to your application. First, we'll set up your layouts, and then we'll customize your view object.

One of the standard resources provided with `Zend_Application` is the "layout" resource. This resource expects you to define configuration values which it will then use to configure your `Zend_Layout` instance.

To use it, all we need to do is update the configuration file.

```
[production]
phpSettings.display_startup_errors = 0
phpSettings.display_errors = 0
```

```
bootstrap.path = APPLICATION_PATH "/Bootstrap.php"
bootstrap.class = "Bootstrap"
resources.frontController.controllerDirectory = APPLICATION_PATH "/controllers"
; ADD THE FOLLOWING LINES
resources.layout.layout = "layout"
resources.layout.layoutPath = APPLICATION_PATH "/layouts/scripts"

[staging : production]

[testing : production]
phpSettings.display_startup_errors = 1
phpSettings.display_errors = 1

[development : production]
phpSettings.display_startup_errors = 1
phpSettings.display_errors = 1
```

If you haven't already, create the directory `application/layouts/scripts/`, and the file `layout.phtml` within that directory. A good starting layout is as follows (and ties in with the view resource covered next):

```
<?php echo $this->doctype() ?>
<html>
<head>
    <?php echo $this->headTitle() ?>
    <?php echo $this->headLink() ?>
    <?php echo $this->headStyle() ?>
    <?php echo $this->headScript() ?>
</head>
<body>
    <?php echo $this->layout()->content ?>
</body>
</html>
```

At this point, you will now have a working layout.

Now, we'll add a custom view resource. When initializing the view, we'll want to set the HTML DocType and a default value for the title to use in the HTML head. This can be accomplished by editing your `Bootstrap` class to add a method:

```
class Bootstrap extends Zend_Application_Bootstrap_Bootstrap
{
    protected function _initView()
    {
        // Initialize view
        $view = new Zend_View();
        $view->doctype('XHTML1_STRICT');
        $view->headTitle('My First Zend Framework Application');

        // Add it to the ViewRenderer
        $viewRenderer = Zend_Controller_Action_HelperBroker::getStaticHelper(
            'ViewRenderer'
        );
        $viewRenderer->setView($view);

        // Return it, so that it can be stored by the bootstrap
        return $view;
    }
}
```

This method will be automatically executed when you bootstrap the application, and will ensure your view is initialized according to your application needs.

2.4. Next steps with Zend_Application

The above should get you started with `Zend_Application` and creating your application bootstrap. From here, you should start creating resource methods, or, for maximum re-usability, resource plugin classes. Continue reading to learn more!

3. Theory of Operation

Getting an MVC application configured and ready to dispatch has required an increasing amount of code as more features become available: setting up the database, configuring your view and view helpers, configuring your layouts, registering plugins, registering action helpers, and more.

Additionally, you will often want to reuse the same code to bootstrap your tests, a cronjob, or a service script. While it's possible to simply include your bootstrap script, oftentimes there are initializations that are environment specific – you may not need the MVC for a cronjob, or just the DB layer for a service script.

`Zend_Application` aims to make this easier and to promote reuse by encapsulating bootstrapping into OOP paradigms.

`Zend_Application` is broken into three realms:

- `Zend_Application`: loads the PHP environment, including include_paths and autoloading, and instantiates the requested bootstrap class.
- `Zend_Application_Bootstrap`: provides interfaces for bootstrap classes. `Zend_Application_Bootstrap_Bootstrap` provides common functionality for most bootstrapping needs, including dependency checking algorithms and the ability to load bootstrap resources on demand.
- `Zend_Application_Resource` provides an interface for standard bootstrapping resources that can be loaded on demand by a bootstrap instance, as well as several default resource implementations.

Developers create a bootstrap class for their application, extending `Zend_Application_Bootstrap_Bootstrap` or implementing (minimally) `Zend_Application_Bootstrap_Bootstrapper`. The entry point (e.g., `public/index.php`) will load `Zend_Application`, and instantiate it by passing:

- The current environment
- Options for bootstrapping

The bootstrap options include the path to the file containing the bootstrap class and optionally:

- Any extra include_paths to set
- Any additional autoloader namespaces to register
- Any `php.ini` settings to initialize
- The class name for the bootstrap class (if not "Bootstrap")
- Resource prefix to path pairs to use
- Any resources to use (by class name or short name)

- Additional path to a configuration file to load
- Additional configuration options

Options may be an array, a `Zend_Config` object, or the path to a configuration file.

3.1. Bootstrapping

`Zend_Application`'s second area of responsibility is executing the application bootstrap. Bootstraps minimally need to implement `Zend_Application_Bootstrap_Bootstrapper`, which defines the following API:

```
interface Zend_Application_Bootstrap_Bootstrapper
{
    public function __construct($application);
    public function setOptions(array $options);
    public function getApplication();
    public function getEnvironment();
    public function getClassResources();
    public function getClassResourceNames();
    public function bootstrap($resource = null);
    public function run();
}
```

This API allows the bootstrap to accept the environment and configuration from the application object, report the resources its responsible for bootstrapping, and then bootstrap and run the application.

You can implement this interface on your own, extend `Zend_Application_Bootstrap_BootstrapAbstract`, or use `Zend_Application_Bootstrap_Bootstrap`.

Besides this functionality, there are a number of other areas of concern you should familiarize yourself with.

3.1.1. Resource Methods

The `Zend_Application_Bootstrap_BootstrapAbstract` implementation provides a simple convention for defining class resource methods. Any protected method beginning with a name prefixed with *_init* will be considered a resource method.

To bootstrap a single resource method, use the `bootstrap()` method, and pass it the name of the resource. The name will be the method name minus the *_init* prefix.

To bootstrap several resource methods, pass an array of names. Too bootstrap all resource methods, pass nothing.

Take the following bootstrap class:

```
class Bootstrap extends Zend_Application_Bootstrap_Bootstrap
{
    protected function _initFoo()
    {
        // ...
    }

    protected function _initBar()
    {
```

```
        // ...
    }

    protected function _initBaz()
    {
        // ...
    }
}
```

To bootstrap just the `_initFoo()` method, do the following:

```
$bootstrap->bootstrap('foo');
```

To bootstrap the `_initFoo()` and `_initBar()` methods, do the following:

```
$bootstrap->bootstrap(array('foo', 'bar'));
```

To bootstrap all resource methods, call `bootstrap()` with no arguments:

```
$bootstrap->bootstrap();
```

3.1.2. Bootstraps that use resource plugins

To make your bootstraps more re-usable, we have provided the ability to push your resources into resource plugin classes. This allows you to mix and match resources simply via configuration. We will cover how to create resources later; in this section we will show you how to utilize them only.

If your bootstrap should be capable of using resource plugins, you will need to implement an additional interface, `Zend_Application_Bootstrap_ResourceBootstrapper`. This interface defines an API for locating, registering, and loading resource plugins:

```
interface Zend_Application_Bootstrap_ResourceBootstrapper
{
    public function registerPluginResource($resource, $options = null);
    public function unregisterPluginResource($resource);
    public function hasPluginResource($resource);
    public function getPluginResource($resource);
    public function getPluginResources();
    public function getPluginResourceNames();
    public function setPluginLoader(Zend_Loader_PluginLoader_Interface $loader);
    public function getPluginLoader();
}
```

Resource plugins basically provide the ability to create resource intializers that can be re-used between applications. This allows you to keep your actual bootstrap relatively clean, and to introduce new resources without needing to touch your bootstrap itself.

`Zend_Application_Bootstrap_BootstrapAbstract` (and `Zend_Application_Bootstrap_Bootstrap` by extension) implement this interface as well, allowing you to utilize resource plugins.

To utilize resource plugins, you must specify them in the options passed to the application object and/or bootstrap. These options may come from a configuration file, or be passed in manually. Options will be of key to options pairs, with the key representing the resource name. The resource name will be the segment following the class prefix. For example, the resources shipped with Zend Framework have the class prefix "`Zend_Application_Resource_`"; anything following this would be the name of the resource. As an example,

```
$application = new Zend_Application(APPLICATION_ENV, array(
    'resources' => array(
        'FrontController' => array(
            'controllerDirectory' => APPLICATION_PATH . '/controllers',
        ),
    ),
));
```

This indicates that the "FrontController" resource should be used, with the options specified.

If you begin writing your own resource plugins, or utilize third-party resource plugins, you will need to tell your bootstrap where to look for them. Internally, the bootstrap utilizes `Zend_Loader_PluginLoader`, so you will only need to indicate the common class prefix an path pairs.

As an example, let's assume you have custom resource plugins in `APPLICATION_PATH/resources/` and that they share the common class prefix of `My_Resource`. You would then pass that information to the application object as follows:

```
$application = new Zend_Application(APPLICATION_ENV, array(
    'pluginPaths' => array(
        'My_Resource' => APPLICATION_PATH . '/resources/',
    ),
    'resources' => array(
        'FrontController' => array(
            'controllerDirectory' => APPLICATION_PATH . '/controllers',
        ),
    ),
));
```

You would now be able to use resources from that directory.

Just like resource methods, you use the `bootstrap()` method to execute resource plugins. Just like with resource methods, you can specify either a single resource plugin, multiple plugins (via an array), or all plugins. Additionally, you can mix and match to execute resource methods as well.

```
// Execute one:
$bootstrap->bootstrap('FrontController');

// Execute several:
$bootstrap->bootstrap(array('FrontController', 'Foo'));

// Execute all resource methods and plugins:
$bootstrap->bootstrap();
```

3.1.3. Resource Registry

Many, if not all, of your resource methods or plugins will initialize objects, and in many cases, these objects will be needed elsewhere in your application. How can you access them?

`Zend_Application_Bootstrap_BootstrapAbstract` provides a local registry for these objects. To store your objects in them, you simply return them from your resources.

For maximum flexibility, this registry is referred to as a "container" internally; its only requirements are that it is an object. Resources are then registered as properties named after the resource name. By default, an instance of `Zend_Registry` is used, but you may also specify any other object you wish. The methods `setContainer()` and `getContainer()` may be used

to manipulate the container itself. `getResource($resource)` can be used to fetch a given resource from the container, and `hasResource($resource)` to check if the resource has actually been registered.

As an example, consider a basic view resource:

```
class Bootstrap extends Zend_Application_Bootstrap_Bootstrap
{
    protected function _initView()
    {
        $view = new Zend_View();
        // more initialization...

        return $view;
    }
}
```

You can then check for it and/or fetch it as follows:

```
// Using the has/getResource() pair:
if ($bootstrap->hasResource('view')) {
    $view = $bootstrap->getResource('view');
}

// Via the container:
$container = $bootstrap->getContainer();
if (isset($container->view)) {
    $view = $container->view;
}
```

Please note that the registry and also the container is not global. This means that you need access to the bootstrap in order to fetch resources. `Zend_Application_Bootstrap_Bootstrap` provides some convenience for this: during its `run()` execution, it registers itself as the front controller parameter "bootstrap", which allows you to fetch it from the router, dispatcher, plugins, and action controllers.

As an example, if you wanted access to the view resource from above within your action controller, you could do the following:

```
class FooController extends Zend_Controller_Action
{
    public function init()
    {
        $bootstrap = $this->getInvokeArg('bootstrap');
        $view = $bootstrap->getResource('view');
        // ...
    }
}
```

3.1.4. Dependency Tracking

In addition to executing resource methods and plugins, it's necessary to ensure that these are executed once and once only; these are meant to bootstrap an application, and executing multiple times can lead to resource overhead.

At the same time, some resources may depend on other resources being executed. To solve these two issues, `Zend_Application_Bootstrap_BootstrapAbstract` provides a simple, effective mechanism for dependency tracking.

As noted previously, all resources -- whether methods or plugins -- are bootstrapped by calling `bootstrap($resource)`, where `$resource` is the name of a resource, an array of resources, or, left empty, indicates all resources should be run.

If a resource depends on another resource, it should call `bootstrap()` within its code to ensure that resource has been executed. Subsequent calls to it will then be ignored.

In a resource method, such a call would look like this:

```
class Bootstrap extends Zend_Application_Bootstrap_Bootstrap
{
    protected function _initRequest()
    {
        // Ensure the front controller is initialized
        $this->bootstrap('FrontController');

        // Retrieve the front controller from the bootstrap registry
        $front = $this->getResource('FrontController');

        $request = new Zend_Controller_Request_Http();
        $request->setBaseUrl('/foo');
        $front->setRequest($request);

        // Ensure the request is stored in the bootstrap registry
        return $request;
    }
}
```

3.2. Resource Plugins

As noted previously, a good way to create re-usable bootstrap resources and to offload much of your coding to discrete classes is to utilize resource plugins. While Zend Framework ships with a number of standard resource plugins, the intention is that developers should write their own to encapsulate their own initialization needs.

Resource plugins need only implement `Zend_Application_Resource_Resource`, or, more simply still, extend `Zend_Application_Resource_ResourceAbstract`. The basic interface is simply this:

```
interface Zend_Application_Resource_Resource
{
    public function __construct($options = null);
    public function setBootstrap(
        Zend_Application_Bootstrap_Bootstrapper $bootstrap
    );
    public function getBootstrap();
    public function setOptions(array $options);
    public function getOptions();
    public function init();
}
```

The interface defines simply that a resource plugin should accept options to the constructor, have mechanisms for setting and retrieving options, have mechanisms for setting and retrieving the bootstrap object, and an initialization method.

As an example, let's assume you have a common view intialization you use in your applications. You have a common doctype, CSS and JavaScript, and you want to be able to pass in a base document title via configuration. Such a resource plugin might look like this:

```
class My_Resource_View extends Zend_Application_Resource_ResourceAbstract
{
    protected $_view;

    public function init()
    {
        // Return view so bootstrap will store it in the registry
        return $this->getView();
    }

    public function getView()
    {
        if (null === $this->_view) {
            $options = $this->getOptions();
            $title   = '';
            if (array_key_exists('title', $options)) {
                $title = $options['title'];
                unset($options['title']);
            }

            $view = new Zend_View($options);
            $view->doctype('XHTML1_STRICT');
            $view->headTitle($title);
            $view->headLink()->appendStylesheet('/css/site.css');
            $view->headScript()->appendfile('/js/analytics.js');

            $viewRenderer =
                Zend_Controller_Action_HelperBroker::getStaticHelper(
                    'ViewRenderer'
                );
            $viewRenderer->setView($view);

            $this->_view = $view;
        }
        return $this->_view;
    }
}
```

As long as you register the prefix path for this resource plugin, you can then use it in your application. Even better, because it uses the plugin loader, you are effectively overriding the shipped "View" resource plugin, ensuring that your own is used instead.

4. Examples

The Bootstrap class itself will typically be fairly minimal; often, it will simply be an empty stub extending the base bootstrap class:

```
class Bootstrap extends Zend_Application_Bootstrap_Bootstrap
{
}
```

With a corresponding configuration file:

```
; APPLICATION_PATH/configs/application.ini
[production]
bootstrap.path = APPLICATION_PATH "/Bootstrap.php"
bootstrap.class = "Bootstrap"
resources.frontController.controllerDirectory = APPLICATION_PATH "/controllers"
```

```
[testing : production]
[development : production]
```

However, should custom initialization be necessary, you have two choices. First, you can write methods prefixed with *_init* to specify discrete code to bootstrap. These methods will be called by `bootstrap()`, and can also be called as if they were public methods: *bootstrap<resource>()*. They should accept an optional array of options.

If your resource method returns a value, it will be stored in a container in the bootstrap. This can be useful when different resources need to interact (such as one resource injecting itself into another). The method `getResource()` can then be used to retrieve those values.

The example below shows a resource method for initializing the request object. It makes use of dependency tracking (it depends on the front controller resource), fetching a resource from the bootstrap, and returning a value to store in the bootstrap.

```
class Bootstrap extends Zend_Application_Bootstrap_Bootstrap
{
    protected function _initRequest()
    {
        // Ensure front controller instance is present, and fetch it
        $this->bootstrap('FrontController');
        $front = $this->getResource('FrontController');

        // Initialize the request object
        $request = new Zend_Controller_Request_Http();
        $request->setBaseUrl('/foo');

        // Add it to the front controller
        $front->setRequest($request);

        // Bootstrap will store this value in the 'request' key of its container
        return $request;
    }
}
```

Note in this example the call to `bootstrap()`; this ensures that the front controller has been initialized prior to calling this method. That call may trigger either a resource or another method in the class.

The other option is to use resource plugins. Resource plugins are objects that perform specific initializations, and may be specified:

- When instantiating the `Zend_Application` object
- During initialization of the bootstrap object
- By explicitly enabling them via method calls to the bootstrap object

Resource plugins implement `Zend_Application_Resource_ResourceAbstract`, which defines simply that they allow injection of the caller and options, and that they have an `init()` method. As an example, a custom "View" bootstrap resource might look like the following:

```
class My_Bootstrap_Resource_View
    extends Zend_Application_Resource_ResourceAbstract
{
```

```
    public function init()
    {
        $view = new Zend_View($this->getOptions());
        Zend_Dojo::enableView($view);

        $view->doctype('XHTML1_STRICT');
        $view->headTitle()->setSeparator(' - ')->append('My Site');
        $view->headMeta()->appendHttpEquiv('Content-Type',
                                           'text/html; charset=utf-8');

        $view->dojo()->setDjConfigOption('parseOnLoad', true)
                     ->setLocalPath('/js/dojo/dojo.js')
                     ->registerModulePath('../spindle', 'spindle')
                     ->addStylesheetModule('spindle.themes.spindle')
                     ->requireModule('spindle.main')
                     ->disable();

        $viewRenderer = Zend_Controller_Action_HelperBroker::getStaticHelper(
            'ViewRenderer'
        );
        $viewRenderer->setView($view);

        return $view;
    }
}
```

To tell the bootstrap to use this, you would need to provide either the class name of the resource plugin, or a combination of a plugin loader prefix path and the short name of the resource plugin (e.g, "view"):

```
$application = new Zend_Application(
    APPLICATION_ENV,
    array(
        'resources' => array(
            'My_Bootstrap_Resource_View' => array(), // full class name; OR
            'view' => array(),                       // short name

            'FrontController' => array(
                'controllerDirectory' => APPLICATION_PATH . '/controllers',
            ),
        ),

        // For short names, define plugin paths:
        'pluginPaths = array(
            'My_Bootstrap_Resource' => 'My/Bootstrap/Resource',
        )
    )
);
```

Resource plugins can call on other resources and initializers by accessing the parent bootstrap:

```
class My_Bootstrap_Resource_Layout
    extends Zend_Application_Resource_ResourceAbstract
{
    public function init()
    {
        // ensure view is initialized...
        $this->getBootstrap()->bootstrap('view');
```

```
        // Get view object:
        $view = $this->getBootstrap()->getResource('view');

        // ...
    }
}
```

In normal usage, you would instantiate the application, bootstrap it, and run it:

```
$application = new Zend_Application(...);
$application->bootstrap()
            ->run();
```

For a custom script, you might need to simply initialize specific resources:

```
$application = new Zend_Application(...);
$application->getBootstrap()->bootstrap('db');

$service = new Zend_XmlRpc_Server();
$service->setClass('Foo');  // uses database...
echo $service->handle();
```

Instead of using the `bootstrap()` method to call the internal methods or resources, you may also use overloading:

```
$application = new Zend_Application(...);
$application->getBootstrap()->bootstrapDb();
```

5. Core Functionality

Here you'll find API-like documentation about all core components of `Zend_Application`.

5.1. Zend_Application

`Zend_Application` provides the base functionality of the component, and the entry point to your Zend Framework application. It's purpose is two-fold: to setup the PHP environment (including autoloading), and to execute your application bootstrap.

Typically, you will pass all configuration to the `Zend_Application` constructor, but you can also configure the object entirely using its own methods. This reference is intended to illustrate both use cases.

Table 2. Zend_Application options

Option	Description
phpSettings	Array of `php.ini` settings to use. Keys should be the `php.ini` keys.
includePaths	Additional paths to prepend to the *include_path*. Should be an array of paths.
autoloaderNamespaces	Array of additional namespaces to register with the `Zend_Loader_Autoloader` instance.
bootstrap	Either the string path to the bootstrap class, or an array with elements for the 'path' and 'class' for the application bootstrap.

Option names

Please note that option names are case insensitive.

Table 3. Zend_Application Methods

Method	Return Value	Parameters	Description

`__construct($environment, $options = null)`	Void	• `$environment`: *required*,. String representing the current application environment. Typical strings might include "development", "testing", "qa", or "production", but will be defined by your organizational requirements.
		• `$options`: *optional*. Argument may be one of the following values:
		  • *String*: path to a `Zend_Config` file to load as configuration for your application. `$environment` will be used to determine what section of the configuration to pull.
		  • *Array*: associative array of configuration data for your application.
		  • *`Zend_Config`*: configuration object instance.
			Constructor. Arguments are as described, and will be used to set initial object state. An instance of `Zend_Loader_Autoloader` is registered during instantiation. Options passed to the constructor are passed to `setOptions()`.

`getEnvironment()`	String	N/A	Retrieve the environment string passed to the constructor.
`getAutoloader()`	`Zend_Loader_Autoloader`	N/A	Retrieve the `Zend_Loader_Autoloader`

Method	Return Value	Parameters	Description
			instance registered during instantiation.
`setOptions(array $options)`	`Zend_Application`	• `$options`: *required*. An array of application options.	All options are stored internally, and calling the method multiple times will merge options. Options matching the various setter methods will be passed to those methods. As an example, the option "phpSettings" will then be passed to `setPhpSettings()`. (Option names are case insensitive.)
`getOptions()`	Array	N/A	Retrieve all options used to initialize the object; could be used to cache `Zend_Config` options to a serialized format between requests.
`hasOption($key)`	Boolean	• `$key`: String option key to lookup	Determine whether or not an option with the specified key has been registered. Keys are case insensitive.
`getOption($key)`	Mixed	• `$key`: String option key to lookup	Retrieve the option value of a given key. Returns `NULL` if the key does not exist.
`setPhpSettings(array $settings, $prefix = '')`	`Zend_Application`	• `$settings`: *required*. Associative array of PHP INI settings. • `$prefix`: *optional*. String prefix with which to prepend option keys. Used internally to allow mapping nested arrays to dot-separated `php.ini` keys. In normal usage, this argument should	Set run-time `php.ini` settings. Dot-separated settings may be nested hierarchically (which may occur with INI `Zend_Config` files) via an array-of-arrays, and will still resolve correctly.

Method	Return Value	Parameters	Description
		never be passed by a user.	
`setAutoloaderNamespaces($namespaces)`	`Zend_Application`	• `$namespaces`: *required*. Array of strings representing the namespaces to register with the `Zend_Loader_Autoloader` instance.	Register namespaces with the `Zend_Loader_Autoloader` instance.
`setBootstrap($path, $class = null)`	`Zend_Application`	• `$path`: *required*. May be either a `Zend_Application_Bootstrap_Bootstrapper` instance, a string path to the bootstrap class, an associative array of classname => filename, or an associative array with the keys 'class' and 'path'. • `$class`: *optional*. If `$path` is a string, `$class` may be specified, and should be a string class name of the class contained in the file represented by path.	
`getBootstrap()`	`NULL` | `Zend_Application_Bootstrap_Bootstrapper`	N/A	Retrieve the registered bootstrap instance.
`bootstrap()`	Void	N/A	Call the bootstrap's `bootstrap()` method to bootstrap the application.
`run()`	Void	N/A	Call the bootstrap's `run()` method to dispatch the application.

5.2. Zend_Application_Bootstrap_Bootstrapper

`Zend_Application_Bootstrap_Bootstrapper` is the base interface all bootstrap classes must implement. The base functionality is aimed at configuration, identifying resources, bootstrapping (either individual resources or the entire application), and dispatching the application.

The following methods make up the definition of the interface.

Table 4. Zend_Application_Bootstrap_Bootstrapper Interface

Method	Return Value	Parameters	Description
`__construct($application)`	Void	• `$application`: *required*. Should accept a `Zend_Application` or a `Zend_Application_Bootstrap_Bootstrapper` object as the sole argument.	Constructor. Accepts a single argument, which should be a `Zend_Application` object, or another bootstrap object.
`setOptions(array $options)`	`Zend_Application_Bootstrap_Bootstrapper`	• `$options`: *required*. Array of options to set.	Typically, any option that has a matching setter will invoke that setter; otherwise, the option will simply be stored for later retrieval.
`getApplication()`	`Zend_Application|Zend_Application_Bootstrap_Bootstrapper`	N/A	Retrieve the application or bootstrap object passed via the constructor.
`getEnvironment()`	String	N/A	Retrieve the environment string registered with the parent application or bootstrap object.
`getClassResources()`	Array	N/A	Retrieve a list of available resource initializer names as defined in the class. This may be implementation specific.
`bootstrap($resource = null)`	Mixed	• `$resource`: *optional*.	If `$resource` is empty, execute all bootstrap resources. If a string, execute that single resource; if an array, execute each resource in the array.
`run()`	Void	N/A	Defines what application logic to run after bootstrapping.

5.3. Zend_Application_Bootstrap_ResourceBootstrapper

`Zend_Application_Bootstrap_ResourceBootstrapper` is an interface to use when a bootstrap class will be loading external resources -- i.e., one or more resources will not be defined directly in the class, but rather via plugins.

It should be used in conjunction with Zend_Application_Bootstrap_Bootstrapper; Zend_Application_Bootstrap_BootstrapAbstract implements this functionality.

The following methods make up the definition of the interface.

Table 5. Zend_Application_Bootstrap_ResourceBootstrapper Interface

Method	Return Value	Parameters	Description
`registerPluginResource($resource, $options = null)`	`Zend_Application_Bootstrap_ResourceBootstrapper`	• `$resource`: *required*. A resource name or `Zend_Application_Resource_Resource` object. • `$options`: *optional*. An array or `Zend_Config` object to pass to the resource on instantiation.	Register a resource with the class, providing optional configuration to pass to the resource.
`unregisterPluginResource($resource)`	`Zend_Application_Bootstrap_ResourceBootstrapper`	• `$resource`: *required*. Name of a resource to unregister from the class.	Remove a plugin resource from the class.
`hasPluginResource($resource)`	`Boolean`	• `$resource`: *required*. Name of the resource.	Determine if a specific resource has been registered with the class.
`getPluginResource($resource)`	`Zend_Application_Resource_Resource`	• `$resource`: *required*. Name of a resource to retrieve (string).	Retrieve a plugin resource instance by name.
`getPluginResourceNames()`	`Array`	N/A	Retrieve a list of all registered plugin resource names.
`setPluginLoader(Zend_Loader_PluginLoader_Interface $loader)`	`Zend_Application_Bootstrap_ResourceBootstrapper`	• `$loader`: *required*. Plugin loader instance to use when resolving plugin names to classes.	Register a plugin loader instance to use when resolving plugin class names.
`getPluginLoader()`	`Zend_Loader_PluginLoader_Interface`	N/A	Retrieve the registered plugin loader.

5.4. Zend_Application_Bootstrap_BootstrapAbstract

`Zend_Application_Bootstrap_BootstrapAbstract` is an abstract class which provides the base functionality of a common bootstrap. It implements both `Zend_Application_Bootstrap_Bootstrapper` and `Zend_Application_Bootstrap_ResourceBootstrapper`.

Table 6. Zend_Application_Bootstrap_BootstrapAbstract Methods

Method	Return Value	Parameters	Description
`__construct($application)`	Void	• `$application`: *required*. Accepts either a `Zend_Application` or a `Zend_Application_Bootstrap_Bootstrapper` object as the sole argument.	Constructor. Accepts a single argument, which should be a `Zend_Application` object, or another bootstrap object.
`setOptions(array $options)`	`Zend_Application_Bootstrap_Bootstrapper`	• `$options`: *required*. Array of options to set.	Any option that has a matching setter will invoke that setter; otherwise, the option will simply be stored for later retrieval. As an example, if your extending class defined a `setFoo()` method, the option 'foo' would pass the value to that method. Two additional, special options keys may also be used. pluginPaths may be used to specify prefix paths to plugin resources; it should be an array of class prefix to filesystem path pairs. resources may be used to specify plugin resources to use, and should consist of plugin resource to instantiation options pairs.
`getOptions()`	Array	N/A	Returns all options registered via `setOptions()`.
`hasOption($key)`	Boolean	• `$key`: *required*. Option key to test.	Determine if an option key is present.
`getOption($key)`	Mixed	• `$key`: *required*. Option key to retrieve.	Retrieve the value associated with an option key; returns `NULL` if no option is registered with that key.

Method	Return Value	Parameters	Description
`setApplication(Zend_Application | Zend_Application_Bootstrap_Bootstrapper $application)`	`Zend_Application_Bootstrap_Bootstrapper`	`$application`, *required*.	Register the parent application or bootstrap object.
`getApplication()`	`Zend_Application|Zend_Application_Bootstrap_Bootstrapper`	N/A	Retrieve the application or bootstrap object passed via the constructor.
`getEnvironment()`	String	N/A	Retrieve the environment string registered with the parent application or bootstrap object.
`getClassResources()`	Array	N/A	Retrieve a list of available resource initializer names as defined in the class. This may be implementation specific.
`getContainer()`	Object	N/A	Retrieves the container that stores resources. If no container is currently registered, it registers a Zend_Registry instance before returning it.
`setContainer($container)`	`Zend_Application_Bootstrap_Bootstrapper`	`$container`, *required*. An object in which to store resources.	Provide a container in which to store resources. When a resource method or plugin returns a value, it will be stored in this container for later retrieval.
`hasResource($name)`	Boolean	• `$name`, *required*. Name of a resource to check.	When a resource method or plugin returns a value, it will be stored in the resource container (see `getContainer()` and `setContainer()`. This method will indicate whether or not a value for that resource has been set.

Method	Return Value	Parameters	Description
`getResource($name)`	Mixed	• `$name`, *required*. Name of a resource to fetch.	When a resource method or plugin returns a value, it will be stored in the resource container (see `getContainer()` and `setContainer()`. This method will retrieve a resources from the container.
`bootstrap($resource = null)`	Mixed	• `$resource`: *optional*.	If `$resource` is empty, execute all bootstrap resources. If a string, execute that single resource; if an array, execute each resource in the array. This method can be used to run individual bootstraps either defined in the class itself or via resource plugin classes. A resource defined in the class will be run in preference over a resource plugin in the case of naming conflicts.
`run()`	Void	N/A	Defines what application logic to run after bootstrapping.
`__call($method, $args)`	Mixed	• `$method`: *required*. The method name to call. • `$args`: *required*. Array of arguments to use in the method call.	Provides convenience to bootstrapping individual resources by allowing you to call 'bootstrap<ResourceName>()' instead of using the `bootstrap()` method.

5.5. Zend_Application_Bootstrap_Bootstrap

`Zend_Application_Bootstrap_Bootstrap` is a concrete implementation of Zend_Application_Bootstrap_BootstrapAbstract. It's primary feature are that it registers the Front Controller resource, and that the `run()` method first checks that a default module is defined and then dispatches the front controller.

In most cases, you will want to extend this class for your bootstrapping needs, or simply use this class and provide a list of resource plugins to utilize.

5.5.1. Enabling Application Autoloading

Additionally, this bootstrap implementation provides the ability to specify the "namespace" or class prefix for resources located in its tree, which will enable autoloading of various application resources; essentially, it instantiates a Zend_Application_Module_Autoloader object, providing the requested namespace and the bootstrap's directory as arguments. You may enable this functionality by providing a namespace to the "appnamespace" configuration option. As an INI example:

```
appnamespace = "Application"
```

Or in XML:

```
<appnamespace>Application</appnamespace>
```

By default, `Zend_Tool` will enable this option with the value "Application".

Alternately, you can simply define the `$_appNamespace` property of your bootstrap class with the appropriate value:

```
class Bootstrap extends Zend_Application_Bootstrap_Bootstrap
{
    protected $_appNamespace = 'Application';
}
```

5.6. Zend_Application_Resource_Resource

`Zend_Application_Resource_Resource` is an interface for plugin resources used with bootstrap classes implementing `Zend_Application_Bootstrap_ResourceBootstrapper`. Resource plugins are expected to allow configuration, be bootstrap aware, and implement a strategy pattern for initializing the resource.

Table 7. Zend_Application_Resource_Resource Interface

Method	Return Value	Parameters	Description
`__construct($options = null)`	Void	• `$options`: *optional*. Options with which to set resource state.	The constructor should allow passing options with which to initialize state.
`setBootstrap(Zend_Application_Bootstrap_ResourceBootstrapper $bootstrap)`	Zend_Application_Bootstrap_ResourceBootstrapper	• `$bootstrap`: *required*. Parent bootstrap initializing this resource.	Should allow registering the parent bootstrap object.
`getBootstrap()`	`Zend_Application_Bootstrap_Bootstrapper`	None	Retrieve the registered bootstrap instance.
`setOptions(array $options)`	`Zend_Application_Resource_Resource`	• `$options`: *required*. Options with which to set state.	Set resource state.

Method	Return Value	Parameters	Description
`getOptions()`	Array	N/A	Retrieve registered options.
`init()`	Mixed	N/A	Strategy pattern: run initialization of the resource.

5.7. Zend_Application_Resource_ResourceAbstract

`Zend_Application_Resource_ResourceAbstract` is an abstract class implementing Zend_Application_Resource_Resource, and is a good starting point for creating your own custom plugin resources.

Note: this abstract class does not implement the `init()` method; this is left for definition in concrete extensions of the class.

Table 8. Zend_Application_Resource_ResourceAbstract Methods

Method	Return Value	Parameters	Description
`__construct($options = null)`	Void	• `$options`: *optional*. Options with which to set resource state.	The constructor should allow passing options with which to initialize state.
`setBootstrap(Zend_Application_Bootstrap_Bootstrapper $bootstrap)`	Zend_Application_Resource_ResourceAbstract	• `$bootstrap`: *required*. Parent bootstrap initializing this resource.	Should allow registering the parent bootstrap object.
`getBootstrap()`	`Zend_Application_Bootstrap_Bootstrapper`	N/A	Retrieve the registered bootstrap instance.
`setOptions(array $options)`	`Zend_Application_Resource_ResourceAbstract`	• `$options`: *required*. Options with which to set state.	Set resource state.
`getOptions()`	Array	N/A	Retrieve registered options.

5.7.1. Resource Names

When registering plugin resources, one issue that arises is how you should refer to them from the parent bootstrap class. There are three different mechanisms that may be used, depending on how you have configured the bootstrap and its plugin resources.

First, if your plugins are defined within a defined prefix path, you may refer to them simply by their "short name" -- i.e., the portion of the class name following the class prefix. As an example, the class "`Zend_Application_Resource_View`" may be referenced as simply "View", as the prefix path "`Zend_Application_Resource`" is already registered. You may register them using the full class name or the short name:

```
$app = new Zend_Application(APPLICATION_ENV, array(
    'pluginPaths' => array(
        'My_Resource' => 'My/Resource/',
```

```
    ),
    'resources' => array(
        // if the following class exists:
        'My_Resource_View' => array(),

        // then this is equivalent:
        'View' => array(),
    ),
));
```

In each case, you can then bootstrap the resource and retrieve it later using the short name:

```
$bootstrap->bootstrap('view');
$view = $bootstrap->getResource('view');
```

Second, if no matching plugin path is defined, you may still pass a resource by the full class name. In this case, you can reference it using the resource's full class name:

```
$app = new Zend_Application(APPLICATION_ENV, array(
    'resources' => array(
        // This will load the standard 'View' resource:
        'View' => array(),

        // While this loads a resource with a specific class name:
        'My_Resource_View' => array(),
    ),
));
```

Obviously, this makes referencing the resource much more verbose:

```
$bootstrap->bootstrap('My_Resource_View');
$view = $bootstrap->getResource('My_Resource_View');
```

This brings us to the third option. You can specify an explicit name that a given resource class will register as. This can be done by adding a public `$_explicitType` property to the resource plugin class, with a string value; that value will then be used whenever you wish to reference the plugin resource via the bootstrap. As an example, let's define our own view class:

```
class My_Resource_View extends Zend_Application_Resource_ResourceAbstract
{
    public $_explicitType = 'My_View';

    public function init()
    {
        // do some initialization...
    }
}
```

We can then bootstrap that resource or retrieve it by the name "`My_View`":

```
$bootstrap->bootstrap('My_View');
$view = $bootstrap->getResource('My_View');
```

Using these various naming approaches, you can override existing resources, add your own, mix multiple resources to achieve complex initialization, and more.

6. Available Resource Plugins

Here you'll find API-like documentation about all resource plugins available by default in `Zend_Application`.

6.1. Zend_Application_Resource_Cachemanager

`Zend_Application_Resource_Cachemanager` may be utilised to configure a set of `Zend_Cache` option bundles for use when lazy loading caches using `Zend_Cache_Manager`

As the Cache Manager is a lazy loading mechanism, the options are translated to option templates used to instantiate a cache object on request.

Example 31. Sample Cachemanager resource configuration

Below is a sample INI file showing how `Zend_Cache_Manager` may be configured. The format is the Cachemanager resource prefix (resources.cachemanager) followed be the name to assign to an option cache template or bundle (e.g. resources.cachemanager.database) and finally followed by a typical `Zend_Cache` option.

```
resources.cachemanager.database.frontend.name = Core
resources.cachemanager.database.frontend.options.lifetime = 7200
resources.cachemanager.database.frontend.options.automatic_serialization = true
resources.cachemanager.database.backend.name = File
resources.cachemanager.database.backend.options.cache_dir = "/path/to/cache"
```

Actually retrieving this cache from the Cache Manager is as simple as accessing an instance of the Manager and calling **$cacheManager->getCache('database');**.

6.2. Zend_Application_Resource_Db

`Zend_Application_Resource_Db` will initialize a `Zend_Db` adapter based on the options passed to it. By default, it also sets the adapter as the default adapter for use with `Zend_Db_Table`. If you want to use mutliple databases simultaneously, you can use the Multidb Resource Plugin.

The following configuration keys are recognized:

- *adapter*: `Zend_Db` adapter type.
- *params*: associative array of configuration parameters to use when retrieving the adapter instance.
- *isDefaultTableAdapter*: whether or not to establish this adapter as the default table adapter.

Example 32. Sample DB adapter resource configuration

Below is an example INI configuration that can be used to initialize the DB resource.

```
[production]
resources.db.adapter = "pdo_mysql"
resources.db.params.host = "localhost"
resources.db.params.username = "webuser"
resources.db.params.password = "XXXXXXX"
resources.db.params.dbname = "test"
resources.db.isDefaultTableAdapter = true
```

Retrieving the Adapter instance

If you choose not to make the adapter instantiated with this resource the default table adapter, how do you retrieve the adapter instance?

As with any resource plugin, you can fetch the DB resource plugin from your bootstrap:

```
$resource = $bootstrap->getPluginResource('db');
```

Once you have the resource object, you can fetch the DB adapter using the `getDbAdapter()` method:

```
$db = $resource->getDbAdapter();
```

6.3. Zend_Application_Resource_Frontcontroller

Probably the most common resource you will load with `Zend_Application` will be the Front Controller resource, which provides the ability to configure `Zend_Controller_Front`. This resource provides the ability to set arbitrary front controller parameters, specify plugins to initialize, and much more.

Once initialized, the resource assigns the `$frontController` property of the bootstrap to the `Zend_Controller_Front` instance.

Available configuration keys include the following, and are case insensitive:

- *controllerDirectory*: either a string value specifying a single controller directory, or an array of module to controller directory pairs.
- *moduleControllerDirectoryName*: a string value indicating the subdirectory under a module that contains controllers.
- *moduleDirectory*: directory under which modules may be found.
- *defaultControllerName*: base name of the default controller (normally "index").
- *defaultAction*: base name of the default action (normally "index").
- *defaultModule*: base name of the default module (normally "default").
- *baseUrl*: explicit base URL to the application (normally auto-detected).
- *plugins*: array of front controller plugin class names. The resource will instantiate each class (with no constructor arguments) and then register the instance with the front controller.
- *params*: array of key to value pairs to register with the front controller.

If an unrecognized key is provided, it is registered as a front controller parameter by passing it to `setParam()`.

Example 33. Sample Front Controller resource configuration

Below is a sample INI snippet showing how to configure the front controller resource.

```
[production]
resources.frontController.controllerDirectory = APPLICATION_PATH "/controllers"
resources.frontController.moduleControllerDirectoryName = "actions"
resources.frontController.moduleDirectory = APPLICATION_PATH "/modules"
resources.frontController.defaultControllerName = "site"
resources.frontController.defaultAction = "home"
resources.frontController.defaultModule = "static"
resources.frontController.baseUrl = "/subdir"
resources.frontController.plugins.foo = "My_Plugin_Foo"
resources.frontController.plugins.bar = "My_Plugin_Bar"
resources.frontController.env = APPLICATION_ENV
```

Example 34. Retrieving the Front Controller in your bootstrap

Once your Front Controller resource has been initialized, you can fetch the Front Controller instance via the `$frontController` property of your bootstrap.

```
$bootstrap->bootstrap('frontController');
$front = $bootstrap->frontController;
```

6.4. Zend_Application_Resource_Layout

`Zend_Application_Resource_Layout` can be used to configure `Zend_Layout`. Configuration options are per the `Zend_Layout` options.

Example 35. Sample Layout configuration

Below is a sample INI snippet showing how to configure the navigation resource.

```
resources.layout.layout = "NameOfDefaultLayout"
resources.layout.layoutPath = "/path/to/layouts"
```

6.5. Zend_Application_Resource_Locale

`Zend_Application_Resource_Locale` can be used to set an application-wide locale which is then used in all classes and components which work with localization or internationalization.

There are basically three usecases for the Locale Resource Plugin. Each of them should be used depending on the applications need.

6.5.1. Autodetect the locale to use

Without specifying any options for `Zend_Application_Resource_Locale`, `Zend_Locale` will detect the locale, which your application will use, automatically.

This detection works because your client sends the wished language within his HTTP request. Normally the clients browser sends the languages he wants to see, and `Zend_Locale` uses this information for detection.

But there are 2 problems with this approach:

- The browser could be setup to send no language

- The user could have manually set a locale which does not exist

In both cases `Zend_Locale` will fallback to other mechanism to detect the locale:

- When a locale has been set which does not exist, `Zend_Locale` tries to downgrade this string.

  When, for example, *en_ZZ* is set it will automatically be degraded to *en*. In this case *en* will be used as locale for your application.

- When the locale could also not be detected by downgrading, the locale of your environment (web server) will be used. Most available environments from Web Hosters use *en* as locale.

- When the systems locale could not be detected `Zend_Locale` will use it's default locale, which is set to *en* per default.

For more informations about locale detection take a look into this chapter on Zend_Locale's automatic detection.

6.5.2. Autodetect the locale and adding a own fallback

The above autodetection could lead to problems when the locale could not be detected and you want to have another default locale than *en*. To prevent this, `Zend_Application_Resource_Locale` allows you to set a own locale which will be used in the case that the locale could not be detected.

Example 36. Autodetect locale and setting a fallback

The following snippet shows how to set a own default locale which will be used when the client does not send a locale himself.

```
; Try to determine automatically first,
; if unsuccessful, use nl_NL as fallback.
resources.locale.default = "nl_NL"
```

6.5.3. Forcing a specific locale to use

Sometimes it is useful to define a single locale which has to be used. This can be done by using the force option.

In this case this single locale will be used and the automatic detection is turned off.

Example 37. Defining a single locale to use

The following snippet shows how to set a single locale for your entire application.

```
; No matter what, the nl_NL locale will be used.
resources.locale.default = "nl_NL"
resources.locale.force = true
```

6.6. Zend_Application_Resource_Log

`Zend_Application_Resource_Log` to instantiate a `Zend_Log` instance with an arbitrary number of log writers. Configuration will be passed to the `Zend_Log::factory()` method, allowing you to specify combinations of log writers and filters. The log instance may then be retrieved from the bootstrap later in order to log events.

Example 38. Sample Log Resource Configuration

Below is a sample INI snippet showing how to configure the log resource.

```
resources.log.stream.writerName = "Stream"
resources.log.stream.writerParams.stream = APPLICATION_PATH "/../data/logs/application.
resources.log.stream.writerParams.mode = "a"
resources.log.stream.filterName = "Priority"
resources.log.stream.filterParams.priority = 4
```

For more information on available options, please review the Zend_Log::factory() documentation.

6.7. Zend_Application_Resource_Mail

`Zend_Application_Resource_Mail` can be used to instantiate a transport for `Zend_Mail` or set the default name and address, as well as the default replyto- name and address.

When instantiating a transport, it's registered automatically to `Zend_Mail`. Though, by setting the transport.register directive to FALSE, this behaviour does no more occur.

Example 39. Sample Mail Resource Configuration

Below is a sample INI snippet showing how to configure the mail resource plugin.

```
resources.mail.transport.type = smtp
resources.mail.transport.host = "smtp.example.com"
resources.mail.transport.auth = login
resources.mail.transport.username = myUsername
resources.mail.transport.password = myPassword
resources.mail.transport.register = true ; True by default

resources.mail.defaultFrom.email = john@example.com
resources.mail.defaultFrom.name = "John Doe"
resources.mail.defaultReplyTo.email = Jane@example.com
resources.mail.defaultReplyTo.name = "Jane Doe"
```

6.8. Zend_Application_Resource_Modules

`Zend_Application_Resource_Modules` is used to initialize your application modules. If your module has a `Bootstrap.php` file in its root, and it contains a class named `Module_Bootstrap` (where "Module" is the module name), then it will use that class to bootstrap the module.

By default, an instance of `Zend_Application_Module_Autoloader` will be created for the module, using the module name and directory to initialize it.

Since the Modules resource does not take any arguments by default, in order to enable it via configuration, you need to create it as an empty array. In INI style configuration, this looks like:

```
resources.modules[] =
```

In XML style configuration, this looks like:

```
<resources>
```

```
    <modules>
        <!-- Placeholder to ensure an array is created -->
        <placeholder />
    </modules>
</resources>
```

Using a standard PHP array, simply create it as an empty array:

```
$options = array(
    'resources' => array(
        'modules' => array(),
    ),
);
```

Front Controller Resource Dependency

The Modules resource has a dependency on the Front Controller resource. You can, of course, provide your own replacement for that resource via a custom Front Controller resource class or a class initializer method -- so long as the resource plugin class ends in "Frontcontroller" or the initializer method is named "_initFrontController" (case insensitive).

Example 40. Configuring Modules

You can specify module-specific configuration using the module name as a prefix or sub-section in your configuration file.

For example, let's assume that your application has a "news" module. The following are INI and XML examples showing configuration of resources in that module.

```
[production]
news.resources.db.adapter = "pdo_mysql"
news.resources.db.params.host = "localhost"
news.resources.db.params.username = "webuser"
news.resources.db.params.password = "XXXXXXX"
news.resources.db.params.dbname = "news"
```

```
<?xml version="1.0"?>
<config>
    <production>
        <news>
            <resources>
                <db>
                    <adapter>pdo_mysql</adapter>
                    <params>
                        <host>localhost</host>
                        <username>webuser</username>
                        <password>XXXXXXX</password>
                        <dbname>news</dbname>
                    </params>
                    <isDefaultAdapter>true</isDefaultAdapter>
                </db>
            </resources>
        </news>
    </production>
</config>
```

Example 41. Retrieving a specific module bootstrap

On occasion, you may need to retrieve the bootstrap object for a specific module -- perhaps to run discrete bootstrap methods, or to fetch the autoloader in order to configure it. This can be done using the Modules resource's `getExecutedBootstraps()` method.

```
$resource = $bootstrap->getPluginResource('modules');
$moduleBootstraps = $resource->getExecutedBootstraps();
$newsBootstrap = $moduleBootstraps['news'];
```

6.9. Zend_Application_Resource_Multidb

`Zend_Application_Resource_Multidb` is used to initialize multiple Database connections. You can use the same options as you can with the Db Resource Plugin. However, for specifying a default connection, you can also use the 'default' directive.

Example 42. Setting up multiple Db Connections

Below is an example INI configuration that can be used to initialize two Db Connections.

```
[production]
resources.multidb.db1.adapter = "pdo_mysql"
resources.multidb.db1.host = "localhost"
resources.multidb.db1.username = "webuser"
resources.multidb.db1.password = "XXXX"
resources.multidb.db1.dbname = "db1"

resources.multidb.db2.adapter = "pdo_pgsql"
resources.multidb.db2.host = "example.com"
resources.multidb.db2.username = "dba"
resources.multidb.db2.password = "notthatpublic"
resources.multidb.db2.dbname = "db2"
resources.multidb.db2.default = true
```

Example 43. Retrieving a specific database adapter

When using this resource plugin you usually will want to retrieve a specific database. This can be done by using the resource's `getDb()`. The method `getDb()` returns an instance of a class that extends `Zend_Db_Adapter_Abstract`. If you have not set a default database, an exception will be thrown when this method is called without specifying a parameter.

```
$resource = $bootstrap->getPluginResource('multidb');
$db1 = $resource->getDb('db1');
$db2 = $resource->getDb('db2');
$defaultDb = $resource->getDb();
```

Example 44. Retrieving the default database adapter

Additionally, you can retrieve the default database adapter by using the method `getDefaultDb()`. If you have not set a default adapter, the first configured db adapter will be returned. Unless you specify FALSE as first parameter, then `NULL` will be returned when no default database adapter was set.

Below is an example that assumes the Multidb resource plugin has been configured with the INI sample above:

```
$resource = $bootstrap->getPluginResource('multidb');
$db2 = $resource->getDefaultDb();

// Same config, but now without a default db:
$db1 = $resource->getDefaultDb();
$null = $resource->getDefaultDb(false); // null
```

6.10. Zend_Application_Resource_Navigation

`Zend_Application_Resource_Navigation` can be used to configure a `Zend_Navigation` instance. Configuration options are per the `Zend_Navigation` options.

Once done configuring the navigation instance, it assigns the instance to `Zend_View_Helper_Navigation` by default -- from which you may retrieve it later.

Example 45. Sample Navigation resource configuration

Below is a sample INI snippet showing how to configure the navigation resource.

```
resources.navigation.pages.page1.label = "Label of the first page"
resources.navigation.pages.page1.route = "Route that belongs to the first page"

; Page 2 is a subpage of page 1
resources.navigation.pages.page1.pages.page2.type = "Zend_Navigation_Page_Uri"
resources.navigation.pages.page1.pages.page2.label = "Label of the second page"
resources.navigation.pages.page1.pages.page2.uri = "/url/to/page/2"
```

6.11. Zend_Application_Resource_Router

`Zend_Application_Resource_Router` can be used to configure the router as it is registered with the Front Controller. Configuration options are per the `Zend_Controller_Router_Route` options.

Example 46. Sample Router Resource configuration

Below is a sample INI snippet showing how to configure the router resource.

```
resources.router.routes.route_id.route = "/login"
resources.router.routes.route_id.defaults.module = "user"
resources.router.routes.route_id.defaults.controller = "login"
resources.router.routes.route_id.defaults.action = "index"

; Optionally you can also set a Chain Name Separator:
resources.router.chainNameSeparator = "_"
```

For more information on the Chain Name Separator, please see its section.

6.12. Zend_Application_Resource_Session

`Zend_Application_Resource_Session` allows you to configure `Zend_Session` as well as optionally initialize a session SaveHandler.

To set a session save handler, simply pass the saveHandler (case insensitive) option key to the resource. The value of this option may be one of the following:

- String: A string indicating a class implementing `Zend_Session_SaveHandler_Interface` that should be instantiated.
- Array: An array with the keys "class" and, optionally, "options", indicating a class implementing `Zend_Session_SaveHandler_Interface` that should be instantiated and an array of options to provide to its constructor.
- `Zend_Session_SaveHandler_Interface`: an object implementing this interface.

Any other option keys passed will be passed to `Zend_Session::setOptions()` to configure `Zend_Session`.

Example 47. Sample Session resource configuration

Below is a sample INI snippet showing how to configure the session resource. It sets several `Zend_Session` options, as well as configures a `Zend_Session_SaveHandler_DbTable` instance.

```
resources.session.save_path = APPLICATION_PATH "/../data/session"
resources.session.use_only_cookies = true
resources.session.remember_me_seconds = 864000
resources.session.saveHandler.class = "Zend_Session_SaveHandler_DbTable"
resources.session.saveHandler.options.name = "session"
resources.session.saveHandler.options.primary.session_id = "session_id"
resources.session.saveHandler.options.primary.save_path = "save_path"
resources.session.saveHandler.options.primary.name = "name"
resources.session.saveHandler.options.primaryAssignment.sessionId = "sessionId"
resources.session.saveHandler.options.primaryAssignment.sessionSavePath = "sessionSavePa
resources.session.saveHandler.options.primaryAssignment.sessionName = "sessionName"
resources.session.saveHandler.options.modifiedColumn = "modified"
resources.session.saveHandler.options.dataColumn = "session_data"
resources.session.saveHandler.options.lifetimeColumn = "lifetime"
```

Bootstrap your database first!

If you are configuring the `Zend_Session_SaveHandler_DbTable` session save handler, you must first configure your database connection for it to work. Do this by either using the Db resource -- and make sure the "resources.db" key comes prior to the "resources.session" key -- or by writing your own resource that initializes the database, and specifically sets the default `Zend_Db_Table` adapter.

6.13. Zend_Application_Resource_View

`Zend_Application_Resource_View` can be used to configure a `Zend_View` instance. Configuration options are per the `Zend_View` options.

Once done configuring the view instance, it creates an instance of `Zend_Controller_Action_Helper_ViewRenderer` and registers the ViewRenderer with `Zend_Controller_Action_HelperBroker` -- from which you may retrieve it later.

Example 48. Sample View resource configuration

Below is a sample INI snippet showing how to configure the view resource.

```
resources.view.encoding = "UTF-8"
resources.view.basePath = APPLICATION_PATH "/views/scripts"
```

Zend_Auth

1. Introduction

`Zend_Auth` provides an API for authentication and includes concrete authentication adapters for common use case scenarios.

`Zend_Auth` is concerned only with *authentication* and not with *authorization*. Authentication is loosely defined as determining whether an entity actually is what it purports to be (i.e., identification), based on some set of credentials. Authorization, the process of deciding whether to allow an entity access to, or to perform operations upon, other entities is outside the scope of `Zend_Auth`. For more information about authorization and access control with Zend Framework, please see `Zend_Acl`.

The `Zend_Auth` class implements the Singleton pattern - only one instance of the class is available - through its static `getInstance()` method. This means that using the *new* operator and the *clone* keyword will not work with the `Zend_Auth` class; use `Zend_Auth::getInstance()` instead.

1.1. Adapters

A `Zend_Auth` adapter is used to authenticate against a particular type of authentication service, such as LDAP, RDBMS, or file-based storage. Different adapters are likely to have vastly different options and behaviors, but some basic things are common among authentication adapters. For example, accepting authentication credentials (including a purported identity), performing queries against the authentication service, and returning results are common to `Zend_Auth` adapters.

Each `Zend_Auth` adapter class implements `Zend_Auth_Adapter_Interface`. This interface defines one method, `authenticate()`, that an adapter class must implement for performing an authentication query. Each adapter class must be prepared prior to calling `authenticate()`. Such adapter preparation includes setting up credentials (e.g., username and password) and defining values for adapter-specific configuration options, such as database connection settings for a database table adapter.

The following is an example authentication adapter that requires a username and password to be set for authentication. Other details, such as how the authentication service is queried, have been omitted for brevity:

```
class MyAuthAdapter implements Zend_Auth_Adapter_Interface
{
    /**
     * Sets username and password for authentication
     *
     * @return void
     */
    public function __construct($username, $password)
    {
        // ...
    }
```

```
    /**
     * Performs an authentication attempt
     *
     * @throws Zend_Auth_Adapter_Exception If authentication cannot
     *                                     be performed
     * @return Zend_Auth_Result
     */
    public function authenticate()
    {
        // ...
    }
}
```

As indicated in its docblock, `authenticate()` must return an instance of `Zend_Auth_Result` (or of a class derived from `Zend_Auth_Result`). If for some reason performing an authentication query is impossible, `authenticate()` should throw an exception that derives from `Zend_Auth_Adapter_Exception`.

1.2. Results

`Zend_Auth` adapters return an instance of `Zend_Auth_Result` with `authenticate()` in order to represent the results of an authentication attempt. Adapters populate the `Zend_Auth_Result` object upon construction, so that the following four methods provide a basic set of user-facing operations that are common to the results of `Zend_Auth` adapters:

- `isValid()` - returns `TRUE` if and only if the result represents a successful authentication attempt
- `getCode()` - returns a `Zend_Auth_Result` constant identifier for determining the type of authentication failure or whether success has occurred. This may be used in situations where the developer wishes to distinguish among several authentication result types. This allows developers to maintain detailed authentication result statistics, for example. Another use of this feature is to provide specific, customized messages to users for usability reasons, though developers are encouraged to consider the risks of providing such detailed reasons to users, instead of a general authentication failure message. For more information, see the notes below.
- `getIdentity()` - returns the identity of the authentication attempt
- `getMessages()` - returns an array of messages regarding a failed authentication attempt

A developer may wish to branch based on the type of authentication result in order to perform more specific operations. Some operations developers might find useful are locking accounts after too many unsuccessful password attempts, flagging an IP address after too many nonexistent identities are attempted, and providing specific, customized authentication result messages to the user. The following result codes are available:

```
Zend_Auth_Result::SUCCESS
Zend_Auth_Result::FAILURE
Zend_Auth_Result::FAILURE_IDENTITY_NOT_FOUND
Zend_Auth_Result::FAILURE_IDENTITY_AMBIGUOUS
Zend_Auth_Result::FAILURE_CREDENTIAL_INVALID
Zend_Auth_Result::FAILURE_UNCATEGORIZED
```

The following example illustrates how a developer may branch on the result code:

```
// inside of AuthController / loginAction
```

```
$result = $this->_auth->authenticate($adapter);

switch ($result->getCode()) {

    case Zend_Auth_Result::FAILURE_IDENTITY_NOT_FOUND:
        /** do stuff for nonexistent identity **/
        break;

    case Zend_Auth_Result::FAILURE_CREDENTIAL_INVALID:
        /** do stuff for invalid credential **/
        break;

    case Zend_Auth_Result::SUCCESS:
        /** do stuff for successful authentication **/
        break;

    default:
        /** do stuff for other failure **/
        break;
}
```

1.3. Identity Persistence

Authenticating a request that includes authentication credentials is useful per se, but it is also important to support maintaining the authenticated identity without having to present the authentication credentials with each request.

HTTP is a stateless protocol, however, and techniques such as cookies and sessions have been developed in order to facilitate maintaining state across multiple requests in server-side web applications.

1.3.1. Default Persistence in the PHP Session

By default, `Zend_Auth` provides persistent storage of the identity from a successful authentication attempt using the PHP session. Upon a successful authentication attempt, `Zend_Auth::authenticate()` stores the identity from the authentication result into persistent storage. Unless configured otherwise, `Zend_Auth` uses a storage class named `Zend_Auth_Storage_Session`, which, in turn, uses `Zend_Session`. A custom class may instead be used by providing an object that implements `Zend_Auth_Storage_Interface` to `Zend_Auth::setStorage()`.

If automatic persistent storage of the identity is not appropriate for a particular use case, then developers may forgot using the `Zend_Auth` class altogether, instead using an adapter class directly.

Example 49. Modifying the Session Namespace

`Zend_Auth_Storage_Session` uses a session namespace of '`Zend_Auth`'. This namespace may be overridden by passing a different value to the constructor of `Zend_Auth_Storage_Session`, and this value is internally passed along to the constructor of `Zend_Session_Namespace`. This should occur before authentication is attempted, since `Zend_Auth::authenticate()` performs the automatic storage of the identity.

```
// Save a reference to the Singleton instance of Zend_Auth
$auth = Zend_Auth::getInstance();

// Use 'someNamespace' instead of 'Zend_Auth'
$auth->setStorage(new Zend_Auth_Storage_Session('someNamespace'));

/**
 * @todo Set up the auth adapter, $authAdapter
 */

// Authenticate, saving the result, and persisting the identity on
// success
$result = $auth->authenticate($authAdapter);
```

1.3.2. Implementing Customized Storage

Sometimes developers may need to use a different identity storage mechanism than that provided by `Zend_Auth_Storage_Session`. For such cases developers may simply implement `Zend_Auth_Storage_Interface` and supply an instance of the class to `Zend_Auth::setStorage()`.

```
class MyStorage implements Zend_Auth_Storage_Interface
{
    /**
     * Returns true if and only if storage is empty
     *
     * @throws Zend_Auth_Storage_Exception If it is impossible to
     *                                     determine whether storage
     *                                     is empty
     * @return boolean
     */
    public function isEmpty()
    {
        /**
         * @todo implementation
         */
    }

    /**
     * Returns the contents of storage
     *
     * Behavior is undefined when storage is empty.
     *
     * @throws Zend_Auth_Storage_Exception If reading contents from
     *                                     storage is impossible
     * @return mixed
     */
    public function read()
    {
        /**
         * @todo implementation
         */
    }

    /**
     * Writes $contents to storage
     *
     * @param  mixed $contents
     * @throws Zend_Auth_Storage_Exception If writing $contents to
     *                                     storage is impossible
     * @return void
     */
    public function write($contents)
    {
        /**
         * @todo implementation
         */
    }

    /**
     * Clears contents from storage
     *
     * @throws Zend_Auth_Storage_Exception If clearing contents from
     *                                     storage is impossible
     * @return void
     */
```

```
// Instruct Zend_Auth to use the custom storage class
Zend_Auth::getInstance()->setStorage(new MyStorage());

/**
 * @todo Set up the auth adapter, $authAdapter
 */

// Authenticate, saving the result, and persisting the identity on
// success
$result = Zend_Auth::getInstance()->authenticate($authAdapter);
```

1.4. Usage

There are two provided ways to use Zend_Auth adapters:

1. indirectly, through `Zend_Auth::authenticate()`

2. directly, through the adapter's `authenticate()` method

The following example illustrates how to use a Zend_Auth adapter indirectly, through the use of the Zend_Auth class:

```
// Get a reference to the singleton instance of Zend_Auth
$auth = Zend_Auth::getInstance();

// Set up the authentication adapter
$authAdapter = new MyAuthAdapter($username, $password);

// Attempt authentication, saving the result
$result = $auth->authenticate($authAdapter);

if (!$result->isValid()) {
    // Authentication failed; print the reasons why
    foreach ($result->getMessages() as $message) {
        echo "$message\n";
    }
} else {
    // Authentication succeeded; the identity ($username) is stored
    // in the session
    // $result->getIdentity() === $auth->getIdentity()
    // $result->getIdentity() === $username
}
```

Once authentication has been attempted in a request, as in the above example, it is a simple matter to check whether a successfully authenticated identity exists:

```
$auth = Zend_Auth::getInstance();
if ($auth->hasIdentity()) {
    // Identity exists; get it
    $identity = $auth->getIdentity();
}
```

To remove an identity from persistent storage, simply use the `clearIdentity()` method. This typically would be used for implementing an application "logout" operation:

```
Zend_Auth::getInstance()->clearIdentity();
```

When the automatic use of persistent storage is inappropriate for a particular use case, a developer may simply bypass the use of the Zend_Auth class, using an adapter class directly. Direct use of an adapter class involves configuring and preparing an adapter object and then calling its `authenticate()` method. Adapter-specific details are discussed in the documentation for each adapter. The following example directly utilizes `MyAuthAdapter`:

```
// Set up the authentication adapter
$authAdapter = new MyAuthAdapter($username, $password);
```

```php
// Attempt authentication, saving the result
$result = $authAdapter->authenticate();

if (!$result->isValid()) {
    // Authentication failed; print the reasons why
    foreach ($result->getMessages() as $message) {
        echo "$message\n";
    }
} else {
    // Authentication succeeded
    // $result->getIdentity() === $username
}
```

2. Database Table Authentication

2.1. Introduction

`Zend_Auth_Adapter_DbTable` provides the ability to authenticate against credentials stored in a database table. Because `Zend_Auth_Adapter_DbTable` requires an instance of `Zend_Db_Adapter_Abstract` to be passed to its constructor, each instance is bound to a particular database connection. Other configuration options may be set through the constructor and through instance methods, one for each option.

The available configuration options include:

- *tableName*: This is the name of the database table that contains the authentication credentials, and against which the database authentication query is performed.

- *identityColumn*: This is the name of the database table column used to represent the identity. The identity column must contain unique values, such as a username or e-mail address.

- *credentialColumn*: This is the name of the database table column used to represent the credential. Under a simple identity and password authentication scheme, the credential value corresponds to the password. See also the credentialTreatment option.

- *credentialTreatment*: In many cases, passwords and other sensitive data are encrypted, hashed, encoded, obscured, salted or otherwise treated through some function or algorithm. By specifying a parameterized treatment string with this method, such as '`MD5(?)`' or '`PASSWORD(?)`', a developer may apply such arbitrary SQL upon input credential data. Since these functions are specific to the underlying RDBMS, check the database manual for the availability of such functions for your database system.

As explained in the introduction, the `Zend_Auth_Adapter_DbTable` constructor requires an instance of `Zend_Db_Adapter_Abstract` that serves as the database connection to which the authentication adapter instance is bound. First, the database connection should

Example 31. Basic Usage

The following code creates an adapter for an in-memory database, creates a simple table

```
// Create an in-memory SQLite database connection
$dbAdapter = new Zend_Db_Adapter_Pdo_Sqlite(array('dbname' =>
                                                  ':memory:'));

// Build a simple table creation query
$sqlCreate = 'CREATE TABLE [users] ('
           . '[id] INTEGER  NOT NULL PRIMARY KEY, '
           . '[username] VARCHAR(50) UNIQUE NOT NULL, '
           . '[password] VARCHAR(32) NULL, '
           . '[real_name] VARCHAR(150) NULL)';

// Create the authentication credentials table
$dbAdapter->query($sqlCreate);

// Build a query to insert a row for which authentication may succeed
$sqlInsert = "INSERT INTO users (username, password, real_name) "
           . "VALUES ('my_username', 'my_password', 'My Real Name')";
```

With the database connection and table data available, an instance of

```
// Configure the instance with constructor parameters...
$authAdapter = new Zend_Auth_Adapter_DbTable(
    $dbAdapter,
    'users',
    'username',
    'password'
);

// ...or configure the instance with setter methods
$authAdapter = new Zend_Auth_Adapter_DbTable($dbAdapter);

$authAdapter
    ->setTableName('users')
    ->setIdentityColumn('username')
    ->setCredentialColumn('password')
;
```

At this point, the authentication adapter instance is ready to accept authentication queries.

```
// Set the input credential values (e.g., from a login form)
$authAdapter
    ->setIdentity('my_username')
    ->setCredential('my_password')
;
```

In addition to the availability of the `getIdentity()` method upon the authentication

```
// Print the identity
echo $result->getIdentity() . "\n\n";

// Print the result row
print_r($authAdapter->getResultRowObject());

/* Output:
my_username

Array
(
    [id] => 1
    [username] => my_username
    [password] => my_password
    [real_name] => My Real Name
)
```

Since the table row contains the credential value, it is important to secure the values against unintended access.

2.2. Advanced Usage: Persisting a DbTable Result Object

By default, `Zend_Auth_Adapter_DbTable` returns the identity supplied back to the auth object upon successful authentication. Another use case scenario, where developers want to store to the persistent storage mechanism of `Zend_Auth` an identity object containing other useful information, is solved by using the `getResultRowObject()` method to return a *stdClass* object. The following code snippet illustrates its use:

```
// authenticate with Zend_Auth_Adapter_DbTable
$result = $this->_auth->authenticate($adapter);

if ($result->isValid()) {
    // store the identity as an object where only the username and
    // real_name have been returned
    $storage = $this->_auth->getStorage();
    $storage->write($adapter->getResultRowObject(array(
        'username',
        'real_name',
    )));

    // store the identity as an object where the password column has
    // been omitted
    $storage->write($adapter->getResultRowObject(
        null,
        'password'
    ));

    /* ... */

} else {

    /* ... */

}
```

2.3. Advanced Usage By Example

While the primary purpose of `Zend_Auth` (and consequently `Zend_Auth_Adapter_DbTable`) is primarily *authentication* and not *authorization*, there are a few instances and problems that toe the line between which domain they fit within. Depending on how you've decided to explain your problem, it sometimes makes sense to solve what could look like an authorization problem within the authentication adapter.

With that disclaimer out of the way, `Zend_Auth_Adapter_DbTable` has some built in mechanisms that can be leveraged for additional checks at authentication time to solve some common user problems.

```
// The status field value of an account is not equal to "compromised"
$adapter = new Zend_Auth_Adapter_DbTable(
    $db,
    'users',
    'username',
    'password',
    'MD5(?) AND status != "compromised"'
);

// The active field value of an account is equal to "TRUE"
```

```
$adapter = new Zend_Auth_Adapter_DbTable(
    $db,
    'users',
    'username',
    'password',
    'MD5(?) AND active = "TRUE"'
```

Another scenario can be the implementation of a salting mechanism. Salting is a term referring to a technique which can highly improve your application's security. It's based on the idea that concatenating a random string to every password makes it impossible to accomplish a successful brute force attack on the database using pre-computed hash values from a dictionary.

Therefore, we need to modify our table to store our salt string:

```
$sqlAlter = "ALTER TABLE [users] "
          . "ADD COLUMN [password_salt] "
          . "AFTER [password]";
```

Here's a simple way to generate a salt string for every user at registration:

```
for ($i = 0; $i < 50; $i++) {
    $dynamicSalt .= chr(rand(33, 126));
```

And now let's build the adapter:

```
$adapter = new Zend_Auth_Adapter_DbTable(
    $db,
    'users',
    'username',
    'password',
    "MD5(CONCAT('"
    . Zend_Registry::get('staticSalt')
    . "', ?, password_salt))"
);
```

You can improve security even more by using a static salt value hard coded into your application. In the case that your database is compromised (e. g. by an SQL injection attack) but your web server is intact your data is still unusable for the attacker.

Another alternative is to use the `getDbSelect()` method of the `Zend_Auth_Adapter_DbTable` after the adapter has been constructed. This method will return the `Zend_Db_Select` object instance it will use to complete the `authenticate()` routine. It is important to note that this method will always return the same object regardless if `authenticate()` has been called or not. This object *will not* have any of the identity or credential information in it as those values are placed into the select object at `authenticate()` time.

An example of a situation where one might want to use the `getDbSelect()` method would check the status of a user, in other words to see if that user's account is enabled.

```
// Continuing with the example from above
$adapter = new Zend_Auth_Adapter_DbTable(
    $db,
    'users',
    'username',
```

```
    'password',
    'MD5(?)'
);

// get select object (by reference)
$select = $adapter->getDbSelect();
$select->where('active = "TRUE"');

// authenticate, this ensures that users.active = TRUE
$adapter->authenticate();
```

3. Digest Authentication

3.1. Introduction

Digest authentication is a method of HTTP authentication that improves upon Basic authentication by providing a way to authenticate without having to transmit the password in clear text across the network.

This adapter allows authentication against text files containing lines having the basic elements of Digest authentication:

- username, such as "*joe.user*"
- realm, such as "*Administrative Area*"
- MD5 hash of the username, realm, and password, separated by colons

The above elements are separated by colons, as in the following example (in which the password is "*somePassword*"):

```
someUser:Some Realm:fde17b91c3a510ecbaf7dbd37f59d4f8
```

3.2. Specifics

The digest authentication adapter, `Zend_Auth_Adapter_Digest`, requires several input parameters:

- filename - Filename against which authentication queries are performed
- realm - Digest authentication realm
- username - Digest authentication user
- password - Password for the user of the realm

These parameters must be set prior to calling `authenticate()`.

3.3. Identity

The digest authentication adapter returns a `Zend_Auth_Result` object, which has been populated with the identity as an array having keys of *realm* and *username*. The respective array values associated with these keys correspond to the values set before `authenticate()` is called.

```
$adapter = new Zend_Auth_Adapter_Digest($filename,
                                        $realm,
```

```
                                        $username,
                                        $password);

$result = $adapter->authenticate();

$identity = $result->getIdentity();

print_r($identity);

/*
Array
(
    [realm] => Some Realm
    [username] => someUser
)
*/
```

4. HTTP Authentication Adapter

4.1. Introduction

`Zend_Auth_Adapter_Http` provides a mostly-compliant implementation of RFC-2617, Basic and Digest HTTP Authentication. Digest authentication is a method of HTTP authentication that improves upon Basic authentication by providing a way to authenticate without having to transmit the password in clear text across the network.

Major Features:

- Supports both Basic and Digest authentication.
- Issues challenges in all supported schemes, so client can respond with any scheme it supports.
- Supports proxy authentication.
- Includes support for authenticating against text files and provides an interface for authenticating against other sources, such as databases.

There are a few notable features of RFC-2617 that are not implemented yet:

- Nonce tracking, which would allow for "stale" support, and increased replay attack protection.
- Authentication with integrity checking, or "auth-int".
- Authentication-Info HTTP header.

4.2. Design Overview

This adapter consists of two sub-components, the HTTP authentication class itself, and the so-called "Resolvers." The HTTP authentication class encapsulates the logic for carrying out both Basic and Digest authentication. It uses a Resolver to look up a client's identity in some data store (text file by default), and retrieve the credentials from the data store. The "resolved" credentials are then compared to the values submitted by the client to determine whether authentication is successful.

4.3. Configuration Options

The `Zend_Auth_Adapter_Http` class requires a configuration array passed to its constructor. There are several configuration options available, and some are required:

Table 9. Configuration Options

Option Name	Required	Description
accept_schemes	Yes	Determines which authentication schemes the adapter will accept from the client. Must be a space-separated list containing *'basic'* and/or *'digest'*.
realm	Yes	Sets the authentication realm; usernames should be unique within a given realm.
digest_domains	Yes, when accept_schemes contains *digest*	Space-separated list of URIs for which the same authentication information is valid. The URIs need not all point to the same server.
nonce_timeout	Yes, when accept_schemes contains *digest*	Sets the number of seconds for which the nonce is valid. See notes below.
proxy_auth	No	Disabled by default. Enable to perform Proxy authentication, instead of normal origin server authentication.

The current implementation of the nonce_timeout has some interesting side effects. This setting is supposed to determine the valid lifetime of a given nonce, or effectively how long a client's authentication information is accepted. Currently, if it's set to 3600 (for example), it will cause the adapter to prompt the client for new credentials every hour, on the hour. This will be resolved in a future release, once nonce tracking and stale support are implemented.

4.4. Resolvers

The resolver's job is to take a username and realm, and return some kind of credential value. Basic authentication expects to receive the Base64 encoded version of the user's password. Digest authentication expects to receive a hash of the user's username, the realm, and their password (each separated by colons). Currently, the only supported hash algorithm is MD5.

`Zend_Auth_Adapter_Http` relies on objects implementing `Zend_Auth_Adapter_Http_Resolver_Interface`. A text file resolver class is included with this adapter, but any other kind of resolver can be created simply by implementing the resolver interface.

4.4.1. File Resolver

The file resolver is a very simple class. It has a single property specifying a filename, which can also be passed to the constructor. Its `resolve()` method walks through the text file, searching for a line with a matching username and realm. The text file format similar to Apache htpasswd files:

```
<username>:<realm>:<credentials>\n
```

Each line consists of three fields - username, realm, and credentials - each separated by a colon. The credentials field is opaque to the file resolver; it simply returns that value as-is to the caller. Therefore, this same file format serves both Basic and Digest authentication. In Basic authentication, the credentials field should be written in clear text. In Digest authentication, it should be the MD5 hash described above.

There are two equally easy ways to create a File resolver:

```
$path     = 'files/passwd.txt';
$resolver = new Zend_Auth_Adapter_Http_Resolver_File($path);
```

or

```
$path     = 'files/passwd.txt';
$resolver = new Zend_Auth_Adapter_Http_Resolver_File();
$resolver->setFile($path);
```

If the given path is empty or not readable, an exception is thrown.

4.5. Basic Usage

First, set up an array with the required configuration values:

```
$config = array(
    'accept_schemes' => 'basic digest',
    'realm'          => 'My Web Site',
    'digest_domains' => '/members_only /my_account',
    'nonce_timeout'  => 3600,
);
```

This array will cause the adapter to accept either Basic or Digest authentication, and will require authenticated access to all the areas of the site under `/members_only` and `/my_account`. The realm value is usually displayed by the browser in the password dialog box. The nonce_timeout, of course, behaves as described above.

Next, create the `Zend_Auth_Adapter_Http` object:

```
$adapter = new Zend_Auth_Adapter_Http($config);
```

Since we're supporting both Basic and Digest authentication, we need two different resolver objects. Note that this could just as easily be two different classes:

```
$basicResolver = new Zend_Auth_Adapter_Http_Resolver_File();
$basicResolver->setFile('files/basicPasswd.txt');

$digestResolver = new Zend_Auth_Adapter_Http_Resolver_File();
$digestResolver->setFile('files/digestPasswd.txt');

$adapter->setBasicResolver($basicResolver);
$adapter->setDigestResolver($digestResolver);
```

Finally, we perform the authentication. The adapter needs a reference to both the Request and Response objects in order to do its job:

```
assert($request instanceof Zend_Controller_Request_Http);
assert($response instanceof Zend_Controller_Response_Http);
```

```
$adapter->setRequest($request);
$adapter->setResponse($response);

$result = $adapter->authenticate();
if (!$result->isValid()) {
    // Bad userame/password, or canceled password prompt
}
```

5. LDAP Authentication

5.1. Introduction

`Zend_Auth_Adapter_Ldap` supports web application authentication with LDAP services. Its features include username and domain name canonicalization, multi-domain authentication, and failover capabilities. It has been tested to work with Microsoft Active Directory and OpenLDAP, but it should also work with other LDAP service providers.

This documentation includes a guide on using `Zend_Auth_Adapter_Ldap`, an exploration of its API, an outline of the various available options, diagnostic information for troubleshooting authentication problems, and example options for both Active Directory and OpenLDAP servers.

5.2. Usage

To incorporate `Zend_Auth_Adapter_Ldap` authentication into your application quickly, even if you're not using `Zend_Controller`, the meat of your code should look something like the following:

```
$username = $this->_request->getParam('username');
$password = $this->_request->getParam('password');

$auth = Zend_Auth::getInstance();

$config = new Zend_Config_Ini('../application/config/config.ini',
                              'production');
$log_path = $config->ldap->log_path;
$options = $config->ldap->toArray();
unset($options['log_path']);

$adapter = new Zend_Auth_Adapter_Ldap($options, $username,
                                      $password);

$result = $auth->authenticate($adapter);

if ($log_path) {
    $messages = $result->getMessages();

    $logger = new Zend_Log();
    $logger->addWriter(new Zend_Log_Writer_Stream($log_path));
    $filter = new Zend_Log_Filter_Priority(Zend_Log::DEBUG);
    $logger->addFilter($filter);

    foreach ($messages as $i => $message) {
        if ($i-- > 1) { // $messages[2] and up are log messages
            $message = str_replace("\n", "\n  ", $message);
            $logger->log("Ldap: $i: $message", Zend_Log::DEBUG);
        }
```

```
    }
}
```

Of course, the logging code is optional, but it is highly recommended that you use a logger. `Zend_Auth_Adapter_Ldap` will record just about every bit of information anyone could want in `$messages` (more below), which is a nice feature in itself for something that has a history of being notoriously difficult to debug.

The `Zend_Config_Ini` code is used above to load the adapter options. It is also optional. A regular array would work equally well. The following is an example `application/config/config.ini` file that has options for two separate servers. With multiple sets of server options the adapter will try each, in order, until the credentials are successfully authenticated. The names of the servers (e.g., 'server1' and 'server2') are largely arbitrary. For details regarding the options array, see the *Server Options* section below. Note that `Zend_Config_Ini` requires that any values with "equals" characters (=) will need to be quoted (like the DNs shown below).

```
[production]

ldap.log_path = /tmp/ldap.log

; Typical options for OpenLDAP
ldap.server1.host = s0.foo.net
ldap.server1.accountDomainName = foo.net
ldap.server1.accountDomainNameShort = FOO
ldap.server1.accountCanonicalForm = 3
ldap.server1.username = "CN=user1,DC=foo,DC=net"
ldap.server1.password = pass1
ldap.server1.baseDn = "OU=Sales,DC=foo,DC=net"
ldap.server1.bindRequiresDn = true

; Typical options for Active Directory
ldap.server2.host = dc1.w.net
ldap.server2.useStartTls = true
ldap.server2.accountDomainName = w.net
ldap.server2.accountDomainNameShort = W
ldap.server2.accountCanonicalForm = 3
ldap.server2.baseDn = "CN=Users,DC=w,DC=net"
```

The above configuration will instruct `Zend_Auth_Adapter_Ldap` to attempt to authenticate users with the OpenLDAP server `s0.foo.net` first. If the authentication fails for any reason, the AD server `dc1.w.net` will be tried.

With servers in different domains, this configuration illustrates multi-domain authentication. You can also have multiple servers in the same domain to provide redundancy.

Note that in this case, even though OpenLDAP has no need for the short NetBIOS style domain name used by Windows, we provide it here for name canonicalization purposes (described in the *Username Canonicalization* section below).

5.3. The API

The `Zend_Auth_Adapter_Ldap` constructor accepts three parameters.

The `$options` parameter is required and must be an array containing one or more sets of options. Note that it is *an array of arrays* of `Zend_Ldap` options. Even if you will be using only one LDAP server, the options must still be within another array.

Below is `print_r()` output of an example options parameter containing two sets of server options for LDAP servers `s0.foo.net` and `dc1.w.net` (the same options as the above INI representation):

```
Array
(
    [server2] => Array
        (
            [host] => dc1.w.net
            [useStartTls] => 1
            [accountDomainName] => w.net
            [accountDomainNameShort] => W
            [accountCanonicalForm] => 3
            [baseDn] => CN=Users,DC=w,DC=net
        )

    [server1] => Array
        (
            [host] => s0.foo.net
            [accountDomainName] => foo.net
            [accountDomainNameShort] => FOO
            [accountCanonicalForm] => 3
            [username] => CN=user1,DC=foo,DC=net
            [password] => pass1
            [baseDn] => OU=Sales,DC=foo,DC=net
            [bindRequiresDn] => 1
        )

)
```

The information provided in each set of options above is different mainly because AD does not require a username be in DN form when binding (see the bindRequiresDn option in the *Server Options* section below), which means we can omit a number of options associated with retrieving the DN for a username being authenticated.

What is a Distinguished Name?

A DN or "distinguished name" is a string that represents the path to an object within the LDAP directory. Each comma-separated component is an attribute and value representing a node. The components are evaluated in reverse. For example, the user account *CN=Bob Carter,CN=Users,DC=w,DC=net* is located directly within the *CN=Users,DC=w,DC=net container*. This structure is best explored with an LDAP browser like the ADSI Edit MMC snap-in for Active Directory or phpLDAPadmin.

The names of servers (e.g. 'server1' and 'server2' shown above) are largely arbitrary, but for the sake of using `Zend_Config`, the identifiers should be present (as opposed to being numeric indexes) and should not contain any special characters used by the associated file formats (e.g. the '.' INI property separator, '&' for XML entity references, etc).

With multiple sets of server options, the adapter can authenticate users in multiple domains and provide failover so that if one server is not available, another will be queried.

The Gory Details: What Happens in the Authenticate Method?

When the `authenticate()` method is called, the adapter iterates over each set of server options, sets them on the internal `Zend_Ldap` instance, and

calls the `Zend_Ldap::bind()` method with the username and password being authenticated. The `Zend_Ldap` class checks to see if the username is qualified with a domain (e.g., has a domain component like `alice@foo.net` or `FOO\alice`). If a domain is present, but does not match either of the server's domain names (`foo.net` or FOO), a special exception is thrown and caught by `Zend_Auth_Adapter_Ldap` that causes that server to be ignored and the next set of server options is selected. If a domain *does* match, or if the user did not supply a qualified username, `Zend_Ldap` proceeds to try to bind with the supplied credentials. if the bind is not successful, `Zend_Ldap` throws a `Zend_Ldap_Exception` which is caught by `Zend_Auth_Adapter_Ldap` and the next set of server options is tried. If the bind is successful, the iteration stops, and the adapter's `authenticate()` method returns a successful result. If all server options have been tried without success, the authentication fails, and `authenticate()` returns a failure result with error messages from the last iteration.

The username and password parameters of the `Zend_Auth_Adapter_Ldap` constructor represent the credentials being authenticated (i.e., the credentials supplied by the user through your HTML login form). Alternatively, they may also be set with the `setUsername()` and `setPassword()` methods.

5.4. Server Options

Each set of server options *in the context of `Zend_Auth_Adapter_Ldap`* consists of the following options, which are passed, largely unmodified, to `Zend_Ldap::setOptions()`:

Table 10. Server Options

Name	Description
host	The hostname of LDAP server that these options represent. This option is required.
port	The port on which the LDAP server is listening. If *useSsl* is TRUE, the default port value is 636. If useSsl is FALSE, the default port value is 389.
useStartTls	Whether or not the LDAP client should use TLS (aka SSLv2) encrypted transport. A value of TRUE is strongly favored in production environments to prevent passwords from be transmitted in clear text. The default value is FALSE, as servers frequently require that a certificate be installed separately after installation. The useSsl and useStartTls options are mutually exclusive. The useStartTls option should be favored over useSsl but not all servers support this newer mechanism.
useSsl	Whether or not the LDAP client should use SSL encrypted transport. The useSsl and useStartTls options are mutually exclusive, but useStartTls should be favored if the server and LDAP client library support it. This value also changes the default port value (see port description above).

Name	Description
username	The DN of the account used to perform account DN lookups. LDAP servers that require the username to be in DN form when performing the "bind" require this option. Meaning, if bindRequiresDn is TRUE, this option is required. This account does not need to be a privileged account; an account with read-only access to objects under the baseDn is all that is necessary (and preferred based on the *Principle of Least Privilege*).
password	The password of the account used to perform account DN lookups. If this option is not supplied, the LDAP client will attempt an "anonymous bind" when performing account DN lookups.
bindRequiresDn	Some LDAP servers require that the username used to bind be in DN form like *CN=Alice Baker,OU=Sales,DC=foo,DC=net* (basically all servers *except* AD). If this option is TRUE, this instructs `Zend_Ldap` to automatically retrieve the DN corresponding to the username being authenticated, if it is not already in DN form, and then re-bind with the proper DN. The default value is FALSE. Currently only Microsoft Active Directory Server (ADS) is known *not* to require usernames to be in DN form when binding, and therefore this option may be FALSE with AD (and it should be, as retrieving the DN requires an extra round trip to the server). Otherwise, this option must be set to TRUE (e.g. for OpenLDAP). This option also controls the default acountFilterFormat used when searching for accounts. See the accountFilterFormat option.
baseDn	The DN under which all accounts being authenticated are located. This option is required. if you are uncertain about the correct baseDn value, it should be sufficient to derive it from the user's DNS domain using *DC=* components. For example, if the user's principal name is `alice@foo.net`, a baseDn of *DC=foo,DC=net* should work. A more precise location (e.g., *OU=Sales,DC=foo,DC=net*) will be more efficient, however.
accountCanonicalForm	A value of 2, 3 or 4 indicating the form to which account names should be canonicalized after successful authentication. Values are as follows: 2 for traditional username style names (e.g., *alice*), 3 for backslash-style

Name	Description
	names (e.g., `FOO\alice`) or 4 for principal style usernames (e.g., `alice@foo.net`). The default value is 4 (e.g., `alice@foo.net`). For example, with a value of 3, the identity returned by `Zend_Auth_Result::getIdentity()` (and `Zend_Auth::getIdentity()`, if `Zend_Auth` was used) will always be `FOO\alice`, regardless of what form Alice supplied, whether it be *alice*, `alice@foo.net`, `FOO\alice`, `FoO\aLicE`, `foo.net\alice`, etc. See the *Account Name Canonicalization* section in the `Zend_Ldap` documentation for details. Note that when using multiple sets of server options it is recommended, but not required, that the same accountCanonicalForm be used with all server options so that the resulting usernames are always canonicalized to the same form (e.g., if you canonicalize to `EXAMPLE\username` with an AD server but to `username@example.com` with an OpenLDAP server, that may be awkward for the application's high-level logic).
accountDomainName	The FQDN domain name for which the target LDAP server is an authority (e.g., `example.com`). This option is used to canonicalize names so that the username supplied by the user can be converted as necessary for binding. It is also used to determine if the server is an authority for the supplied username (e.g., if accountDomainName is `foo.net` and the user supplies `bob@bar.net`, the server will not be queried, and a failure will result). This option is not required, but if it is not supplied, usernames in principal name form (e.g., `alice@foo.net`) are not supported. It is strongly recommended that you supply this option, as there are many use-cases that require generating the principal name form.
accountDomainNameShort	The 'short' domain for which the target LDAP server is an authority (e.g., FOO). Note that there is a 1:1 mapping between the accountDomainName and accountDomainNameShort. This option should be used to specify the NetBIOS domain name for Windows networks, but may also be used by non-AD servers (e.g., for consistency when multiple sets of server options with the backslash style accountCanonicalForm). This option is not required but if it is not

Name	Description
	supplied, usernames in backslash form (e.g., `FOO\alice`) are not supported.
accountFilterFormat	The LDAP search filter used to search for accounts. This string is a `printf()`-style expression that must contain one '*%s*' to accomodate the username. The default value is '*(&(objectClass=user)(sAMAccountName=%s))*', unless bindRequiresDn is set to TRUE, in which case the default is '*(&(objectClass=posixAccount)(uid=%s))*'. For example, if for some reason you wanted to use *bindRequiresDn = true* with AD you would need to set *accountFilterFormat = '(&(objectClass=user)(sAMAccountName=%s))*'.
optReferrals	If set to TRUE, this option indicates to the LDAP client that referrals should be followed. The default value is FALSE.

If you enable *useStartTls = TRUE* or *useSsl = TRUE* you may find that the LDAP client generates an error claiming that it cannot validate the server's certificate. Assuming the PHP LDAP extension is ultimately linked to the OpenLDAP client libraries, to resolve this issue you can set "**TLS_REQCERT never**" in the OpenLDAP client `ldap.conf` (and restart the web server) to indicate to the OpenLDAP client library that you trust the server. Alternatively, if you are concerned that the server could be spoofed, you can export the LDAP server's root certificate and put it on the web server so that the OpenLDAP client can validate the server's identity.

5.5. Collecting Debugging Messages

`Zend_Auth_Adapter_Ldap` collects debugging information within its `authenticate()` method. This information is stored in the `Zend_Auth_Result` object as messages. The array returned by `Zend_Auth_Result::getMessages()` is described as follows

Table 11. Debugging Messages

Messages Array Index	Description
Index 0	A generic, user-friendly message that is suitable for displaying to users (e.g., "Invalid credentials"). If the authentication is successful, this string is empty.
Index 1	A more detailed error message that is not suitable to be displayed to users but should be logged for the benefit of server operators. If the authentication is successful, this string is empty.
Indexes 2 and higher	All log messages in order starting at index 2.

In practice, index 0 should be displayed to the user (e.g., using the FlashMessenger helper), index 1 should be logged and, if debugging information is being collected, indexes 2 and higher could be logged as well (although the final message always includes the string from index 1).

5.6. Common Options for Specific Servers

5.6.1. Options for Active Directory

For ADS, the following options are noteworthy:

Table 12. Options for Active Directory

Name	Additional Notes
host	As with all servers, this option is required.
useStartTls	For the sake of security, this should be `TRUE` if the server has the necessary certificate installed.
useSsl	Possibly used as an alternative to *useStartTls* (see above).
baseDn	As with all servers, this option is required. By default AD places all user accounts under the *Users* container (e.g., *CN=Users,DC=foo,DC=net*), but the default is not common in larger organizations. Ask your AD administrator what the best DN for accounts for your application would be.
accountCanonicalForm	You almost certainly want this to be 3 for backslash style names (e.g., `FOO\alice`), which are most familiar to Windows users. You should *not* use the unqualified form 2 (e.g., *alice*), as this may grant access to your application to users with the same username in other trusted domains (e.g., `BAR\alice` and `FOO\alice` will be treated as the same user). (See also note below.)
accountDomainName	This is required with AD unless accountCanonicalForm 2 is used, which, again, is discouraged.
accountDomainNameShort	The NetBIOS name of the domain that users are in and for which the AD server is an authority. This is required if the backslash style accountCanonicalForm is used.

Technically there should be no danger of accidental cross-domain authentication with the current `Zend_Auth_Adapter_Ldap` implementation, since server domains are explicitly checked, but this may not be true of a future implementation that discovers the domain at runtime, or if an alternative adapter is used (e.g., Kerberos). In general, account name ambiguity is known to be the source of security issues, so always try to use qualified account names.

5.6.2. Options for OpenLDAP

For OpenLDAP or a generic LDAP server using a typical posixAccount style schema, the following options are noteworthy:

Table 13. Options for OpenLDAP

Name	Additional Notes
host	As with all servers, this option is required.
useStartTls	For the sake of security, this should be TRUE if the server has the necessary certificate installed.
useSsl	Possibly used as an alternative to useStartTls (see above).
username	Required and must be a DN, as OpenLDAP requires that usernames be in DN form when performing a bind. Try to use an unprivileged account.
password	The password corresponding to the username above, but this may be omitted if the LDAP server permits an anonymous binding to query user accounts.
bindRequiresDn	Required and must be TRUE, as OpenLDAP requires that usernames be in DN form when performing a bind.
baseDn	As with all servers, this option is required and indicates the DN under which all accounts being authenticated are located.
accountCanonicalForm	Optional, but the default value is 4 (principal style names like `alice@foo.net`), which may not be ideal if your users are used to backslash style names (e.g., `FOO\alice`). For backslash style names use value 3.
accountDomainName	Required unless you're using accountCanonicalForm 2, which is not recommended.
accountDomainNameShort	If AD is not also being used, this value is not required. Otherwise, if accountCanonicalForm 3 is used, this option is required and should be a short name that corresponds adequately to the accountDomainName (e.g., if your accountDomainName is `foo.net`, a good accountDomainNameShort value might be FOO).

6. Open ID Authentication

6.1. Introduction

The `Zend_Auth_Adapter_OpenId` adapter can be used to authenticate users using remote OpenID servers. This authentication method assumes that the user submits only their OpenID

identity to the web application. They are then redirected to their OpenID provider to prove identity ownership using a password or some other method. This password is never provided to the web application.

The OpenID identity is just a URI that points to a web site with information about a user, along with special tags that describes which server to use and which identity to submit there. You can read more about OpenID at the OpenID official site.

The `Zend_Auth_Adapter_OpenId` class wraps the `Zend_OpenId_Consumer` component, which implements the OpenID authentication protocol itself.

`Zend_OpenId` takes advantage of the GMP extension, where available. Consider enabling the GMP extension for better performance when using `Zend_Auth_Adapter_OpenId`.

6.2. Specifics

As is the case for all `Zend_Auth` adapters, the `Zend_Auth_Adapter_OpenId` class implements `Zend_Auth_Adapter_Interface`, which defines one method: `authenticate()`. This method performs the authentication itself, but the object must be prepared prior to calling it. Such adapter preparation includes setting up the OpenID identity and some other `Zend_OpenId` specific options.

However, as opposed to other `Zend_Auth` adapters, `Zend_Auth_Adapter_OpenId` performs authentication on an external server in two separate HTTP requests. So the `Zend_Auth_Adapter_OpenId::authenticate()` method must be called twice. On the first invocation the method won't return, but will redirect the user to their OpenID server. Then after the user is authenticated on the remote server, they will be redirected back and the script for this second request must call `Zend_Auth_Adapter_OpenId::authenticate()` again to verify the signature which comes with the redirected request from the server to complete the authentication process. On this second invocation, the method will return the `Zend_Auth_Result` object as expected.

The following example shows the usage of `Zend_Auth_Adapter_OpenId`. As previously mentioned, the `Zend_Auth_Adapter_OpenId::authenticate()` must be called two times. The first time is after the user submits the HTML form with the `$_POST['openid_action']` set to *"login"*, and the second time is after the HTTP redirection from OpenID server with `$_GET['openid_mode']` or `$_POST['openid_mode']` set.

```
<?php
$status = "";
$auth = Zend_Auth::getInstance();
if ((isset($_POST['openid_action']) &&
     $_POST['openid_action'] == "login" &&
     !empty($_POST['openid_identifier'])) ||
    isset($_GET['openid_mode']) ||
    isset($_POST['openid_mode'])) {
    $result = $auth->authenticate(
        new Zend_Auth_Adapter_OpenId(@$_POST['openid_identifier']));
    if ($result->isValid()) {
        $status = "You are logged in as "
                . $auth->getIdentity()
                . "<br>\n";
    } else {
        $auth->clearIdentity();
```

```
        foreach ($result->getMessages() as $message) {
            $status .= "$message<br>\n";
        }
    }
} else if ($auth->hasIdentity()) {
    if (isset($_POST['openid_action']) &&
        $_POST['openid_action'] == "logout") {
        $auth->clearIdentity();
    } else {
        $status = "You are logged in as "
                . $auth->getIdentity()
                . "<br>\n";
    }
}
?>
<html><body>
<?php echo htmlspecialchars($status);?>
<form method="post"><fieldset>
<legend>OpenID Login</legend>
<input type="text" name="openid_identifier" value="">
<input type="submit" name="openid_action" value="login">
<input type="submit" name="openid_action" value="logout">
</fieldset></form></body></html>
*/
```

You may customize the OpenID authentication process in several way. You can, for example, receive the redirect from the OpenID server on a separate page, specifying the "root" of web site and using a custom `Zend_OpenId_Consumer_Storage` or a custom `Zend_Controller_Response`. You may also use the Simple Registration Extension to retrieve information about user from the OpenID server. All of these possibilities are described in more detail in the `Zend_OpenId_Consumer` chapter.

Zend_Barcode

1. Introduction

`Zend_Barcode` provides a generic way to generate barcodes. The `Zend_Barcode` component is divided into two subcomponents: barcode objects and renderers. Objects allow you to create barcodes independently of the renderer. Renderer allow you to draw barcodes based on the support required.

2. Barcode creation using Zend_Barcode class

2.1. Using Zend_Barcode::factory

`Zend_Barcode` uses a factory method to create an instance of a renderer that extends `Zend_Barcode_Renderer_RendererAbstract`. The factory method accepts five arguments.

1. The name of the barcode format (e.g., "code39") (required)
2. The name of the renderer (e.g., "image") (required)
3. Options to pass to the barcode object (an array or `Zend_Config` object) (optional)
4. Options to pass to the renderer object (an array or `Zend_Config` object) (optional)
5. Boolean to indicate whether or not to automatically render errors. If an exception occurs, the provided barcode object will be replaced with an Error representation (optional default `TRUE`)

Example 52. Getting a Renderer with Zend_Barcode::factory()

`Zend_Barcode::factory()` instantiates barcode objects and renderers and ties them together. In this first example, we will use the *Code39* barcode type together with the *Image* renderer.

```
// Only the text to draw is required
$barcodeOptions = array('text' => 'ZEND-FRAMEWORK');

// No required options
$rendererOptions = array();
$renderer = Zend_Barcode::factory(
    'code39', 'image', $barcodeOptions, $rendererOptions
);
```

Example 53. Using Zend_Barcode::factory() with Zend_Config objects

You may pass a `Zend_Config` object to the factory in order to create the necessary objects. The following example is functionally equivalent to the previous.

```
// Using only one Zend_Config object
$config = new Zend_Config(array(
    'barcode'        => 'code39',
    'barcodeParams'  => array('text' => 'ZEND-FRAMEWORK'),
    'renderer'       => 'image',
    'rendererParams' => array('imageType' => 'gif'),
));

$renderer = Zend_Barcode::factory($config);
```

2.2. Drawing a barcode

When you *draw* the barcode, you retrieve the resource in which the barcode is drawn. To draw a barcode, you can call the `draw()` of the renderer, or simply use the proxy method provided by `Zend_Barcode`.

Example 54. Drawing a barcode with the renderer object

```
// Only the text to draw is required
$barcodeOptions = array('text' => 'ZEND-FRAMEWORK');

// No required options
$rendererOptions = array();

// Draw the barcode in a new image,
$imageResource = Zend_Barcode::factory(
    'code39', 'image', $barcodeOptions, $rendererOptions
)->draw();
```

Example 55. Drawing a barcode with Zend_Barcode::draw()

```
// Only the text to draw is required
$barcodeOptions = array('text' => 'ZEND-FRAMEWORK');

// No required options
$rendererOptions = array();

// Draw the barcode in a new image,
$imageResource = Zend_Barcode::draw(
    'code39', 'image', $barcodeOptions, $rendererOptions
);
```

2.3. Renderering a barcode

When you render a barcode, you draw the barcode, you send the headers and you send the resource (e.g. to a browser). To render a barcode, you can call the `render()` method of the renderer or simply use the proxy method provided by `Zend_Barcode`.

Example 56. Renderering a barcode with the renderer object

```
// Only the text to draw is required
$barcodeOptions = array('text' => 'ZEND-FRAMEWORK');

// No required options
$rendererOptions = array();

// Draw the barcode in a new image,
// send the headers and the image
Zend_Barcode::factory(
    'code39', 'image', $barcodeOptions, $rendererOptions
)->render();
```

This will generate this barcode:

Example 57. Renderering a barcode with Zend_Barcode::render()

```
// Only the text to draw is required
$barcodeOptions = array('text' => 'ZEND-FRAMEWORK');

// No required options
$rendererOptions = array();

// Draw the barcode in a new image,
// send the headers and the image
Zend_Barcode::render(
    'code39', 'image', $barcodeOptions, $rendererOptions
);
```

This will generate the same barcode as the previous example.

3. Zend_Barcode Objects

Barcode objects allow you to generate barcodes independently of the rendering support. After generation, you can retrieve the barcode as an array of drawing instructions that you can provide to a renderer.

Objects have a large number of options. Most of them are common to all objects. These options can be set in four ways:

- As an array or a `Zend_Config` object passed to the constructor.
- As an array passed to the `setOptions()` method.
- As a `Zend_Config` object passed to the `setConfig()` method.
- Via individual setters for each configuration type.

Example 58. Different ways to parameterize a barcode object

```
$options = array('text' => 'ZEND-FRAMEWORK', 'barHeight' => 40);

// Case 1: constructor
$barcode = new Zend_Barcode_Object_Code39($options);

// Case 2: setOptions()
$barcode = new Zend_Barcode_Object_Code39();
$barcode->setOptions($options);

// Case 3: setConfig()
$config  = new Zend_Config($options);
$barcode = new Zend_Barcode_Object_Code39();
$barcode->setConfig($config);

// Case 4: individual setters
$barcode = new Zend_Barcode_Object_Code39();
$barcode->setText('ZEND-FRAMEWORK')
        ->setBarHeight(40);
```

3.1. Common Options

In the following list, the values have no units; we will use the term "unit." For example, the default value of the "thin bar" is "1 unit". The real units depend on the rendering support (see the renderers documentation for more information). Setters are each named by uppercasing the initial letter of the option and prefixing the name with "set" (e.g. "barHeight" becomes "setBarHeight"). All options have a corresponding getter prefixed with "get" (e.g. "getBarHeight"). Available options are:

Table 14. Common Options

Option	Data Type	Default Value	Description
barcodeNamespace	String	`Zend_Barcode_Object`	Namespace of the barcode; for example, if you need to extend the embedding objects
barHeight	Integer	50	Height of the bars
barThickWidth	Integer	3	Width of the thick bar
barThinWidth	Integer	1	Width of the thin
factor	Integer	1	Factor by which to multiply bar widths and font sizes
foreColor	Integer	0 (black)	Color of the bar and the text. Could be provided as an integer or as a HTML value (e.g. "#333333")
backgroundColor	Integer or String	16777125 (white)	Color of the background. Could be provided as an integer or as a HTML value (e.g. "#333333")

Option	Data Type	Default Value	Description
reverseColor	Boolean	FALSE	Allow switching the color of the bar and the background
orientation	Integer	0	Orientation of the barcode
font	String or Integer	NULL	Font path to a TTF font or a number between 1 and 5 if using image generation with GD (internal fonts)
fontSize	Integer	10	Size of the font (not applicable with numeric fonts)
withBorder	Boolean	FALSE	Draw a border around the barcode and the quiet zones
drawText	Boolean	TRUE	Set if the text is displayed below the barcode
stretchText	Boolean	FALSE	Specify if the text is stretched all along the barcode
withChecksum	Boolean	FALSE	Indicate whether or not the checksum is automatically added to the barcode
withChecksumInText	Boolean	FALSE	Indicate whether or not the checksum is displayed in the textual representation
text	String	NULL	The text to represent as a barcode

3.1.1. Particular case of static setBarcodeFont()

You can set a commont font for all your objects by using the static method `Zend_Barcode_Object::setBarcodeFont()`. This value can be always be overridden for individual objects by using the `setFont()` method.

```
// In your bootstrap:
Zend_Barcode_Object::setBarcodeFont('my_font.ttf');

// Later in your code:
Zend_Barcode::render(
    'code39',
    'pdf',
    array('text' => 'ZEND-FRAMEWORK')
); // will use 'my_font.ttf'

// or:
```

```
Zend_Barcode::render(
    'code39',
    'image',
    array(
        'text' => 'ZEND-FRAMEWORK',
        'font' => 3
    )
); // will use the 3rd GD internal font
```

3.2. Common Additional Getters

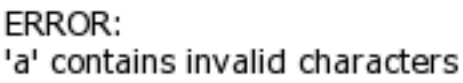
Table 15. Common Getters

Getter	Data Type	Description
`getType()`	String	Return the name of the barcode class without the namespace (e.g. `Zend_Barcode_Object_Code39` returns simply "code39")
`getRawText()`	String	Return the original text provided to the object
`getTextToDisplay()`	String	Return the text to display, including, if activated, the checksum value
`getQuietZone()`	Integer	Return the size of the space needed before and after the barcode without any drawing
`getInstructions()`	Array	Return drawing instructions as an array.
`getHeight($recalculate = false)`	Integer	Return the height of the barcode calculated after possible rotation
`getWidth($recalculate = false)`	Integer	Return the width of the barcode calculated after possible rotation
`getOffsetTop($recalculate = false)`	Integer	Return the position of the top of the barcode calculated after possible rotation
`getOffsetLeft($recalculate = false)`	Integer	Return the position of the left of the barcode calculated after possible rotation

3.3. Description of shipped barcodes

You will find below detailed information about all barcode types shipped by default with Zend Framework.

3.3.1. Zend_Barcode_Object_Error



This barcode is a special case. It is internally used to automatically render an exception caught by the `Zend_Barcode` component.

3.3.2. Zend_Barcode_Object_Code25

- *Name:* Code 25 (or Code 2 of 5 or Code 25 Industrial)
- *Allowed characters:* '0123456789'
- *Checksum:* optional (modulo 10)
- *Length:* variable

There are no particular options for this barcode.

3.3.3. Zend_Barcode_Object_Code25interleaved

This barcode extends `Zend_Barcode_Object_Code25` (Code 2 of 5), and has the same particulars and options, and adds the following:

- *Name:* Code 2 of 5 Interleaved
- *Allowed characters:* '0123456789'
- *Checksum:* optional (modulo 10)
- *Length:* variable (always even number of characters)

Available options include:

Table 16. Zend_Barcode_Object_Code25interleaved Options

Option	Data Type	Default Value	Description
withBearerBars	Boolean	FALSE	Draw a thick bar at the top and the bottom of the barcode.

If the number of characters is not even, `Zend_Barcode_Object_Code25interleaved` will automatically prepend the missing zero to the barcode text.

3.3.4. Zend_Barcode_Object_Ean2

This barcode extends `Zend_Barcode_Object_Ean5` (EAN 5), and has the same particulars and options, and adds the following:

- *Name:* EAN-2
- *Allowed characters:* '0123456789'
- *Checksum:* only use internally but not displayed
- *Length:* 2 characters

There are no particular options for this barcode.

If the number of characters is lower than 2, `Zend_Barcode_Object_Ean2` will automatically prepend the missing zero to the barcode text.

3.3.5. Zend_Barcode_Object_Ean5

This barcode extends `Zend_Barcode_Object_Ean13` (EAN 13), and has the same particulars and options, and adds the following:

- *Name:* EAN-5
- *Allowed characters:* '0123456789'
- *Checksum:* only use internally but not displayed
- *Length:* 5 characters

There are no particular options for this barcode.

If the number of characters is lower than 5, `Zend_Barcode_Object_Ean5` will automatically prepend the missing zero to the barcode text.

3.3.6. Zend_Barcode_Object_Ean8

This barcode extends `Zend_Barcode_Object_Ean13` (EAN 13), and has the same particulars and options, and adds the following:

- *Name:* EAN-8
- *Allowed characters:* '0123456789'
- *Checksum:* mandatory (modulo 10)
- *Length:* 8 characters (including checksum)

There are no particular options for this barcode.

If the number of characters is lower than 8, `Zend_Barcode_Object_Ean8` will automatically prepend the missing zero to the barcode text.

3.3.7. Zend_Barcode_Object_Ean13

- *Name:* EAN-13
- *Allowed characters:* '0123456789'
- *Checksum:* mandatory (modulo 10)
- *Length:* 13 characters (including checksum)

There are no particular options for this barcode.

If the number of characters is lower than 13, `Zend_Barcode_Object_Ean13` will automatically prepend the missing zero to the barcode text.

3.3.8. Zend_Barcode_Object_Code39

- *Name:* Code 39

- *Allowed characters:* '0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ -.$/+%'
- *Checksum:* optional (modulo 43)
- *Length:* variable

`Zend_Barcode_Object_Code39` will automatically add the start and stop characters ('*') for you.

There are no particular options for this barcode.

3.3.9. Zend_Barcode_Object_Identcode

This barcode extends `Zend_Barcode_Object_Code25interleaved` (Code 2 of 5 Interleaved), and inherits some of its capabilities; it also has a few particulars of its own.

- *Name:* Identcode (Deutsche Post Identcode)
- *Allowed characters:* '0123456789'
- *Checksum:* mandatory (modulo 10 different from Code25)
- *Length:* 12 characters (including checksum)

There are no particular options for this barcode.

If the number of characters is lower than 12, `Zend_Barcode_Object_Identcode` will automatically prepend missing zeros to the barcode text.

3.3.10. Zend_Barcode_Object_Itf14

This barcode extends `Zend_Barcode_Object_Code25interleaved` (Code 2 of 5 Interleaved), and inherits some of its capabilities; it also has a few particulars of its own.

- *Name:* ITF-14
- *Allowed characters:* '0123456789'
- *Checksum:* mandatory (modulo 10)
- *Length:* 14 characters (including checksum)

There are no particular options for this barcode.

If the number of characters is lower than 14, `Zend_Barcode_Object_Itf14` will automatically prepend missing zeros to the barcode text.

3.3.11. Zend_Barcode_Object_Leitcode

This barcode extends `Zend_Barcode_Object_Identcode` (Deutsche Post Identcode), and inherits some of its capabilities; it also has a few particulars of its own.

- *Name:* Leitcode (Deutsche Post Leitcode)
- *Allowed characters:* '0123456789'
- *Checksum:* mandatory (modulo 10 different from Code25)
- *Length:* 14 characters (including checksum)

There are no particular options for this barcode.

If the number of characters is lower than 14, `Zend_Barcode_Object_Leitcode` will automatically prepend missing zeros to the barcode text.

3.3.12. Zend_Barcode_Object_Planet

- *Name:* Planet (PostaL Alpha Numeric Encoding Technique)
- *Allowed characters:* '0123456789'
- *Checksum:* mandatory (modulo 10)
- *Length:* 12 or 14 characters (including checksum)

There are no particular options for this barcode.

3.3.13. Zend_Barcode_Object_Postnet

- *Name:* Postnet (POSTal Numeric Encoding Technique)
- *Allowed characters:* '0123456789'
- *Checksum:* mandatory (modulo 10)
- *Length:* 6, 7, 10 or 12 characters (including checksum)

There are no particular options for this barcode.

3.3.14. Zend_Barcode_Object_Royalmail

- *Name:* Royal Mail or RM4SCC (Royal Mail 4-State Customer Code)
- *Allowed characters:* '0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ'
- *Checksum:* mandatory
- *Length:* variable

There are no particular options for this barcode.

3.3.15. Zend_Barcode_Object_Upca

This barcode extends `Zend_Barcode_Object_Ean13` (EAN-13), and inherits some of its capabilities; it also has a few particulars of its own.

- *Name:* UPC-A (Universal Product Code)
- *Allowed characters:* '0123456789'
- *Checksum:* mandatory (modulo 10)
- *Length:* 12 characters (including checksum)

There are no particular options for this barcode.

If the number of characters is lower than 12, `Zend_Barcode_Object_Upca` will automatically prepend missing zeros to the barcode text.

3.3.16. Zend_Barcode_Object_Upce

This barcode extends `Zend_Barcode_Object_Upca` (UPC-A), and inherits some of its capabilities; it also has a few particulars of its own. The first character of the text to encode is the system (0 or 1).

- *Name:* UPC-E (Universal Product Code)
- *Allowed characters:* '0123456789'
- *Checksum:* mandatory (modulo 10)

- *Length:* 8 characters (including checksum)

There are no particular options for this barcode.

If the number of characters is lower than 8, `Zend_Barcode_Object_Upce` will automatically prepend missing zeros to the barcode text.

If the first character of the text to encode is not 0 or 1, `Zend_Barcode_Object_Upce` will automatically replace it by 0.

4. Zend_Barcode Renderers

Renderers have some common options. These options can be set in four ways:

- As an array or a `Zend_Config` object passed to the constructor.
- As an array passed to the `setOptions()` method.
- As a `Zend_Config` object passed to the `setConfig()` method.
- As discrete values passed to individual setters.

Example 59. Different ways to parameterize a renderer object

```
$options = array('topOffset' => 10);

// Case 1
$renderer = new Zend_Barcode_Renderer_Pdf($options);

// Case 2
$renderer = new Zend_Barcode_Renderer_Pdf();
$renderer->setOptions($options);

// Case 3
$config   = new Zend_Config($options);
$renderer = new Zend_Barcode_Renderer_Pdf();
$renderer->setConfig($config);

// Case 4
$renderer = new Zend_Barcode_Renderer_Pdf();
$renderer->setTopOffset(10);
```

4.1. Common Options

In the following list, the values have no unit; we will use the term "unit." For example, the default value of the "thin bar" is "1 unit." The real units depend on the rendering support. The individual setters are obtained by uppercasing the initial letter of the option and prefixing the name with "set" (e.g. "barHeight" => "setBarHeight"). All options have a correspondant getter prefixed with "get" (e.g. "getBarHeight"). Available options are:

Table 17. Common Options

Option	Data Type	Default Value	Description
rendererNamespace	String	`Zend_Barcode_Rende`	Namespace of the renderer; for example,

Option	Data Type	Default Value	Description
			if you need to extend the renderers
horizontalPosition	String	"left"	Can be "left", "center" or "right". Can be useful with PDF or if the `setWidth()` method is used with an image renderer.
verticalPosition	String	"top"	Can be "top", "middle" or "bottom". Can be useful with PDF or if the `setHeight()` method is used with an image renderer.
leftOffset	Integer	0	Top position of the barcode inside the renderer. If used, this value will override the "horizontalPosition" option.
topOffset	Integer	0	Top position of the barcode inside the renderer. If used, this value will override the "verticalPosition" option.
automaticRenderError	Boolean	TRUE	Whether or not to automatically render errors. If an exception occurs, the provided barcode object will be replaced with an Error representation. Note that some errors (or exceptions) can not be rendered.
moduleSize	Float	1	Size of a rendering module in the support.
barcode	`Zend_Barcode_Object`	`NULL`	The barcode object to render.

An additional getter exists: `getType()`. It returns the name of the renderer class without the namespace (e.g. `Zend_Barcode_Renderer_Image` returns "image").

4.2. Zend_Barcode_Renderer_Image

The Image renderer will draw the instruction list of the barcode object in an image resource. The component requires the GD extension. The default width of a module is 1 pixel.

Available option are:

Table 18. Zend_Barcode_Renderer_Image Options

Option	Data Type	Default Value	Description
height	Integer	0	Allow you to specify the height of the result image. If "0", the height will be calculated by the barcode object.
width	Integer	0	Allow you to specify the width of the result image. If "0", the width will be calculated by the barcode object.
imageType	String	"png"	Specify the image format. Can be "png", "jpeg", "jpg" or "gif".

4.3. Zend_Barcode_Renderer_Pdf

The PDF renderer will draw the instruction list of the barcode object in a PDF document. The default width of a module is 0.5 point.

There are no particular options for this renderer.

Zend_Cache

1. Introduction

`Zend_Cache` provides a generic way to cache any data.

Caching in Zend Framework is operated by frontends while cache records are stored through backend adapters (*File*, *Sqlite*, *Memcache*...) through a flexible system of IDs and tags. Using those, it is easy to delete specific types of records afterwards (for example: "delete all cache records marked with a given tag").

The core of the module (`Zend_Cache_Core`) is generic, flexible and configurable. Yet, for your specific needs there are cache frontends that extend `Zend_Cache_Core` for convenience: *Output*, *File*, *Function* and *Class*.

Example 60. Getting a Frontend with Zend_Cache::factory()

`Zend_Cache::factory()` instantiates correct objects and ties them together. In this first example, we will use *Core* frontend together with *File* backend.

```
$frontendOptions = array(
   'lifetime' => 7200, // cache lifetime of 2 hours
   'automatic_serialization' => true
);

$backendOptions = array(
    'cache_dir' => './tmp/' // Directory where to put the cache files
);

// getting a Zend_Cache_Core object
$cache = Zend_Cache::factory('Core',
                             'File',
                             $frontendOptions,
                             $backendOptions);
```

Frontends and Backends Consisting of Multiple Words

Some frontends and backends are named using multiple words, such as 'ZendPlatform'. When specifying them to the factory, separate them using a word separator, such as a space (' '), hyphen ('-'), or period ('.').

Example 61. Caching a Database Query Result

Now that we have a frontend, we can cache any type of data (we turned on serialization). for example, we can cache a result from a very expensive database query. After it is cached, there is no need to even connect to the database; records are fetched from cache and unserialized.

```
// $cache initialized in previous example

// see if a cache already exists:
if(!$result = $cache->load('myresult')) {

    // cache miss; connect to the database

    $db = Zend_Db::factory( [...] );

    $result = $db->fetchAll('SELECT * FROM huge_table');

    $cache->save($result, 'myresult');

} else {

    // cache hit! shout so that we know
    echo "This one is from cache!\n\n";

}

print_r($result);
```

Example 62. Caching Output with Zend_Cache Output Frontend

We 'mark up' sections in which we want to cache output by adding some conditional logic, encapsulating the section within `start()` and `end()` methods (this resembles the first example and is the core strategy for caching).

Inside, output your data as usual - all output will be cached when execution hits the `end()` method. On the next run, the whole section will be skipped in favor of fetching data from cache (as long as the cache record is valid).

```
$frontendOptions = array(
   'lifetime' => 30,                  // cache lifetime of 30 seconds
   'automatic_serialization' => false  // this is the default anyways
);

$backendOptions = array('cache_dir' => './tmp/');

$cache = Zend_Cache::factory('Output',
                             'File',
                             $frontendOptions,
                             $backendOptions);

// we pass a unique identifier to the start() method
if(!$cache->start('mypage')) {
    // output as usual:

    echo 'Hello world! ';
    echo 'This is cached ('.time().') ';

    $cache->end(); // the output is saved and sent to the browser
}

echo 'This is never cached ('.time().').';
```

Notice that we output the result of `time()` twice; this is something dynamic for demonstration purposes. Try running this and then refreshing several times; you will notice that the first number doesn't change while second changes as time passes. That is because the first number was output in the cached section and is saved among other output. After half a minute (we've set lifetime to 30 seconds) the numbers should match again because the cache record expired -- only to be cached again. You should try this in your browser or console.

When using `Zend_Cache`, pay attention to the important cache identifier (passed to `save()` and `start()`). It must be unique for every resource you cache, otherwise unrelated cache records may wipe each other or, even worse, be displayed in place of the other.

2. The Theory of Caching

There are three key concepts in `Zend_Cache`. One is the unique identifier (a string) that is used to identify cache records. The second one is the *'lifetime'* directive as seen in the examples; it defines for how long the cached resource is considered 'fresh'. The third key concept is conditional execution so that parts of your code can be skipped entirely, boosting performance. The main frontend function (eg. `Zend_Cache_Core::get()`) is always designed to return FALSE for a cache miss if that makes sense for the nature of a frontend. That enables end-users

to wrap parts of the code they would like to cache (and skip) in *if(){...}* statements where the condition is a Zend_Cache method itself. On the end if these blocks you must save what you've generated, however (eg. Zend_Cache_Core::save()).

The conditional execution design of your generating code is not necessary in some frontends (*Function*, for an example) when the whole logic is implemented inside the frontend.

'Cache hit' is a term for a condition when a cache record is found, is valid and is 'fresh' (in other words hasn't expired yet). 'Cache miss' is everything else. When a cache miss happens, you must generate your data (as you would normally do) and have it cached. When you have a cache hit, on the other hand, the backend automatically fetches the record from cache transparently.

2.1. The Zend_Cache Factory Method

A good way to build a usable instance of a Zend_Cache Frontend is given in the following example :

```
// We choose a backend (for example 'File' or 'Sqlite'...)
$backendName = '[...]';

// We choose a frontend (for example 'Core', 'Output', 'Page'...)
$frontendName = '[...]';

// We set an array of options for the choosen frontend
$frontendOptions = array([...]);

// We set an array of options for the choosen backend
$backendOptions = array([...]);

// We create an instance of Zend_Cache
// (of course, the two last arguments are optional)
$cache = Zend_Cache::factory($frontendName,
                             $backendName,
                             $frontendOptions,
                             $backendOptions);
```

In the following examples we will assume that the $cache variable holds a valid, instantiated frontend as shown and that you understand how to pass parameters to your chosen backends.

Always use Zend_Cache::factory() to get frontend instances. Instantiating frontends and backends yourself will not work as expected.

2.2. Tagging Records

Tags are a way to categorize cache records. When you save a cache with the save() method, you can set an array of tags to apply for this record. Then you will be able to clean all cache records tagged with a given tag (or tags):

```
$cache->save($huge_data, 'myUniqueID', array('tagA', 'tagB', 'tagC'));
```

note than the `save()` method accepts an optional fourth argument: `$specificLifetime` (if != FALSE, it sets a specific lifetime for this particular cache record)

2.3. Cleaning the Cache

To remove or invalidate in particular cache id, you can use the `remove()` method :

```
$cache->remove('idToRemove');
```

To remove or invalidate several cache ids in one operation, you can use the `clean()` method. For example to remove all cache records :

```
// clean all records
$cache->clean(Zend_Cache::CLEANING_MODE_ALL);

// clean only outdated
$cache->clean(Zend_Cache::CLEANING_MODE_OLD);
```

If you want to remove cache entries matching the tags 'tagA' and 'tagC':

```
$cache->clean(
    Zend_Cache::CLEANING_MODE_MATCHING_TAG,
    array('tagA', 'tagC')
);
```

If you want to remove cache entries not matching the tags 'tagA' or 'tagC':

```
$cache->clean(
    Zend_Cache::CLEANING_MODE_NOT_MATCHING_TAG,
    array('tagA', 'tagC')
);
```

If you want to remove cache entries matching the tags 'tagA' or 'tagC':

```
$cache->clean(
    Zend_Cache::CLEANING_MODE_MATCHING_ANY_TAG,
    array('tagA', 'tagC')
);
```

Available cleaning modes are: `CLEANING_MODE_ALL`, `CLEANING_MODE_OLD`, `CLEANING_MODE_MATCHING_TAG`, `CLEANING_MODE_NOT_MATCHING_TAG` and `CLEANING_MODE_MATCHING_ANY_TAG`. The latter are, as their names suggest, combined with an array of tags in cleaning operations.

3. Zend_Cache Frontends

3.1. Zend_Cache_Core

3.1.1. Introduction

`Zend_Cache_Core` is a special frontend because it is the core of the module. It is a generic cache frontend and is extended by other classes.

All frontends inherit from `Zend_Cache_Core` so that its methods and options (described below) would also be available in other frontends, therefore they won't be documented there.

3.1.2. Available options

These options are passed to the factory method as demonstrated in previous examples.

Table 19. Core Frontend Options

Option	Data Type	Default Value	Description
caching	Boolean	`TRUE`	enable / disable caching (can be very useful for the debug of cached scripts)
cache_id_prefix	String	`NULL`	A prefix for all cache ids, if set to `NULL`, no cache id prefix will be used. The cache id prefix essentially creates a namespace in the cache, allowing multiple applications or websites to use a shared cache. Each application or website can use a different cache id prefix so specific cache ids can be used more than once.
lifetime	Integer	3600	cache lifetime (in seconds), if set to `NULL`, the cache is valid forever.
logging	Boolean	`FALSE`	if set to `TRUE`, logging through `Zend_Log` is activated (but the system is slower)
write_control	Boolean	`TRUE`	Enable / disable write control (the cache is read just after writing to detect corrupt entries), enabling write_control will lightly slow the cache writing but not the cache reading (it can detect some corrupt cache files but it's not a perfect control)

Option	Data Type	Default Value	Description
automatic_serialization	Boolean	FALSE	Enable / disable automatic serialization, it can be used to save directly datas which aren't strings (but it's slower)
automatic_cleaning_factor	Integer	10	Disable / Tune the automatic cleaning process (garbage collector): 0 means no automatic cache cleaning, 1 means systematic cache cleaning and x > 1 means automatic random cleaning 1 times in x write operations.
ignore_user_abort	Boolean	FALSE	if set to TRUE, the core will set the ignore_user_abort PHP flag inside the `save()` method to avoid cache corruptions in some cases

3.1.3. Examples

An example is given in the manual at the very beginning.

If you store only strings into cache (because with "automatic_serialization" option, it's possible to store some booleans), you can use a more compact construction like:

```
// we assume you already have $cache

$id = 'myBigLoop'; // cache id of "what we want to cache"

if (!($data = $cache->load($id))) {
    // cache miss

    $data = '';
    for ($i = 0; $i < 10000; $i++) {
        $data = $data . $i;
    }

    $cache->save($data);

}

// [...] do something with $data (echo it, pass it on etc.)
```

If you want to cache multiple blocks or data instances, the idea is the same:

```
// make sure you use unique identifiers:
$id1 = 'foo';
$id2 = 'bar';

// block 1
if (!($data = $cache->load($id1))) {
    // cache missed

    $data = '';
    for ($i=0;$i<10000;$i++) {
        $data = $data . $i;
    }

    $cache->save($data);

}
echo($data);

// this isn't affected by caching
echo('NEVER CACHED! ');

// block 2
if (!($data = $cache->load($id2))) {
    // cache missed

    $data = '';
    for ($i=0;$i<10000;$i++) {
        $data = $data . '!';
    }

    $cache->save($data);

}
echo($data);
```

If you want to cache special values (boolean with "automatic_serialization" option) or empty strings you can't use the compact construction given above. You have to test formally the cache record.

```
// the compact construction
// (not good if you cache empty strings and/or booleans)
if (!($data = $cache->load($id))) {

    // cache missed

    // [...] we make $data

    $cache->save($data);

}

// we do something with $data

// [...]

// the complete construction (works in any case)
if (!($cache->test($id))) {

    // cache missed
```

```
    // [...] we make $data

    $cache->save($data);

} else {

    // cache hit

    $data = $cache->load($id);

}

// we do something with $data
```

3.2. Zend_Cache_Frontend_Output

3.2.1. Introduction

`Zend_Cache_Frontend_Output` is an output-capturing frontend. It utilizes output buffering in PHP to capture everything between its `start()` and `end()` methods.

3.2.2. Available Options

This frontend doesn't have any specific options other than those of `Zend_Cache_Core`.

3.2.3. Examples

An example is given in the manual at the very beginning. Here it is with minor changes:

```
// if it is a cache miss, output buffering is triggered
if (!($cache->start('mypage'))) {

    // output everything as usual
    echo 'Hello world! ';
    echo 'This is cached ('.time().') ';

    $cache->end(); // output buffering ends

}

echo 'This is never cached ('.time().').';
```

Using this form it is fairly easy to set up output caching in your already working project with little or no code refactoring.

3.3. Zend_Cache_Frontend_Function

3.3.1. Introduction

`Zend_Cache_Frontend_Function` caches the results of function calls. It has a single main method named `call()` which takes a function name and parameters for the call in an array.

3.3.2. Available Options

Table 20. Function Frontend Options

Option	Data Type	Default Value	Description
cache_by_default	Boolean	TRUE	if TRUE, function calls will be cached by default
cached_functions	Array		function names which will always be cached
non_cached_functions	Array		function names which must never be cached

3.3.3. Examples

Using the `call()` function is the same as using `call_user_func_array()` in PHP:

```
$cache->call('veryExpensiveFunc', $params);

// $params is an array
// For example to call veryExpensiveFunc(1, 'foo', 'bar') with
// caching, you can use
// $cache->call('veryExpensiveFunc', array(1, 'foo', 'bar'))
```

`Zend_Cache_Frontend_Function` is smart enough to cache both the return value of the function and its internal output.

You can pass any built in or user defined function with the exception of `array()`, `echo()`, `empty()`, `eval()`, `exit()`, `isset()`, `list()`, `print()` and `unset()`.

3.4. Zend_Cache_Frontend_Class

3.4.1. Introduction

`Zend_Cache_Frontend_Class` is different from `Zend_Cache_Frontend_Function` because it allows caching of object and static method calls.

3.4.2. Available Options

Table 21. Class Frontend Options

Option	Data Type	Default Value	Description
cached_entity (required)	Mixed		if set to a class name, we will cache an abstract class and will use only static calls; if set to an object, we will cache this object methods
cache_by_default	Boolean	TRUE	if TRUE, calls will be cached by default

Option	Data Type	Default Value	Description
cached_methods	Array		method names which will always be cached
non_cached_methods	Array		method names which must never be cached

3.4.3. Examples

For example, to cache static calls :

```
class Test {

    // Static method
    public static function foobar($param1, $param2) {
        echo "foobar_output($param1, $param2)";
        return "foobar_return($param1, $param2)";
    }

}

// [...]
$frontendOptions = array(
    'cached_entity' => 'Test' // The name of the class
);
// [...]

// The cached call
$result = $cache->foobar('1', '2');
```

To cache classic method calls :

```
class Test {

    private $_string = 'hello !';

    public function foobar2($param1, $param2) {
        echo($this->_string);
        echo "foobar2_output($param1, $param2)";
        return "foobar2_return($param1, $param2)";
    }

}

// [...]
$frontendOptions = array(
    'cached_entity' => new Test() // An instance of the class
);
// [...]

// The cached call
$result = $cache->foobar2('1', '2');
```

3.5. Zend_Cache_Frontend_File

3.5.1. Introduction

`Zend_Cache_Frontend_File` is a frontend driven by the modification time of a "master file". It's really interesting for examples in configuration or templates issues. It's also possible to use multiple master files.

For instance, you have an XML configuration file which is parsed by a function which returns a "config object" (like with `Zend_Config`). With `Zend_Cache_Frontend_File`, you can store the "config object" into cache (to avoid the parsing of the XML config file at each time) but with a sort of strong dependency on the "master file". So, if the XML config file is modified, the cache is immediately invalidated.

3.5.2. Available Options

Table 22. File Frontend Options

Option	Data Type	Default Value	Description
master_file (deprecated)	String	''	the complete path and name of the master file
master_files	Array	`array()`	an array of complete path of master files
master_files_mode	String	`Zend_Cache_Frontend_File::MODE_OR`	`Zend_Cache_Frontend_File::MODE_OR` or `Zend_Cache_Frontend_File::MODE_AND`; if `MODE_AND`, then all master files have to be touched to get a cache invalidation if `MODE_OR`, then a single touched master file is enough to get a cache invalidation
ignore_missing_master_files	Boolean	`FALSE`	if `TRUE`, missing master files are ignored silently (an exception is raised else)

3.5.3. Examples

Use of this frontend is the same than of `Zend_Cache_Core`. There is no need of a specific example - the only thing to do is to define the *master_file* when using the factory.

3.6. Zend_Cache_Frontend_Page

3.6.1. Introduction

`Zend_Cache_Frontend_Page` is like `Zend_Cache_Frontend_Output` but designed for a complete page. It's impossible to use `Zend_Cache_Frontend_Page` for caching only a single block.

On the other hand, the "cache id" is calculated automatically with `$_SERVER['REQUEST_URI']` and (depending on options) `$_GET`, `$_POST`, `$_SESSION`, `$_COOKIE`, `$_FILES`. More over, you have only one method to call (`start()`) because the `end()` call is fully automatic when the page is ended.

For the moment, it's not implemented but we plan to add a HTTP conditional system to save bandwidth (the system will send a HTTP 304 Not Modified if the cache is hit and if the browser has already the good version).

3.6.2. Available Options

Table 23. Page Frontend Options

Option	Data Type	Default Value	Description
http_conditional	Boolean	FALSE	use the http_conditional system (not implemented for the moment)
debug_header	Boolean	FALSE	if TRUE, a debug text is added before each cached pages
default_options	Array	`array(...see below...)`	an associative array of default options: • *(boolean, TRUE by default) cache*: cache is on if TRUE • *(boolean, FALSE by default) cache_with_get_variables*: if TRUE, cache is still on even if there are some variables in `$_GET` array • *(boolean, FALSE by default) cache_with_post_variables*: if TRUE, cache is still on even if there are some variables in `$_POST` array • *(boolean, FALSE by default) cache_with_session_variables*: if TRUE, cache is still on even if there are some variables in `$_SESSION` array

Option	Data Type	Default Value	Description
			• *(boolean, FALSE by default)* *cache_with_files_variables*: if TRUE, cache is still on even if there are some variables in $_FILES array • *(boolean, FALSE by default)* *cache_with_cookie_variables*: if TRUE, cache is still on even if there are some variables in $_COOKIE array • *(boolean, TRUE by default)* *make_id_with_get_variables*: if TRUE, the cache id will be dependent of the content of the $_GET array • *(boolean, TRUE by default)* *make_id_with_post_variables*: if TRUE, the cache id will be dependent of the content of the $_POST array • *(boolean, TRUE by default)* *make_id_with_session_variables*: if TRUE, the cache id will be dependent of the content of the $_SESSION array • *(boolean, TRUE by default)* *make_id_with_files_variables*: if TRUE, the cache id will be dependent of the content of the $_FILES array • *(boolean, TRUE by default)* *make_id_with_cookie_variables*: if TRUE, the cache id will be dependent

Option	Data Type	Default Value	Description
			of the content of the `$_COOKIE` array • *(int, `FALSE` by default) specific_lifetime*: if not `FALSE`, the given lifetime will be used for the choosen regexp • *(array, `array()` by default) tags*: tags for the cache record • *(int, `NULL` by default) priority*: priority (if the backend supports it)
regexps	Array	`array()`	an associative array to set options only for some `REQUEST_URI`, keys are (PCRE) regexps, values are associative arrays with specific options to set if the regexp matchs on `$_SERVER['REQUEST_URI']` (see default_options for the list of available options); if several regexps match the `$_SERVER['REQUEST_URI']`, only the last one will be used
memorize_headers	Array	`array()`	an array of strings corresponding to some HTTP headers name. Listed headers will be stored with cache datas and "replayed" when the cache is hit

3.6.3. Examples

Use of `Zend_Cache_Frontend_Page` is really trivial:

```
// [...] // require, configuration and factory

$cache->start();
// if the cache is hit, the result is sent to the browser
// and the script stop here
```

```
// rest of the page ...
```

a more complex example which shows a way to get a centralized cache management in a bootstrap file (for using with `Zend_Controller` for example)

```
/*
 * You should avoid putting too many lines before the cache section.
 * For example, for optimal performances, "require_once" or
 * "Zend_Loader::loadClass" should be after the cache section.
 */

$frontendOptions = array(
   'lifetime' => 7200,
   'debug_header' => true, // for debugging
   'regexps' => array(
       // cache the whole IndexController
       '^/$' => array('cache' => true),

       // cache the whole IndexController
       '^/index/' => array('cache' => true),

       // we don't cache the ArticleController...
       '^/article/' => array('cache' => false),

       // ... but we cache the "view" action of this ArticleController
       '^/article/view/' => array(
           'cache' => true,

           // and we cache even there are some variables in $_POST
           'cache_with_post_variables' => true,

           // but the cache will be dependent on the $_POST array
           'make_id_with_post_variables' => true
       )
   )
);

$backendOptions = array(
    'cache_dir' => '/tmp/'
);

// getting a Zend_Cache_Frontend_Page object
$cache = Zend_Cache::factory('Page',
                             'File',
                             $frontendOptions,
                             $backendOptions);

$cache->start();
// if the cache is hit, the result is sent to the browser and the
// script stop here

// [...] the end of the bootstrap file
// these lines won't be executed if the cache is hit
```

3.6.4. The Specific Cancel Method

Because of design issues, in some cases (for example when using non HTTP 200 return codes), you could need to cancel the current cache process. So we introduce for this particular frontend, the `cancel()` method.

```
// [...] // require, configuration and factory

$cache->start();

// [...]

if ($someTest) {
    $cache->cancel();
    // [...]
}

// [...]
```

4. Zend_Cache Backends

There are two kinds of backends: standard ones and extended ones. Of course, extended backends offer more features.

4.1. Zend_Cache_Backend_File

This (extended) backends stores cache records into files (in a choosen directory).

Available options are :

Table 24. File Backend Options

Option	Data Type	Default Value	Description
cache_dir	String	'/tmp/'	Directory where to store cache files
file_locking	Boolean	TRUE	Enable or disable file_locking : Can avoid cache corruption under bad circumstances but it doesn't help on multithread webservers or on NFS filesystems...
read_control	Boolean	TRUE	Enable / disable read control : if enabled, a control key is embedded in the cache file and this key is compared with the one calculated after the reading.
read_control_type	String	'crc32'	Type of read control (only if read control is enabled). Available values are : 'md5' (best but slowest), 'crc32' (lightly less safe but faster, better choice), 'adler32' (new choice,

Option	Data Type	Default Value	Description
			faster than crc32), 'strlen' for a length only test (fastest).
hashed_directory_level	Integer	0	Hashed directory structure level : 0 means "no hashed directory structure", 1 means "one level of directory", 2 means "two levels"... This option can speed up the cache only when you have many thousands of cache files. Only specific benchs can help you to choose the perfect value for you. Maybe, 1 or 2 is a good start.
hashed_directory_umask	Integer	0700	Umask for the hashed directory structure
file_name_prefix	String	'zend_cache'	prefix for cache files ; be really careful with this option because a too generic value in a system cache dir (like `/tmp`) can cause disasters when cleaning the cache
cache_file_umask	Integer	0700	umask for cache files
metatadatas_array_max_size	Integer	100	internal max size for the metadatas array (don't change this value unless you know what you are doing)

4.2. Zend_Cache_Backend_Sqlite

This (extended) backends stores cache records into a SQLite database.

Available options are :

Table 25. Sqlite Backend Options

Option	Data Type	Default Value	Description
cache_db_complete_path (mandatory)	String	`NULL`	The complete path (filename included) of the SQLite database
automatic_vacuum_factor	Integer	10	Disable / Tune the automatic vacuum

Option	Data Type	Default Value	Description
			process. The automatic vacuum process defragment the database file (and make it smaller) when a `clean()` or `delete()` is called: 0 means no automatic vacuum ; 1 means systematic vacuum (when `delete()` or `clean()` methods are called) ; x (integer) > 1 => automatic vacuum randomly 1 times on x `clean()` or `delete()`.

4.3. Zend_Cache_Backend_Memcached

This (extended) backends stores cache records into a memcached server. memcached is a high-performance, distributed memory object caching system. To use this backend, you need a memcached daemon and the memcache PECL extension.

Be careful : with this backend, "tags" are not supported for the moment as the "doNotTestCacheValidity=true" argument.

Available options are :

Table 26. Memcached Backend Options

Option	Data Type	Default Value	Description
servers	Array	**array(array('host' => 'localhost', 'port' => 11211, 'persistent' => true, 'weight' => 1, 'timeout' => 5, 'retry_interval' => 15, 'status' => true, 'failure_callback' => ''))**	An array of memcached servers ; each memcached server is described by an associative array : 'host' => (string) : the name of the memcached server, 'port' => (int) : the port of the memcached server, 'persistent' => (bool) : use or not persistent connections to this memcached server 'weight' => (int) :the weight of the memcached server, 'timeout' => (int) :the time out of the memcached server, 'retry_interval' =>

Option	Data Type	Default Value	Description
			(int) :the retry interval of the memcached server, 'status' => (bool) :the status of the memcached server, 'failure_callback' => (callback) : the failure_callback of the memcached server
compression	Boolean	FALSE	TRUE if you want to use on-the-fly compression
compatibility	Boolean	FALSE	TRUE if you want to use this compatibility mode with old memcache servers or extensions

4.4. Zend_Cache_Backend_Apc

This (extended) backends stores cache records in shared memory through the APC (Alternative PHP Cache) extension (which is of course need for using this backend).

Be careful : with this backend, "tags" are not supported for the moment as the "doNotTestCacheValidity=true" argument.

There is no option for this backend.

4.5. Zend_Cache_Backend_Xcache

This backends stores cache records in shared memory through the XCache extension (which is of course need for using this backend).

Be careful : with this backend, "tags" are not supported for the moment as the "doNotTestCacheValidity=true" argument.

Available options are :

Table 27. Xcache Backend Options

Option	Data Type	Default Value	Description
user	String	`NULL`	`xcache.admin.user`, necessary for the `clean()` method
password	String	`NULL`	`xcache.admin.pass` (in clear form, not MD5), necessary for the `clean()` method

4.6. Zend_Cache_Backend_ZendPlatform

This backend uses content caching API of the Zend Platform product. Naturally, to use this backend you need to have Zend Platform installed.

This backend supports tags, but does not support `CLEANING_MODE_NOT_MATCHING_TAG` cleaning mode.

Specify this backend using a word separator -- '-', '.', ' ', or '_' -- between the words 'Zend' and 'Platform' when using the `Zend_Cache::factory()` method:

```
$cache = Zend_Cache::factory('Core', 'Zend Platform');
```

There are no options for this backend.

4.7. Zend_Cache_Backend_TwoLevels

This (extend) backend is an hybrid one. It stores cache records in two other backends : a fast one (but limited) like Apc, Memcache... and a "slow" one like File, Sqlite...

This backend will use the priority parameter (given at the frontend level when storing a record) and the remaining space in the fast backend to optimize the usage of these two backends.

Specify this backend using a word separator -- '-', '.', ' ', or '_' -- between the words 'Two' and 'Levels' when using the `Zend_Cache::factory()` method:

```
$cache = Zend_Cache::factory('Core', 'Two Levels');
```

Available options are :

Table 28. TwoLevels Backend Options

Option	Data Type	Default Value	Description
slow_backend	String	File	the "slow" backend name
fast_backend	String	Apc	the "fast" backend name
slow_backend_options	Array	`array()`	the "slow" backend options
fast_backend_options	Array	`array()`	the "fast" backend options
slow_backend_custom_naming	Boolean	FALSE	if TRUE, the slow_backend argument is used as a complete class name; if FALSE, the frontend argument is used as the end of "Zend_Cache_Backend_[...]" class name
fast_backend_custom_naming	Boolean	FALSE	if TRUE, the fast_backend argument is used as a complete class name; if FALSE, the frontend argument is used as the end of

Option	Data Type	Default Value	Description
			"Zend_Cache_Backend_[...]" class name
slow_backend_autoload	Boolean	FALSE	if TRUE, there will no require_once for the slow backend (useful only for custom backends)
fast_backend_autoload	Boolean	FALSE	if TRUE, there will no require_once for the fast backend (useful only for custom backends)
auto_refresh_fast_cache	Boolean	TRUE	if TRUE, auto refresh the fast cache when a cache record is hit
stats_update_factor	Integer	10	disable / tune the computation of the fast backend filling percentage (when saving a record into cache, computation of the fast backend filling percentage randomly 1 times on x cache writes)

4.8. Zend_Cache_Backend_ZendServer_Disk and Zend_Cache_Backend_ZendServer_ShMem

These backends store cache records using Zend Server caching functionality.

Be careful: with these backends, "tags" are not supported for the moment as the "doNotTestCacheValidity=true" argument.

These backend work only withing Zend Server environment for pages requested through HTTP or HTTPS and don't work for command line script execution

Specify this backend using parameter *customBackendNaming* as TRUE when using the `Zend_Cache::factory()` method:

```
$cache = Zend_Cache::factory('Core', 'Zend_Cache_Backend_ZendServer_Disk',
                             $frontendOptions, $backendOptions, false, true);
```

There is no option for this backend.

5. The Cache Manager

It's the nature of applications to require a multitude of caches of any type often dependent on the controller, library or domain model being accessed. To allow for a simple means of defining `Zend_Cache` options in advance (such as from a bootstrap) so that accessing a cache object requires minimum setup within the application

source code, the `Zend_Cache_Manager` class was written. This class is accompanied by `Zend_Application_Resource_Cachemanager` ensuring bootstrap configuration is available and `Zend_Controller_Action_Helper_Cache` to allow simple cache access and instantiation from controllers and other helpers.

The basic operation of this component is as follows. The Cache Manager allows users to setup "option templates", basically options for a set of named caches. These can be set using the method `Zend_Cache_Manager::setCacheTemplate()`. These templates do not give rise to a cache until the user attempts to retrieve a named cache (associated with an existing option template) using the method `Zend_Cache_Manager::getCache()`.

```
$manager = new Zend_Cache_Manager;

$dbCache = array(
    'frontend' => array(
        'name' => 'Core',
        'options' => array(
            'lifetime' => 7200,
            'automatic_serialization' => true
        )
    ),
    'backend' => array(
        'name' => 'Core',
        'options' => array(
            'cache_dir' => '/path/to/cache'
        )
    )
);

$manager->setCacheTemplate('database', $dbCache);

/**
 * Anywhere else where the Cache Manager is available...
 */
$databaseCache = $manager->getCache('database');
```

The Cache Manager also allows simple setting of pre-instantiated caches using the method `Zend_Cache_Manager::setCache()`.

```
$frontendOptions = array(
   'lifetime' => 7200,
   'automatic_serialization' => true
);

$backendOptions = array(
    'cache_dir' => '/path/to/cache'
);

$dbCache = Zend_Cache::factory('Core',
                               'File',
                               $frontendOptions,
                               $backendOptions);

$manager = new Zend_Cache_Manager;
$manager->setCache('database', $dbCache);

/**
 * Anywhere else where the Cache Manager is available...
```

```
 */
$databaseCache = $manager->getCache('database');
```

If for any reason, you are unsure where the Cache Manager contains a pre-instantiated cache or a relevant option cache template to create one on request, you can check for the existance of a name cache configuration or instance using the method `Zend_Cache_Manager::hasCache()`.

```
$manager = new Zend_Cache_Manager;

$dbCache = array(
    'frontend' => array(
        'name' => 'Core',
        'options' => array(
            'lifetime' => 7200,
            'automatic_serialization' => true
        )
    ),
    'backend' => array(
        'name' => 'Core',
        'options' => array(
            'cache_dir' => '/path/to/cache'
        )
    )
);

$manager->setCacheTemplate('database', $dbCache);

/**
 * Anywhere else where the Cache Manager is available...
 */
if ($manager->hasCache('database')) {
    $databaseCache = $manager->getCache('database');
} else {
    // create a cache from scratch if none available from Manager
}
```

In some scenarios, you may have defined a number of general use caches using `Zend_Cache_Manager` but need to fine-tune their options before use depending on the circumstances. You can edit previously set cache templates on the fly before they are instantiated using the method `Zend_Cache_Manager::setTemplateOptions()`.

```
$manager = new Zend_Cache_Manager;

$dbCache = array(
    'frontend' => array(
        'name' => 'Core',
        'options' => array(
            'lifetime' => 7200,
            'automatic_serialization' => true
        )
    ),
    'backend' => array(
        'name' => 'Core',
        'options' => array(
            'cache_dir' => '/path/to/cache'
        )
    )
```

```
);

$manager->setCacheTemplate('database', $dbCache);

/**
 * Anywhere else where the Cache Manager is available...
 * Here we decided to store some upcoming database queries to Memcached instead
 * of the preconfigured File backend.
 */
$fineTuning = array(
    'backend' => array(
        'name' => 'Memcached',
        'options' => array(
            'servers' => array(
                array(
                    'host' => 'localhost',
                    'port' => 11211,
                    'persistent' => true,
                    'weight' => 1,
                    'timeout' => 5,
                    'retry_interval' => 15,
                    'status' => true,
                    'failure_callback' => ''
                )
            )
        )
    )
);
$manager->setTemplateOptions('database', $fineTuning);
$databaseCache = $manager->getCache('database');
```

To assist in making the Cache Manager more useful, it is accompanied by `Zend_Application_Resource_Cachemanager` and also the `Zend_Controller_Action_Helper_Cache` Action Helper. Both of these are described in their relevant areas of the Reference Guide.

Out of the box, `Zend_Cache_Manager` already includes four pre-defined cache templates called "skeleton", "default", "page" and "tagcache". The default cache is a simple File based cache using the Core frontend which assumes a cache_dir called "cache" exists at the same level as the conventional "public" directory of a Zend Framework application. The skeleton cache is actually a `NULL` cache, i.e. it contains no options. The remaining two caches are used to implement a default Static Page Cache where static HTML, XML or even JSON may be written to static files in `/public`. Control over a Static Page Cache is offered via `Zend_Controller_Action_Helper_Cache`, though you may alter the settings of this "page" the "tagcache" it uses to track tags using `Zend_Cache_Manager::setTemplateOptions()` or even `Zend_Cache_Manager::setCacheTemplate()` if overloading all of their options.

Zend_Captcha

1. Introduction

CAPTCHA stands for "Completely Automated Public Turing test to tell Computers and Humans Apart"; it is used as a challenge-response to ensure that the individual submitting information is a human and not an automated process. Typically, a captcha is used with form submissions where authenticated users are not necessary, but you want to prevent spam submissions.

Captchas can take a variety of forms, including asking logic questions, presenting skewed fonts, and presenting multiple images and asking how they relate. `Zend_Captcha` aims to provide a variety of back ends that may be utilized either standalone or in conjunction with `Zend_Form`.

2. Captcha Operation

All CAPTCHA adapter implement `Zend_Captcha_Adapter`, which looks like the following:

```
interface Zend_Captcha_Adapter extends Zend_Validate_Interface
{
    public function generate();

    public function render(Zend_View $view, $element = null);

    public function setName($name);

    public function getName();

    public function getDecorator();

    // Additionally, to satisfy Zend_Validate_Interface:
    public function isValid($value);

    public function getMessages();

    public function getErrors();
}
```

The name setter and getter are used to specify and retrieve the CAPTCHA identifier. `getDecorator()` can be used to specify a `Zend_Form` decorator either by name or returning an actual decorator object. The most interesting methods are `generate()` and `render()`. `generate()` is used to create the CAPTCHA token. This process typically will store the token in the session so that you may compare against it in subsequent requests. `render()` is used to render the information that represents the CAPTCHA, be it an image, a figlet, a logic problem, or some other CAPTCHA.

A typical use case might look like the following:

```
// Creating a Zend_View instance
$view = new Zend_View();

// Originating request:
$captcha = new Zend_Captcha_Figlet(array(
    'name' => 'foo',
    'wordLen' => 6,
```

```
    'timeout' => 300,
));

$id = $captcha->generate();
echo "<form method=\"post\" action=\"\">";
echo $captcha->render($view);
echo "</form>";

// On subsequent request:
// Assume captcha setup as before, the value of $_POST['foo']
// would be key/value array: id => captcha ID, input => captcha value
if ($captcha->isValid($_POST['foo'], $_POST)) {
    // Validated!
}
```

3. CAPTCHA Adapters

The following adapters are shipped with Zend Framework by default.

3.1. Zend_Captcha_Word

`Zend_Captcha_Word` is an abstract adapter that serves as the base class for most other CAPTCHA adapters. It provides mutators for specifying word length, session TTL, the session namespace object to use, and the session namespace class to use for persistence if you do not wish to use `Zend_Session_Namespace`. `Zend_Captcha_Word` encapsulates validation logic.

By default, the word length is 8 characters, the session timeout is 5 minutes, and `Zend_Session_Namespace` is used for persistence (using the namespace "`Zend_Form_Captcha_<captcha ID>`").

In addition to the methods required by the `Zend_Captcha_Adapter` interface, `Zend_Captcha_Word` exposes the following methods:

- `setWordLen($length)` and `getWordLen()` allow you to specify the length of the generated "word" in characters, and to retrieve the current value.
- `setTimeout($ttl)` and `getTimeout()` allow you to specify the time-to-live of the session token, and to retrieve the current value. `$ttl` should be specified in seconds.
- `setSessionClass($class)` and `getSessionClass()` allow you to specify an alternate `Zend_Session_Namespace` implementation to use to persist the CAPTCHA token and to retrieve the current value.
- `getId()` allows you to retrieve the current token identifier.
- `getWord()` allows you to retrieve the generated word to use with the CAPTCHA. It will generate the word for you if none has been generated yet.
- `setSession(Zend_Session_Namespace $session)` allows you to specify a session object to use for persisting the CAPTCHA token. `getSession()` allows you to retrieve the current session object.

All word CAPTCHAs allow you to pass an array of options to the constructor, or, alternately, pass them to `setOptions()`. You can also pass a `Zend_Config` object to `setConfig()`. By default, the *wordLen*, *timeout*, and *sessionClass* keys may all be used. Each concrete implementation may define additional keys or utilize the options in other ways.

`Zend_Captcha_Word` is an abstract class and may not be instantiated directly.

3.2. Zend_Captcha_Dumb

The `Zend_Captch_Dumb` adapter is mostly self-descriptive. It provides a random string that must be typed in reverse to validate. As such, it's not a good CAPTCHA solution and should only be used for testing. It extends `Zend_Captcha_Word`.

3.3. Zend_Captcha_Figlet

The `Zend_Captcha_Figlet` adapter utilizes `Zend_Text_Figlet` to present a figlet to the user.

Options passed to the constructor will also be passed to the `Zend_Text_Figlet` object. See the `Zend_Text_Figlet` documentation for details on what configuration options are available.

3.4. Zend_Captcha_Image

The `Zend_Captcha_Image` adapter takes the generated word and renders it as an image, performing various skewing permutations to make it difficult to automatically decipher. It requires the GD extension compiled with TrueType or Freetype support. Currently, the `Zend_Captcha_Image` adapter can only generate PNG images.

`Zend_Captcha_Image` extends `Zend_Captcha_Word`, and additionally exposes the following methods:

- `setExpiration($expiration)` and `getExpiration()` allow you to specify a maximum lifetime the CAPTCHA image may reside on the filesystem. This is typically a longer than the session lifetime. Garbage collection is run periodically each time the CAPTCHA object is invoked, deleting all images that have expired. Expiration values should be specified in seconds.

- `setGcFreq($gcFreq)` and `getGcFreg()` allow you to specify how frequently garbage collection should run. Garbage collection will run every **1/$gcFreq** calls. The default is 100.

- `setFont($font)` and `getFont()` allow you to specify the font you will use. `$font` should be a fully qualified path to the font file. This value is required; the CAPTCHA will throw an exception during generation if the font file has not been specified.

- `setFontSize($fsize)` and `getFontSize()` allow you to specify the font size in pixels for generating the CAPTCHA. The default is 24px.

- `setHeight($height)` and `getHeight()` allow you to specify the height in pixels of the generated CAPTCHA image. The default is 50px.

- `setWidth($width)` and `getWidth()` allow you to specify the width in pixels of the generated CAPTCHA image. The default is 200px.

- `setImgDir($imgDir)` and `getImgDir()` allow you to specify the directory for storing CAPTCHA images. The default is "`./images/captcha/`", relative to the bootstrap script.

- `setImgUrl($imgUrl)` and `getImgUrl()` allow you to specify the relative path to a CAPTCHA image to use for HTML markup. The default is "`/images/captcha/`".

- `setSuffix($suffix)` and `getSuffix()` allow you to specify the filename suffix for the CAPTCHA image. The default is "`.png`". Note: changing this value will not change the type of the generated image.

All of the above options may be passed to the constructor by simply removing the 'set' method prefix and casting the initial letter to lowercase: "suffix", "height", "imgUrl", etc.

3.5. Zend_Captcha_ReCaptcha

The `Zend_Captcha_ReCaptcha` adapter uses Zend_Service_ReCaptcha to generate and validate CAPTCHAs. It exposes the following methods:

- `setPrivKey($key)` and `getPrivKey()` allow you to specify the private key to use for the ReCaptcha service. This must be specified during construction, although it may be overridden at any point.
- `setPubKey($key)` and `getPubKey()` allow you to specify the public key to use with the ReCaptcha service. This must be specified during construction, although it may be overridden at any point.
- `setService(Zend_Service_ReCaptcha $service)` and `getService()` allow you to set and get the ReCaptcha service object.

Zend_CodeGenerator

1. Introduction

`Zend_CodeGenerator` provides facilities to generate arbitrary code using an object oriented interface, both to create new code as well as to update existing code. While the current implementation is limited to generating PHP code, you can easily extend the base class in order to provide code generation for other tasks: JavaScript, configuration files, apache vhosts, etc.

1.1. Theory of Operation

In the most typical use case, you will simply instantiate a code generator class and either pass it the appropriate configuration or configure it after instantiation. To generate the code, you will simply echo the object or call its `generate()` method.

```
// Passing configuration to the constructor:
$file = new Zend_CodeGenerator_Php_File(array(
    'classes' => array(
        new Zend_CodeGenerator_Php_Class(array(
            'name'    => 'World',
            'methods' => array(
                new Zend_CodeGenerator_Php_Method(array(
                    'name' => 'hello',
                    'body' => 'echo \'Hello world!\';',
                )),
            ),
        )),
    )
));

// Configuring after instantiation
$method = new Zend_CodeGenerator_Php_Method();
$method->setName('hello')
       ->setBody('echo \'Hello world!\';');

$class = new Zend_CodeGenerator_Php_Class();
$class->setName('World')
      ->setMethod($method);

$file = new Zend_CodeGenerator_Php_File();
$file->setClass($class);

// Render the generated file
echo $file;

// or write it to a file:
file_put_contents('World.php', $file->generate());
```

Both of the above samples will render the same result:

```
<?php

class World
{
```

```
    public function hello()
    {
        echo 'Hello world!';
    }

}
```

Another common use case is to update existing code -- for instance, to add a method to a class. In such a case, you must first inspect the existing code using reflection, and then add your new method. `Zend_CodeGenerator` makes this trivially simple, by leveraging Zend_Reflection.

As an example, let's say we've saved the above to the file "`World.php`", and have already included it. We could then do the following:

```
$class = Zend_CodeGenerator_Php_Class::fromReflection(
    new Zend_Reflection_Class('World')
);

$method = new Zend_CodeGenerator_Php_Method();
$method->setName('mrMcFeeley')
       ->setBody('echo \'Hello, Mr. McFeeley!\';');
$class->setMethod($method);

$file = new Zend_CodeGenerator_Php_File();
$file->setClass($class);

// Render the generated file
echo $file;

// Or, better yet, write it back to the original file:
file_put_contents('World.php', $file->generate());
```

The resulting class file will now look like this:

```
<?php

class World
{

    public function hello()
    {
        echo 'Hello world!';
    }

    public function mrMcFeeley()
    {
        echo 'Hellow Mr. McFeeley!';
    }

}
```

2. Zend_CodeGenerator Examples

Example 63. Generating PHP classes

The following example generates an empty class with a class-level DocBlock.

```
$foo      = new Zend_CodeGenerator_Php_Class();
$docblock = new Zend_CodeGenerator_Php_Docblock(array(
    'shortDescription' => 'Sample generated class',
    'longDescription'  => 'This is a class generated with Zend_CodeGenerator.',
    'tags'             => array(
        array(
            'name'        => 'version',
            'description' => '$Rev:$',
        ),
        array(
            'name'        => 'license',
            'description' => 'New BSD',
        ),
    ),
));
$foo->setName('Foo')
    ->setDocblock($docblock);
echo $foo->generate();
```

The above code will result in the following:

```
/**
 * Sample generated class
 *
 * This is a class generated with Zend_CodeGenerator.
 *
 * @version $Rev:$
 * @license New BSD
 *
 */
class Foo
{

}
```

Example 64. Generating PHP classes with class properties

Building on the previous example, we now add properties to our generated class.

```
$foo      = new Zend_CodeGenerator_Php_Class();
$docblock = new Zend_CodeGenerator_Php_Docblock(array(
    'shortDescription' => 'Sample generated class',
    'longDescription'  => 'This is a class generated with Zend_CodeGenerator.',
    'tags'             => array(
        array(
            'name'        => 'version',
            'description' => '$Rev:$',
        ),
        array(
            'name'        => 'license',
            'description' => 'New BSD',
        ),
    ),
));
$foo->setName('Foo')
    ->setDocblock($docblock)
    ->setProperties(array(
        array(
            'name'         => '_bar',
            'visibility'   => 'protected',
            'defaultValue' => 'baz',
        ),
        array(
            'name'         => 'baz',
            'visibility'   => 'public',
            'defaultValue' => 'bat',
        ),
        array(
            'name'         => 'bat',
            'const'        => true,
            'defaultValue' => 'foobarbazbat',
        ),
    ));
echo $foo->generate();
```

The above results in the following class definition:

```
/**
 * Sample generated class
 *
 * This is a class generated with Zend_CodeGenerator.
 *
 * @version $Rev:$
 * @license New BSD
 *
 */
class Foo
{

    protected $_bar = 'baz';

    public $baz = 'bat';

    const bat = 'foobarbazbat';

}
```

```
        array(
            'name'         => '_bar',
            'visibility'   => 'protected',
            'defaultValue' => 'baz',
        ),
        array(
            'name'         => 'baz',
            'visibility'   => 'public',
            'defaultValue' => 'bat',
        ),
        array(
            'name'         => 'bat',
            'const'        => true,
            'defaultValue' => 'foobarbazbat',
        ),
    ))
    ->setMethods(array(
        // Method passed as array
        array(
            'name'       => 'setBar',
            'parameters' => array(
                array('name' => 'bar'),
            ),
            'body'       => '$this->_bar = $bar;' . "\n" . 'return $this;',
            'docblock'   => new Zend_CodeGenerator_Php_Docblock(array(
                'shortDescription' => 'Set the bar property'
```

```
/**
 * Sample generated class
 *
 * This is a class generated with Zend_CodeGenerator.
 *
 * @version $Rev:$
 * @license New BSD
 */
class Foo
{

    protected $_bar = 'baz';

    public $baz = 'bat';

    const bat = 'foobarbazbat';

    /**
     * Set the bar property
     *
     * @param string bar
     * @return string
     */
    public function setBar($bar)
    {
        $this->_bar = $bar;
        return $this;
    }

    /**
     * Retrieve the bar property
     *
     * @return string|null
     */
    public function getBar()
    {
        return $this->_bar;
    }

}
```

Zend_CodeGenerator_Php_File can be used to generate the contents of a PHP file. You can include classes as well as arbitrary content body. When attaching classes, you

```
$file = new Zend_CodeGenerator_Php_File(array(
    'classes'  => array($foo);
    'docblock' => new Zend_CodeGenerator_Php_Docblock(array(
        'shortDescription' => 'Foo class file',
        'tags'             => array(
            array(
                'name'        => 'license',
                'description' => 'New BSD',
            ),
        ),
```

```
<?php
/**
 * Foo class file
 *
 * @license New BSD
 */

/**
 * Sample generated class
 *
 * This is a class generated with Zend_CodeGenerator.
 *
 * @version $Rev:$
 * @license New BSD
 */
class Foo
{

    protected $_bar = 'baz';

    public $baz = 'bat';

    const bat = 'foobarbazbat';

    /**
     * Set the bar property
     *
     * @param string bar
     * @return string
     */
    public function setBar($bar)
    {
        $this->_bar = $bar;
        return $this;
    }

    /**
     * Retrieve the bar property
     *
     * @return string|null
     */
    public function getBar()
    {
        return $this->_bar;
    }

}

define('APPLICATION_ENV', 'testing');
```

Example 67. Seeding PHP file code generation via reflection

You can add PHP code to an existing PHP file using the code generator. To do so, you need to first do reflection on it. The static method `fromReflectedFileName()` allows you to do this.

```
$generator = Zend_CodeGenerator_Php_File::fromReflectedFileName($path);
$body = $generator->getBody();
$body .= "\n\$foo->bar();";
file_put_contents($path, $generator->generate());
```

Example 68. Seeding PHP class generation via reflection

You may add code to an existing class. To do so, first use the static `fromReflection()` method to map the class into a generator object. From there, you may add additional properties or methods, and then regenerate the class.

```
$generator = Zend_CodeGenerator_Php_Class::fromReflection(
    new Zend_Reflection_Class($class)
);
$generator->setMethod(array(
    'name'       => 'setBaz',
    'parameters' => array(
        array('name' => 'baz'),
    ),
    'body'       => '$this->_baz = $baz;' . "\n" . 'return $this;',
    'docblock'   => new Zend_CodeGenerator_Php_Docblock(array(
        'shortDescription' => 'Set the baz property',
        'tags'             => array(
            new Zend_CodeGenerator_Php_Docblock_Tag_Param(array(
                'paramName' => 'baz',
                'datatype'  => 'string'
            )),
            new Zend_CodeGenerator_Php_Docblock_Tag_Return(array(
                'datatype'  => 'string',
            )),
        ),
    )),
));
$code = $generator->generate();
```

3. Zend_CodeGenerator Reference

3.1. Abstract Classes and Interfaces

3.1.1. Zend_CodeGenerator_Abstract

The base class from which all CodeGenerator classes inherit provides the minimal functionality necessary. It's API is as follows:

```
abstract class Zend_CodeGenerator_Abstract
{
    final public function __construct(Array $options = array())
    public function setOptions(Array $options)
    public function setSourceContent($sourceContent)
```

```
    public function getSourceContent()
    protected function _init()
    protected function _prepare()
    abstract public function generate();
    final public function __toString()
}
```

The constructor first calls `_init()` (which is left empty for the concrete extending class to implement), then passes the `$options` parameter to `setOptions()`, and finally calls `_prepare()` (again, to be implemented by an extending class).

Like most classes in Zend Framework, `setOptions()` compares an option key to existing setters in the class, and passes the value on to that method if found.

`__toString()` is marked as final, and proxies to `generate()`.

`setSourceContent()` and `getSourceContent()` are intended to either set the default content for the code being generated, or to replace said content once all generation tasks are complete.

3.1.2. Zend_CodeGenerator_Php_Abstract

`Zend_CodeGenerator_Php_Abstract` extends `Zend_CodeGenerator_Abstract`, and adds some properties for tracking whether content has changed as well as the amount of indentation that should appear before generated content. Its API is as follows:

```
abstract class Zend_CodeGenerator_Php_Abstract
    extends Zend_CodeGenerator_Abstract
{
    public function setSourceDirty($isSourceDirty = true)
    public function isSourceDirty()
    public function setIndentation($indentation)
    public function getIndentation()
}
```

3.1.3. Zend_CodeGenerator_Php_Member_Abstract

`Zend_CodeGenerator_Php_Member_Abstract` is a base class for generating class members -- properties and methods -- and provides accessors and mutators for establishing visibility; whether or not the member is abstract, static, or final; and the name of the member. Its API is as follows:

```
abstract class Zend_CodeGenerator_Php_Member_Abstract
    extends Zend_CodeGenerator_Php_Abstract
{
    public function setAbstract($isAbstract)
    public function isAbstract()
    public function setStatic($isStatic)
    public function isStatic()
    public function setVisibility($visibility)
    public function getVisibility()
    public function setName($name)
    public function getName()
}
```

3.2. Concrete CodeGenerator Classes

3.2.1. Zend_CodeGenerator_Php_Body

`Zend_CodeGenerator_Php_Body` is intended for generating arbitrary procedural code to include within a file. As such, you simply set content for the object, and it will return that content when you invoke `generate()`.

The API of the class is as follows:

```
class Zend_CodeGenerator_Php_Body extends Zend_CodeGenerator_Php_Abstract
{
    public function setContent($content)
    public function getContent()
    public function generate()
}
```

3.2.2. Zend_CodeGenerator_Php_Class

`Zend_CodeGenerator_Php_Class` is intended for generating PHP classes. The basic functionality just generates the PHP class itself, as well as optionally the related PHP DocBlock. Classes may implement or inherit from other classes, and may be marked as abstract. Utilizing other code generator classes, you can also attach class constants, properties, and methods.

The API is as follows:

```
class Zend_CodeGenerator_Php_Class extends Zend_CodeGenerator_Php_Abstract
{
    public static function fromReflection(
        Zend_Reflection_Class $reflectionClass
    )
    public function setDocblock(Zend_CodeGenerator_Php_Docblock $docblock)
    public function getDocblock()
    public function setName($name)
    public function getName()
    public function setAbstract($isAbstract)
    public function isAbstract()
    public function setExtendedClass($extendedClass)
    public function getExtendedClass()
    public function setImplementedInterfaces(Array $implementedInterfaces)
    public function getImplementedInterfaces()
    public function setProperties(Array $properties)
    public function setProperty($property)
    public function getProperties()
    public function getProperty($propertyName)
    public function setMethods(Array $methods)
    public function setMethod($method)
    public function getMethods()
    public function getMethod($methodName)
    public function hasMethod($methodName)
    public function isSourceDirty()
    public function generate()
}
```

The `setProperty()` method accepts an array of information that may be used to generate a `Zend_CodeGenerator_Php_Property` instance -- or simply an instance of `Zend_CodeGenerator_Php_Property`. Likewise, `setMethod()` accepts either an array

of information for generating a `Zend_CodeGenerator_Php_Method` instance or a concrete instance of that class.

Note that `setDocBlock()` expects an instance of `Zend_CodeGenerator_Php_DocBlock`.

3.2.3. Zend_CodeGenerator_Php_Docblock

`Zend_CodeGenerator_Php_Docblock` can be used to generate arbitrary PHP docblocks, including all the standard docblock features: short and long descriptions and annotation tags.

Annotation tags may be set using the `setTag()` and `setTags()` methods; these each take either an array describing the tag that may be passed to the `Zend_CodeGenerator_Php_Docblock_Tag` constructor, or an instance of that class.

The API is as follows:

```
class Zend_CodeGenerator_Php_Docblock extends Zend_CodeGenerator_Php_Abstract
{
    public static function fromReflection(
        Zend_Reflection_Docblock $reflectionDocblock
    )
    public function setShortDescription($shortDescription)
    public function getShortDescription()
    public function setLongDescription($longDescription)
    public function getLongDescription()
    public function setTags(Array $tags)
    public function setTag($tag)
    public function getTags()
    public function generate()
}
```

3.2.4. Zend_CodeGenerator_Php_Docblock_Tag

`Zend_CodeGenerator_Php_Docblock_Tag` is intended for creating arbitrary annotation tags for inclusion in PHP docblocks. Tags are expected to contain a name (the portion immediately following the '@' symbol) and a description (everything following the tag name).

The class API is as follows:

```
class Zend_CodeGenerator_Php_Docblock_Tag
    extends Zend_CodeGenerator_Php_Abstract
{
    public static function fromReflection(
        Zend_Reflection_Docblock_Tag $reflectionTag
    )
    public function setName($name)
    public function getName()
    public function setDescription($description)
    public function getDescription()
    public function generate()
}
```

3.2.5. Zend_CodeGenerator_Php_DocBlock_Tag_Param

`Zend_CodeGenerator_Php_DocBlock_Tag_Param` is a specialized version of `Zend_CodeGenerator_Php_DocBlock_Tag`, and represents a method parameter. The tag

name is therefor known ("param"), but due to the format of this annotation tag, additional information is required in order to generate it: the parameter name and data type it represents.

The class API is as follows:

```
class Zend_CodeGenerator_Php_Docblock_Tag_Param
    extends Zend_CodeGenerator_Php_Docblock_Tag
{
    public static function fromReflection(
        Zend_Reflection_Docblock_Tag $reflectionTagParam
    )
    public function setDatatype($datatype)
    public function getDatatype()
    public function setParamName($paramName)
    public function getParamName()
    public function generate()
}
```

3.2.6. Zend_CodeGenerator_Php_DocBlock_Tag_Return

Like the param docblock tag variant, `Zend_CodeGenerator_Php_Docblock_Tab_Return` is an annotation tag variant for representing a method return value. In this case, the annotation tag name is known ("return"), but requires a return type.

The class API is as follows:

```
class Zend_CodeGenerator_Php_Docblock_Tag_Param
    extends Zend_CodeGenerator_Php_Docblock_Tag
{
    public static function fromReflection(
        Zend_Reflection_Docblock_Tag $reflectionTagReturn
    )
    public function setDatatype($datatype)
    public function getDatatype()
    public function generate()
}
```

3.2.7. Zend_CodeGenerator_Php_File

`Zend_CodeGenerator_Php_File` is used to generate the full contents of a file that will contain PHP code. The file may contain classes or arbitrary PHP code, as well as a file-level docblock if desired.

When adding classes to the file, you will need to pass either an array of information to pass to the `Zend_CodeGenerator_Php_Class` constructor, or an instance of that class. Similarly, with docblocks, you will need to pass information for the `Zend_CodeGenerator_Php_Docblock` constructor to consume or an instance of the class.

The API of the class is as follows:

```
class Zend_CodeGenerator_Php_File extends Zend_CodeGenerator_Php_Abstract
{
    public static function fromReflectedFilePath(
        $filePath,
        $usePreviousCodeGeneratorIfItExists = true,
        $includeIfNotAlreadyIncluded = true)
    public static function fromReflection(Zend_Reflection_File $reflectionFile)
```

```
    public function setDocblock(Zend_CodeGenerator_Php_Docblock $docblock)
    public function getDocblock()
    public function setRequiredFiles($requiredFiles)
    public function getRequiredFiles()
    public function setClasses(Array $classes)
    public function getClass($name = null)
    public function setClass($class)
    public function setFilename($filename)
    public function getFilename()
    public function getClasses()
    public function setBody($body)
    public function getBody()
    public function isSourceDirty()
    public function generate()
}
```

3.2.8. Zend_CodeGenerator_Php_Member_Container

`Zend_CodeGenerator_Php_Member_Container` is used internally by `Zend_CodeGenerator_Php_Class` to keep track of class members -- properties and methods alike. These are indexed by name, using the concrete instances of the members as values.

The API of the class is as follows:

```
class Zend_CodeGenerator_Php_Member_Container extends ArrayObject
{
    public function __construct($type = self::TYPE_PROPERTY)
}
```

3.2.9. Zend_CodeGenerator_Php_Method

`Zend_CodeGenerator_Php_Method` describes a class method, and can generate both the code and the docblock for the method. The visibility and status as static, abstract, or final may be indicated, per its parent class, `Zend_CodeGenerator_Php_Member_Abstract`. Finally, the parameters and return value for the method may be specified.

Parameters may be set using `setParameter()` or `setParameters()`. In each case, a parameter should either be an array of information to pass to the `Zend_CodeGenerator_Php_Parameter` constructor or an instance of that class.

The API of the class is as follows:

```
class Zend_CodeGenerator_Php_Method
    extends Zend_CodeGenerator_Php_Member_Abstract
{
    public static function fromReflection(
        Zend_Reflection_Method $reflectionMethod
    )
    public function setDocblock(Zend_CodeGenerator_Php_Docblock $docblock)
    public function getDocblock()
    public function setFinal($isFinal)
    public function setParameters(Array $parameters)
    public function setParameter($parameter)
    public function getParameters()
    public function setBody($body)
    public function getBody()
    public function generate()
```

```
}
```

3.2.10. Zend_CodeGenerator_Php_Parameter

`Zend_CodeGenerator_Php_Parameter` may be used to specify method parameters. Each parameter may have a position (if unspecified, the order in which they are registered with the method will be used), a default value, and a data type; a parameter name is required.

The API of the class is as follows:

```
class Zend_CodeGenerator_Php_Parameter extends Zend_CodeGenerator_Php_Abstract
{
    public static function fromReflection(
        Zend_Reflection_Parameter $reflectionParameter
    )
    public function setType($type)
    public function getType()
    public function setName($name)
    public function getName()
    public function setDefaultValue($defaultValue)
    public function getDefaultValue()
    public function setPosition($position)
    public function getPosition()
    public function getPassedByReference()
    public function setPassedByReference($passedByReference)
    public function generate()
}
```

There are several problems that might occur when trying to set `NULL`, booleans or arrays as default values. For this the value holder object `Zend_CodeGenerator_Php_ParameterDefaultValue` can be used, for example:

```
$parameter = new Zend_CodeGenerator_Php_Parameter();
$parameter->setDefaultValue(
    new Zend_CodeGenerator_Php_Parameter_DefaultValue("null")
);
$parameter->setDefaultValue(
    new Zend_CodeGenerator_Php_Parameter_DefaultValue("array('foo', 'bar')")
);
```

Internally `setDefaultValue()` also converts the values which can't be expressed in PHP into the value holder.

3.2.11. Zend_CodeGenerator_Php_Property

`Zend_CodeGenerator_Php_Property` describes a class property, which may be either a constant or a variable. In each case, the property may have an optional default value associated with it. Additionally, the visibility of variable properties may be set, per the parent class, `Zend_CodeGenerator_Php_Member_Abstract`.

The API of the class is as follows:

```
class Zend_CodeGenerator_Php_Property
    extends Zend_CodeGenerator_Php_Member_Abstract
{
    public static function fromReflection(
        Zend_Reflection_Property $reflectionProperty
```

```
    )
    public function setConst($const)
    public function isConst()
    public function setDefaultValue($defaultValue)
    public function getDefaultValue()
    public function generate()
}
```

Zend_Config

1. Introduction

`Zend_Config` is designed to simplify the access to, and the use of, configuration data within applications. It provides a nested object property based user interface for accessing this configuration data within application code. The configuration data may come from a variety of media supporting hierarchical data storage. Currently `Zend_Config` provides adapters for configuration data that are stored in text files with `Zend_Config_Ini` and `Zend_Config_Xml`.

Example 69. Using Zend_Config

Normally it is expected that users would use one of the adapter classes such as `Zend_Config_Ini` or `Zend_Config_Xml`, but if configuration data are available in a PHP array, one may simply pass the data to the `Zend_Config` constructor in order to utilize a simple object-oriented interface:

```
// Given an array of configuration data
$configArray = array(
    'webhost'  => 'www.example.com',
    'database' => array(
        'adapter' => 'pdo_mysql',
        'params'  => array(
            'host'     => 'db.example.com',
            'username' => 'dbuser',
            'password' => 'secret',
            'dbname'   => 'mydatabase'
        )
    )
);

// Create the object-oriented wrapper upon the configuration data
$config = new Zend_Config($configArray);

// Print a configuration datum (results in 'www.example.com')
echo $config->webhost;

// Use the configuration data to connect to the database
$db = Zend_Db::factory($config->database->adapter,
                       $config->database->params->toArray());

// Alternative usage: simply pass the Zend_Config object.
// The Zend_Db factory knows how to interpret it.
$db = Zend_Db::factory($config->database);
```

As illustrated in the example above, `Zend_Config` provides nested object property syntax to access configuration data passed to its constructor.

Along with the object oriented access to the data values, `Zend_Config` also has `get()` which will return the supplied default value if the data element doesn't exist. For example:

```
$host = $config->database->get('host', 'localhost');
```

Example 70. Using Zend_Config with a PHP Configuration File

It is often desirable to use a pure PHP-based configuration file. The following code illustrates how easily this can be accomplished:

```
// config.php
return array(
    'webhost'  => 'www.example.com',
    'database' => array(
        'adapter' => 'pdo_mysql',
        'params'  => array(
            'host'     => 'db.example.com',
            'username' => 'dbuser',
            'password' => 'secret',
            'dbname'   => 'mydatabase'
        )
    )
);
```

```
// Configuration consumption
$config = new Zend_Config(require 'config.php');

// Print a configuration datum (results in 'www.example.com')
echo $config->webhost;
```

2. Theory of Operation

Configuration data are made accessible to the `Zend_Config` constructor through an associative array, which may be multi-dimensional, in order to support organizing the data from general to specific. Concrete adapter classes adapt configuration data from storage to produce the associative array for the `Zend_Config` constructor. User scripts may provide such arrays directly to the `Zend_Config` constructor, without using an adapter class, since it may be appropriate to do so in certain situations.

Each configuration data array value becomes a property of the `Zend_Config` object. The key is used as the property name. If a value is itself an array, then the resulting object property is created as a new `Zend_Config` object, loaded with the array data. This occurs recursively, such that a hierarchy of configuration data may be created with any number of levels.

`Zend_Config` implements the *Countable* and *Iterator* interfaces in order to facilitate simple access to configuration data. Thus, one may use the `count()` function and PHP constructs such as *foreach* with `Zend_Config` objects.

By default, configuration data made available through `Zend_Config` are read-only, and an assignment (e.g., **$config->database->host = 'example.com';**) results in a thrown exception. This default behavior may be overridden through the constructor, however, to allow modification of data values. Also, when modifications are allowed, `Zend_Config` supports unsetting of values (i.e. `unset($config->database->host)`). The `readOnly()` method can be used to determine if modifications to a given `Zend_Config` object are allowed and the `setReadOnly()` method can be used to stop any further modifications to a `Zend_Config` object that was created allowing modifications.

It is important not to confuse such in-memory modifications with saving configuration data out to specific storage media. Tools for creating and modifying configuration data for various storage media are out of scope with respect to

Zend_Config. Third-party open source solutions are readily available for the purpose of creating and modifying configuration data for various storage media.

Adapter classes inherit from the Zend_Config class since they utilize its functionality.

The Zend_Config family of classes enables configuration data to be organized into sections. Zend_Config adapter objects may be loaded with a single specified section, multiple specified sections, or all sections (if none are specified).

Zend_Config adapter classes support a single inheritance model that enables configuration data to be inherited from one section of configuration data into another. This is provided in order to reduce or eliminate the need for duplicating configuration data for different purposes. An inheriting section may also override the values that it inherits through its parent section. Like PHP class inheritance, a section may inherit from a parent section, which may inherit from a grandparent section, and so on, but multiple inheritance (i.e., section C inheriting directly from parent sections A and B) is not supported.

If you have two Zend_Config objects, you can merge them into a single object using the merge() function. For example, given $config and $localConfig, you can merge data from $localConfig to $config using **$config->merge($localConfig);**. The items in $localConfig will override any items with the same name in $config.

The Zend_Config object that is performing the merge must have been constructed to allow modifications, by passing TRUE as the second parameter of the constructor. The setReadOnly() method can then be used to prevent any further modifications after the merge is complete.

3. Zend_Config_Ini

Zend_Config_Ini enables developers to store configuration data in a familiar INI format and read them in the application by using nested object property syntax. The INI format is specialized to provide both the ability to have a hierarchy of configuration data keys and inheritance between configuration data sections. Configuration data hierarchies are supported by separating the keys with the dot or period character ("."). A section may extend or inherit from another section by following the section name with a colon character (":") and the name of the section from which data are to be inherited.

Parsing the INI File

Zend_Config_Ini utilizes the parse_ini_file() PHP function. Please review this documentation to be aware of its specific behaviors, which propagate to Zend_Config_Ini, such as how the special values of "TRUE", "FALSE", "yes", "no", and "NULL" are handled.

Key Separator

By default, the key separator character is the period character ("."). This can be changed, however, by changing the $options key nestSeparator when constructing the Zend_Config_Ini object. For example:

```
$options['nestSeparator'] = ':';
$config = new Zend_Config_Ini('/path/to/config.ini',
                              'staging',
```

```
                                    $options);
```

Example 71. Using Zend_Config_Ini

This example illustrates a basic use of `Zend_Config_Ini` for loading configuration data from an INI file. In this example there are configuration data for both a production system and for a staging system. Because the staging system configuration data are very similar to those for production, the staging section inherits from the production section. In this case, the decision is arbitrary and could have been written conversely, with the production section inheriting from the staging section, though this may not be the case for more complex situations. Suppose, then, that the following configuration data are contained in `/path/to/config.ini`:

```
; Production site configuration data
[production]
webhost                  = www.example.com
database.adapter         = pdo_mysql
database.params.host     = db.example.com
database.params.username = dbuser
database.params.password = secret
database.params.dbname   = dbname

; Staging site configuration data inherits from production and
; overrides values as necessary
[staging : production]
database.params.host     = dev.example.com
database.params.username = devuser
database.params.password = devsecret
```

Next, assume that the application developer needs the staging configuration data from the INI file. It is a simple matter to load these data by specifying the INI file and the staging section:

```
$config = new Zend_Config_Ini('/path/to/config.ini', 'staging');

echo $config->database->params->host;   // prints "dev.example.com"
echo $config->database->params->dbname; // prints "dbname"
```

Table 29. Zend_Config_Ini Constructor Parameters

Parameter	Notes
`$filename`	The INI file to load.
`$section`	The [section] within the ini file that is to be loaded. Setting this parameter to `NULL` will load all sections. Alternatively, an array of section names may be supplied to load multiple sections.
`$options` (default FALSE)	Options array. The following keys are supported: • *allowModifications*: Set to `TRUE` to allow subsequent modification

Parameter	Notes
	of loaded configuration data in-memory. Defaults to `NULL` • *nestSeparator*: Set to the character to be used as the nest separator. Defaults to "."

4. Zend_Config_Xml

`Zend_Config_Xml` enables developers to store configuration data in a simple XML format and read them via nested object property syntax. The root element of the XML file or string is irrelevant and may be named arbitrarily. The first level of XML elements correspond with configuration data sections. The XML format supports hierarchical organization through nesting of XML elements below the section-level elements. The content of a leaf-level XML element corresponds to the value of a configuration datum. Section inheritance is supported by a special XML attribute named *extends*, and the value of this attribute corresponds with the section from which data are to be inherited by the extending section.

Return Type

Configuration data read into `Zend_Config_Xml` are always returned as strings. Conversion of data from strings to other types is left to developers to suit their particular needs.

Example 72. Using Zend_Config_Xml

This example illustrates a basic use of `Zend_Config_Xml` for loading configuration data from an XML file. In this example there are configuration data for both a production system and for a staging system. Because the staging system configuration data are very similar to those for production, the staging section inherits from the production section. In this case, the decision is arbitrary and could have been written conversely, with the production section inheriting from the staging section, though this may not be the case for more complex situations. Suppose, then, that the following configuration data are contained in `/path/to/config.xml`:

```
<?xml version="1.0"?>
<configdata>
    <production>
        <webhost>www.example.com</webhost>
        <database>
            <adapter>pdo_mysql</adapter>
            <params>
                <host>db.example.com</host>
                <username>dbuser</username>
                <password>secret</password>
                <dbname>dbname</dbname>
            </params>
        </database>
    </production>
    <staging extends="production">
        <database>
            <params>
                <host>dev.example.com</host>
                <username>devuser</username>
                <password>devsecret</password>
            </params>
        </database>
    </staging>
</configdata>
```

Next, assume that the application developer needs the staging configuration data from the XML file. It is a simple matter to load these data by specifying the XML file and the staging section:

```
$config = new Zend_Config_Xml('/path/to/config.xml', 'staging');

echo $config->database->params->host;   // prints "dev.example.com"
echo $config->database->params->dbname; // prints "dbname"
```

Example 73. Using Tag Attributes in Zend_Config_Xml

`Zend_Config_Xml` also supports two additional ways of defining nodes in the configuration. Both make use of attributes. Since the *extends* and the *value* attributes are reserved keywords (the latter one by the second way of using attributes), they may not be used. The first way of making usage of attributes is to add attributes in a parent node, which then will be translated into children of that node:

```
<?xml version="1.0"?>
<configdata>
    <production webhost="www.example.com">
        <database adapter="pdo_mysql">
            <params host="db.example.com" username="dbuser" password="secret"
                    dbname="dbname"/>
        </database>
    </production>
    <staging extends="production">
        <database>
            <params host="dev.example.com" username="devuser"
                    password="devsecret"/>
        </database>
    </staging>
</configdata>
```

The other way does not really shorten the config, but keeps it easier to maintain since you don't have to write the tag name twice. You simply create an empty tag with the value in the *value* attribute:

```
<?xml version="1.0"?>
<configdata>
    <production>
        <webhost>www.example.com</webhost>
        <database>
            <adapter value="pdo_mysql"/>
            <params>
                <host value="db.example.com"/>
                <username value="dbuser"/>
                <password value="secret"/>
                <dbname value="dbname"/>
            </params>
        </database>
    </production>
    <staging extends="production">
        <database>
            <params>
                <host value="dev.example.com"/>
                <username value="devuser"/>
                <password value="devsecret"/>
            </params>
        </database>
    </staging>
</configdata>
```

XML strings

`Zend_Config_Xml` is able to load an XML string directly, such as that retrieved from a database. The string is passed as the first parameter to the constructor and must start with the characters *'<?xml'*:

```
$string = <<<EOT
<?xml version="1.0"?>
<config>
    <production>
        <db>
            <adapter value="pdo_mysql"/>
            <params>
                <host value="db.example.com"/>
            </params>
        </db>
    </production>
    <staging extends="production">
        <db>
            <params>
                <host value="dev.example.com"/>
            </params>
        </db>
    </staging>
</config>
EOT;

$config = new Zend_Config_Xml($string, 'staging');
```

Zend_Config XML namespace

`Zend_Config` comes with it's own XML namespace, which adds additional functionality to the parsing process. To take advantage of it, you have to define a namespace with the namespace URI `http://framework.zend.com/xml/zend-config-xml/1.0/` in your config root node.

With the namespace enabled, you can now use PHP constants within your configuration files. Additionally, the *extends* attribute was moved to the new namespace and is deprecated in the `NULL` namespace. It will be completely removed there in Zend Framework 2.0.

```
$string = <<<EOT
<?xml version="1.0"?>
<config xmlns:zf="http://framework.zend.com/xml/zend-config-xml/1.0/">
    <production>
        <includePath>
            <zf:const zf:name="APPLICATION_PATH"/>/library</includePath>
        <db>
            <adapter value="pdo_mysql"/>
            <params>
                <host value="db.example.com"/>
            </params>
        </db>
    </production>
    <staging zf:extends="production">
        <db>
            <params>
                <host value="dev.example.com"/>
            </params>
        </db>
    </staging>
</config>
```

```
EOT;

define('APPLICATION_PATH', dirname(__FILE__));
$config = new Zend_Config_Xml($string, 'staging');

echo $config->includePath; // Prints "/var/www/something/library"
```

Zend_Config_Writer

1. Zend_Config_Writer

`Zend_Config_Writer` gives you the ability to write config files out of `Zend_Config` objects. It works with an adapter-less system and thus is very easy to use. By default `Zend_Config_Writer` ships with three adapters, which are all file-based. You instantiate a writer with specific options, which can be *filename* and *config*. Then you call the `write()` method of the writer and the config file is created. You can also give `$filename` and `$config` directly to the `write()` method. Currently the following writers are shipped with `Zend_Config_Writer`:

- `Zend_Config_Writer_Array`
- `Zend_Config_Writer_Ini`
- `Zend_Config_Writer_Xml`

The INI writer has two modes for rendering with regard to sections. By default the top-level configuration is always written into section names. By calling **$writer->setRenderWithoutSections();** all options are written into the global namespace of the INI file and no sections are applied.

As an addition `Zend_Config_Writer_Ini` has an additional option parameter *nestSeparator*, which defines with which character the single nodes are separated. The default is a single dot, like it is accepted by `Zend_Config_Ini` by default.

When modifying or creating a `Zend_Config` object, there are some things to know. To create or modify a value, you simply say set the parameter of the `Zend_Config` object via the parameter accessor (->). To create a section in the root or to create a branch, you just create a new array ("**$config->branch = array();**"). To define which section extends another one, you call the `setExtend()` method on the root `Zend_Config` object.

Example 74. Using Zend_Config_Writer

This example illustrates the basic use of `Zend_Config_Writer_Xml` to create a new config file:

```
// Create the config object
$config = new Zend_Config(array(), true);
$config->production = array();
$config->staging    = array();

$config->setExtend('staging', 'production');

$config->production->db = array();
$config->production->db->hostname = 'localhost';
$config->production->db->username = 'production';

$config->staging->db = array();
$config->staging->db->username = 'staging';

// Write the config file in one of the following ways:
// a)
$writer = new Zend_Config_Writer_Xml(array('config'   => $config,
                                           'filename' => 'config.xml'));
$writer->write();

// b)
$writer = new Zend_Config_Writer_Xml();
$writer->setConfig($config)
       ->setFilename('config.xml')
       ->write();

// c)
$writer = new Zend_Config_Writer_Xml();
$writer->write('config.xml', $config);
```

This will create an XML config file with the sections production and staging, where staging extends production.

Example 75. Modifying an Existing Config

This example demonstrates how to edit an existing config file.

```
// Load all sections from an existing config file, while skipping the extends.
$config = new Zend_Config_Ini('config.ini',
                              null,
                              array('skipExtends'        => true,
                                    'allowModifications' => true));

// Modify a value
$config->production->hostname = 'foobar';

// Write the config file
$writer = new Zend_Config_Writer_Ini(array('config'   => $config,
                                           'filename' => 'config.ini'));
$writer->write();
```

Loading a Config File

When loading an existing config file for modifications it is very important to load all sections and to skip the extends, so that no values are merged. This is done by giving the *skipExtends* as option to the constructor.

For all the File-Based writers (INI, XML and PHP Array) internally the `render()` is used to build the configuration string. This method can be used from the outside also if you need to access the string-representation of the configuration data.

Zend_Console_Getopt

1. Introduction

The `Zend_Console_Getopt` class helps command-line applications to parse their options and arguments.

Users may specify command-line arguments when they execute your application. These arguments have meaning to the application, to change the behavior in some way, or choose resources, or specify parameters. Many options have developed customary meaning, for example **--verbose** enables extra output from many applications. Other options may have a meaning that is different for each application. For example, **-c** enables different features in **grep**, **ls**, and **tar**.

Below are a few definitions of terms. Common usage of the terms varies, but this documentation will use the definitions below.

- "argument": a string that occurs on the command-line following the name of the command. Arguments may be options or else may appear without an option, to name resources on which the command operates.

- "option": an argument that signifies that the command should change its default behavior in some way.

- "flag": the first part of an option, identifies the purpose of the option. A flag is preceded conventionally by one or two dashes (**-** or **--**). A single dash precedes a single-character flag or a cluster of single-character flags. A double-dash precedes a multi-character flag. Long flags cannot be clustered.

- "parameter": the secondary part of an option; a data value that may accompany a flag, if it is applicable to the given option. For example, many commands accept a **--verbose** option, but typically this option has no parameter. However, an option like **--user** almost always requires a following parameter.

  A parameter may be given as a separate argument following a flag argument, or as part of the same argument string, separated from the flag by an equals symbol (**=**). The latter form is supported only by long flags. For example, **-u username**, **--user username**, and **--user=username** are forms supported by `Zend_Console_Getopt`.

- "cluster": multiple single-character flags combined in a single string argument and preceded by a single dash. For example, "**ls -1str**" uses a cluster of four short flags. This command is equivalent to "**ls -1 -s -t -r**". Only single-character flags can be clustered. You cannot make a cluster of long flags.

For example, in **mysql --user=root mydatabase**, **mysql** is a *command*, **--user=root** is an *option*, **--user** is a *flag*, **root** is a *parameter* to the option, and **mydatabase** is an argument but not an option by our definition.

`Zend_Console_Getopt` provides an interface to declare which flags are valid for your application, output an error and usage message if they use an invalid flag, and report to your application code which flags the user specified.

Getopt is not an Application Framework

`Zend_Console_Getopt` does *not* interpret the meaning of flags and parameters, nor does this class implement application workflow or invoke application code. You must implement those actions in your own application code. You can use the `Zend_Console_Getopt` class to parse the command-line and provide object-oriented methods for querying which options were given by a user, but code to use this information to invoke parts of your application should be in another PHP class.

The following sections describe usage of `Zend_Console_Getopt`.

2. Declaring Getopt Rules

The constructor for the `Zend_Console_Getopt` class takes from one to three arguments. The first argument declares which options are supported by your application. This class supports alternative syntax forms for declaring the options. See the sections below for the format and usage of these syntax forms.

The constructor takes two more arguments, both of which are optional. The second argument may contain the command-line arguments. This defaults to `$_SERVER['argv']`.

The third argument of the constructor may contain an configuration options to customize the behavior of `Zend_Console_Getopt`. See Adding Configuration for reference on the options available.

2.1. Declaring Options with the Short Syntax

`Zend_Console_Getopt` supports a compact syntax similar to that used by GNU Getopt (see http://www.gnu.org/software/libc/manual/html_node/Getopt.html. This syntax supports only single-character flags. In a single string, you type each of the letters that correspond to flags supported by your application. A letter followed by a colon character (:) indicates a flag that requires a parameter.

Example 76. Using the Short Syntax

```
$opts = new Zend_Console_Getopt('abp:');
```

The example above shows using `Zend_Console_Getopt` to declare that options may be given as **-a**, **-b**, or **-p**. The latter flag requires a parameter.

The short syntax is limited to flags of a single character. Aliases, parameter types, and help strings are not supported in the short syntax.

2.2. Declaring Options with the Long Syntax

A different syntax with more features is also available. This syntax allows you to specify aliases for flags, types of option parameters, and also help strings to describe usage to the user. Instead of the single string used in the short syntax to declare the options, the long syntax uses an associative array as the first argument to the constructor.

The key of each element of the associative array is a string with a format that names the flag, with any aliases, separated by the pipe symbol ("|"). Following this series of flag aliases, if the option requires a parameter, is an equals symbol ("=") with a letter that stands for the *type* of the parameter:

- "=*s*" for a string parameter
- "=*w*" for a word parameter (a string containing no whitespace)
- "=*i*" for an integer parameter

If the parameter is optional, use a dash ("-") instead of the equals symbol.

The value of each element in the associative array is a help string to describe to a user how to use your program.

Example 77. Using the Long Syntax

```
$opts = new Zend_Console_Getopt(
  array(
    'apple|a'    => 'apple option, with no parameter',
    'banana|b=i' => 'banana option, with required integer parameter',
    'pear|p-s'   => 'pear option, with optional string parameter'
  )
);
```

In the example declaration above, there are three options. **--apple** and **-a** are aliases for each other, and the option takes no parameter. **--banana** and **-b** are aliases for each other, and the option takes a mandatory integer parameter. Finally, **--pear** and **-p** are aliases for each other, and the option may take an optional string parameter.

3. Fetching Options and Arguments

After you have declared the options that the `Zend_Console_Getopt` object should recognize, and supply arguments from the command-line or an array, you can query the object to find out which options were specified by a user in a given command-line invocation of your program. The class implements magic methods so you can query for options by name.

The parsing of the data is deferred until the first query you make against the `Zend_Console_Getopt` object to find out if an option was given, the object performs its parsing. This allows you to use several method calls to configure the options, arguments, help strings, and configuration options before parsing takes place.

3.1. Handling Getopt Exceptions

If the user gave any invalid options on the command-line, the parsing function throws a `Zend_Console_Getopt_Exception`. You should catch this exception in your application code. You can use the `parse()` method to force the object to parse the arguments. This is useful because you can invoke `parse()` in a *try* block. If it passes, you can be sure that the parsing won't throw an exception again. The exception thrown has a custom method `getUsageMessage()`, which returns as a string the formatted set of usage messages for all declared options.

Example 78. Catching Getopt Exceptions

```
try {
    $opts = new Zend_Console_Getopt('abp:');
    $opts->parse();
} catch (Zend_Console_Getopt_Exception $e) {
    echo $e->getUsageMessage();
    exit;
}
```

Cases where parsing throws an exception include:

- Option given is not recognized.
- Option requires a parameter but none was given.
- Option parameter is of the wrong type. E.g. a non-numeric string when an integer was required.

3.2. Fetching Options by Name

You can use the `getOption()` method to query the value of an option. If the option had a parameter, this method returns the value of the parameter. If the option had no parameter but the user did specify it on the command-line, the method returns `TRUE`. Otherwise the method returns `NULL`.

Example 79. Using getOption()

```
$opts = new Zend_Console_Getopt('abp:');
$b = $opts->getOption('b');
$p_parameter = $opts->getOption('p');
```

Alternatively, you can use the magic `__get()` function to retrieve the value of an option as if it were a class member variable. The `__isset()` magic method is also implemented.

Example 80. Using __get() and __isset() Magic Methods

```
$opts = new Zend_Console_Getopt('abp:');
if (isset($opts->b)) {
    echo "I got the b option.\n";
}
$p_parameter = $opts->p; // null if not set
```

If your options are declared with aliases, you may use any of the aliases for an option in the methods above.

3.3. Reporting Options

There are several methods to report the full set of options given by the user on the current command-line.

- As a string: use the `toString()` method. The options are returned as a space-separated string of **flag=value** pairs. The value of an option that does not have a parameter is the literal string "TRUE".
- As an array: use the `toArray()` method. The options are returned in a simple integer-indexed array of strings, the flag strings followed by parameter strings, if any.
- As a string containing JSON data: use the `toJson()` method.
- As a string containing XML data: use the `toXml()` method.

In all of the above dumping methods, the flag string is the first string in the corresponding list of aliases. For example, if the option aliases were declared like **verbose|v**, then the first string, **verbose**, is used as the canonical name of the option. The name of the option flag does not include any preceding dashes.

3.4. Fetching Non-option Arguments

After option arguments and their parameters have been parsed from the command-line, there may be additional arguments remaining. You can query these arguments using the `getRemainingArgs()` method. This method returns an array of the strings that were not part of any options.

Example 81. Using getRemainingArgs()

```
$opts = new Zend_Console_Getopt('abp:');
$opts->setArguments(array('-p', 'p_parameter', 'filename'));
$args = $opts->getRemainingArgs(); // returns array('filename')
```

`Zend_Console_Getopt` supports the GNU convention that an argument consisting of a double-dash signifies the end of options. Any arguments following this signifier must be treated as non-option arguments. This is useful if you might have a non-option argument that begins with a dash. For example: "**rm -- -filename-with-dash**".

4. Configuring Zend_Console_Getopt

4.1. Adding Option Rules

You can add more option rules in addition to those you specified in the `Zend_Console_Getopt` constructor, using the `addRules()` method. The argument to `addRules()` is the same as the first argument to the class constructor. It is either a string in the format of the short syntax options specification, or else an associative array in the format of a long syntax options specification. See Declaring Getopt Rules for details on the syntax for specifying options.

Example 82. Using addRules()

```
$opts = new Zend_Console_Getopt('abp:');
$opts->addRules(
  array(
    'verbose|v' => 'Print verbose output'
  )
);
```

The example above shows adding the **--verbose** option with an alias of **-v** to a set of options defined in the call to the constructor. Notice that you can mix short format options and long format options in the same instance of `Zend_Console_Getopt`.

4.2. Adding Help Messages

In addition to specifying the help strings when declaring option rules in the long format, you can associate help strings with option rules using the `setHelp()` method. The argument to the `setHelp()` method is an associative array, in which the key is a flag, and the value is a corresponding help string.

Example 83. Using setHelp()

```
$opts = new Zend_Console_Getopt('abp:');
$opts->setHelp(
    array(
        'a' => 'apple option, with no parameter',
        'b' => 'banana option, with required integer parameter',
        'p' => 'pear option, with optional string parameter'
    )
);
```

If you declared options with aliases, you can use any of the aliases as the key of the associative array.

The `setHelp()` method is the only way to define help strings if you declared the options using the short syntax.

4.3. Adding Option Aliases

You can declare aliases for options using the `setAliases()` method. The argument is an associative array, whose key is a flag string declared previously, and whose value is a new alias for that flag. These aliases are merged with any existing aliases. In other words, aliases you declared earlier are still in effect.

An alias may be declared only once. If you try to redefine an alias, a `Zend_Console_Getopt_Exception` is thrown.

Example 84. Using setAliases()

```
$opts = new Zend_Console_Getopt('abp:');
$opts->setAliases(
    array(
        'a' => 'apple',
        'a' => 'apfel',
        'p' => 'pear'
    )
);
```

In the example above, after declaring these aliases, **-a**, **--apple** and **--apfel** are aliases for each other. Also **-p** and **--pear** are aliases for each other.

The `setAliases()` method is the only way to define aliases if you declared the options using the short syntax.

4.4. Adding Argument Lists

By default, `Zend_Console_Getopt` uses `$_SERVER['argv']` for the array of command-line arguments to parse. You can alternatively specify the array of arguments as the second constructor argument. Finally, you can append more arguments to those already used using the `addArguments()` method, or you can replace the current array of arguments using the `setArguments()` method. In both cases, the parameter to these methods is a simple array of strings. The former method appends the array to the current arguments, and the latter method substitutes the array for the current arguments.

Example 85. Using addArguments() and setArguments()

```
// By default, the constructor uses $_SERVER['argv']
$opts = new Zend_Console_Getopt('abp:');

// Append an array to the existing arguments
$opts->addArguments(array('-a', '-p', 'p_parameter', 'non_option_arg'));

// Substitute a new array for the existing arguments
$opts->setArguments(array('-a', '-p', 'p_parameter', 'non_option_arg'));
```

4.5. Adding Configuration

The third parameter to the `Zend_Console_Getopt` constructor is an array of configuration options that affect the behavior of the object instance returned. You can also specify configuration options using the `setOptions()` method, or you can set an individual option using the `setOption()` method.

Clarifying the Term "option"

The term "option" is used for configuration of the `Zend_Console_Getopt` class to match terminology used elsewhere in Zend Framework. These are not the same things as the command-line options that are parsed by the `Zend_Console_Getopt` class.

The currently supported options have const definitions in the class. The options, their const identifiers (with literal values in parentheses) are listed below:

- `Zend_Console_Getopt::CONFIG_DASHDASH` ("dashDash"), if `TRUE`, enables the special flag **--** to signify the end of flags. Command-line arguments following the double-dash signifier are not interpreted as options, even if the arguments start with a dash. This configuration option is `TRUE` by default.

- `Zend_Console_Getopt::CONFIG_IGNORECASE` ("ignoreCase"), if `TRUE`, makes flags aliases of each other if they differ only in their case. That is, **-a** and **-A** will be considered to be synonymous flags. This configuration option is `FALSE` by default.

- `Zend_Console_Getopt::CONFIG_RULEMODE` ("ruleMode") may have values `Zend_Console_Getopt::MODE_ZEND` ("zend") and `Zend_Console_Getopt::MODE_GNU` ("gnu"). It should not be necessary to use this option unless you extend the class with additional syntax forms. The two modes supported in the base `Zend_Console_Getopt` class are unambiguous. If the specifier is a string, the class assumes `MODE_GNU`, otherwise it assumes `MODE_ZEND`. But if you extend the class and add more syntax forms, you may need to specify the mode using this option.

More configuration options may be added as future enhancements of this class.

The two arguments to the `setOption()` method are a configuration option name and an option value.

Example 86. Using setOption()

```
$opts = new Zend_Console_Getopt('abp:');
$opts->setOption('ignoreCase', true);
```

The argument to the `setOptions()` method is an associative array. The keys of this array are the configuration option names, and the values are configuration values. This is also the array format used in the class constructor. The configuration values you specify are merged with the current configuration; you don't have to list all options.

Example 87. Using setOptions()

```
$opts = new Zend_Console_Getopt('abp:');
$opts->setOptions(
    array(
        'ignoreCase' => true,
        'dashDash'   => false
    )
);
```

Zend_Controller

1. Zend_Controller Quick Start

1.1. Introduction

`Zend_Controller` is the heart of Zend Framework's MVC system. MVC stands for Model-View-Controller and is a design pattern targeted at separating application logic from display logic. `Zend_Controller_Front` implements a Front Controller pattern, in which all requests are intercepted by the front controller and dispatched to individual Action Controllers based on the URL requested.

The `Zend_Controller` system was built with extensibility in mind, either by subclassing the existing classes, writing new classes that implement the various interfaces and abstract classes that form the foundation of the controller family of classes, or writing plugins or action helpers to augment or manipulate the functionality of the system.

1.2. Quick Start

If you need more in-depth information, see the following sections. If you just want to get up and running quickly, read on.

1.2.1. Create the Filesystem Layout

The first step is to create your file system layout. The typical layout is as follows:

```
application/
    controllers/
        IndexController.php
    models/
    views/
        scripts/
            index/
                index.phtml
        helpers/
        filters/
html/
    .htaccess
    index.php
```

1.2.2. Set the Document Root

In your web server, point your document root to the `html/` directory of the above file system layout.

1.2.3. Create the Rewrite Rules

Edit the `html/.htaccess` file above to read as follows:

```
RewriteEngine On
RewriteCond %{REQUEST_FILENAME} -s [OR]
RewriteCond %{REQUEST_FILENAME} -l [OR]
RewriteCond %{REQUEST_FILENAME} -d
RewriteRule ^.*$ - [NC,L]
RewriteRule ^.*$ index.php [NC,L]
```

Learn about mod_rewrite

The above rewrite rules allow access to any file under your virtual host's document root. If there are files you do not want exposed in this way, you may want to be more restrictive in your rules. Go to the Apache website to learn more about mod_rewrite.

If using IIS 7.0, use the following as your rewrite configuration:

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
     <system.webServer>
         <rewrite>
             <rules>
                 <rule name="Imported Rule 1" stopProcessing="true">
                     <match url="^.*$" />
                     <conditions logicalGrouping="MatchAny">
                         <add input="{REQUEST_FILENAME}"
                             matchType="IsFile" pattern=""
                             ignoreCase="false" />
                         <add input="{REQUEST_FILENAME}"
                             matchType="IsDirectory"
                             pattern="" ignoreCase="false" />
                     </conditions>
                     <action type="None" />
                 </rule>
                 <rule name="Imported Rule 2" stopProcessing="true">
                     <match url="^.*$" />
                     <action type="Rewrite" url="index.php" />
                 </rule>
             </rules>
         </rewrite>
     </system.webServer>
</configuration>
```

The above rules will route requests to existing resources (existing symlinks, non-empty files, or non-empty directories) accordingly, and all other requests to the front controller.

The above rewrite rules are for Apache; for examples of rewrite rules for other web servers, see the router documentation.

1.2.4. Create the Bootstrap File

The bootstrap file is the page all requests are routed through -- `html/index.php` in this case. Open up `html/index.php` in the editor of your choice and add the following:

```
Zend_Controller_Front::run('/path/to/app/controllers');
```

This will instantiate and dispatch the front controller, which will route requests to action controllers.

1.2.5. Create the Default Action Controller

Before discussing action controllers, you should first understand how requests are routed in Zend Framework. By default, the first segment of a URL path maps to a controller, and the

second to an action. For example, given the URL `http://framework.zend.com/roadmap/components`, the path is `/roadmap/components`, which will map to the controller *roadmap* and the action *components*. If no action is provided, the action *index* is assumed, and if no controller is provided, the controller *index* is assumed (following the Apache convention that maps a *DirectoryIndex* automatically).

`Zend_Controller`'s dispatcher then takes the controller value and maps it to a class. By default, it Title-cases the controller name and appends the word *Controller*. Thus, in our example above, the controller *roadmap* is mapped to the class `RoadmapController`.

Similarly, the action value is mapped to a method of the controller class. By default, the value is lower-cased, and the word *Action* is appended. Thus, in our example above, the action *components* becomes `componentsAction()`, and the final method called is `RoadmapController::componentsAction()`.

So, moving on, let's now create a default action controller and action method. As noted earlier, the default controller and action called are both *index*. Open the file `application/controllers/IndexController.php`, and enter the following:

```
/** Zend_Controller_Action */
class IndexController extends Zend_Controller_Action
{
    public function indexAction()
    {
    }
}
```

By default, the ViewRenderer action helper is enabled. What this means is that by simply defining an action method and a corresponding view script, you will immediately get content rendered. By default, `Zend_View` is used as the View layer in the MVC. The *ViewRenderer* does some magic, and uses the controller name (e.g., *index*) and the current action name (e.g., *index*) to determine what template to pull. By default, templates end in the `.phtml` extension, so this means that, in the above example, the template `index/index.phtml` will be rendered. Additionally, the *ViewRenderer* automatically assumes that the directory `views/` at the same level as the controller directory will be the base view directory, and that the actual view scripts will be in the `views/scripts/` subdirectory. Thus, the template rendered will be found in `application/views/scripts/index/index.phtml`.

1.2.6. Create the View Script

As mentioned in the previous section, view scripts are found in `application/views/scripts/`; the view script for the default controller and action is in `application/views/scripts/index/index.phtml`. Create this file, and type in some HTML:

```
<!DOCTYPE html
PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html>
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
  <title>My first Zend Framework App</title>
</head>
<body>
    <h1>Hello, World!</h1>
</body>
```

```
</html>
```

1.2.7. Create the Error Controller

By default, the error handler plugin is registered. This plugin expects that a controller exists to handle errors. By default, it assumes an *ErrorController* in the default module with an `errorAction()` method:

```
class ErrorController extends Zend_Controller_Action
{
    public function errorAction()
    {
    }
}
```

Assuming the already discussed directory layout, this file will go in `application/controllers/ErrorController.php`. You will also need to create a view script in `application/views/scripts/error/error.phtml`; sample content might look like:

```
<!DOCTYPE html
PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html>
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
  <title>Error</title>
</head>
<body>
    <h1>An error occurred</h1>
    <p>An error occurred; please try again later.</p>
</body>
</html>
```

1.2.8. View the Site!

With your first controller and view under your belt, you can now fire up your browser and browse to the site. Assuming `example.com` is your domain, any of the following URLs will get to the page we've just created:

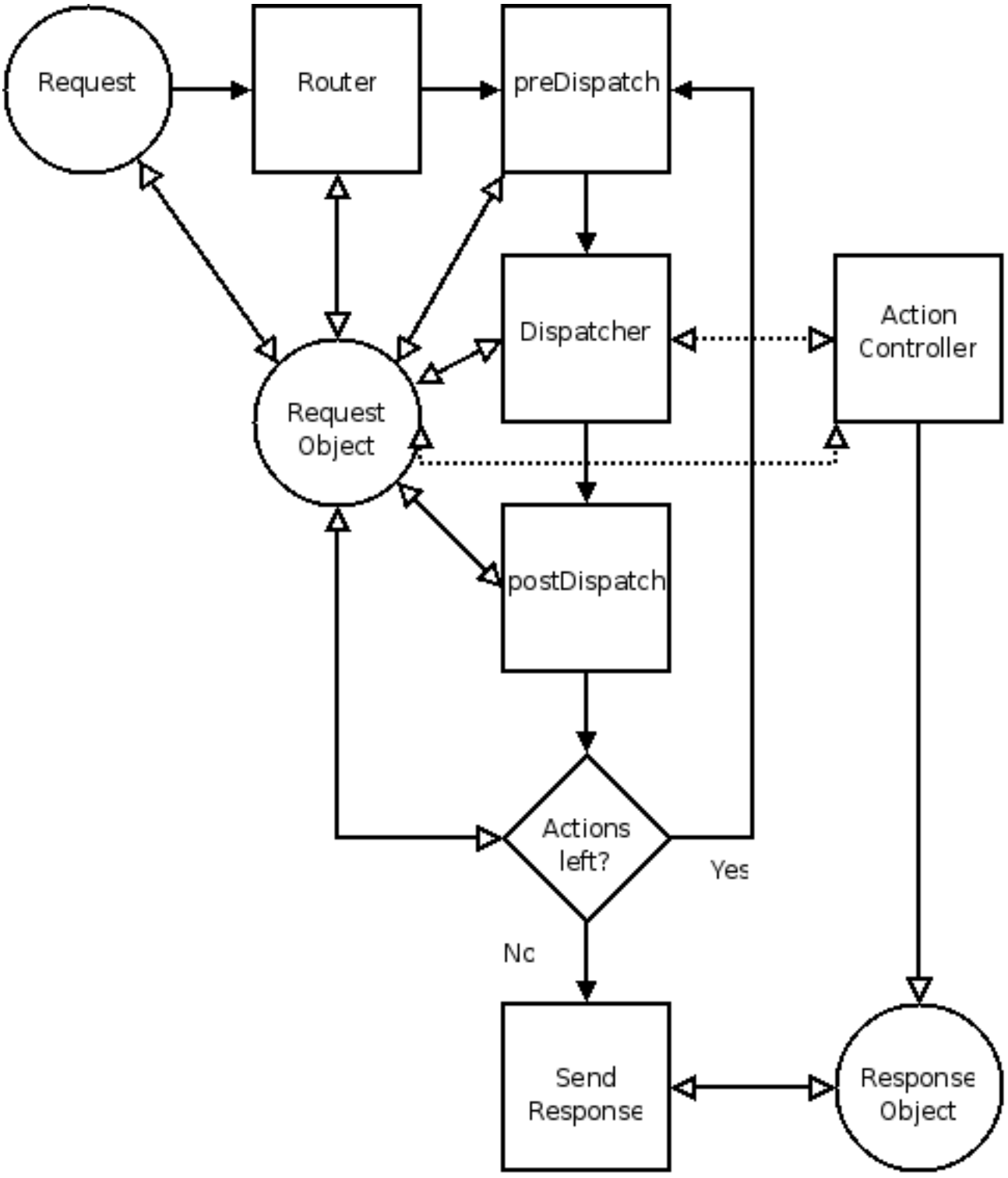
- `http://example.com/`
- `http://example.com/index`
- `http://example.com/index/index`

You're now ready to start creating more controllers and action methods. Congratulations!

2. Zend_Controller Basics

The `Zend_Controller` system is designed to be lightweight, modular, and extensible. It is a minimalist design to permit flexibility and some freedom to users while providing enough structure so that systems built around `Zend_Controller` share some common conventions and similar code layout.

The following diagram depicts the workflow, and the narrative following describes in detail the interactions:



The `Zend_Controller` workflow is implemented by several components. While it is not necessary to completely understand the underpinnings of all of these components to use the system, having a working knowledge of the process is helpful.

- `Zend_Controller_Front` orchestrates the entire workflow of the `Zend_Controller` system. It is an interpretation of the FrontController pattern. `Zend_Controller_Front`

processes all requests received by the server and is ultimately responsible for delegating requests to ActionControllers (`Zend_Controller_Action`).

- `Zend_Controller_Request_Abstract` (often referred to as the *Request Object*) represents the request environment and provides methods for setting and retrieving the controller and action names and any request parameters. Additionally it keeps track of whether or not the action it contains has been dispatched by `Zend_Controller_Dispatcher`. Extensions to the abstract request object can be used to encapsulate the entire request environment, allowing routers to pull information from the request environment in order to set the controller and action names.

  By default, `Zend_Controller_Request_Http` is used, which provides access to the entire HTTP request environment.

- `Zend_Controller_Router_Interface` is used to define routers. Routing is the process of examining the request environment to determine which controller, and action of that controller, should receive the request. This controller, action, and optional parameters are then set in the request object to be processed by `Zend_Controller_Dispatcher_Standard`. Routing occurs only once: when the request is initially received and before the first controller is dispatched.

  The default router, `Zend_Controller_Router_Rewrite`, takes a URI endpoint as specified in `Zend_Controller_Request_Http` and decomposes it into a controller, action, and parameters based on the path information in the URL. As an example, the URL `http://localhost/foo/bar/key/value` would be decoded to use the *foo* controller, *bar* action, and specify a parameter *key* with a value of *value*.

  `Zend_Controller_Router_Rewrite` can also be used to match arbitrary paths; see the router documentation for more information.

- `Zend_Controller_Dispatcher_Interface` is used to define dispatchers. Dispatching is the process of pulling the controller and action from the request object and mapping them to a controller file (or class) and action method in the controller class. If the controller or action do not exist, it handles determining default controllers and actions to dispatch.

  The actual dispatching process consists of instantiating the controller class and calling the action method in that class. Unlike routing, which occurs only once, dispatching occurs in a loop. If the request object's dispatched status is reset at any point, the loop will be repeated, calling whatever action is currently set in the request object. The first time the loop finishes with the request object's dispatched status set (Boolean TRUE), it will finish processing.

  The default dispatcher is `Zend_Controller_Dispatcher_Standard`. It defines controllers as MixedCasedClasses ending in the word Controller, and action methods as camelCasedMethods ending in the word Action: `FooController::barAction()`. In this case, the controller would be referred to as *foo* and the action as *bar*.

Case Naming Conventions

Since humans are notoriously inconsistent at maintaining case sensitivity when typing links, Zend Framework actually normalizes path information to lowercase. This, of course, will affect how you name your controller and actions... or refer to them in links.

If you wish to have your controller class or action method name have multiple MixedCasedWords or camelCasedWords, you will need to separate those

words on the url with either a '-' or '.' (though you can configure the character used).

As an example, if you were going to the action in `FooBarController::bazBatAction()`, you'd refer to it on the url as `/foo-bar/baz-bat` or `/foo.bar/baz.bat`.

- `Zend_Controller_Action` is the base action controller component. Each controller is a single class that extends the `Zend_Controller_Action` class and should contain one or more action methods.

- `Zend_Controller_Response_Abstract` defines a base response class used to collect and return responses from the action controllers. It collects both headers and body content.

  The default response class is `Zend_Controller_Response_Http`, which is suitable for use in an HTTP environment.

The workflow of `Zend_Controller` is relatively simple. A request is received by `Zend_Controller_Front`, which in turn calls `Zend_Controller_Router_Rewrite` to determine which controller (and action in that controller) to dispatch. `Zend_Controller_Router_Rewrite` decomposes the URI in order to set the controller and action names in the request. `Zend_Controller_Front` then enters a dispatch loop. It calls `Zend_Controller_Dispatcher_Standard`, passing it the request, to dispatch to the controller and action specified in the request (or use defaults). After the controller has finished, control returns to `Zend_Controller_Front`. If the controller has indicated that another controller should be dispatched by resetting the dispatched status of the request, the loop continues and another dispatch is performed. Otherwise, the process ends.

3. The Front Controller

3.1. Overview

`Zend_Controller_Front` implements a Front Controller pattern used in Model-View-Controller (MVC) applications. Its purpose is to initialize the request environment, route the incoming request, and then dispatch any discovered actions; it aggregates any responses and returns them when the process is complete.

`Zend_Controller_Front` also implements the Singleton pattern, meaning only a single instance of it may be available at any given time. This allows it to also act as a registry on which the other objects in the dispatch process may draw.

`Zend_Controller_Front` registers a plugin broker with itself, allowing various events it triggers to be observed by plugins. In most cases, this gives the developer the opportunity to tailor the dispatch process to the site without the need to extend the front controller to add functionality.

At a bare minimum, the front controller needs one or more paths to directories containing action controllers in order to do its work. A variety of methods may also be invoked to further tailor the front controller environment and that of its helper classes.

Default Behaviour

By default, the front controller loads the ErrorHandler plugin, as well as the ViewRenderer action helper plugin. These are to simplify error handling and view renderering in your controllers, respectively.

To disable the *ErrorHandler*, perform the following at any point prior to calling `dispatch()`:

```
// Disable the ErrorHandler plugin:
$front->setParam('noErrorHandler', true);
```

To disable the *ViewRenderer*, do the following prior to calling `dispatch()`:

```
// Disable the ViewRenderer helper:
$front->setParam('noViewRenderer', true);
```

3.2. Primary Methods

The front controller has several accessors for setting up its environment. However, there are three primary methods key to the front controller's functionality:

3.2.1. getInstance()

`getInstance()` is used to retrieve a front controller instance. As the front controller implements a Singleton pattern, this is also the only means possible for instantiating a front controller object.

```
$front = Zend_Controller_Front::getInstance();
```

3.2.2. setControllerDirectory() and addControllerDirectory

`setControllerDirectory()` is used to tell the dispatcher where to look for action controller class files. It accepts either a single path or an associative array of module and path pairs.

As some examples:

```
// Set the default controller directory:
$front->setControllerDirectory('../application/controllers');

// Set several module directories at once:
$front->setControllerDirectory(array(
    'default' => '../application/controllers',
    'blog'    => '../modules/blog/controllers',
    'news'    => '../modules/news/controllers',
));

// Add a 'foo' module directory:
$front->addControllerDirectory('../modules/foo/controllers', 'foo');
```

If you use `addControllerDirectory()` without a module name, it will set the directory for the *default* module -- overwriting it if it already exists.

You can get the current settings for the controller directory using `getControllerDirectory()`; this will return an array of module and directory pairs.

3.2.3. addModuleDirectory() and getModuleDirectory()

One aspect of the front controller is that you may define a modular directory structure for creating standalone components; these are called "modules".

Each module should be in its own directory and mirror the directory structure of the default module -- i.e., it should have a `/controllers/` subdirectory at the minimum, and typically a `/views/` subdirectory and other application subdirectories.

`addModuleDirectory()` allows you to pass the name of a directory containing one or more module directories. It then scans it and adds them as controller directories to the front controller.

Later, if you want to determine the path to a particular module or the current module, you can call `getModuleDirectory()`, optionally passing a module name to get that specific module directory.

3.2.4. dispatch()

`dispatch(Zend_Controller_Request_Abstract $request = null, Zend_Controller_Response_Abstract $response = null)` does the heavy work of the front controller. It may optionally take a request object and/or a response object, allowing the developer to pass in custom objects for each.

If no request or response object are passed in, `dispatch()` will check for previously registered objects and use those or instantiate default versions to use in its process (in both cases, the HTTP flavor will be used as the default).

Similarly, `dispatch()` checks for registered router and dispatcher objects, instantiating the default versions of each if none is found.

The dispatch process has three distinct events:

- Routing
- Dispatching
- Response

Routing takes place exactly once, using the values in the request object when `dispatch()` is called. Dispatching takes place in a loop; a request may either indicate multiple actions to dispatch, or the controller or a plugin may reset the request object to force additional actions to dispatch. When all is done, the front controller returns a response.

3.2.5. run()

`Zend_Controller_Front::run($path)` is a static method taking simply a path to a directory containing controllers. It fetches a front controller instance (via getInstance(), registers the path provided via setControllerDirectory(), and finally dispatches.

Basically, `run()` is a convenience method that can be used for site setups that do not require customization of the front controller environment.

```
// Instantiate front controller, set controller directory, and dispatch in one
// easy step:
Zend_Controller_Front::run('../application/controllers');
```

3.3. Environmental Accessor Methods

In addition to the methods listed above, there are a number of accessor methods that can be used to affect the front controller environment -- and thus the environment of the classes to which the front controller delegates.

- `resetInstance()` can be used to clear all current settings. Its primary purpose is for testing, but it can also be used for instances where you wish to chain together multiple front controllers.

- `setDefaultControllerName()` and `getDefaultControllerName()` let you specify a different name to use for the default controller ('index' is used otherwise) and retrieve the current value. They proxy to the dispatcher.

- `setDefaultAction()` and `getDefaultAction()` let you specify a different name to use for the default action ('index' is used otherwise) and retrieve the current value. They proxy to the dispatcher.

- `setRequest()` and `getRequest()` let you specify the request class or object to use during the dispatch process and to retrieve the current object. When setting the request object, you may pass in a request class name, in which case the method will load the class file and instantiate it.

- `setRouter()` `getRouter()` let you specify the router class or object to use during the dispatch process and to retrieve the current object. When setting the router object, you may pass in a router class name, in which case the method will load the class file and instantiate it.

  When retrieving the router object, it first checks to see if one is present, and if not, instantiates the default router (rewrite router).

- `setBaseUrl()` and `getBaseUrl()` let you specify the base URL to strip when routing requests and to retrieve the current value. The value is provided to the request object just prior to routing.

- `setDispatcher()` and `getDispatcher()` let you specify the dispatcher class or object to use during the dispatch process and retrieve the current object. When setting the dispatcher object, you may pass in a dispatcher class name, in which case the method will load the class file and instantiate it.

  When retrieving the dispatcher object, it first checks to see if one is present, and if not, instantiates the default dispatcher.

- `setResponse()` and `getResponse()` let you specify the response class or object to use during the dispatch process and to retrieve the current object. When setting the response object, you may pass in a response class name, in which case the method will load the class file and instantiate it.

- `registerPlugin(Zend_Controller_Plugin_Abstract $plugin, $stackIndex = null)` allows you to register plugin objects. By setting the optional `$stackIndex`, you can control the order in which plugins will execute.

- `unregisterPlugin($plugin)` let you unregister plugin objects. `$plugin` may be either a plugin object or a string denoting the class of plugin to unregister.

- `throwExceptions($flag)` is used to turn on/off the ability to throw exceptions during the dispatch process. By default, exceptions are caught and placed in the response object; turning on `throwExceptions()` will override this behaviour.

  For more information, read MVC Exceptions.

- `returnResponse($flag)` is used to tell the front controller whether to return the response (TRUE) from `dispatch()`, or if the response should be automatically emitted (FALSE). By default, the response is automatically emitted

(by calling `Zend_Controller_Response_Abstract::sendResponse()`); turning on `returnResponse()` will override this behaviour.

Reasons to return the response include a desire to check for exceptions prior to emitting the response, needing to log various aspects of the response (such as headers), etc.

3.4. Front Controller Parameters

In the introduction, we indicated that the front controller also acts as a registry for the various controller components. It does so through a family of "param" methods. These methods allow you to register arbitrary data -- objects and variables -- with the front controller to be retrieved at any time in the dispatch chain. These values are passed on to the router, dispatcher, and action controllers. The methods include:

- `setParam($name, $value)` allows you to set a single parameter of `$name` with value `$value`.
- `setParams(array $params)` allows you to set multiple parameters at once using an associative array.
- `getParam($name)` allows you to retrieve a single parameter at a time, using `$name` as the identifier.
- `getParams()` allows you to retrieve the entire list of parameters at once.
- `clearParams()` allows you to clear a single parameter (by passing a string identifier), multiple named parameters (by passing an array of string identifiers), or the entire parameter stack (by passing nothing).

There are several pre-defined parameters that may be set that have specific uses in the dispatch chain:

- *useDefaultControllerAlways* is used to hint to the dispatcher to use the default controller in the default module for any request that is not dispatchable (i.e., the module, controller, and/or action do not exist). By default, this is off.

  See MVC Exceptions You May Encounter for more detailed information on using this setting.

- *disableOutputBuffering* is used to hint to the dispatcher that it should not use output buffering to capture output generated by action controllers. By default, the dispatcher captures any output and appends it to the response object body content.
- *noViewRenderer* is used to disable the ViewRenderer. Set this parameter to `TRUE` to disable it.
- *noErrorHandler* is used to disable the Error Handler plugin. Set this parameter to `TRUE` to disable it.

3.5. Extending the Front Controller

To extend the Front Controller, at the very minimum you will need to override the `getInstance()` method:

```
class My_Controller_Front extends Zend_Controller_Front
{
    public static function getInstance()
    {
        if (null === self::$_instance) {
            self::$_instance = new self();
```

```
        }

        return self::$_instance;
    }
}
```

Overriding the `getInstance()` method ensures that subsequent calls to `Zend_Controller_Front::getInstance()` will return an instance of your new subclass instead of a `Zend_Controller_Front` instance -- this is particularly useful for some of the alternate routers and view helpers.

Typically, you will not need to subclass the front controller unless you need to add new functionality (for instance, a plugin autoloader, or a way to specify action helper paths). Some points where you may want to alter behaviour may include modifying how controller directories are stored, or what default router or dispatcher are used.

4. The Request Object

4.1. Introduction

The request object is a simple value object that is passed between `Zend_Controller_Front` and the router, dispatcher, and controller classes. It packages the names of the requested module, controller, action, and optional parameters, as well as the rest of the request environment, be it HTTP, the CLI, or PHP-GTK.

- The module name is accessed by `getModuleName()` and `setModuleName()`.
- The controller name is accessed by `getControllerName()` and `setControllerName()`.
- The name of the action to call within that controller is accessed by `getActionName()` and `setActionName()`.
- Parameters to be accessible by the action are an associative array of key and value pairs that are retrieved by `getParams()` and set with `setParams()`, or individually by `getParam()` and `setParam()`.

Based on the type of request, there may be more methods available. The default request used, `Zend_Controller_Request_Http`, for instance, has methods for retrieving the request URI, path information, `$_GET` and `$_POST` parameters, etc.

The request object is passed to the front controller, or if none is provided, it is instantiated at the beginning of the dispatch process, before routing occurs. It is passed through to every object in the dispatch chain.

Additionally, the request object is particularly useful in testing. The developer may craft the request environment, including module, controller, action, parameters, URI, etc, and pass the request object to the front controller to test application flow. When paired with the response object, elaborate and precise unit testing of MVC applications becomes possible.

4.2. HTTP Requests

4.2.1. Accessing Request Data

`Zend_Controller_Request_Http` encapsulates access to relevant values such as the key name and value for the controller and action router variables, and all additional parameters parsed from the URI. It additionally allows access to values contained in the superglobals as public members, and manages the current Base URL and Request URI. Superglobal values

cannot be set on a request object, instead use the `setParam()` and `getParam()` methods to set or retrieve user parameters.

Superglobal Data

When accessing superglobal data through `Zend_Controller_Request_Http` as public member properties, it is necessary to keep in mind that the property name (superglobal array key) is matched to a superglobal in a specific order of precedence: 1. `GET`, 2. `POST`, 3. `COOKIE`, 4. `SERVER`, 5. `ENV`.

Specific superglobals can be accessed using a public method as an alternative. For example, the raw value of `$_POST['user']` can be accessed by calling `getPost('user')` on the request object. These include `getQuery()` for retrieving `$_GET` elements, and `getHeader()` for retrieving request headers.

GET and POST Data

Be cautious when accessing data from the request object as it is not filtered in any way. The router and dispatcher validate and filter data for use with their tasks, but leave the data untouched in the request object.

Retrieving the Raw POST Data

As of 1.5.0, you can also retrieve the raw post data via the `getRawBody()` method. This method returns `FALSE` if no data was submitted in that fashion, but the full body of the post otherwise.

This is primarily useful for accepting content when developing a RESTful MVC application.

You may also set user parameters in the request object using `setParam()` and retrieve these later using `getParam()`. The router makes use of this functionality to set parameters matched in the request URI into the request object.

getParam() Retrieves More than User Parameters

In order to do some of its work, `getParam()` actually retrieves from several sources. In order of priority, these include: user parameters set via `setParam()`, `GET` parameters, and finally `POST` parameters. Be aware of this when pulling data via this method.

If you wish to pull only from parameters you set via `setParam()`, use the `getUserParam()`.

Additionally, as of 1.5.0, you can lock down which parameter sources will be searched. `setParamSources()` allows you to specify an empty array or an array with one or more of the values '_GET' or '_POST' indicating which parameter sources are allowed (by default, both are allowed); if you wish to restrict access to only '_GET' specify `setParamSources(array('_GET'))`.

Apache Quirks

If you are using Apache's 404 handler to pass incoming requests to the front controller, or using a PT flag with rewrite rules, `$_SERVER['REDIRECT_URL']`

contains the URI you need, not `$_SERVER['REQUEST_URI']`. If you are using such a setup and getting invalid routing, you should use the `Zend_Controller_Request_Apache404` class instead of the default Http class for your request object:

```
$request = new Zend_Controller_Request_Apache404();
$front->setRequest($request);
```

This class extends the `Zend_Controller_Request_Http` class and simply modifies the autodiscovery of the request URI. It can be used as a drop-in replacement.

4.2.2. Base Url and Subdirectories

`Zend_Controller_Request_Http` allows `Zend_Controller_Router_Rewrite` to be used in subdirectories. `Zend_Controller_Request_Http` will attempt to automatically detect your base URL and set it accordingly.

For example, if you keep your `index.php` in a webserver subdirectory named `/projects/myapp/index.php`, base URL (rewrite base) should be set to `/projects/myapp`. This string will then be stripped from the beginning of the path before calculating any route matches. This frees one from the necessity of prepending it to any of your routes. A route of **'user/:username'** will match URIs like `http://localhost/projects/myapp/user/martel` and `http://example.com/user/martel`.

URL Detection is Case Sensitive

Automatic base URL detection is case sensitive, so make sure your URL will match a subdirectory name in a filesystem (even on Windows machines). If it doesn't, an exception will be raised.

Should base URL be detected incorrectly you can override it with your own base path with the help of the `setBaseUrl()` method of either the `Zend_Controller_Request_Http` class, or the `Zend_Controller_Front` class. The easiest method is to set it in `Zend_Controller_Front`, which will proxy it into the request object. Example usage to set a custom base URL:

```
/**
 * Dispatch Request with custom base URL with Zend_Controller_Front.
 */
$router     = new Zend_Controller_Router_Rewrite();
$controller = Zend_Controller_Front::getInstance();
$controller->setControllerDirectory('./application/controllers')
           ->setRouter($router)
           ->setBaseUrl('/projects/myapp'); // set the base url!
$response   = $controller->dispatch();
```

4.2.3. Determining the Request Method

`getMethod()` allows you to determine the HTTP request method used to request the current resource. Additionally, a variety of methods exist that allow you to get boolean responses when asking if a specific type of request has been made:

- `isGet()`

- `isPost()`
- `isPut()`
- `isDelete()`
- `isHead()`
- `isOptions()`

The primary use case for these is for creating RESTful MVC architectures.

4.2.4. Detecting AJAX Requests

`Zend_Controller_Request_Http` has a rudimentary method for detecting AJAX requests: `isXmlHttpRequest()`. This method looks for an HTTP request header *X-Requested-With* with the value 'XMLHttpRequest'; if found, it returns `TRUE`.

Currently, this header is known to be passed by default with the following JS libraries:

- Prototype and Scriptaculous (and libraries derived from Prototype)
- Yahoo! UI Library
- jQuery
- MochiKit

Most AJAX libraries allow you to send custom HTTP request headers; if your library does not send this header, simply add it as a request header to ensure the `isXmlHttpRequest()` method works for you.

4.3. Subclassing the Request Object

The base request class used for all request objects is the abstract class `Zend_Controller_Request_Abstract`. At its most basic, it defines the following methods:

```
abstract class Zend_Controller_Request_Abstract
{
    /**
     * @return string
     */
    public function getControllerName();

    /**
     * @param string $value
     * @return self
     */
    public function setControllerName($value);

    /**
     * @return string
     */
    public function getActionName();

    /**
     * @param string $value
```

```
     * @return self
     */
    public function setActionName($value);

    /**
     * @return string
     */
    public function getControllerKey();

    /**
     * @param string $key
     * @return self
     */
    public function setControllerKey($key);

    /**
     * @return string
     */
    public function getActionKey();

    /**
     * @param string $key
     * @return self
     */
    public function setActionKey($key);

    /**
     * @param string $key
     * @return mixed
     */
    public function getParam($key);

    /**
     * @param string $key
     * @param mixed $value
     * @return self
     */
    public function setParam($key, $value);

    /**
     * @return array
     */
     public function getParams();

    /**
     * @param array $array
     * @return self
     */
    public function setParams(array $array);

    /**
     * @param boolean $flag
     * @return self
     */
    public function setDispatched($flag = true);

    /**
     * @return boolean
     */
    public function isDispatched();
```

```
}
```

The request object is a container for the request environment. The controller chain really only needs to know how to set and retrieve the controller, action, optional parameters, and dispatched status. By default, the request will search its own parameters using the controller or action keys in order to determine the controller and action.

Extend this class, or one of its derivatives, when you need the request class to interact with a specific environment in order to retrieve data for use in the above tasks. Examples include the HTTP environment, a CLI environment, or a PHP-GTK environment.

5. The Standard Router

5.1. Introduction

`Zend_Controller_Router_Rewrite` is the standard framework router. Routing is the process of taking a URI endpoint (that part of the URI which comes after the base URL) and decomposing it into parameters to determine which module, controller, and action of that controller should receive the request. This values of the module, controller, action and other parameters are packaged into a `Zend_Controller_Request_Http` object which is then processed by `Zend_Controller_Dispatcher_Standard`. Routing occurs only once: when the request is initially received and before the first controller is dispatched.

`Zend_Controller_Router_Rewrite` is designed to allow for mod_rewrite-like functionality using pure PHP structures. It is very loosely based on Ruby on Rails routing and does not require any prior knowledge of webserver URL rewriting. It is designed to work with a single Apache mod_rewrite rule (one of):

```
RewriteEngine on
RewriteRule !\.(js|ico|gif|jpg|png|css|html)$ index.php
```

or (preferred):

```
RewriteEngine On
RewriteCond %{REQUEST_FILENAME} -s [OR]
RewriteCond %{REQUEST_FILENAME} -l [OR]
RewriteCond %{REQUEST_FILENAME} -d
RewriteRule ^.*$ - [NC,L]
RewriteRule ^.*$ index.php [NC,L]
```

The rewrite router can also be used with the IIS webserver (versions <= 7.0) if Isapi_Rewrite has been installed as an Isapi extension with the following rewrite rule:

```
RewriteRule ^[\w/\%]*(?:\.(?!(?:js|ico|gif|jpg|png|css|html)$)[\w\%]*$)? /index.php [I]
```

IIS Isapi_Rewrite

When using IIS, `$_SERVER['REQUEST_URI']` will either not exist, or be set as an empty string. In this case, `Zend_Controller_Request_Http` will attempt to use the `$_SERVER['HTTP_X_REWRITE_URL']` value set by the `Isapi_Rewrite` extension.

IIS 7.0 introduces a native URL rewriting module, and it can be configured as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
     <system.webServer>
         <rewrite>
             <rules>
                 <rule name="Imported Rule 1" stopProcessing="true">
                     <match url="^.*$" />
                     <conditions logicalGrouping="MatchAny">
                         <add input="{REQUEST_FILENAME}"
                             matchType="IsFile" pattern=""
                             ignoreCase="false" />
                         <add input="{REQUEST_FILENAME}"
                             matchType="IsDirectory"
                             pattern="" ignoreCase="false" />
                     </conditions>
                     <action type="None" />
                 </rule>
                 <rule name="Imported Rule 2" stopProcessing="true">
                     <match url="^.*$" />
                     <action type="Rewrite" url="index.php" />
                 </rule>
             </rules>
         </rewrite>
     </system.webServer>
</configuration>
```

If using Lighttpd, the following rewrite rule is valid:

```
url.rewrite-once = (
    ".*\?(.*)$" => "/index.php?$1",
    ".*\.(js|ico|gif|jpg|png|css|html)$" => "$0",
    "" => "/index.php"
)
```

5.2. Using a Router

To properly use the rewrite router you have to instantiate it, add some user defined routes and inject it into the controller. The following code illustrates the procedure:

```
// Create a router

$router = $ctrl->getRouter(); // returns a rewrite router by default
$router->addRoute(
    'user',
    new Zend_Controller_Router_Route('user/:username',
                                     array('controller' => 'user',
                                           'action' => 'info'))
);
```

5.3. Basic Rewrite Router Operation

The heart of the RewriteRouter is the definition of user defined routes. Routes are added by calling the addRoute method of RewriteRouter and passing in a new instance of a class implementing `Zend_Controller_Router_Route_Interface`. Eg.:

```
$router->addRoute('user',
                  new Zend_Controller_Router_Route('user/:username'));
```

Rewrite Router comes with six basic types of routes (one of which is special):

- Zend_Controller_Router_Route
- Zend_Controller_Router_Route_Static
- Zend_Controller_Router_Route_Regex
- Zend_Controller_Router_Route_Hostname
- Zend_Controller_Router_Route_Chain
- Zend_Controller_Router_Rewrite *

Routes may be used numerous times to create a chain or user defined application routing schema. You may use any number of routes in any configuration, with the exception of the Module route, which should rather be used once and probably as the most generic route (i.e., as a default). Each route will be described in greater detail later on.

The first parameter to addRoute is the name of the route. It is used as a handle for getting the routes out of the router (e.g., for URL generation purposes). The second parameter being the route itself.

The most common use of the route name is through the means of `Zend_View` url helper:

```
<a href=
"<?php echo $this->url(array('username' => 'martel'), 'user') ?>">Martel</a>
```

Which would result in the href: `user/martel`.

Routing is a simple process of iterating through all provided routes and matching its definitions to current request URI. When a positive match is found, variable values are returned from the Route instance and are injected into the `Zend_Controller_Request` object for later use in the dispatcher as well as in user created controllers. On a negative match result, the next route in the chain is checked.

If you need to determine which route was matched, you can use the `getCurrentRouteName()` method, which will return the identifier used when registering the route with the router. If you want the actual route object, you can use `getCurrentRoute()`.

Reverse Matching

Routes are matched in reverse order so make sure your most generic routes are defined first.

Returned Values

Values returned from routing come from URL parameters or user defined route defaults. These variables are later accessible through the `Zend_Controller_Request::getParam()` or `Zend_Controller_Action::_getParam()` methods.

There are three special variables which can be used in your routes - 'module', 'controller' and 'action'. These special variables are used by `Zend_Controller_Dispatcher` to find a controller and action to dispatch to.

Special Variables

The names of these special variables may be different if you choose to alter the defaults in `Zend_Controller_Request_Http` by means of the `setControllerKey()` and `setActionKey()` methods.

5.4. Default Routes

`Zend_Controller_Router_Rewrite` comes preconfigured with a default route, which will match URIs in the shape of `controller/action`. Additionally, a module name may be specified as the first path element, allowing URIs of the form `module/controller/action`. Finally, it will also match any additional parameters appended to the URI by default - `controller/action/var1/value1/var2/value2`.

Some examples of how such routes are matched:

```
// Assuming the following:
$ctrl->setControllerDirectory(
    array(
        'default' => '/path/to/default/controllers',
        'news'    => '/path/to/news/controllers',
        'blog'    => '/path/to/blog/controllers'
    )
);

Module only:
http://example/news
    module == news

Invalid module maps to controller name:
http://example/foo
    controller == foo

Module + controller:
http://example/blog/archive
    module     == blog
    controller == archive

Module + controller + action:
http://example/blog/archive/list
    module     == blog
    controller == archive
    action     == list

Module + controller + action + params:
http://example/blog/archive/list/sort/alpha/date/desc
    module     == blog
    controller == archive
    action     == list
    sort       == alpha
    date       == desc
```

The default route is simply a `Zend_Controller_Router_Route_Module` object stored under the name (index) of 'default' in RewriteRouter. It's created more-or-less like below:

```
$compat = new Zend_Controller_Router_Route_Module(array(),
                                                  $dispatcher,
```

```
                                                    $request);
$this->addRoute('default', $compat);
```

If you do not want this particular default route in your routing schema, you may override it by creating your own 'default' route (i.e., storing it under the name of 'default') or removing it altogether by using `removeDefaultRoutes()`:

```
// Remove any default routes
$router->removeDefaultRoutes();
```

5.5. Base URL and Subdirectories

The rewrite router can be used in subdirectories (e.g., `http://domain.com/user/application-root/`) in which case the base URL of the application (`/user/application-root`) should be automatically detected by `Zend_Controller_Request_Http` and used accordingly.

Should the base URL be detected incorrectly you can override it with your own base path by using `Zend_Controller_Request_Http` and calling the `setBaseUrl()` method (see Base URL and Subdirectories):

```
$request->setBaseUrl('/~user/application-root/');
```

5.6. Global Parameters

You can set global parameters in a router which are automatically supplied to a route when assembling through `setGlobalParam()`. If a global parameter is set but also given to the assemble method directly, the user parameter overrides the global parameter. You can set a global parameter this way:

```
$router->setGlobalParam('lang', 'en');
```

5.7. Route Types

5.7.1. Zend_Controller_Router_Route

`Zend_Controller_Router_Route` is the standard framework route. It combines ease of use with flexible route definition. Each route consists primarily of URL mapping (of static and dynamic parts (variables)) and may be initialized with defaults as well as with variable requirements.

Let's imagine our fictional application will need some informational page about the content authors. We want to be able to point our web browsers to `http://domain.com/author/martel` to see the information about this "martel" guy. And the route for such functionality could look like:

```
$route = new Zend_Controller_Router_Route(
    'author/:username',
    array(
        'controller' => 'profile',
        'action'     => 'userinfo'
    )
);
```

```
$router->addRoute('user', $route);
```

The first parameter in the `Zend_Controller_Router_Route` constructor is a route definition that will be matched to a URL. Route definitions consist of static and dynamic parts separated by the slash ('/') character. Static parts are just simple text: **author**. Dynamic parts, called variables, are marked by prepending a colon to the variable name: **:username**.

Character Usage

The current implementation allows you to use any character (except a slash) as a variable identifier, but it is strongly recommended that one uses only characters that are valid for PHP variable identifiers. Future implementations may alter this behaviour, which could result in hidden bugs in your code.

This example route should be matched when you point your browser to `http://domain.com/author/martel`, in which case all its variables will be injected to the `Zend_Controller_Request` object and will be accessible in your `ProfileController`. Variables returned by this example may be represented as an array of the following key and value pairs:

```
$values = array(
    'username'   => 'martel',
    'controller' => 'profile',
    'action'     => 'userinfo'
);
```

Later on, `Zend_Controller_Dispatcher_Standard` should invoke the `userinfoAction()` method of your `ProfileController` class (in the default module) based on these values. There you will be able to access all variables by means of the `Zend_Controller_Action::_getParam()` or `Zend_Controller_Request::getParam()` methods:

```
public function userinfoAction()
{
    $request = $this->getRequest();
    $username = $request->getParam('username');

    $username = $this->_getParam('username');
}
```

Route definition can contain one more special character - a wildcard - represented by '*' symbol. It is used to gather parameters similarly to the default Module route (var => value pairs defined in the URI). The following route more-or-less mimics the Module route behavior:

```
$route = new Zend_Controller_Router_Route(
    ':module/:controller/:action/*',
    array('module' => 'default')
);
$router->addRoute('default', $route);
```

5.7.1.1. Variable Defaults

Every variable in the route can have a default and this is what the second parameter of the `Zend_Controller_Router_Route` constructor is used for. This parameter is an array with keys representing variable names and with values as desired defaults:

```
$route = new Zend_Controller_Router_Route(
    'archive/:year',
    array('year' => 2006)
);
$router->addRoute('archive', $route);
```

The above route will match URLs like `http://domain.com/archive/2005` and `http://example.com/archive`. In the latter case the variable year will have an initial default value of 2006.

This example will result in injecting a year variable to the request object. Since no routing information is present (no controller and action parameters are defined), the application will be dispatched to the default controller and action method (which are both defined in `Zend_Controller_Dispatcher_Abstract`). To make it more usable, you have to provide a valid controller and a valid action as the route's defaults:

```
$route = new Zend_Controller_Router_Route(
    'archive/:year',
    array(
        'year'       => 2006,
        'controller' => 'archive',
        'action'     => 'show'
    )
);
$router->addRoute('archive', $route);
```

This route will then result in dispatching to the method `showAction()` of the class `ArchiveController`.

5.7.1.2. Variable Requirements

One can add a third parameter to the `Zend_Controller_Router_Route` constructor where variable requirements may be set. These are defined as parts of a regular expression:

```
$route = new Zend_Controller_Router_Route(
    'archive/:year',
    array(
        'year'       => 2006,
        'controller' => 'archive',
        'action'     => 'show'
    ),
    array('year' => '\d+')
);
$router->addRoute('archive', $route);
```

With a route defined like above, the router will match it only when the year variable will contain numeric data, eg. `http://domain.com/archive/2345`. A URL like `http://example.com/archive/test` will not be matched and control will be passed to the next route in the chain instead.

5.7.1.3. Translated segments

The standard route supports translated segments. To use this feature, you have to define at least a translator (an instance of `Zend_Translate`) via one of the following ways:

- Put it into the registry with the key `Zend_Translate`.

- Set it via the static method `Zend_Controller_Router_Route::setDefaultTranslator()`.
- Pass it as fourth parameter to the constructor.

By default, the locale specified in the `Zend_Translate` instance will be used. To override it, you set it (an instance of `Zend_Locale` or a locale string) in one of the following ways:

- Put it into the registry with the key `Zend_Locale`.
- Set it via the static method `Zend_Controller_Router_Route::setDefaultLocale()`.
- Pass it as fifth parameter to the constructor.
- Pass it as **@locale** parameter to the assemble method.

Translated segments are separated into two parts. Fixed segments are prefixed by a single @-sign, and will be translated to the current locale when assembling and reverted to the message ID when matching again. Dynamic segments are prefixed by **:@**. When assembling, the given parameter will be translated and inserted into the parameter position. When matching, the translated parameter from the URL will be reverted to the message ID again.

Message IDs and separate language file

Occasionally a message ID which you want to use in one of your routes is already used in a view script or somewhere else. To have full control over safe URLs, you should use a separate language file for the messages used in the route.

The following is the simplest way to prepare the standard route for translated segment usage:

```
// Prepare the translator
$translator = new Zend_Translate('array', array(), 'en');
$translator->addTranslation(array('archive' => 'archiv',
                                  'year'    => 'jahr',
                                  'month'   => 'monat',
                                  'index'   => 'uebersicht'),
                            'de');

// Set the current locale for the translator
$translator->setLocale('en');

// Set it as default translator for routes
Zend_Controller_Router_Route::setDefaultTranslator($translator);
```

This example demonstrates the usage of static segments:

```
// Create the route
$route = new Zend_Controller_Router_Route(
    '@archive',
    array(
        'controller' => 'archive',
        'action'     => 'index'
    )
);
$router->addRoute('archive', $route);

// Assemble the URL in default locale: archive
$route->assemble(array());
```

```
// Assemble the URL in german: archiv
$route->assemble(array());
```

You can use the dynamic segments to create a module-route like translated version:

```
// Create the route
$route = new Zend_Controller_Router_Route(
    ':@controller/:@action/*',
    array(
        'controller' => 'index',
        'action'     => 'index'
    )
);
$router->addRoute('archive', $route);

// Assemble the URL in default locale: archive/index/foo/bar
$route->assemble(array('controller' => 'archive', 'foo' => 'bar'));

// Assemble the URL in german: archiv/uebersicht/foo/bar
$route->assemble(array('controller' => 'archive', 'foo' => 'bar'));
```

You can also mix static and dynamic segments:

```
// Create the route
$route = new Zend_Controller_Router_Route(
    '@archive/:@mode/:value',
    array(
        'mode'       => 'year'
        'value'      => 2005,
        'controller' => 'archive',
        'action'     => 'show'
    ),
    array('mode'  => '(month|year)'
          'value' => '\d+')
);
$router->addRoute('archive', $route);

// Assemble the URL in default locale: archive/month/5
$route->assemble(array('mode' => 'month', 'value' => '5'));

// Assemble the URL in german: archiv/monat/5
$route->assemble(array('mode' => 'month', 'value' => '5', '@locale' => 'de'));
```

5.7.2. Zend_Controller_Router_Route_Static

The examples above all use dynamic routes -- routes that contain patterns to match against. Sometimes, however, a particular route is set in stone, and firing up the regular expression engine would be an overkill. The answer to this situation is to use static routes:

```
$route = new Zend_Controller_Router_Route_Static(
    'login',
    array('controller' => 'auth', 'action' => 'login')
);
$router->addRoute('login', $route);
```

Above route will match a URL of `http://domain.com/login`, and dispatch to `AuthController::loginAction()`.

Warning: Static Routes must Contain Sane Defaults

Since a static route does not pass any part of the URL to the request object as parameters, you *must* pass all parameters necessary for dispatching a request as defaults to the route. Omitting the "controller" or "action" default values will have unexpected results, and will likely result in the request being undispatchable.

As a rule of thumb, always provide each of the following default values:

- controller
- action
- module (if not default)

Optionally, you can also pass the "useDefaultControllerAlways" parameter to the front controller during bootstrapping:

```
$front->setParam('useDefaultControllerAlways', true);
```

However, this is considered a workaround; it is always better to explicitly define sane defaults.

5.7.3. Zend_Controller_Router_Route_Regex

In addition to the default and static route types, a Regular Expression route type is available. This route offers more power and flexibility over the others, but at a slight cost of complexity. At the same time, it should be faster than the standard Route.

Like the standard route, this route has to be initialized with a route definition and some defaults. Let's create an archive route as an example, similar to the previously defined one, only using the Regex route this time:

```
$route = new Zend_Controller_Router_Route_Regex(
    'archive/(\d+)',
    array(
        'controller' => 'archive',
        'action'     => 'show'
    )
);
$router->addRoute('archive', $route);
```

Every defined regex subpattern will be injected to the request object. With our above example, after successful matching `http://domain.com/archive/2006`, the resulting value array may look like:

```
$values = array(
    1            => '2006',
    'controller' => 'archive',
    'action'     => 'show'
);
```

Leading and trailing slashes are trimmed from the URL in the Router prior to a match. As a result, matching the URL `http://domain.com/foo/bar/`, would involve a regex of `foo/bar`, and not `/foo/bar`.

Line start and line end anchors ('^' and '$', respectively) are automatically pre- and appended to all expressions. Thus, you should not use these in your regular expressions, and you should match the entire string.

This route class uses the '#' character for a delimiter. This means that you will need to escape hash characters ('#') but not forward slashes ('/') in your route definitions. Since the '#' character (named anchor) is rarely passed to the webserver, you will rarely need to use that character in your regex.

You can get the contents of the defined subpatterns the usual way:

```
public function showAction()
{
    $request = $this->getRequest();
    $year    = $request->getParam(1); // $year = '2006';
}
```

Notice the key is an integer (1) instead of a string ('1').

This route will not yet work exactly the same as its standard route counterpart since the default for 'year' is not yet set. And what may not yet be evident is that we will have a problem with a trailing slash even if we declare a default for the year and make the subpattern optional. The solution is to make the whole year part optional along with the slash but catch only the numeric part:

```
$route = new Zend_Controller_Router_Route_Regex(
    'archive(?:/(\d+))?',
    array(
        1            => '2006',
        'controller' => 'archive',
        'action'     => 'show'
    )
);
$router->addRoute('archive', $route);
```

Now let's get to the problem you have probably noticed on your own by now. Using integer based keys for parameters is not an easily manageable solution and may be potentially problematic in the long run. And that's where the third parameter comes in. This parameter is an associative array that represents a map of regex subpatterns to parameter named keys. Let's work on our easier example:

```
$route = new Zend_Controller_Router_Route_Regex(
    'archive/(\d+)',
    array(
        'controller' => 'archive',
        'action' => 'show'
    ),
    array(
        1 => 'year'
    )
);
$router->addRoute('archive', $route);
```

This will result in following values injected into Request:

```
$values = array(
    'year'       => '2006',
    'controller' => 'archive',
    'action'     => 'show'
);
```

The map may be defined in either direction to make it work in any environment. Keys may contain variable names or subpattern indexes:

```
$route = new Zend_Controller_Router_Route_Regex(
    'archive/(\d+)',
    array( ... ),
    array(1 => 'year')
);

// OR

$route = new Zend_Controller_Router_Route_Regex(
    'archive/(\d+)',
    array( ... ),
    array('year' => 1)
);
```

Subpattern keys have to be represented by integers.

Notice that the numeric index in Request values is now gone and a named variable is shown in its place. Of course you can mix numeric and named variables if you wish:

```
$route = new Zend_Controller_Router_Route_Regex(
    'archive/(\d+)/page/(\d+)',
    array( ... ),
    array('year' => 1)
);
```

Which will result in mixed values available in the Request. As an example, the URL `http://domain.com/archive/2006/page/10` will result in following values:

```
$values = array(
    'year'       => '2006',
    2            => 10,
    'controller' => 'archive',
    'action'     => 'show'
);
```

Since regex patterns are not easily reversed, you will need to prepare a reverse URL if you wish to use a URL helper or even an assemble method of this class. This reversed path is represented by a string parsable by `sprintf()` and is defined as a fourth construct parameter:

```
$route = new Zend_Controller_Router_Route_Regex(
    'archive/(\d+)',
    array( ... ),
    array('year' => 1),
```

```
    'archive/%s'
);
```

All of this is something which was already possible by the means of a standard route object, so where's the benefit in using the Regex route, you ask? Primarily, it allows you to describe any type of URL without any restrictions. Imagine you have a blog and wish to create URLs like: `http://domain.com/blog/archive/01-Using_the_Regex_Router.html`, and have it decompose the last path element, `01-Using_the_Regex_Router.html`, into an article ID and article title or description; this is not possible with the standard route. With the Regex route, you can do something like the following solution:

```
$route = new Zend_Controller_Router_Route_Regex(
    'blog/archive/(\d+)-(.+)\.html',
    array(
        'controller' => 'blog',
        'action'     => 'view'
    ),
    array(
        1 => 'id',
        2 => 'description'
    ),
    'blog/archive/%d-%s.html'
);
$router->addRoute('blogArchive', $route);
```

As you can see, this adds a tremendous amount of flexibility over the standard route.

5.7.4. Zend_Controller_Router_Route_Hostname

`Zend_Controller_Router_Route_Hostname` is the hostname route of the framework. It works similar to the standard route, but it works on the with the hostname of the called URL instead with the path.

Let's use the example from the standard route and see how it would look like in a hostname based way. Instead of calling the user via a path, we'd want to have a user to be able to call `http://martel.users.example.com` to see the information about the user "martel":

```
$hostnameRoute = new Zend_Controller_Router_Route_Hostname(
    ':username.users.example.com',
    array(
        'controller' => 'profile',
        'action'     => 'userinfo'
    )
);

$plainPathRoute = new Zend_Controller_Router_Route_Static('');

$router->addRoute('user', $hostnameRoute->chain($plainPathRoute);
```

The first parameter in the `Zend_Controller_Router_Route_Hostname` constructor is a route definition that will be matched to a hostname. Route definitions consist of static and dynamic parts separated by the dot ('.') character. Dynamic parts, called variables, are marked by prepending a colon to the variable name: **:username**. Static parts are just simple text: **user**.

Hostname routes can, but never should be used as is. The reason behind that is, that a hostname route alone would match any path. So what you have to do is to chain a path

route to the hostname route. This is done like in the example by calling **$hostnameRoute->chain($pathRoute);**. By doing this, `$hostnameRoute` isn't modified, but a new route (`Zend_Controller_Router_Route_Chain`) is returned, which can then be given to the router.

5.7.5. Zend_Controller_Router_Route_Chain

`Zend_Controller_Router_Route_Chain` is a route which allows to chain multiple routes together. This allows you to chain hostname-routes and path routes, or multiple path routes for example. Chaining can be done either programatically or within a configuration file.

Parameter Priority

When chaining routes together, the parameters of the outer route have a higher priority than the parameters of the inner route. Thus if you define a controller in the outer and in the inner route, the controller of the outer route will be selected.

When chaining programatically, there are two ways to achieve this. The first one is to create a new `Zend_Controller_Router_Route_Chain` instance and then calling the `chain()` method multiple times with all routes which should be chained together. The other way is to take the first route, e.g. a hostname route, and calling the `chain()` method on it with the route which should be appended to it. This will not modify the hostname route, but return a new instance of `Zend_Controller_Router_Route_Chain`, which then has both routes chained together:

```
// Create two routes
$hostnameRoute = new Zend_Controller_Router_Route_Hostname(...);
$pathRoute     = new Zend_Controller_Router_Route(...);

// First way, chain them via the chain route
$chainedRoute = new Zend_Controller_Router_Route_Chain();
$chainedRoute->chain($hostnameRoute)
             ->chain($pathRoute);

// Second way, chain them directly
$chainedRoute = $hostnameRoute->chain($pathRoute);
```

When chaining routes together, their separator is a slash by default. There may be cases when you want to have a different separator:

```
// Create two routes
$firstRoute  = new Zend_Controller_Router_Route('foo');
$secondRoute = new Zend_Controller_Router_Route('bar');

// Chain them together with a different separator
$chainedRoute = $firstRoute->chain($secondRoute, '-');

// Assemble the route: "foo-bar"
echo $chainedRoute->assemble();
```

5.7.5.1. Chain Routes via Zend_Config

To chain routes together in a config file, there are additional parameters for the configuration of those. The simpler approach is to use the chains parameters. This one is simply a list of routes, which will be chained with the parent route. Neither the parent- nor the child-route will be added directly to the router but only the resulting chained route. The name of the chained route in the

router will be the parent route name and the child route name concatenated with a dash (-) by default. A simple config in XML would look like this:

```
<routes>
    <www type="Zend_Controller_Router_Route_Hostname">
        <route>www.example.com</route>
        <chains>
            <language type="Zend_Controller_Router_Route">
                <route>:language</route>
                <reqs language="[a-z]{2}">
                <chains>
                    <index type="Zend_Controller_Router_Route_Static">
                        <route></route>
                        <defaults module="default" controller="index"
                                  action="index" />
                    </index>
                    <imprint type="Zend_Controller_Router_Route_Static">
                        <route>imprint</route>
                        <defaults module="default" controller="index"
                                  action="index" />
                    </imprint>
                </chains>
            </language>
        </chains>
    </www>
    <users type="Zend_Controller_Router_Route_Hostname">
        <route>users.example.com</route>
        <chains>
            <profile type="Zend_Controller_Router_Route">
                <route>:username</route>
                <defaults module="users" controller="profile" action="index" />
            </profile>
        </chains>
    </users>
    <misc type="Zend_Controller_Router_Route_Static">
        <route>misc</route>
    </misc>
</routes>
```

This will result in the three routes **www-language-index**, **www-language-imprint** and **users-language-profile** which will only match based on the hostname and the route **misc**, which will match with any hostname.

The alternative way of creating a chained route is via the chain parameter, which can only be used with the chain-route type directly, and also just works in the root level:

```
<routes>
    <www type="Zend_Controller_Router_Route_Chain">
        <route>www.example.com</route>
    </www>
    <language type="Zend_Controller_Router_Route">
        <route>:language</route>
        <reqs language="[a-z]{2}">
    </language>
    <index type="Zend_Controller_Router_Route_Static">
        <route></route>
        <defaults module="default" controller="index" action="index" />
    </index>
    <imprint type="Zend_Controller_Router_Route_Static">
```

```
        <route>imprint</route>
        <defaults module="default" controller="index" action="index" />
    </imprint>

    <www-index type="Zend_Controller_Router_Route_Chain">
        <chain>www, language, index</chain>
    </www-index>
    <www-imprint type="Zend_Controller_Router_Route_Chain">
        <chain>www, language, imprint</chain>
    </www-imprint>
</routes>
```

You can also give the chain parameter as array instead of separating the routes with a comma:

```
<routes>
    <www-index type="Zend_Controller_Router_Route_Chain">
        <chain>www</chain>
        <chain>language</chain>
        <chain>index</chain>
    </www-index>
    <www-imprint type="Zend_Controller_Router_Route_Chain">
        <chain>www</chain>
        <chain>language</chain>
        <chain>imprint</chain>
    </www-imprint>
</routes>
```

When you configure chain routes with `Zend_Config` and want the chain name separator to be different from a dash, you need to specify this separator separately:

```
$config = new Zend_Config(array(
    'chainName' => array(
        'type'   => 'Zend_Controller_Router_Route_Static',
        'route'  => 'foo',
        'chains' => array(
            'subRouteName' => array(
                'type'     => 'Zend_Controller_Router_Route_Static',
                'route'    => 'bar',
                'defaults' => array(
                    'module'      => 'module',
                     'controller' => 'controller',
                     'action'     => 'action'
                )
            )
        )
    )
));

// Set separator before adding config
$router->setChainNameSeparator('_separator_')

// Add config
$router->addConfig($config);

// The name of our route now is: chainName_separator_subRouteName
echo $this->_router->assemble(array(), 'chainName_separator_subRouteName');

// The proof: it echoes /foo/bar
```

5.7.6. Zend_Rest_Route

The `Zend_Rest` component contains a RESTful route for `Zend_Controller_Router_Rewrite`. This route offers a standardized routing scheme that routes requests by translating the HTTP method and the URI to a module, controller, and action. The table below provides an overview of how request methods and URI's are routed.

Table 30. Zend_Rest_Route Behavior

Method	URI	Module_Controller::action
GET	/product/ratings/	Product_RatingsController::indexAct
GET	/product/ratings/:id	Product_RatingsController::getActio
POST	/product/ratings	Product_RatingsController::postActi
PUT	/product/ratings/:id	Product_RatingsController::putActio
DELETE	/product/ratings/:id	Product_RatingsController::deleteAc
POST	/product/ratings/:id?_method="PUT"	Product_RatingsController::putActio
POST	/product/ratings/:id?_method="DELETE"	Product_RatingsController::deleteAc

5.7.6.1. Zend_Rest_Route Usage

To enable `Zend_Rest_Route` for an entire application, construct it with no config params and add it as the default route on the front controller:

```
$front     = Zend_Controller_Front::getInstance();
$restRoute = new Zend_Rest_Route($front);
$front->getRouter()->addRoute('default', $restRoute);
```

If `Zend_Rest_Route` cannot match a valid module, controller, or action, it will return FALSE and the router will attempt to match using the next route in the router.

To enable `Zend_Rest_Route` for specific modules, construct it with an array of module names as the 3rd constructor argument:

```
$front     = Zend_Controller_Front::getInstance();
$restRoute = new Zend_Rest_Route($front, array(), array('product'));
$front->getRouter()->addRoute('rest', $restRoute);
```

To enable `Zend_Rest_Route` for specific controllers, add an array of controller names as the value of each module array element.

```
$front     = Zend_Controller_Front::getInstance();
$restRoute = new Zend_Rest_Route($front, array(), array(
    'product' => array('ratings')
));
$front->getRouter()->addRoute('rest', $restRoute);
```

5.7.6.2. Zend_Rest_Route with Zend_Config_Ini

To use `Zend_Rest_Route` from an INI config file, use a route type parameter and set the config options:

```
routes.rest.type = Zend_Rest_Route
routes.rest.defaults.controller = object
routes.rest.mod = project,user
```

The 'type' option designates the RESTful routing config type. The 'defaults' option is used to specify custom default module, controller, and/or actions for the route. All other options in the config group are treated as RESTful module names, and their values are RESTful controller names. The example config defines `Mod_ProjectController` and `Mod_UserController` as RESTful controllers.

Then use the `addConfig()` method of the Rewrite router object:

```
$config = new Zend_Config_Ini('path/to/routes.ini');
$router = new Zend_Controller_Router_Rewrite();
$router->addConfig($config, 'routes');
```

5.7.6.3. Zend_Rest_Controller

To help or guide development of Controllers for use with `Zend_Rest_Route`, extend your Controllers from `Zend_Rest_Controller`. `Zend_Rest_Controller` defines the 5 most-commonly needed operations for RESTful resources in the form of abstract action methods.

- *`indexAction()`* - Should retrieve an index of resources and assign it to view.
- *`getAction()`* - Should retrieve a single resource identified by URI and assign it to view.
- *`postAction()`* - Should accept a new single resource and persist its state.
- *`putAction()`* - Should accept a single resource idenitifed by URI and persist its state.
- *`deleteAction()`* - Should delete a single resource identified by URI.

5.8. Using Zend_Config with the RewriteRouter

Sometimes it is more convenient to update a configuration file with new routes than to change the code. This is possible via the `addConfig()` method. Basically, you create a `Zend_Config`-compatible configuration, and in your code read it in and pass it to the RewriteRouter.

As an example, consider the following INI file:

```
[production]
routes.archive.route = "archive/:year/*"
routes.archive.defaults.controller = archive
routes.archive.defaults.action = show
routes.archive.defaults.year = 2000
routes.archive.reqs.year = "\d+"

routes.news.type = "Zend_Controller_Router_Route_Static"
routes.news.route = "news"
routes.news.defaults.controller = "news"
routes.news.defaults.action = "list"

routes.archive.type = "Zend_Controller_Router_Route_Regex"
routes.archive.route = "archive/(\d+)"
routes.archive.defaults.controller = "archive"
```

```
routes.archive.defaults.action = "show"
routes.archive.map.1 = "year"
; OR: routes.archive.map.year = 1
```

The above INI file can then be read into a Zend_Config object as follows:

```
$config = new Zend_Config_Ini('/path/to/config.ini', 'production');
$router = new Zend_Controller_Router_Rewrite();
$router->addConfig($config, 'routes');
```

In the above example, we tell the router to use the 'routes' section of the INI file to use for its routes. Each first-level key under that section will be used to define a route name; the above example defines the routes 'archive' and 'news'. Each route then requires, at minimum, a 'route' entry and one or more 'defaults' entries; optionally one or more 'reqs' (short for 'required') may be provided. All told, these correspond to the three arguments provided to a Zend_Controller_Router_Route_Interface object. An option key, 'type', can be used to specify the route class type to use for that particular route; by default, it uses Zend_Controller_Router_Route. In the example above, the 'news' route is defined to use Zend_Controller_Router_Route_Static.

5.9. Subclassing the Router

The standard rewrite router should provide most functionality you may need; most often, you will only need to create a new route type in order to provide new or modified functionality over the provided routes.

That said, you may at some point find yourself wanting to use a different routing paradigm. The interface Zend_Controller_Router_Interface provides the minimal information required to create a router, and consists of a single method.

```
interface Zend_Controller_Router_Interface
{
  /**
   * @param  Zend_Controller_Request_Abstract $request
   * @throws Zend_Controller_Router_Exception
   * @return Zend_Controller_Request_Abstract
   */
  public function route(Zend_Controller_Request_Abstract $request);
}
```

Routing only occurs once: when the request is first received into the system. The purpose of the router is to determine the controller, action, and optional parameters based on the request environment, and then set them in the request. The request object is then passed to the dispatcher. If it is not possible to map a route to a dispatch token, the router should do nothing to the request object.

6. The Dispatcher

6.1. Overview

Dispatching is the process of taking the request object, Zend_Controller_Request_Abstract, extracting the module name, controller name, action name, and optional parameters contained in it, and then instantiating a controller and

calling an action of that controller. If any of the module, controller, or action are not found, it will use default values for them. `Zend_Controller_Dispatcher_Standard` specifies *index* for each of the controller and action defaults and *default* for the module default value, but allows the developer to change the default values for each using the `setDefaultController()`, `setDefaultAction()`, and `setDefaultModule()` methods, respectively.

Default Module

When creating modular applications, you may find that you want your default module namespaced as well (the default configuration is that the default module is *not* namespaced). As of 1.5.0, you can now do so by specifying the prefixDefaultModule as TRUE in either the front controller or your dispatcher:

```
// In your front controller:
$front->setParam('prefixDefaultModule', true);

// In your dispatcher:
$dispatcher->setParam('prefixDefaultModule', true);
```

This allows you to re-purpose an existing module to be the default module for an application.

Dispatching happens in a loop in the front controller. Before dispatching occurs, the front controller routes the request to find user specified values for the module, controller, action, and optional parameters. It then enters a dispatch loop, dispatching the request.

At the beginning of each iteration, it sets a flag in the request object indicating that the action has been dispatched. If an action or pre or postDispatch plugin resets that flag, the dispatch loop will continue and attempt to dispatch the new request. By changing the controller and/or action in the request and resetting the dispatched flag, the developer may define a chain of requests to perform.

The action controller method that controls such dispatching is `_forward()`; call this method from any of the `preDispatch()`, `postDispatch()` or action methods, providing an action, controller, module, and optionally any additional parameters you may wish to send to the new action:

```
public function fooAction()
{
    // forward to another action in the current controller and module:
    $this->_forward('bar', null, null, array('baz' => 'bogus'));
}

public function barAction()
{
    // forward to an action in another controller:
    // FooController::bazAction(),
    // in the current module:
    $this->_forward('baz', 'foo', null, array('baz' => 'bogus'));
}

public function bazAction()
{
    // forward to an action in another controller in another module,
    // Foo_BarController::bazAction():
    $this->_forward('baz', 'bar', 'foo', array('baz' => 'bogus'));
```

```
}
```

6.2. Subclassing the Dispatcher

`Zend_Controller_Front` will first call the router to determine the first action in the request. It then enters a dispatch loop, which calls on the dispatcher to dispatch the action.

The dispatcher needs a variety of data in order to do its work - it needs to know how to format controller and action names, where to look for controller class files, whether or not a provided module name is valid, and an API for determining if a given request is even dispatchable based on the other information available.

`Zend_Controller_Dispatcher_Interface` defines the following methods as required for any dispatcher implementation:

```
interface Zend_Controller_Dispatcher_Interface
{
    /**
     * Format a string into a controller class name.
     *
     * @param string $unformatted
     * @return string
     */
    public function formatControllerName($unformatted);

    /**
     * Format a string into an action method name.
     *
     * @param string $unformatted
     * @return string
     */
    public function formatActionName($unformatted);

    /**
     * Determine if a request is dispatchable
     *
     * @param  Zend_Controller_Request_Abstract $request
     * @return boolean
     */
    public function isDispatchable(
        Zend_Controller_Request_Abstract $request
    );

    /**
     * Set a user parameter (via front controller, or for local use)
     *
     * @param string $name
     * @param mixed $value
     * @return Zend_Controller_Dispatcher_Interface
     */
    public function setParam($name, $value);

    /**
     * Set an array of user parameters
     *
     * @param array $params
     * @return Zend_Controller_Dispatcher_Interface
     */
    public function setParams(array $params);
```

```
    /**
     * Retrieve a single user parameter
     *
     * @param string $name
     * @return mixed
     */
    public function getParam($name);

    /**
     * Retrieve all user parameters
     *
     * @return array
     */
    public function getParams();

    /**
     * Clear the user parameter stack, or a single user parameter
     *
     * @param null|string|array single key or array of keys for
     *        params to clear
     * @return Zend_Controller_Dispatcher_Interface
     */
    public function clearParams($name = null);

    /**
     * Set the response object to use, if any
     *
     * @param Zend_Controller_Response_Abstract|null $response
     * @return void
     */
    public function setResponse(
        Zend_Controller_Response_Abstract $response = null
    );

    /**
     * Retrieve the response object, if any
     *
     * @return Zend_Controller_Response_Abstract|null
     */
    public function getResponse();

    /**
     * Add a controller directory to the controller directory stack
     *
     * @param string $path
     * @param string $args
     * @return Zend_Controller_Dispatcher_Interface
     */
    public function addControllerDirectory($path, $args = null);

    /**
     * Set the directory (or directories) where controller files are
     * stored
     *
     * @param string|array $dir
     * @return Zend_Controller_Dispatcher_Interface
     */
    public function setControllerDirectory($path);
```

```
    /**
     * Return the currently set directory(ies) for controller file
     * lookup
     *
     * @return array
     */
    public function getControllerDirectory();

    /**
     * Dispatch a request to a (module/)controller/action.
     *
     * @param  Zend_Controller_Request_Abstract $request
     * @param  Zend_Controller_Response_Abstract $response
     * @return Zend_Controller_Request_Abstract|boolean
     */
    public function dispatch(
        Zend_Controller_Request_Abstract $request,
        Zend_Controller_Response_Abstract $response
    );

    /**
     * Whether or not a given module is valid
     *
     * @param string $module
     * @return boolean
     */
    public function isValidModule($module);

    /**
     * Retrieve the default module name
     *
     * @return string
     */
    public function getDefaultModule();

    /**
     * Retrieve the default controller name
     *
     * @return string
     */
    public function getDefaultControllerName();

    /**
     * Retrieve the default action
     *
     * @return string
     */
    public function getDefaultAction();
}
```

In most cases, however, you should simply extend the abstract class `Zend_Controller_Dispatcher_Abstract`, in which each of these have already been defined, or `Zend_Controller_Dispatcher_Standard` to modify functionality of the standard dispatcher.

Possible reasons to subclass the dispatcher include a desire to use a different class or method naming schema in your action controllers, or a desire to use a different dispatching paradigm such as dispatching to action files under controller directories (instead of dispatching to class methods).

7. Action Controllers

7.1. Introduction

`Zend_Controller_Action` is an abstract class you may use for implementing Action Controllers for use with the Front Controller when building a website based on the Model-View-Controller (MVC) pattern.

To use `Zend_Controller_Action`, you will need to subclass it in your actual action controller classes (or subclass it to create your own base class for action controllers). The most basic operation is to subclass it, and create action methods that correspond to the various actions you wish the controller to handle for your site. `Zend_Controller`'s routing and dispatch handling will autodiscover any methods ending in 'Action' in your class as potential controller actions.

For example, let's say your class is defined as follows:

```
class FooController extends Zend_Controller_Action
{
    public function barAction()
    {
        // do something
    }

    public function bazAction()
    {
        // do something
    }
}
```

The above *FooController* class (controller *foo*) defines two actions, *bar* and *baz*.

There's much more that can be accomplished than this, such as custom initialization actions, default actions to call should no action (or an invalid action) be specified, pre- and post-dispatch hooks, and a variety of helper methods. This chapter serves as an overview of the action controller functionality

Default Behaviour

By default, the front controller enables the ViewRenderer action helper. This helper takes care of injecting the view object into the controller, as well as automatically rendering views. You may disable it within your action controller via one of the following methods:

```
class FooController extends Zend_Controller_Action
{
    public function init()
    {
        // Local to this controller only; affects all actions,
        // as loaded in init:
        $this->_helper->viewRenderer->setNoRender(true);

        // Globally:
        $this->_helper->removeHelper('viewRenderer');

        // Also globally, but would need to be in conjunction with the
        // local version in order to propagate for this controller:
        Zend_Controller_Front::getInstance()
```

```
            ->setParam('noViewRenderer', true);
    }
}
```

`initView()`, `getViewScript()`, `render()`, and `renderScript()` each proxy to the *ViewRenderer* unless the helper is not in the helper broker or the *noViewRenderer* flag has been set.

You can also simply disable rendering for an individual view by setting the *ViewRenderer*'s *noRender* flag:

```
class FooController extends Zend_Controller_Action
{
    public function barAction()
    {
        // disable autorendering for this action only:
        $this->_helper->viewRenderer->setNoRender();
    }
}
```

The primary reasons to disable the *ViewRenderer* are if you simply do not need a view object or if you are not rendering via view scripts (for instance, when using an action controller to serve web service protocols such as SOAP, XML-RPC, or REST). In most cases, you will never need to globally disable the *ViewRenderer*, only selectively within individual controllers or actions.

7.2. Object Initialization

While you can always override the action controller's constructor, we do not recommend this. `Zend_Controller_Action::__construct()` performs some important tasks, such as registering the request and response objects, as well as any custom invocation arguments passed in from the front controller. If you must override the constructor, be sure to call `parent::__construct($request, $response, $invokeArgs)`.

The more appropriate way to customize instantiation is to use the `init()` method, which is called as the last task of `__construct()`. For example, if you want to connect to a database at instantiation:

```
class FooController extends Zend_Controller_Action
{
    public function init()
    {
        $this->db = Zend_Db::factory('Pdo_Mysql', array(
            'host'     => 'myhost',
            'username' => 'user',
            'password' => 'XXXXXXX',
            'dbname'   => 'website'
        ));
    }
}
```

7.3. Pre- and Post-Dispatch Hooks

`Zend_Controller_Action` specifies two methods that may be called to bookend a requested action, `preDispatch()` and `postDispatch()`. These can be useful in a variety of ways: verifying authentication and ACL's prior to running an action (by calling `_forward()` in

`preDispatch()`, the action will be skipped), for instance, or placing generated content in a sitewide template (`postDispatch()`).

Usage of init() vs. preDispatch()

In the previous section, we introduced the `init()` method, and in this section, the `preDispatch()` method. What is the difference between them, and what actions would you take in each?

The `init()` method is primarily intended for extending the constructor. Typically, your constructor should simply set object state, and not perform much logic. This might include initializing resources used in the controller (such as models, configuration objects, etc.), or assigning values retrieved from the front controller, bootstrap, or a registry.

The `preDispatch()` method can also be used to set object or environmental (e.g., view, action helper, etc.) state, but its primary purpose is to make decisions about whether or not the requested action should be dispatched. If not, you should then `_forward()` to another action, or throw an exception.

Note: `_forward()` actually will not work correctly when executed from `init()`, which is a formalization of the intentions of the two methods.

7.4. Accessors

A number of objects and variables are registered with the object, and each has accessor methods.

- *Request Object*: `getRequest()` may be used to retrieve the request object used to call the action.
- *Response Object*: `getResponse()` may be used to retrieve the response object aggregating the final response. Some typical calls might look like:

```
$this->getResponse()->setHeader('Content-Type', 'text/xml');
$this->getResponse()->appendBody($content);
```

- *Invocation Arguments*: the front controller may push parameters into the router, dispatcher, and action controller. To retrieve these, use `getInvokeArg($key)`; alternatively, fetch the entire list using `getInvokeArgs()`.
- *Request parameters*: The request object aggregates request parameters, such as any `_GET` or `_POST` parameters, or user parameters specified in the URL's path information. To retrieve these, use `_getParam($key)` or `_getAllParams()`. You may also set request parameters using `_setParam()`; this is useful when forwarding to additional actions.

  To test whether or not a parameter exists (useful for logical branching), use `_hasParam($key)`.

`_getParam()` may take an optional second argument containing a default value to use if the parameter is not set or is empty. Using it eliminates the need to call `_hasParam()` prior to retrieving a value:

```
// Use default value of 1 if id is not set
```

```
$id = $this->_getParam('id', 1);

// Instead of:
if ($this->_hasParam('id') {
    $id = $this->_getParam('id');
} else {
    $id = 1;
}
```

7.5. View Integration

Default View Integration is Via the ViewRenderer

The content in this section is only valid when you have explicitly disabled the ViewRenderer. Otherwise, you can safely skip over this section.

`Zend_Controller_Action` provides a rudimentary and flexible mechanism for view integration. Two methods accomplish this, `initView()` and `render()`; the former method lazy-loads the `$view` public property, and the latter renders a view based on the current requested action, using the directory hierarchy to determine the script path.

7.5.1. View Initialization

`initView()` initializes the view object. `render()` calls `initView()` in order to retrieve the view object, but it may be initialized at any time; by default it populates the `$view` property with a `Zend_View` object, but any class implementing `Zend_View_Interface` may be used. If `$view` is already initialized, it simply returns that property.

The default implementation makes the following assumption of the directory structure:

```
applicationOrModule/
    controllers/
        IndexController.php
    views/
        scripts/
            index/
                index.phtml
        helpers/
        filters/
```

In other words, view scripts are assumed to be in the `/views/scripts/` subdirectory, and the `/views/` subdirectory is assumed to contain sibling functionality (helpers, filters). When determining the view script name and path, the `/views/scripts/` directory will be used as the base path, with directories named after the individual controllers providing a hierarchy of view scripts.

7.5.2. Rendering Views

`render()` has the following signature:

```
string render(string $action = null,
              string $name = null,
              bool $noController = false);
```

`render()` renders a view script. If no arguments are passed, it assumes that the script requested is `[controller]/[action].phtml` (where `.phtml` is the value of the `$viewSuffix` property). Passing a value for `$action` will render that template in the `/[controller]/` subdirectory. To override using the `/[controller]/` subdirectory, pass a `TRUE` value for `$noController`. Finally, templates are rendered into the response object; if you wish to render to a specific named segment in the response object, pass a value to `$name`.

Since controller and action names may contain word delimiter characters such as '_', '.', and '-', `render()` normalizes these to '-' when determining the script name. Internally, it uses the dispatcher's word and path delimiters to do this normalization. Thus, a request to `/foo.bar/baz-bat` will render the script `foo-bar/baz-bat.phtml`. If your action method contains camelCasing, please remember that this will result in '-' separated words when determining the view script file name.

Some examples:

```
class MyController extends Zend_Controller_Action
{
    public function fooAction()
    {
        // Renders my/foo.phtml
        $this->render();

        // Renders my/bar.phtml
        $this->render('bar');

        // Renders baz.phtml
        $this->render('baz', null, true);

        // Renders my/login.phtml to the 'form' segment of the
        // response object
        $this->render('login', 'form');

        // Renders site.phtml to the 'page' segment of the response
        // object; does not use the 'my/' subirectory
        $this->render('site', 'page', true);
    }

    public function bazBatAction()
    {
        // Renders my/baz-bat.phtml
        $this->render();
    }
}
```

7.6. Utility Methods

Besides the accessors and view integration methods, `Zend_Controller_Action` has several utility methods for performing common tasks from within your action methods (or from pre- and post-dispatch).

- `_forward($action, $controller = null, $module = null, array $params = null)`: perform another action. If called in `preDispatch()`, the currently requested action will be skipped in favor of the new one. Otherwise, after the current action is processed, the action requested in `_forward()` will be executed.

- `_redirect($url, array $options = array())`: redirect to another location. This method takes a URL and an optional set of options. By default, it performs an HTTP 302 redirect.

  The options may include one or more of the following:

  - *exit:* whether or not to exit immediately. If requested, it will cleanly close any open sessions and perform the redirect.

    You may set this option globally within the controller using the `setRedirectExit()` accessor.

  - *prependBase:* whether or not to prepend the base URL registered with the request object to the URL provided.

    You may set this option globally within the controller using the `setRedirectPrependBase()` accessor.

  - *code:* what HTTP code to utilize in the redirect. By default, an HTTP 302 is utilized; any code between 301 and 306 may be used.

    You may set this option globally within the controller using the `setRedirectCode()` accessor.

7.7. Subclassing the Action Controller

By design, `Zend_Controller_Action` must be subclassed in order to create an action controller. At the minimum, you will need to define action methods that the controller may call.

Besides creating useful functionality for your web applications, you may also find that you're repeating much of the same setup or utility methods in your various controllers; if so, creating a common base controller class that extends `Zend_Controller_Action` could solve such redundancy.

Example 88. Handling Non-Existent Actions

If a request to a controller is made that includes an undefined action method, `Zend_Controller_Action::__call()` will be invoked. `__call()` is, of course, PHP's magic method for method overloading.

By default, this method throws a `Zend_Controller_Action_Exception` indicating the requested method was not found in the controller. If the method requested ends in 'Action', the assumption is that an action was requested and does not exist; such errors result in an exception with a code of 404. All other methods result in an exception with a code of 500. This allows you to easily differentiate between page not found and application errors in your error handler.

You should override this functionality if you wish to perform other operations. For instance, if you wish to display an error message, you might write something like this:

```
class MyController extends Zend_Controller_Action
{
    public function __call($method, $args)
    {
        if ('Action' == substr($method, -6)) {
            // If the action method was not found, render the error
            // template
            return $this->render('error');
        }

        // all other methods throw an exception
        throw new Exception('Invalid method "'
                            . $method
                            . '" called',
                            500);
    }
}
```

Another possibility is that you may want to forward on to a default controller page:

```
class MyController extends Zend_Controller_Action
{
    public function indexAction()
    {
        $this->render();
    }

    public function __call($method, $args)
    {
        if ('Action' == substr($method, -6)) {
            // If the action method was not found, forward to the
            // index action
            return $this->_forward('index');
        }

        // all other methods throw an exception
        throw new Exception('Invalid method "'
                            . $method
                            . '" called',
                            500);
    }
}
```

Besides overriding `__call()`, each of the initialization, utility, accessor, view, and dispatch hook methods mentioned previously in this chapter may be overridden in order to customize your controllers. As an example, if you are storing your view object in a registry, you may want to modify your `initView()` method with code resembling the following:

```
abstract class My_Base_Controller extends Zend_Controller_Action
{
    public function initView()
    {
        if (null === $this->view) {
            if (Zend_Registry::isRegistered('view')) {
                $this->view = Zend_Registry::get('view');
            } else {
                $this->view = new Zend_View();
                $this->view->setBasePath(dirname(__FILE__) . '/../views');
            }
        }

        return $this->view;
    }
}
```

Hopefully, from the information in this chapter, you can see the flexibility of this particular component and how you can shape it to your application's or site's needs.

8. Action Helpers

8.1. Introduction

Action Helpers allow developers to inject runtime and/or on-demand functionality into any Action Controllers that extend `Zend_Controller_Action`. Action Helpers aim to minimize the necessity to extend the abstract Action Controller in order to inject common Action Controller functionality.

There are a number of ways to use Action Helpers. Action Helpers employ the use of a brokerage system, similar to the types of brokerage you see in Zend_View_Helper, and that of Zend_Controller_Plugin. Action Helpers (like `Zend_View_Helper`) may be loaded and called on demand, or they may be instantiated at request time (bootstrap) or action controller creation time (`init()`). To understand this more fully, please see the usage section below.

8.2. Helper Initialization

A helper can be initialized in several different ways, based on your needs as well as the functionality of that helper.

The helper broker is stored as the `$_helper` member of `Zend_Controller_Action`; use the broker to retrieve or call on helpers. Some methods for doing so include:

- Explicitly using `getHelper()`. Simply pass it a name, and a helper object is returned:

```
$flashMessenger = $this->_helper->getHelper('FlashMessenger');
$flashMessenger->addMessage('We did something in the last request');
```

- Use the helper broker's `__get()` functionality and retrieve the helper as if it were a member property of the broker:

```
$flashMessenger = $this->_helper->FlashMessenger;
$flashMessenger->addMessage('We did something in the last request');
```

- Finally, most action helpers implement the method `direct()` which will call a specific, default method in the helper. In the example of the *FlashMessenger*, it calls `addMessage()`:

```
$this->_helper->FlashMessenger('We did something in the last request');
```

All of the above examples are functionally equivalent.

You may also instantiate helpers explicitly. You may wish to do this if using the helper outside of an action controller, or if you wish to pass a helper to the helper broker for use by any action. Instantiation is as per any other PHP class.

8.3. The Helper Broker

`Zend_Controller_Action_HelperBroker` handles the details of registering helper objects and helper paths, as well as retrieving helpers on-demand.

To register a helper with the broker, use `addHelper()`:

```
Zend_Controller_Action_HelperBroker::addHelper($helper);
```

Of course, instantiating and passing helpers to the broker is a bit time and resource intensive, so two methods exists to automate things slightly: `addPrefix()` and `addPath()`.

- `addPrefix()` takes a class prefix and uses it to determine a path where helper classes have been defined. It assumes the prefix follows Zend Framework class naming conventions.

```
// Add helpers prefixed with My_Action_Helpers in My/Action/Helpers/
Zend_Controller_Action_HelperBroker::addPrefix('My_Action_Helpers');
```

- `addPath()` takes a directory as its first argument and a class prefix as the second argument (defaulting to '`Zend_Controller_Action_Helper`'). This allows you to map your own class prefixes to specific directories.

```
// Add helpers prefixed with Helper in Plugins/Helpers/
Zend_Controller_Action_HelperBroker::addPath('./Plugins/Helpers',
                                             'Helper');
```

Since these methods are static, they may be called at any point in the controller chain in order to dynamically add helpers as needed.

Internally, the helper broker uses a PluginLoader instance to maintain paths. You can retrieve the PluginLoader using the static method `getPluginLoader()`, or, alternately, inject a custom PluginLoader instance using `setPluginLoader()`.

To determine if a helper exists in the helper broker, use `hasHelper($name)`, where `$name` is the short name of the helper (minus the prefix):

```
// Check if 'redirector' helper is registered with the broker:
```

```
if (Zend_Controller_Action_HelperBroker::hasHelper('redirector')) {
    echo 'Redirector helper registered';
}
```

There are also two static methods for retrieving helpers from the helper broker: `getExistingHelper()` and `getStaticHelper()`. `getExistingHelper()` will retrieve a helper only if it has previously been invoked by or explicitly registered with the helper broker; it will throw an exception if not. `getStaticHelper()` does the same as `getExistingHelper()`, but will attempt to instantiate the helper if has not yet been registered with the helper stack. `getStaticHelper()` is a good choice for retrieving helpers which you wish to configure.

Both methods take a single argument, `$name`, which is the short name of the helper (minus the prefix).

```
// Check if 'redirector' helper is registered with the broker, and fetch:
if (Zend_Controller_Action_HelperBroker::hasHelper('redirector')) {
    $redirector =
        Zend_Controller_Action_HelperBroker::getExistingHelper('redirector');
}

// Or, simply retrieve it, not worrying about whether or not it was
// previously registered:
$redirector =
    Zend_Controller_Action_HelperBroker::getStaticHelper('redirector');
}
```

Finally, to delete a registered helper from the broker, use `removeHelper($name)`, where `$name` is the short name of the helper (minus the prefix):

```
// Conditionally remove the 'redirector' helper from the broker:
if (Zend_Controller_Action_HelperBroker::hasHelper('redirector')) {
    Zend_Controller_Action_HelperBroker::removeHelper('redirector')
}
```

8.4. Built-in Action Helpers

Zend Framework includes several action helpers by default: *AutoComplete* for automating responses for AJAX autocompletion; *ContextSwitch* and *AjaxContext* for serving alternate response formats for your actions; a *FlashMessenger* for handling session flash messages; *Json* for encoding and sending JSON responses; a *Redirector*, to provide different implementations for redirecting to internal and external pages from your application; and a *ViewRenderer* to automate the process of setting up the view object in your controllers and rendering views.

8.4.1. ActionStack

The *ActionStack* helper allows you to push requests to the ActionStack front controller plugin, effectively helping you create a queue of actions to execute during the request. The helper allows you to add actions either by specifying new request objects or action - controller - module sets.

Invoking ActionStack Helper Initializes the ActionStack Plugin

Invoking the *ActionStack* helper implicitly registers the *ActionStack* plugin -- which means you do not need to explicitly register the *ActionStack* plugin to use this functionality.

Example 89. Adding a Task Using Action, Controller and Module Names

Often, it's simplest to simply specify the action, controller, and module (and optional request parameters), much as you would when calling `Zend_Controller_Action::_forward()`:

```
class FooController extends Zend_Controller_Action
{
    public function barAction()
    {
        // Add two actions to the stack
        // Add call to /foo/baz/bar/baz
        // (FooController::bazAction() with request var bar == baz)
        $this->_helper->actionStack('baz',
                                    'foo',
                                    'default',
                                    array('bar' => 'baz'));

        // Add call to /bar/bat
        // (BarController::batAction())
        $this->_helper->actionStack('bat', 'bar');
    }
}
```

Example 90. Adding a Task Using a Request Object

Sometimes the OOP nature of a request object makes most sense; you can pass such an object to the *ActionStack* helper as well.

```
class FooController extends Zend_Controller_Action
{
    public function barAction()
    {
        // Add two actions to the stack
        // Add call to /foo/baz/bar/baz
        // (FooController::bazAction() with request var bar == baz)
        $request = clone $this->getRequest();
        // Don't set controller or module; use current values
        $request->setActionName('baz')
                ->setParams(array('bar' => 'baz'));
        $this->_helper->actionStack($request);

        // Add call to /bar/bat
        // (BarController::batAction())
        $request = clone $this->getRequest();
        // don't set module; use current value
        $request->setActionName('bat')
                ->setControllerName('bar');
        $this->_helper->actionStack($request);
    }
}
```

8.4.2. AutoComplete

Many AJAX javascript libraries offer functionality for providing autocompletion whereby a selectlist of potentially matching results is displayed as the user types. The *AutoComplete* helper aims to simplify returning acceptable responses to such methods.

Since not all JS libraries implement autocompletion in the same way, the *AutoComplete* helper provides some abstract base functionality necessary to many libraries, and concrete implementations for individual libraries. Return types are generally either JSON arrays of strings, JSON arrays of arrays (with each member array being an associative array of metadata used to create the selectlist), or HTML.

Basic usage for each implementation is the same:

```
class FooController extends Zend_Controller_Action
{
    public function barAction()
    {
        // Perform some logic...

        // Encode and send response;
        $this->_helper->autoCompleteDojo($data);

        // Or explicitly:
        $response = $this->_helper->autoCompleteDojo
                                  ->sendAutoCompletion($data);

        // Or simply prepare autocompletion response:
        $response = $this->_helper->autoCompleteDojo
                                  ->prepareAutoCompletion($data);
    }
}
```

By default, autocompletion does the following:

- Disables layouts and ViewRenderer.
- Sets appropriate response headers.
- Sets response body with encoded or formatted autocompletion data.
- Sends response.

Available methods of the helper include:

- `disableLayouts()` can be used to disable layouts and the ViewRenderer. Typically, this is called within `prepareAutoCompletion()`.
- `encodeJson($data, $keepLayouts = false)` will encode data to JSON, optionally enabling or disabling layouts. Typically, this is called within `prepareAutoCompletion()`.
- `prepareAutoCompletion($data, $keepLayouts = false)` is used to prepare data in the response format necessary for the concrete implementation, optionally enabling or disabling layouts. The return value will vary based on the implementation.
- `sendAutoCompletion($data, $keepLayouts = false)` is used to send data in the response format necessary for the concrete implementation. It calls `prepareAutoCompletion()`, and then sends the response.
- `direct($data, $sendNow = true, $keepLayouts = false)` is used when calling the helper as a method of the helper broker. The `$sendNow` flag is used to determine whether to call `sendAutoCompletion()` or `prepareAutoCompletion()`, respectively.

Currently, *AutoComplete* supports the Dojo and Scriptaculous AJAX libraries.

8.4.2.1. AutoCompletion with Dojo

Dojo does not have an AutoCompletion widget per se, but has two widgets that can perform AutoCompletion: ComboBox and FilteringSelect. In both cases, they require a data store that implements the QueryReadStore; for more information on these topics, see the dojo.data documentation.

In Zend Framework, you can pass a simple indexed array to the AutoCompleteDojo helper, and it will return a JSON response suitable for use with such a store:

```
// within a controller action:
$this->_helper->autoCompleteDojo($data);
```

```
    public function getForm()
    {
        if (null === $this->_form) {
            $this->_form = new Zend_Form();
            $this->_form->setMethod('get')
                ->setAction(
                    $this->getRequest()->getBaseUrl() . '/test/process'
                )
                ->addElements(array(
                    'test' => array('type' => 'text', 'options' => array(
                        'filters'        => array('StringTrim'),
                        'dojoType'       => array('dijit.form.ComboBox'),
                        'store'          => 'testStore',
                        'autoComplete'   => 'false',
                        'hasDownArrow'   => 'true',
                        'label' => 'Your input:',
                    )),
                    'go' => array('type' => 'submit',
```

```
class TestController extends Zend_Controller_Action
{
    // ...

    /**
     * Landing page
     */
    public function indexAction()
    {
        $this->view->form = $this->getForm();
    }

    public function autocompleteAction()
    {
        if ('ajax' != $this->_getParam('format', false)) {
            return $this->_helper->redirector('index');
        }
        if ($this->getRequest()->isPost()) {
            return $this->_helper->redirector('index');
        }

        $match = trim($this->getRequest()->getQuery('test', ''));

        $matches = array();
        foreach ($this->getData() as $datum) {
            if (0 === strpos($datum, $match)) {
                $matches[] = $datum;
```

```
<?php // setup our data store: ?>
<div dojoType="custom.TestNameReadStore" jsId="testStore"
    url="<?php echo $this->baseUrl() ?>/unit-test/autocomplete/format/ajax"
    requestMethod="get"></div>

<?php // render our form: ?>
<?php echo $this->form ?>
<?php // setup Dojo-related CSS to load in HTML head: ?>
<?php $this->headStyle()->captureStart() ?>
@import "<?php echo $this->baseUrl()
?>/javascript/dijit/themes/tundra/tundra.css";
@import "<?php echo $this->baseUrl() ?>/javascript/dojo/resources/dojo.css";
<?php $this->headStyle()->captureEnd() ?>
<?php // setup javascript to load in HTML head, including all required
   // Dojo libraries: ?>
<?php $this->headScript()
        ->setAllowArbitraryAttributes(true)
        ->appendFile($this->baseUrl() . '/javascript/dojo/dojo.js',
            'text/javascript',
            array('djConfig' => 'parseOnLoad: true'))
        ->captureStart() ?>
djConfig.usePlainJson=true;
dojo.registerModulePath("custom", "../custom");
dojo.require("dojo.parser");
dojo.require("dojox.data.QueryReadStore");
dojo.require("dijit.form.ComboBox");
dojo.require("custom.TestNameReadStore");
```

We now have all the pieces to get Dojo AutoCompletion working.

Note the calls to view helpers such as headStyle and headScript; these are placeholders, which we can then render in the HTML head section of our layout view script.

8.4.2.2. AutoCompletion with Scriptaculous

Scriptaculous expects an HTML response in a specific format.

The helper to use with this library is 'AutoCompleteScriptaculous'. Simply provide it an array of data, and the helper will create an HTML response compatible with Ajax.Autocompleter.

8.4.3. ContextSwitch and AjaxContext

The *ContextSwitch* action helper is intended for facilitating returning different response formats on request. The *AjaxContext* helper is a specialized version of *ContextSwitch* that facilitates returning responses to XmlHttpRequests.

To enable either one, you must provide hinting in your controller as to what actions can respond to which contexts. If an incoming request indicates a valid context for the given action, the helper will then:

- Disable layouts, if enabled.
- Set an alternate view suffix, effectively requiring a separate view script for the context.
- Send appropriate response headers for the context desired.
- Optionally, call specified callbacks to setup the context and/or perform post-processing.

As an example, let's consider the following controller:

```
class NewsController extends Zend_Controller_Action
{
    /**
     * Landing page; forwards to listAction()
     */
    public function indexAction()
    {
        $this->_forward('list');
    }

    /**
     * List news items
     */
    public function listAction()
    {
    }

    /**
     * View a news item
     */
    public function viewAction()
    {
    }
}
```

Let's say that we want the `listAction()` to also be available in an XML format. Instead of creating a different action, we can hint that it can return an XML response:

```
class NewsController extends Zend_Controller_Action
{
    public function init()
    {
        $contextSwitch = $this->_helper->getHelper('contextSwitch');
```

```
        $contextSwitch->addActionContext('list', 'xml')
                      ->initContext();
    }

    // ...
}
```

What this will do is:

- Set the 'Content-Type' response header to '`text/xml`'.
- Change the view suffix to '`xml.phtml`' (or, if you use an alternate view suffix, 'xml.[your suffix]').

Now, you'll need to create a new view script, '`news/list.xml.phtml`', which will create and render the XML.

To determine if a request should initiate a context switch, the helper checks for a token in the request object. By default, it looks for the 'format' parameter, though this may be configured. This means that, in most cases, to trigger a context switch, you can add a 'format' parameter to your request:

- Via URL parameter: `/news/list/format/xml` (recall, the default routing schema allows for arbitrary key to value pairs following the action)
- Via `GET` parameter: **/news/list?format=xml**

ContextSwitch allows you to specify arbitrary contexts, including what suffix change will occur (if any), any response headers that should be sent, and arbitrary callbacks for initialization and post processing.

8.4.3.1. Default Contexts Available

By default, two contexts are available to the *ContextSwitch* helper: json and xml.

- *JSON*. The JSON context sets the 'Content-Type' response header to '`application/json`', and the view script suffix to '`json.phtml`'.

  By default, however, no view script is required. It will simply serialize all view variables, and emit the JSON response immediately.

  This behaviour can be disabled by turning off the automatic JSON serialization:

  ```
  $this->_helper->contextSwitch()->setAutoJsonSerialization(false);
  ```

- *XML*. The XML context sets the 'Content-Type' response header to '`text/xml`', and the view script suffix to '`xml.phtml`'. You will need to create a new view script for the context.

8.4.3.2. Creating Custom Contexts

Sometimes, the default contexts are not enough. For instance, you may wish to return YAML, or serialized PHP, an RSS or ATOM feed, etc. *ContextSwitch* allows you to do so.

The easiest way to add a new context is via the `addContext()` method. This method takes two arguments, the name of the context, and an array specification. The specification should include one or more of the following:

- *suffix*: the suffix to prepend to the default view suffix as registered in the ViewRenderer.
- *headers*: an array of header to value pairs you wish sent as part of the response.

- *callbacks*: an array containing one or more of the keys 'init' or 'post', pointing to valid PHP callbacks that can be used for context initialization and post processing.

  Initialization callbacks occur when the context is detected by *ContextSwitch*. You can use it to perform arbitrary logic that should occur. As an example, the JSON context uses a callback to disable the ViewRenderer when the automatic JSON serialization is on.

  Post processing occurs during the action's `postDispatch()` routine, and can be used to perform arbitrary logic. As an example, the JSON context uses a callback to determine if the automatic JSON serialization is on; if so, it serializes the view variables to JSON and sends the response, but if not, it re-enables the ViewRenderer.

There are a variety of methods for interacting with contexts:

- `addContext($context, array $spec)`: add a new context. Throws an exception if the context already exists.
- `setContext($context, array $spec)`: add a new context or overwrite an existing context. Uses the same specification as `addContext()`.
- `addContexts(array $contexts)`: add many contexts at once. The `$contexts` array should be an array of context to specification pairs. If any of the contexts already exists, it will throw an exception.
- `setContexts(array $contexts)`: add new contexts and overwrite existing ones. Uses the same specification as `addContexts()`.
- `hasContext($context)`: returns TRUE if the context exists, FALSE otherwise.
- `getContext($context)`: retrieve a single context by name. Returns an array following the specification used in `addContext()`.
- `getContexts()`: retrieve all contexts. Returns an array of context to specification pairs.
- `removeContext($context)`: remove a single context by name. Returns TRUE if successful, FALSE if the context was not found.
- `clearContexts()`: remove all contexts.

8.4.3.3. Setting Contexts Per Action

There are two mechanisms for setting available contexts. You can either manually create arrays in your controller, or use several methods in *ContextSwitch* to assemble them.

The principle method for adding action to context relations is `addActionContext()`. It expects two arguments, the action to which the context is being added, and either the name of a context or an array of contexts. As an example, consider the following controller class:

```
class FooController extends Zend_Controller_Action
{
    public function listAction()
    {
    }

    public function viewAction()
    {
    }

    public function commentsAction()
```

```
    {
    }

    public function updateAction()
    {
    }
}
```

Let's say we wanted to add an XML context to the 'list' action, and XML and JSON contexts to the 'comments' action. We could do so in the `init()` method:

```
class FooController extends Zend_Controller_Action
{
    public function init()
    {
        $this->_helper->contextSwitch()
             ->addActionContext('list', 'xml')
             ->addActionContext('comments', array('xml', 'json'))
             ->initContext();
    }
}
```

Alternately, you could simply define the array property `$contexts`:

```
class FooController extends Zend_Controller_Action
{
    public $contexts = array(
        'list'     => array('xml'),
        'comments' => array('xml', 'json')
    );

    public function init()
    {
        $this->_helper->contextSwitch()->initContext();
    }
}
```

The above is less overhead, but also prone to potential errors.

The following methods can be used to build the context mappings:

- `addActionContext($action, $context)`: marks one or more contexts as available to an action. If mappings already exists, simply appends to those mappings. `$context` may be a single context, or an array of contexts.

  A value of TRUE for the context will mark all available contexts as available for the action.

  An empty value for `$context` will disable all contexts for the given action.

- `setActionContext($action, $context)`: marks one or more contexts as available to an action. If mappings already exists, it replaces them with those specified. `$context` may be a single context, or an array of contexts.

- `addActionContexts(array $contexts)`: add several action to context pairings at once. `$contexts` should be an associative array of action to context pairs. It proxies to `addActionContext()`, meaning that if pairings already exist, it appends to them.

- `setActionContexts(array $contexts)`: acts like `addActionContexts()`, but overwrites existing action to context pairs.

- `hasActionContext($action, $context)`: determine if a particular action has a given context.
- `getActionContexts($action = null)`: returns either all contexts for a given action, or all action to context pairs.
- `removeActionContext($action, $context)`: remove one or more contexts from a given action. `$context` may be a single context or an array of contexts.
- `clearActionContexts($action = null)`: remove all contexts from a given action, or from all actions with contexts.

8.4.3.4. Initializing Context Switching

To initialize context switching, you need to call `initContext()` in your action controller:

```
class NewsController extends Zend_Controller_Action
{
    public function init()
    {
        $this->_helper->contextSwitch()->initContext();
    }
}
```

In some cases, you may want to force the context used; for instance, you may only want to allow the XML context if context switching is activated. You can do so by passing the context to `initContext()`:

```
$contextSwitch->initContext('xml');
```

8.4.3.5. Additional Functionality

A variety of methods can be used to alter the behaviour of the *ContextSwitch* helper. These include:

- `setAutoJsonSerialization($flag)`: By default, JSON contexts will serialize any view variables to JSON notation and return this as a response. If you wish to create your own response, you should turn this off; this needs to be done prior to the call to `initContext()`.

  ```
  $contextSwitch->setAutoJsonSerialization(false);
  $contextSwitch->initContext();
  ```

  You can retrieve the value of the flag with `getAutoJsonSerialization()`.

- `setSuffix($context, $suffix, $prependViewRendererSuffix)`: With this method, you can specify a different suffix to use for a given context. The third argument is used to indicate whether or not to prepend the current ViewRenderer suffix with the new suffix; this flag is enabled by default.

  Passing an empty value to the suffix will cause only the ViewRenderer suffix to be used.

- `addHeader($context, $header, $content)`: Add a response header for a given context. `$header` is the header name, and `$content` is the value to pass for that header.

  Each context can have multiple headers; `addHeader()` adds additional headers to the context's header stack.

  If the `$header` specified already exists for the context, an exception will be thrown.

- `setHeader($context, $header, $content)`: `setHeader()` acts just like `addHeader()`, except it allows you to overwrite existing context headers.
- `addHeaders($context, array $headers)`: Add multiple headers at once to a given context. Proxies to `addHeader()`, so if the header already exists, an exception will be thrown. `$headers` is an array of header to context pairs.
- `setHeaders($context, array $headers.)`: like `addHeaders()`, except it proxies to `setHeader()`, allowing you to overwrite existing headers.
- `getHeader($context, $header)`: retrieve the value of a header for a given context. Returns `NULL` if not found.
- `removeHeader($context, $header)`: remove a single header for a given context.
- `clearHeaders($context, $header)`: remove all headers for a given context.
- `setCallback($context, $trigger, $callback)`: set a callback at a given trigger for a given context. Triggers may be either 'init' or 'post' (indicating callback will be called at either context initialization or postDispatch). `$callback` should be a valid PHP callback.
- `setCallbacks($context, array $callbacks)`: set multiple callbacks for a given context. `$callbacks` should be trigger to callback pairs. In actuality, the most callbacks that can be registered are two, one for initialization and one for post processing.
- `getCallback($context, $trigger)`: retrieve a callback for a given trigger in a given context.
- `getCallbacks($context)`: retrieve all callbacks for a given context. Returns an array of trigger to callback pairs.
- `removeCallback($context, $trigger)`: remove a callback for a given trigger and context.
- `clearCallbacks($context)`: remove all callbacks for a given context.
- `setContextParam($name)`: set the request parameter to check when determining if a context switch has been requested. The value defaults to 'format', but this accessor can be used to set an alternate value.

  `getContextParam()` can be used to retrieve the current value.
- `setAutoDisableLayout($flag)`: By default, layouts are disabled when a context switch occurs; this is because typically layouts will only be used for returning normal responses, and have no meaning in alternate contexts. However, if you wish to use layouts (perhaps you may have a layout for the new context), you can change this behaviour by passing a `FALSE` value to `setAutoDisableLayout()`. You should do this *before* calling `initContext()`.

  To get the value of this flag, use the accessor `getAutoDisableLayout()`.
- `getCurrentContext()` can be used to determine what context was detected, if any. This returns `NULL` if no context switch occurred, or if called before `initContext()` has been invoked.

8.4.3.6. AjaxContext Functionality

The *AjaxContext* helper extends *ContextSwitch*, so all of the functionality listed for *ContextSwitch* is available to it. There are a few key differences, however.

First, it uses a different action controller property for determining contexts, `$ajaxable`. This is so you can have different contexts used for AJAX versus normal HTTP requests. The various *`ActionContext()`* methods of *AjaxContext* will write to this property.

Second, it will only trigger if an XmlHttpRequest has occurred, as determined by the request object's `isXmlHttpRequest()` method. Thus, if the context parameter ('format') is passed in the request, but the request was not made as an XmlHttpRequest, no context switch will trigger.

Third, *AjaxContext* adds an additional context, HTML. In this context, it sets the suffix to '`ajax.phtml`' in order to differentiate the context from a normal request. No additional headers are returned.

Example 92. Allowing Actions to Respond To Ajax Requests

In this following example, we're allowing requests to the actions 'view', 'form', and 'process' to respond to AJAX requests. In the first two cases, 'view' and 'form', we'll return HTML snippets with which to update the page; in the latter, we'll return JSON.

```
class CommentController extends Zend_Controller_Action
{
    public function init()
    {
        $ajaxContext = $this->_helper->getHelper('AjaxContext');
        $ajaxContext->addActionContext('view', 'html')
                    ->addActionContext('form', 'html')
                    ->addActionContext('process', 'json')
                    ->initContext();
    }

    public function viewAction()
    {
        // Pull a single comment to view.
        // When AjaxContext detected, uses the comment/view.ajax.phtml
        // view script.
    }

    public function formAction()
    {
        // Render the "add new comment" form.
        // When AjaxContext detected, uses the comment/form.ajax.phtml
        // view script.
    }

    public function processAction()
    {
        // Process a new comment
        // Return the results as JSON; simply assign the results as
        // view variables, and JSON will be returned.
    }
}
```

On the client end, your AJAX library will simply request the endpoints '`/comment/view`', '`/comment/form`', and '`/comment/process`', and pass the 'format' parameter: '`/comment/view/format/html`', '`/comment/form/format/html`', '`/comment/process/format/json`'. (Or you can pass the parameter via query string: e.g., "?format=json".)

Assuming your library passes the 'X-Requested-With: XmlHttpRequest' header, these actions will then return the appropriate response format.

8.4.4. FlashMessenger

8.4.4.1. Introduction

The *FlashMessenger* helper allows you to pass messages that the user may need to see on the next request. To accomplish this, *FlashMessenger* uses `Zend_Session_Namespace` to store messages for future or next request retrieval. It is generally a good idea that if you plan on using `Zend_Session` or `Zend_Session_Namespace`, that you initialize with `Zend_Session::start()` in your bootstrap file. (See the Zend_Session documentation for more details on its usage.)

8.4.4.2. Basic Usage Example

The usage example below shows the use of the flash messenger at its most basic. When the action `/some/my` is called, it adds the flash message "Record Saved!" A subsequent request to the action `/some/my-next-request` will retrieve it (and thus delete it as well).

```
class SomeController extends Zend_Controller_Action
{
    /**
     * FlashMessenger
     *
     * @var Zend_Controller_Action_Helper_FlashMessenger
     */
    protected $_flashMessenger = null;

    public function init()
    {
        $this->_flashMessenger =
            $this->_helper->getHelper('FlashMessenger');
        $this->initView();
    }

    public function myAction()
    {
        /**
         * default method of getting
         * Zend_Controller_Action_Helper_FlashMessenger instance
         * on-demand
         */
        $this->_flashMessenger->addMessage('Record Saved!');
    }

    public function myNextRequestAction()
    {
        $this->view->messages = $this->_flashMessenger->getMessages();
        $this->render();
    }
}
```

8.4.5. JSON

JSON responses are rapidly becoming the response of choice when dealing with AJAX requests that expect dataset responses; JSON can be immediately parsed on the client-side, leading to quick execution.

The JSON action helper does several things:

- Disables layouts if currently enabled.

- Optionally, an array of options to pass as the second argument to `Zend_Json::encode()`. This array of options allows enabling layouts and encoding using `Zend_Json_Expr`.

```
$this->_helper->json($data, array('enableJsonExprFinder' => true));
```

- Disables the ViewRenderer if currently enabled.
- Sets the 'Content-Type' response header to '`application/json`'.
- By default, immediately returns the response, without waiting for the action to finish execution.

Usage is simple: either call it as a method of the helper broker, or call one of the methods `encodeJson()` or `sendJson()`:

```
class FooController extends Zend_Controller_Action
{
    public function barAction()
    {
        // do some processing...
        // Send the JSON response:
        $this->_helper->json($data);

        // or...
        $this->_helper->json->sendJson($data);

        // or retrieve the json:
        $json = $this->_helper->json->encodeJson($data);
    }
}
```

Keeping Layouts

If you have a separate layout for JSON responses -- perhaps to wrap the JSON response in some sort of context -- each method in the JSON helper accepts a second, optional argument: a flag to enable or disable layouts. Passing a boolean TRUE value will keep layouts enabled:

```
$this->_helper->json($data, true);
```

Optionally, you can pass an array as the second parameter. This array may contain a variety of options, including the *keepLayouts* option:

```
$this->_helper->json($data, array('keepLayouts' => true);
```

Enabling encoding using Zend_Json_Expr

`Zend_Json::encode()` allows the encoding of native JSON expressions using `Zend_Json_Expr` objects. This option is disabled by default. To enable this option, pass a boolean TRUE value to the *enableJsonExprFinder* option:

```
$this->_helper->json($data, array('enableJsonExprFinder' => true);
```

If you desire to do this, you *must* pass an array as the second argument. This also allows you to combine other options, such as the *keepLayouts* option. All such options are then passed to `Zend_Json::encode()`.

```
$this->_helper->json($data, array(
    'enableJsonExprFinder' => true,
    'keepLayouts'          => true,
));
```

8.4.6. Redirector

8.4.6.1. Introduction

The *Redirector* helper allows you to use a redirector object to fulfill your application's needs for redirecting to a new URL. It provides numerous benefits over the `_redirect()` method, such as being able to preconfigure sitewide behavior into the redirector object or using the built in `gotoSimple($action, $controller, $module, $params)` interface similar to that of `Zend_Controller_Action::_forward()`.

The *Redirector* has a number of methods that can be used to affect the behaviour at redirect:

- `setCode()` can be used to set the HTTP response code to use during the redirect.
- `setExit()` can be used to force an `exit()` following a redirect. By default this is `TRUE`.
- `setGotoSimple()` can be used to set a default URL to use if none is passed to `gotoSimple()`. Uses the API of `Zend_Controller_Action::_forward()`: `setGotoSimple($action, $controller = null, $module = null, array $params = array())`
- `setGotoRoute()` can be used to set a URL based on a registered route. Pass in an array of key / value pairs and a route name, and it will assemble the URL according to the route type and definition.
- `setGotoUrl()` can be used to set a default URL to use if none is passed to `gotoUrl()`. Accepts a single URL string.
- `setPrependBase()` can be used to prepend the request object's base URL to a URL specified with `setGotoUrl()`, `gotoUrl()`, or `gotoUrlAndExit()`.
- `setUseAbsoluteUri()` can be used to force the *Redirector* to use absolute URIs when redirecting. When this option is set, it uses the value of `$_SERVER['HTTP_HOST']`, `$_SERVER['SERVER_PORT']`, and `$_SERVER['HTTPS']` to form a full URI to the URL specified by one of the redirect methods. This option is off by default, but may be enabled by default in later releases.

Additionally, there are a variety of methods in the redirector for performing the actual redirects:

- `gotoSimple()` uses `setGotoSimple()` (`_forward()`-like API) to build a URL and perform a redirect.
- `gotoRoute()` uses `setGotoRoute()` (*route-assembly*) to build a URL and perform a redirect.
- `gotoUrl()` uses `setGotoUrl()` (*URL string*) to build a URL and perform a redirect.

Finally, you can determine the current redirect URL at any time using `getRedirectUrl()`.

8.4.6.2. Basic Usage Examples

Example 93. Setting Options

This example overrides several options, including setting the HTTP status code to use in the redirect ('303'), not defaulting to exit on redirect, and defining a default URL to use when redirecting.

```
class SomeController extends Zend_Controller_Action
{
    /**
     * Redirector - defined for code completion
     *
     * @var Zend_Controller_Action_Helper_Redirector
     */
    protected $_redirector = null;

    public function init()
    {
        $this->_redirector = $this->_helper->getHelper('Redirector');

        // Set the default options for the redirector
        // Since the object is registered in the helper broker, these
        // become relevant for all actions from this point forward
        $this->_redirector->setCode(303)
                          ->setExit(false)
                          ->setGotoSimple("this-action",
                                          "some-controller");
    }

    public function myAction()
    {
        /* do some stuff */

        // Redirect to a previously registered URL, and force an exit
        // to occur when done:
        $this->_redirector->redirectAndExit();
        return; // never reached
    }
}
```

Example 94. Using Defaults

This example assumes that the defaults are used, which means that any redirect will result in an immediate `exit()`.

```
// ALTERNATIVE EXAMPLE
class AlternativeController extends Zend_Controller_Action
{
    /**
     * Redirector - defined for code completion
     *
     * @var Zend_Controller_Action_Helper_Redirector
     */
    protected $_redirector = null;

    public function init()
    {
        $this->_redirector = $this->_helper->getHelper('Redirector');
    }

    public function myAction()
    {
        /* do some stuff */

        $this->_redirector
            ->gotoUrl('/my-controller/my-action/param1/test/param2/test2');
        return; // never reached since default is to goto and exit
    }
}
```

Example 95. Using goto()'s _forward() API

`gotoSimple()`'s API mimics that of `Zend_Controller_Action::_forward()`. The primary difference is that it builds a URL from the parameters passed, and using the default `:module/:controller/:action/*` format of the default router. It then redirects instead of chaining the action.

```
class ForwardController extends Zend_Controller_Action
{
    /**
     * Redirector - defined for code completion
     *
     * @var Zend_Controller_Action_Helper_Redirector
     */
    protected $_redirector = null;

    public function init()
    {
        $this->_redirector = $this->_helper->getHelper('Redirector');
    }

    public function myAction()
    {
        /* do some stuff */

        // Redirect to 'my-action' of 'my-controller' in the current
        // module, using the params param1 => test and param2 => test2
        $this->_redirector->gotoSimple('my-action',
                                       'my-controller',
                                       null,
                                       array('param1' => 'test',
                                             'param2' => 'test2'
                                             )
                                       );
    }
}
```

Example 96. Using Route Assembly with gotoRoute()

The following example uses the router's `assemble()` method to create a URL based on an associative array of parameters passed. It assumes the following route has been registered:

```
$route = new Zend_Controller_Router_Route(
    'blog/:year/:month/:day/:id',
    array('controller' => 'archive',
          'module' => 'blog',
          'action' => 'view')
);
$router->addRoute('blogArchive', $route);
```

Given an array with year set to 2006, month to 4, day to 24, and id to 42, it would then build the URL `/blog/2006/4/24/42`.

```
class BlogAdminController extends Zend_Controller_Action
{
    /**
     * Redirector - defined for code completion
     *
     * @var Zend_Controller_Action_Helper_Redirector
     */
    protected $_redirector = null;

    public function init()
    {
        $this->_redirector = $this->_helper->getHelper('Redirector');
    }

    public function returnAction()
    {
        /* do some stuff */

        // Redirect to blog archive. Builds the following URL:
        // /blog/2006/4/24/42
        $this->_redirector->gotoRoute(
            array('year' => 2006,
                  'month' => 4,
                  'day' => 24,
                  'id' => 42),
            'blogArchive'
        );
    }
}
```

8.4.7. ViewRenderer

8.4.7.1. Introduction

The *ViewRenderer* helper is designed to satisfy the following goals:

- Eliminate the need to instantiate view objects within controllers; view objects will be automatically registered with the controller.
- Automatically set view script, helper, and filter paths based on the current module, and automatically associate the current module name as a class prefix for helper and filter classes.
- Create a globally available view object for all dispatched controllers and actions.

- Allow the developer to set default view rendering options for all controllers.
- Add the ability to automatically render a view script with no intervention.
- Allow the developer to create her own specifications for the view base path and for view script paths.

If you perform a `_forward()`, `redirect()`, or `render()` manually, autorendering will not occur, as by performing any of these actions you are telling the *ViewRenderer* that you are determining your own output.

The *ViewRenderer* is enabled by default. You may disable it via the front controller *noViewRenderer* param (**$front->setParam('noViewRenderer', true);**) or removing the helper from the helper broker stack (`Zend_Controller_Action_HelperBroker::removeHelper('viewRenderer')`).

If you wish to modify settings of the *ViewRenderer* prior to dispatching the front controller, you may do so in one of two ways:

- Instantiate and register your own *ViewRenderer* object and pass it to the helper broker:

```
$viewRenderer = new Zend_Controller_Action_Helper_ViewRenderer();
$viewRenderer->setView($view)
             ->setViewSuffix('php');
Zend_Controller_Action_HelperBroker::addHelper($viewRenderer);
```

- Initialize and/or retrieve a *ViewRenderer* object on demand via the helper broker:

```
$viewRenderer =
    Zend_Controller_Action_HelperBroker::getStaticHelper('viewRenderer');
$viewRenderer->setView($view)
             ->setViewSuffix('php');
```

8.4.7.2. API

At its most basic usage, you simply instantiate the *ViewRenderer* and pass it to the action helper broker. The easiest way to instantiate it and register in one go is to use the helper broker's `getStaticHelper()` method:

```
Zend_Controller_Action_HelperBroker::getStaticHelper('viewRenderer');
```

The first time an action controller is instantiated, it will trigger the *ViewRenderer* to instantiate a view object. Each time a controller is instantiated, the *ViewRenderer*'s `init()` method is called, which will cause it to set the view property of the action controller, and call `addScriptPath()` with a path relative to the current module; this will be called with a class prefix named after the current module, effectively namespacing all helper and filter classes you define for the module.

Each time `postDispatch()` is called, it will call `render()` for the current action.

As an example, consider the following class:

```
// A controller class, foo module:
class Foo_BarController extends Zend_Controller_Action
{
    // Render bar/index.phtml by default; no action required
    public function indexAction()
    {
    }

    // Render bar/populate.phtml with variable 'foo' set to 'bar'.
    // Since view object defined at preDispatch(), it's already available.
    public function populateAction()
    {
        $this->view->foo = 'bar';
    }
}

...

// in one of your view scripts:
$this->foo(); // call Foo_View_Helper_Foo::foo()
```

The *ViewRenderer* also defines a number of accessors to allow setting and retrieving view options:

- `setView($view)` allows you to set the view object for the *ViewRenderer*. It gets set as the public class property `$view`.
- `setNeverRender($flag = true)` can be used to disable or enable autorendering globally, i.e., for all controllers. If set to `TRUE`, `postDispatch()` will not automatically call `render()` in the current controller. `getNeverRender()` retrieves the current value.
- `setNoRender($flag = true)` can be used to disable or enable autorendering. If set to `TRUE`, `postDispatch()` will not automatically call `render()` in the current controller. This setting is reset each time `preDispatch()` is called (i.e., you need to set this flag for each controller for which you don't want autorenderering to occur). `getNoRender()` retrieves the current value.
- `setNoController($flag = true)` can be used to tell `render()` not to look for the action script in a subdirectory named after the controller (which is the default behaviour). `getNoController()` retrieves the current value.
- `setNeverController($flag = true)` is analogous to `setNoController()`, but works on a global level -- i.e., it will not be reset for each dispatched action. `getNeverController()` retrieves the current value.
- `setScriptAction($name)` can be used to specify the action script to render. `$name` should be the name of the script minus the file suffix (and without the controller subdirectory, unless *noController* has been turned on). If not specified, it looks for a view script named after the action in the request object. `getScriptAction()` retrieves the current value.
- `setResponseSegment($name)` can be used to specify which response object named segment to render into. If not specified, it renders into the default segment. `getResponseSegment()` retrieves the current value.
- `initView($path, $prefix, $options)` may be called to specify the base view path, class prefix for helper and filter scripts, and *ViewRenderer* options. You may pass any of the following flags: *neverRender*, *noRender*, *noController*, *scriptAction*, and *responseSegment*.

- `setRender($action = null, $name = null, $noController = false)` allows you to set any of *scriptAction*, *responseSegment*, and *noController* in one pass. `direct()` is an alias to this method, allowing you to call this method easily from your controller:

```
// Render 'foo' instead of current action script
$this->_helper->viewRenderer('foo');

// render form.phtml to the 'html' response segment, without using a
// controller view script subdirectory:
$this->_helper->viewRenderer('form', 'html', true);
```

`setRender()` and `direct()` don't actually render the view script, but instead set hints that `postDispatch()` and `render()` will use to render the view.

The constructor allows you to optionally pass the view object and *ViewRenderer* options; it accepts the same flags as `initView()`:

```
$view    = new Zend_View(array('encoding' => 'UTF-8'));
$options = array('noController' => true, 'neverRender' => true);
$viewRenderer =
    new Zend_Controller_Action_Helper_ViewRenderer($view, $options);
```

There are several additional methods for customizing path specifications used for determining the view base path to add to the view object, and the view script path to use when autodetermining the view script to render. These methods each take one or more of the following placeholders:

- *:moduleDir* refers to the current module's base directory (by convention, the parent directory of the module's controller directory).
- *:module* refers to the current module name.
- *:controller* refers to the current controller name.
- *:action* refers to the current action name.
- *:suffix* refers to the view script suffix (which may be set via `setViewSuffix()`).

The methods for controlling path specifications are:

- `setViewBasePathSpec($spec)` allows you to change the path specification used to determine the base path to add to the view object. The default specification is `:moduleDir/views`. You may retrieve the current specification at any time using `getViewBasePathSpec()`.
- `setViewScriptPathSpec($spec)` allows you to change the path specification used to determine the path to an individual view script (minus the base view script path). The default specification is `:controller/:action.:suffix`. You may retrieve the current specification at any time using `getViewScriptPathSpec()`.
- `setViewScriptPathNoControllerSpec($spec)` allows you to change the path specification used to determine the path to an individual view script when *noController* is in effect (minus the base view script path). The default specification is `:action.:suffix`. You may retrieve the current specification at any time using `getViewScriptPathNoControllerSpec()`.

For fine-grained control over path specifications, you may use Zend_Filter_Inflector. Under the hood, the *ViewRenderer* uses an inflector to perform path mappings already. To interact with the inflector -- either to set your own for use, or to modify the default inflector, the following methods may be used:

- `getInflector()` will retrieve the inflector. If none exists yet in the *ViewRenderer*, it creates one using the default rules.

  By default, it uses static rule references for the suffix and module directory, as well as a static target; this allows various *ViewRenderer* properties the ability to dynamically modify the inflector.

- `setInflector($inflector, $reference)` allows you to set a custom inflector for use with the *ViewRenderer*. If `$reference` is `TRUE`, it will set the suffix and module directory as static references to *ViewRenderer* properties, as well as the target.

Default Lookup Conventions

The *ViewRenderer* does some path normalization to make view script lookups easier. The default rules are as follows:

- *:module*: MixedCase and camelCasedWords are separated by dashes, and the entire string cast to lowercase. E.g.: "FooBarBaz" becomes "foo-bar-baz".

  Internally, the inflector uses the filters `Zend_Filter_Word_CamelCaseToDash` and `Zend_Filter_StringToLower`.

- *:controller*: MixedCase and camelCasedWords are separated by dashes; underscores are converted to directory separators, and the entire string cast to lower case. Examples: "`FooBar`" becomes "foo-bar"; "`FooBar_Admin`" becomes "`foo-bar/admin`".

  Internally, the inflector uses the filters `Zend_Filter_Word_CamelCaseToDash`, `Zend_Filter_Word_UnderscoreToSeparator`, and `Zend_Filter_StringToLower`.

- *:action*: MixedCase and camelCasedWords are separated by dashes; non-alphanumeric characters are translated to dashes, and the entire string cast to lower case. Examples: "fooBar" becomes "foo-bar"; "foo-barBaz" becomes "foo-bar-baz".

  Internally, the inflector uses the filters `Zend_Filter_Word_CamelCaseToDash`, `Zend_Filter_PregReplace`, and `Zend_Filter_StringToLower`.

The final items in the *ViewRenderer* API are the methods for actually determining view script paths and rendering views. These include:

- `renderScript($script, $name)` allows you to render a script with a path you specify, optionally to a named path segment. When using this method, the *ViewRenderer* does no autodetermination of the script name, but instead directly passes the `$script` argument directly to the view object's `render()` method.

Once the view has been rendered to the response object, it sets the *noRender* to prevent accidentally rendering the same view script multiple times.

By default, `Zend_Controller_Action::renderScript()` proxies to the *ViewRenderer*'s `renderScript()` method.

- `getViewScript($action, $vars)` creates the path to a view script based on the action passed and/or any variables passed in `$vars`. Keys for this array may include any of the path specification keys ('moduleDir', 'module', 'controller', 'action', and 'suffix'). Any variables passed will be used; otherwise, values based on the current request will be utlized.

  `getViewScript()` will use either the *viewScriptPathSpec* or *viewScriptPathNoControllerSpec* based on the setting of the *noController* flag.

  Word delimiters occurring in module, controller, or action names will be replaced with dashes ('-'). Thus, if you have the controller name '**foo.bar**' and the action '**baz:bat**', using the default path specification will result in a view script path of '`foo-bar/baz-bat.phtml`'.

  By default, `Zend_Controller_Action::getViewScript()` proxies to the *ViewRenderer*'s `getViewScript()` method.

- `render($action, $name, $noController)` checks first to see if either `$name` or `$noController` have been passed, and if so, sets the appropriate flags (responseSegment and noController, respectively) in the ViewRenderer. It then passes the `$action` argument, if any, on to `getViewScript()`. Finally, it passes the calculated view script path to `renderScript()`.

  Be aware of the side-effects of using `render()`: the values you pass for the response segment name and for the noController flag will persist in the object. Additionally, noRender will be set after rendering is completed.

  By default, `Zend_Controller_Action::render()` proxies to the *ViewRenderer*'s `render()` method.

- `renderBySpec($action, $vars, $name)` allows you to pass path specification variables in order to determine the view script path to create. It passes `$action` and `$vars` to `getScriptPath()`, and then passes the resulting script path and `$name` on to `renderScript()`.

8.4.7.3. Basic Usage Examples

Example 97. Basic Usage

At its most basic, you simply initialize and register the *ViewRenderer* helper with the helper broker in your bootstrap, and then set variables in your action methods.

```
// In your bootstrap:
Zend_Controller_Action_HelperBroker::getStaticHelper('viewRenderer');

...

// 'foo' module, 'bar' controller:
class Foo_BarController extends Zend_Controller_Action
{
    // Render bar/index.phtml by default; no action required
    public function indexAction()
    {
    }

    // Render bar/populate.phtml with variable 'foo' set to 'bar'.
    // Since view object defined at preDispatch(), it's already available.
    public function populateAction()
    {
        $this->view->foo = 'bar';
    }

    // Renders nothing as it forwards to another action; the new action
    // will perform any rendering
    public function bazAction()
    {
        $this->_forward('index');
    }

    // Renders nothing as it redirects to another location
    public function batAction()
    {
        $this->_redirect('/index');
    }
}
```

Naming Conventions: Word Delimiters in Controller and Action Names

If your controller or action name is composed of several words, the dispatcher requires that these are separated on the URL by specific path and word delimiter characters. The *ViewRenderer* replaces any path delimiter found in the controller name with an actual path delimiter ('/'), and any word delimiter found with a dash ('-') when creating paths. Thus, a call to the action `/foo.bar/baz.bat` would dispatch to `FooBarController::bazBatAction()` in `FooBarController.php`, which would render `foo-bar/baz-bat.phtml`; a call to the action `/bar_baz/baz-bat` would dispatch to `Bar_BazController::bazBatAction()` in `Bar/BazController.php` (note the path separation) and render `bar/baz/baz-bat.phtml`.

Note that the in the second example, the module is still the default module, but that, because of the existence of a path separator, the controller receives the name `Bar_BazController`, in `Bar/BazController.php`. The ViewRenderer mimics the controller directory hierarchy.

Example 98. Disabling Autorender

For some actions or controllers, you may want to turn off the autorendering -- for instance, if you're wanting to emit a different type of output (XML, JSON, etc), or if you simply want to emit nothing. You have two options: turn off all cases of autorendering (`setNeverRender()`), or simply turn it off for the current action (`setNoRender()`).

```
// Baz controller class, bar module:
class Bar_BazController extends Zend_Controller_Action
{
    public function fooAction()
    {
        // Don't auto render this action
        $this->_helper->viewRenderer->setNoRender();
    }
}

// Bat controller class, bar module:
class Bar_BatController extends Zend_Controller_Action
{
    public function preDispatch()
    {
        // Never auto render this controller's actions
        $this->_helper->viewRenderer->setNoRender();
    }
}
```

In most cases, it makes no sense to turn off autorendering globally (ala `setNeverRender()`), as the only thing you then gain from *ViewRenderer* is the autosetup of the view object.

Example 99. Choosing a Different View Script

Some situations require that you render a different script than one named after the action. For instance, if you have a controller that has both add and edit actions, they may both display the same 'form' view, albeit with different values set. You can easily change the script name used with either `setScriptAction()`, `setRender()`, or calling the helper as a method, which will invoke `setRender()`.

```
// Bar controller class, foo module:
class Foo_BarController extends Zend_Controller_Action
{
    public function addAction()
    {
        // Render 'bar/form.phtml' instead of 'bar/add.phtml'
        $this->_helper->viewRenderer('form');
    }

    public function editAction()
    {
        // Render 'bar/form.phtml' instead of 'bar/edit.phtml'
        $this->_helper->viewRenderer->setScriptAction('form');
    }

    public function processAction()
    {
        // do some validation...
        if (!$valid) {
            // Render 'bar/form.phtml' instead of 'bar/process.phtml'
            $this->_helper->viewRenderer->setRender('form');
            return;
        }

        // otherwise continue processing...
    }

}
```

Example 100. Modifying the Registered View

What if you need to modify the view object -- for instance, change the helper paths, or the encoding? You can do so either by modifying the view object set in your controller, or by grabbing the view object out of the *ViewRenderer*; both are references to the same object.

```
// Bar controller class, foo module:
class Foo_BarController extends Zend_Controller_Action
{
    public function preDispatch()
    {
        // change view encoding
        $this->view->setEncoding('UTF-8');
    }

    public function bazAction()
    {
        // Get view object and set escape callback to 'htmlspecialchars'
        $view = $this->_helper->viewRenderer->view;
        $view->setEscape('htmlspecialchars');
    }
}
```

8.4.7.4. Advanced Usage Examples

Example 101. Changing the Path Specifications

In some circumstances, you may decide that the default path specifications do not fit your site's needs. For instance, you may want to have a single template tree to which you may then give access to your designers (this is very typical when using Smarty, for instance). In such a case, you may want to hardcode the view base path specification, and create an alternate specification for the action view script paths themselves.

For purposes of this example, let's assume that the base path to views should be '`/opt/vendor/templates`', and that you wish for view scripts to be referenced by '`:moduleDir/:controller/:action.:suffix`'; if the *noController* flag has been set, you want to render out of the top level instead of in a subdirectory ('`:action.:suffix`'). Finally, you want to use 'tpl' as the view script filename suffix.

```
/**
 * In your bootstrap:
 */

// Different view implementation
$view = new ZF_Smarty();

$viewRenderer = new Zend_Controller_Action_Helper_ViewRenderer($view);
$viewRenderer->setViewBasePathSpec('/opt/vendor/templates')
             ->setViewScriptPathSpec(':module/:controller/:action.:suffix')
             ->setViewScriptPathNoControllerSpec(':action.:suffix')
             ->setViewSuffix('tpl');
Zend_Controller_Action_HelperBroker::addHelper($viewRenderer);
```

Example 102. Rendering Multiple View Scripts from a Single Action

At times, you may need to render multiple view scripts from a single action. This is very straightforward -- simply make multiple calls to `render()`:

```
class SearchController extends Zend_Controller_Action
{
    public function resultsAction()
    {
        // Assume $this->model is the current model
        $this->view->results =
            $this->model->find($this->_getParam('query', ''));

        // render() by default proxies to the ViewRenderer
        // Render first the search form and then the results
        $this->render('form');
        $this->render('results');
    }

    public function formAction()
    {
        // do nothing; ViewRenderer autorenders the view script
    }
}
```

8.5. Writing Your Own Helpers

Action helpers extend `Zend_Controller_Action_Helper_Abstract`, an abstract class that provides the basic interface and functionality required by the helper broker. These include the following methods:

- `setActionController()` is used to set the current action controller.
- `init()`, triggered by the helper broker at instantiation, can be used to trigger initialization in the helper; this can be useful for resetting state when multiple controllers use the same helper in chained actions.
- `preDispatch()`, is triggered prior to a dispatched action.
- `postDispatch()` is triggered when a dispatched action is done -- even if a `preDispatch()` plugin has skipped the action. Mainly useful for cleanup.
- `getRequest()` retrieves the current request object.
- `getResponse()` retrieves the current response object.
- `getName()` retrieves the helper name. It retrieves the portion of the class name following the last underscore character, or the full class name otherwise. As an example, if the class is named `Zend_Controller_Action_Helper_Redirector`, it will return *Redirector*; a class named *FooMessage* will simply return itself.

You may optionally include a `direct()` method in your helper class. If defined, it allows you to treat the helper as a method of the helper broker, in order to allow easy, one-off usage of the helper. As an example, the redirector defines `direct()` as an alias of `goto()`, allowing use of the helper like this:

```
// Redirect to /blog/view/item/id/42
$this->_helper->redirector('item', 'view', 'blog', array('id' => 42));
```

Internally, the helper broker's `__call()` method looks for a helper named *redirector*, then checks to see if that helper has a defined `direct()` method, and calls it with the arguments provided.

Once you have created your own helper class, you may provide access to it as described in the sections above.

9. The Response Object

9.1. Usage

The response object is the logical counterpart to the request object. Its purpose is to collate content and/or headers so that they may be returned en masse. Additionally, the front controller will pass any caught exceptions to the response object, allowing the developer to gracefully handle exceptions. This functionality may be overridden by setting `Zend_Controller_Front::throwExceptions(true)`:

```
$front->throwExceptions(true);
```

To send the response output, including headers, use `sendResponse()`.

```
$response->sendResponse();
```

By default, the front controller calls `sendResponse()` when it has finished dispatching the request; typically you will never need to call it. However, if you wish to manipulate the response or use it in testing, you can override this behaviour by setting the returnResponse flag with `Zend_Controller_Front::returnResponse(true)`:

```
$front->returnResponse(true);
$response = $front->dispatch();

// do some more processing, such as logging...
// and then send the output:
$response->sendResponse();
```

Developers should make use of the response object in their action controllers. Instead of directly rendering output and sending headers, push them to the response object:

```
// Within an action controller action:
// Set a header
$this->getResponse()
    ->setHeader('Content-Type', 'text/html')
    ->appendBody($content);
```

By doing this, all headers get sent at once, just prior to displaying the content.

If using the action controller view integration, you do not need to set the rendered view script content in the response object, as `Zend_Controller_Action::render()` does this by default.

Should an exception occur in an application, check the response object's `isException()` flag, and retrieve the exception using `getException()`. Additionally, one may create custom response objects that redirect to error pages, log exception messages, do pretty formatting of exception messages (for development environments), etc.

You may retrieve the response object following the front controller `dispatch()`, or request the front controller to return the response object instead of rendering output.

```
// retrieve post-dispatch:
$front->dispatch();
$response = $front->getResponse();
if ($response->isException()) {
    // log, mail, etc...
}

// Or, have the front controller dispatch() process return it
$front->returnResponse(true);
$response = $front->dispatch();

// do some processing...

// finally, echo the response
$response->sendResponse();
```

By default, exception messages are not displayed. This behaviour may be overridden by calling `renderExceptions()`, or enabling the front controller to `throwExceptions()`, as shown above:

```
$response->renderExceptions(true);
$front->dispatch($request, $response);

// or:
$front->returnResponse(true);
$response = $front->dispatch();
$response->renderExceptions();
$response->sendResponse();

// or:
$front->throwExceptions(true);
$front->dispatch();
```

9.2. Manipulating Headers

As stated previously, one aspect of the response object's duties is to collect and emit HTTP response headers. A variety of methods exist for this:

- `canSendHeaders()` is used to determine if headers have already been sent. It takes an optional flag indicating whether or not to throw an exception if headers have already been sent. This can be overridden by setting the property headersSentThrowsException to FALSE.
- `setHeader($name, $value, $replace = false)` is used to set an individual header. By default, it does not replace existing headers of the same name in the object; however, setting `$replace` to TRUE will force it to do so.

  Before setting the header, it checks with `canSendHeaders()` to see if this operation is allowed at this point, and requests that an exception be thrown.
- `setRedirect($url, $code = 302)` sets an HTTP Location header for a redirect. If an HTTP status code has been provided, it will use that status code.

  Internally, it calls `setHeader()` with the `$replace` flag on to ensure only one such header is ever sent.
- `getHeaders()` returns an array of all headers. Each array element is an array with the keys 'name' and 'value'.
- `clearHeaders()` clears all registered headers.
- `setRawHeader()` can be used to set headers that are not key and value pairs, such as an HTTP status header.
- `getRawHeaders()` returns any registered raw headers.
- `clearRawHeaders()` clears any registered raw headers.
- `clearAllHeaders()` clears both regular key and value headers as well as raw headers.

In addition to the above methods, there are accessors for setting and retrieving the HTTP response code for the current request, `setHttpResponseCode()` and `getHttpResponseCode()`.

9.3. Named Segments

The response object has support for "named segments". This allows you to segregate body content into different segments and order those segments so output is returned in a specific order. Internally, body content is saved as an array, and the various accessor methods can be used to indicate placement and names within that array.

As an example, you could use a `preDispatch()` hook to add a header to the response object, then have the action controller add body content, and a `postDispatch()` hook add a footer:

```
// Assume that this plugin class is registered with the front controller
class MyPlugin extends Zend_Controller_Plugin_Abstract
{
    public function preDispatch(Zend_Controller_Request_Abstract $request)
    {
        $response = $this->getResponse();
        $view = new Zend_View();
        $view->setBasePath('../views/scripts');

        $response->prepend('header', $view->render('header.phtml'));
    }

    public function postDispatch(Zend_Controller_Request_Abstract $request)
    {
        $response = $this->getResponse();
        $view = new Zend_View();
        $view->setBasePath('../views/scripts');

        $response->append('footer', $view->render('footer.phtml'));
    }
}

// a sample action controller
class MyController extends Zend_Controller_Action
{
    public function fooAction()
    {
        $this->render();
    }
}
```

In the above example, a call to `/my/foo` will cause the final body content of the response object to have the following structure:

```
array(
    'header'  => ..., // header content
    'default' => ..., // body content from MyController::fooAction()
    'footer'  => ...  // footer content
);
```

When this is rendered, it will render in the order in which elements are arranged in the array.

A variety of methods can be used to manipulate the named segments:

- `setBody()` and `appendBody()` both allow you to pass a second value, `$name`, indicating a named segment. In each case, if you provide this, it will overwrite that specific named segment or create it if it does not exist (appending to the array by default). If no named segment is

passed to `setBody()`, it resets the entire body content array. If no named segment is passed to `appendBody()`, the content is appended to the value in the 'default' name segment.

- `prepend($name, $content)` will create a segment named `$name` and place it at the beginning of the array. If the segment exists already, it will be removed prior to the operation (i.e., overwritten and replaced).

- `append($name, $content)` will create a segment named `$name` and place it at the end of the array. If the segment exists already, it will be removed prior to the operation (i.e., overwritten and replaced).

- `insert($name, $content, $parent = null, $before = false)` will create a segment named `$name`. If provided with a `$parent` segment, the new segment will be placed either before or after that segment (based on the value of `$before`) in the array. If the segment exists already, it will be removed prior to the operation (i.e., overwritten and replaced).

- `clearBody($name = null)` will remove a single named segment if a `$name` is provided (and the entire array otherwise).

- `getBody($spec = false)` can be used to retrieve a single array segment if `$spec` is the name of a named segment. If `$spec` is FALSE, it returns a string formed by concatenating all named segments in order. If `$spec` is TRUE, it returns the body content array.

9.4. Testing for Exceptions in the Response Object

As mentioned earlier, by default, exceptions caught during dispatch are registered with the response object. Exceptions are registered in a stack, which allows you to keep all exceptions thrown -- application exceptions, dispatch exceptions, plugin exceptions, etc. Should you wish to check for particular exceptions or to log exceptions, you'll want to use the response object's exception API:

- `setException(Exception $e)` allows you to register an exception.

- `isException()` will tell you if an exception has been registered.

- `getException()` returns the entire exception stack.

- `hasExceptionOfType($type)` allows you to determine if an exception of a particular class is in the stack.

- `hasExceptionOfMessage($message)` allows you to determine if an exception with a specific message is in the stack.

- `hasExceptionOfCode($code)` allows you to determine if an exception with a specific code is in the stack.

- `getExceptionByType($type)` allows you to retrieve all exceptions of a specific class from the stack. It will return FALSE if none are found, and an array of exceptions otherwise.

- `getExceptionByMessage($message)` allows you to retrieve all exceptions with a specific message from the stack. It will return FALSE if none are found, and an array of exceptions otherwise.

- `getExceptionByCode($code)` allows you to retrieve all exceptions with a specific code from the stack. It will return FALSE if none are found, and an array of exceptions otherwise.

- `renderExceptions($flag)` allows you to set a flag indicating whether or not exceptions should be emitted when the response is sent.

9.5. Subclassing the Response Object

The purpose of the response object is to collect headers and content from the various actions and plugins and return them to the client; secondarily, it also collects any errors (exceptions) that occur in order to process them, return them, or hide them from the end user.

The base response class is `Zend_Controller_Response_Abstract`, and any subclass you create should extend that class or one of its derivatives. The various methods available have been listed in the previous sections.

Reasons to subclass the response object include modifying how output is returned based on the request environment (e.g., not sending headers for CLI or PHP-GTK requests), adding functionality to return a final view based on content stored in named segments, etc.

10. Plugins

10.1. Introduction

The controller architecture includes a plugin system that allows user code to be called when certain events occur in the controller process lifetime. The front controller uses a plugin broker as a registry for user plugins, and the plugin broker ensures that event methods are called on each plugin registered with the front controller.

The event methods are defined in the abstract class `Zend_Controller_Plugin_Abstract`, from which user plugin classes inherit:

- `routeStartup()` is called before `Zend_Controller_Front` calls on the router to evaluate the request against the registered routes.
- `routeShutdown()` is called after the router finishes routing the request.
- `dispatchLoopStartup()` is called before `Zend_Controller_Front` enters its dispatch loop.
- `preDispatch()` is called before an action is dispatched by the dispatcher. This callback allows for proxy or filter behavior. By altering the request and resetting its dispatched flag (via `Zend_Controller_Request_Abstract::setDispatched(false)`), the current action may be skipped and/or replaced.
- `postDispatch()` is called after an action is dispatched by the dispatcher. This callback allows for proxy or filter behavior. By altering the request and resetting its dispatched flag (via `Zend_Controller_Request_Abstract::setDispatched(false)`), a new action may be specified for dispatching.
- `dispatchLoopShutdown()` is called after `Zend_Controller_Front` exits its dispatch loop.

10.2. Writing Plugins

In order to write a plugin class, simply include and extend the abstract class `Zend_Controller_Plugin_Abstract`:

```
class MyPlugin extends Zend_Controller_Plugin_Abstract
{
    // ...
}
```

None of the methods of `Zend_Controller_Plugin_Abstract` are abstract, and this means that plugin classes are not forced to implement any of the available event methods listed above. Plugin writers may implement only those methods required by their particular needs.

`Zend_Controller_Plugin_Abstract` also makes the request and response objects available to controller plugins via the `getRequest()` and `getResponse()` methods, respectively.

10.3. Using Plugins

Plugin classes are registered with `Zend_Controller_Front::registerPlugin()`, and may be registered at any time. The following snippet illustrates how a plugin may be used in the controller chain:

```
class MyPlugin extends Zend_Controller_Plugin_Abstract
{
    public function routeStartup(Zend_Controller_Request_Abstract $request)
    {
        $this->getResponse()
             ->appendBody("<p>routeStartup() called</p>\n");
    }

    public function routeShutdown(Zend_Controller_Request_Abstract $request)
    {
        $this->getResponse()
             ->appendBody("<p>routeShutdown() called</p>\n");
    }

    public function dispatchLoopStartup(
        Zend_Controller_Request_Abstract $request)
    {
        $this->getResponse()
             ->appendBody("<p>dispatchLoopStartup() called</p>\n");
    }

    public function preDispatch(Zend_Controller_Request_Abstract $request)
    {
        $this->getResponse()
             ->appendBody("<p>preDispatch() called</p>\n");
    }

    public function postDispatch(Zend_Controller_Request_Abstract $request)
    {
        $this->getResponse()
             ->appendBody("<p>postDispatch() called</p>\n");
    }

    public function dispatchLoopShutdown()
    {
        $this->getResponse()
             ->appendBody("<p>dispatchLoopShutdown() called</p>\n");
    }
}
```

```
$front = Zend_Controller_Front::getInstance();
$front->setControllerDirectory('/path/to/controllers')
      ->setRouter(new Zend_Controller_Router_Rewrite())
      ->registerPlugin(new MyPlugin());
$front->dispatch();
```

Assuming that no actions called emit any output, and only one action is called, the functionality of the above plugin would still create the following output:

```
<p>routeStartup() called</p>
<p>routeShutdown() called</p>
<p>dispatchLoopStartup() called</p>
<p>preDispatch() called</p>
<p>postDispatch() called</p>
<p>dispatchLoopShutdown() called</p>
```

Plugins may be registered at any time during the front controller execution. However, if an event has passed for which the plugin has a registered event method, that method will not be triggered.

10.4. Retrieving and Manipulating Plugins

On occasion, you may need to unregister or retrieve a plugin. The following methods of the front controller allow you to do so:

- `getPlugin($class)` allows you to retrieve a plugin by class name. If no plugins match, it returns FALSE. If more than one plugin of that class is registered, it returns an array.
- `getPlugins()` retrieves the entire plugin stack.
- `unregisterPlugin($plugin)` allows you to remove a plugin from the stack. You may pass a plugin object, or the class name of the plugin you wish to unregister. If you pass the class name, any plugins of that class will be removed.

10.5. Plugins Included in the Standard Distribution

Zend Framework includes a plugin for error handling in its standard distribution.

10.5.1. ActionStack

The *ActionStack* plugin allows you to manage a stack of requests, and operates as a *postDispatch* plugin. If a forward (i.e., a call to another action) is already detected in the current request object, it does nothing. However, if not, it checks its stack and pulls the topmost item off it and forwards to the action specified in that request. The stack is processed in LIFO order.

You can retrieve the plugin from the front controller at any time using `Zend_Controller_Front::getPlugin('Zend_Controller_Plugin_ActionStack')`. Once you have the plugin object, there are a variety of mechanisms you can use to manipulate it.

- `getRegistry()` and `setRegistry()`. Internally, *ActionStack* uses a `Zend_Registry` instance to store the stack. You can substitute a different registry instance or retrieve it with these accessors.

- `getRegistryKey()` and `setRegistryKey()`. These can be used to indicate which registry key to use when pulling the stack. Default value is 'Zend_Controller_Plugin_ActionStack'.
- `getStack()` allows you to retrieve the stack of actions in its entirety.
- `pushStack()` and `popStack()` allow you to add to and pull from the stack, respectively. `pushStack()` accepts a request object.

An additional method, `forward()`, expects a request object, and sets the state of the current request object in the front controller to the state of the provided request object, and markes it as undispatched (forcing another iteration of the dispatch loop).

10.5.2. Zend_Controller_Plugin_ErrorHandler

`Zend_Controller_Plugin_ErrorHandler` provides a drop-in plugin for handling exceptions thrown by your application, including those resulting from missing controllers or actions; it is an alternative to the methods listed in the MVC Exceptions section.

The primary targets of the plugin are:

- Intercept exceptions raised when no route matched
- Intercept exceptions raised due to missing controllers or action methods
- Intercept exceptions raised within action controllers

In other words, the *ErrorHandler* plugin is designed to handle HTTP 404-type errors (page missing) and 500-type errors (internal error). It is not intended to catch exceptions raised in other plugins.

By default, `Zend_Controller_Plugin_ErrorHandler` will forward to `ErrorController::errorAction()` in the default module. You may set alternate values for these by using the various accessors available to the plugin:

- `setErrorHandlerModule()` sets the controller module to use.
- `setErrorHandlerController()` sets the controller to use.
- `setErrorHandlerAction()` sets the controller action to use.
- `setErrorHandler()` takes an associative array, which may contain any of the keys 'module', 'controller', or 'action', with which it will set the appropriate values.

Additionally, you may pass an optional associative array to the constructor, which will then proxy to `setErrorHandler()`.

`Zend_Controller_Plugin_ErrorHandler` registers a `postDispatch()` hook and checks for exceptions registered in the response object. If any are found, it attempts to forward to the registered error handler action.

If an exception occurs dispatching the error handler, the plugin will tell the front controller to throw exceptions, and rethrow the last exception registered with the response object.

10.5.2.1. Using the ErrorHandler as a 404 Handler

Since the *ErrorHandler* plugin captures not only application errors, but also errors in the controller chain arising from missing controller classes and/or action methods, it can be used as a 404 handler. To do so, you will need to have your error controller check the exception type.

Exceptions captured are logged in an object registered in the request. To retrieve it, use `Zend_Controller_Action::_getParam('error_handler')`:

```
class ErrorController extends Zend_Controller_Action
{
    public function errorAction()
    {
        $errors = $this->_getParam('error_handler');
    }
}
```

Once you have the error object, you can get the type via **$errors->type;**. It will be one of the following:

- `Zend_Controller_Plugin_ErrorHandler::EXCEPTION_NO_ROUTE`, indicating no route matched.
- `Zend_Controller_Plugin_ErrorHandler::EXCEPTION_NO_CONTROLLER`, indicating the controller was not found.
- `Zend_Controller_Plugin_ErrorHandler::EXCEPTION_NO_ACTION`, indicating the requested action was not found.
- `Zend_Controller_Plugin_ErrorHandler::EXCEPTION_OTHER`, indicating other exceptions.

You can then test for either of the first three types, and, if so, indicate a 404 page:

```
class ErrorController extends Zend_Controller_Action
{
    public function errorAction()
    {
        $errors = $this->_getParam('error_handler');

        switch ($errors->type) {
            case Zend_Controller_Plugin_ErrorHandler::EXCEPTION_NO_ROUTE:
            case Zend_Controller_Plugin_ErrorHandler::EXCEPTION_NO_CONTROLLER:
            case Zend_Controller_Plugin_ErrorHandler::EXCEPTION_NO_ACTION:
                // 404 error -- controller or action not found
                $this->getResponse()
                     ->setRawHeader('HTTP/1.1 404 Not Found');

                // ... get some output to display...
                break;
            default:
                // application error; display error page, but don't
                // change status code
                break;
        }
    }
}
```

Finally, you can retrieve the exception that triggered the error handler by grabbing the exception property of the *error_handler* object:

```
public function errorAction()
{
```

```
        $errors = $this->_getParam('error_handler');

        switch ($errors->type) {
            case Zend_Controller_Plugin_ErrorHandler::EXCEPTION_NO_ROUTE:
            case Zend_Controller_Plugin_ErrorHandler::EXCEPTION_NO_CONTROLLER:
            case Zend_Controller_Plugin_ErrorHandler::EXCEPTION_NO_ACTION:
                // 404 error -- controller or action not found
                $this->getResponse()
                     ->setRawHeader('HTTP/1.1 404 Not Found');

                // ... get some output to display...
                break;
            default:
                // application error; display error page, but don't change
                // status code

                // ...

                // Log the exception:
                $exception = $errors->exception;
                $log = new Zend_Log(
                    new Zend_Log_Writer_Stream(
                        '/tmp/applicationException.log'
                    )
                );
                $log->debug($exception->getMessage() . "\n" .
                            $exception->getTraceAsString());
                break;
        }
}
```

10.5.2.2. Handling Previously Rendered Output

If you dispatch multiple actions in a request, or if your action makes multiple calls to `render()`, it's possible that the response object already has content stored within it. This can lead to rendering a mixture of expected content and error content.

If you wish to render errors inline in such pages, no changes will be necessary. If you do not wish to render such content, you should clear the response body prior to rendering any views:

```
$this->getResponse()->clearBody();
```

10.5.2.3. Plugin Usage Examples

Example 103. Standard Usage

```
$front = Zend_Controller_Front::getInstance();
$front->registerPlugin(new Zend_Controller_Plugin_ErrorHandler());
```

Example 104. Setting a Different Error Handler

```
$front = Zend_Controller_Front::getInstance();
$front->registerPlugin(new Zend_Controller_Plugin_ErrorHandler(array(
    'module'     => 'mystuff',
    'controller' => 'static',
    'action'     => 'error'
)));
```

Example 105. Using Accessors

```
$plugin = new Zend_Controller_Plugin_ErrorHandler();
$plugin->setErrorHandlerModule('mystuff')
       ->setErrorHandlerController('static')
       ->setErrorHandlerAction('error');

$front = Zend_Controller_Front::getInstance();
$front->registerPlugin($plugin);
```

10.5.2.4. Error Controller Example

In order to use the Error Handler plugin, you need an error controller. Below is a simple example.

```
class ErrorController extends Zend_Controller_Action
{
    public function errorAction()
    {
        $errors = $this->_getParam('error_handler');

        switch ($errors->type) {
            case Zend_Controller_Plugin_ErrorHandler::EXCEPTION_NO_ROUTE:
            case Zend_Controller_Plugin_ErrorHandler::EXCEPTION_NO_CONTROLLER:
            case Zend_Controller_Plugin_ErrorHandler::EXCEPTION_NO_ACTION:
                // 404 error -- controller or action not found
                $this->getResponse()->setRawHeader('HTTP/1.1 404 Not Found');

                $content =<<<EOH
<h1>Error!</h1>
<p>The page you requested was not found.</p>
EOH;
                break;
            default:
                // application error
                $content =<<<EOH
<h1>Error!</h1>
<p>An unexpected error occurred. Please try again later.</p>
EOH;
                break;
        }

        // Clear previous content
        $this->getResponse()->clearBody();

        $this->view->content = $content;
    }
}
```

10.5.3. Zend_Controller_Plugin_PutHandler

`Zend_Controller_Plugin_PutHandler` provides a drop-in plugin for marshalling `PUT` request bodies into request parameters, just like `POST` request bodies. It will inspect the request and, if `PUT`, will use parse_str to parse the raw `PUT` body into an array of params which is then set on the request. E.g.,

```
PUT /notes/5.xml HTTP/1.1

title=Hello&body=World
```

To receive the 'title' and 'body' params as regular request params, register the plugin:

```
$front = Zend_Controller_Front::getInstance();
$front->registerPlugin(new Zend_Controller_Plugin_PutHandler());
```

Then you can access the PUT body params by name from the request inside your controller:

```
...
public function putAction()
{
    $title = $this->getRequest()->getParam('title'); // $title = "Hello"
    $body = $this->getRequest()->getParam('body'); // $body = "World"
}
...
```

11. Using a Conventional Modular Directory Structure

11.1. Introduction

The Conventional Modular directory structure allows you to separate different MVC applications into self-contained units, and re-use them with different front controllers. To illustrate such a directory structure:

```
docroot/
    index.php
application/
    default/
        controllers/
            IndexController.php
            FooController.php
        models/
        views/
            scripts/
                index/
                foo/
            helpers/
            filters/
    blog/
        controllers/
            IndexController.php
        models/
        views/
            scripts/
                index/
            helpers/
            filters/
    news/
        controllers/
            IndexController.php
            ListController.php
        models/
        views/
            scripts/
                index/
                list/
            helpers/
            filters/
```

In this paradigm, the module name serves as a prefix to the controllers it contains. The above example contains three module controllers, 'Blog_IndexController', 'News_IndexController', and 'News_ListController'. Two global controllers, 'IndexController' and 'FooController' are also defined; neither of these will be namespaced. This directory structure will be used for examples in this chapter.

No Namespacing in the Default Module

Note that in the default module, controllers do not need a namespace prefix. Thus, in the example above, the controllers in the default module do not need a prefix of 'Default_' -- they are simply dispatched according to their base controller name: 'IndexController' and 'FooController'. A namespace prefix is used in all other modules, however.

So, how do you implement such a directory layout using the Zend Framework MVC components?

11.2. Specifying Module Controller Directories

The first step to making use of modules is to modify how you specify the controller directory list in the front controller. In the basic MVC series, you pass either an array or a string to `setControllerDirectory()`, or a path to `addControllerDirectory()`. When using modules, you need to alter your calls to these methods slightly.

With `setControllerDirectory()`, you will need to pass an associative array and specify key and value pairs of module name and directory paths. The special key default will be used for global controllers (those not needing a module namespace). All entries should contain a string key pointing to a single path, and the default key must be present. As an example:

```
$front->setControllerDirectory(array(
    'default' => '/path/to/application/controllers',
    'blog'    => '/path/to/application/blog/controllers'
));
```

`addControllerDirectory()` will take an optional second argument. When using modules, pass the module name as the second argument; if not specified, the path will be added to the *default* namespace. As an example:

```
$front->addControllerDirectory('/path/to/application/news/controllers',
                               'news');
```

Saving the best for last, the easiest way to specify module directories is to do so en masse, with all modules under a common directory and sharing the same structure. This can be done with `addModuleDirectory()`:

```
/**
 * Assuming the following directory structure:
 * application/
 *     modules/
 *         default/
 *             controllers/
 *         foo/
 *             controllers/
 *         bar/
 *             controllers/
 */
$front->addModuleDirectory('/path/to/application/modules');
```

The above example will define the *default*, *foo*, and *bar* modules, each pointing to the `controllers/` subdirectory of their respective module.

You may customize the controller subdirectory to use within your modules by using `setModuleControllerDirectoryName()`:

```
/**
 * Change the controllers subdirectory to be 'con'
 * application/
 *     modules/
 *         default/
 *             con/
 *         foo/
 *             con/
 *         bar/
 *             con/
 */
$front->setModuleControllerDirectoryName('con');
$front->addModuleDirectory('/path/to/application/modules');
```

You can indicate that no controller subdirectory be used for your modules by passing an empty value to `setModuleControllerDirectoryName()`.

11.3. Routing to Modules

The default route in `Zend_Controller_Router_Rewrite` is an object of type `Zend_Controller_Router_Route_Module`. This route expects one of the following routing schemas:

- `:module/:controller/:action/*`
- `:controller/:action/*`

In other words, it will match a controller and action by themselves or with a preceding module. The rules for matching specify that a module will only be matched if a key of the same name exists in the controller directory array passed to the front controller and dispatcher.

11.4. Module or Global Default Controller

In the default router, if a controller was not specified in the URL, a default controller is used (`IndexController`, unless otherwise requested). With modular controllers, if a module has been specified but no controller, the dispatcher first looks for this default controller in the module path, and then falls back on the default controller found in the 'default', global, namespace.

If you wish to always default to the global namespace, set the `$useDefaultControllerAlways` parameter in the front controller:

```
$front->setParam('useDefaultControllerAlways', true);
```

12. MVC Exceptions

12.1. Introduction

The MVC components in Zend Framework utilize a Front Controller, which means that all requests to a given site will go through a single entry point. As a result, all exceptions bubble up to the Front Controller eventually, allowing the developer to handle them in a single location.

However, exception messages and backtrace information often contain sensitive system information, such as SQL statements, file locations, and more. To help protect your site, by default `Zend_Controller_Front` catches all exceptions and registers them with the response object; in turn, by default, the response object does not display exception messages.

12.2. Handling Exceptions

Several mechanisms are built in to the MVC components already to allow you to handle exceptions.

- By default, the error handler plugin is registered and active. This plugin was designed to handle:
  - Errors due to missing controllers or actions
  - Errors occurring within action controllers

  It operates as a `postDispatch()` plugin, and checks to see if a dispatcher, action controller, or other exception has occurred. If so, it forwards to an error handler controller.

  This handler will cover most exceptional situations, and handle missing controllers and actions gracefully.

- `Zend_Controller_Front::throwExceptions()`

  By passing a boolean `TRUE` value to this method, you can tell the front controller that instead of aggregating exceptions in the response object or using the error handler plugin, you'd rather handle them yourself. As an example:

```
$front->throwExceptions(true);
try {
    $front->dispatch();
} catch (Exception $e) {
    // handle exceptions yourself
}
```

  This method is probably the easiest way to add custom exception handling covering the full range of possible exceptions to your front controller application.

- `Zend_Controller_Response_Abstract::renderExceptions()`

  By passing a boolean `TRUE` value to this method, you tell the response object that it should render an exception message and backtrace when rendering itself. In this scenario, any exception raised by your application will be displayed. This is only recommended for non-production environments.

- `Zend_Controller_Front::returnResponse()` and `Zend_Controller_Response_Abstract::isException()`.

  By passing a boolean `TRUE` to `Zend_Controller_Front::returnResponse()`, `Zend_Controller_Front::dispatch()` will not render the response, but instead return it. Once you have the response, you may then test to see if any exceptions were trapped using its `isException()` method, and retrieving the exceptions via the `getException()` method. As an example:

```
$front->returnResponse(true);
$response = $front->dispatch();
if ($response->isException()) {
```

```
    $exceptions = $response->getException();
    // handle exceptions ...
} else {
    $response->sendHeaders();
    $response->outputBody();
}
```

The primary advantage this method offers over `Zend_Controller_Front::throwExceptions()` is to allow you to conditionally render the response after handling the exception. This will catch any exception in the controller chain, unlike the error handler plugin.

12.3. MVC Exceptions You May Encounter

The various MVC components -- request, router, dispatcher, action controller, and response objects -- may each throw exceptions on occasion. Some exceptions may be conditionally overridden, and others are used to indicate the developer may need to consider their application structure.

As some examples:

- `Zend_Controller_Dispatcher::dispatch()` will, by default, throw an exception if an invalid controller is requested. There are two recommended ways to deal with this.
  - Set the useDefaultControllerAlways parameter.

    In your front controller, or your dispatcher, add the following directive:

    ```
    $front->setParam('useDefaultControllerAlways', true);

    // or

    $dispatcher->setParam('useDefaultControllerAlways', true);
    ```

    When this flag is set, the dispatcher will use the default controller and action instead of throwing an exception. The disadvantage to this method is that any typos a user makes when accessing your site will still resolve and display your home page, which can wreak havoc with search engine optimization.
  - The exception thrown by `dispatch()` is a `Zend_Controller_Dispatcher_Exception` containing the text 'Invalid controller specified'. Use one of the methods outlined in the previous section to catch the exception, and then redirect to a generic error page or the home page.
- `Zend_Controller_Action::__call()` will throw a `Zend_Controller_Action_Exception` if it cannot dispatch a non-existent action to a method. Most likely, you will want to use some default action in the controller in cases like this. Ways to achieve this include:
  - Subclass `Zend_Controller_Action` and override the `__call()` method. As an example:

    ```
    class My_Controller_Action extends Zend_Controller_Action
    {
        public function __call($method, $args)
        {
            if ('Action' == substr($method, -6)) {
    ```

```
            $controller = $this->getRequest()->getControllerName();
            $url = '/' . $controller . '/index';
            return $this->_redirect($url);
        }

        throw new Exception('Invalid method');
    }
}
```

The example above intercepts any undefined action method called and redirects it to the default action in the controller.

- Subclass `Zend_Controller_Dispatcher` and override the `getAction()` method to verify the action exists. As an example:

```
class My_Controller_Dispatcher extends Zend_Controller_Dispatcher
{
    public function getAction($request)
    {
        $action = $request->getActionName();
        if (empty($action)) {
            $action = $this->getDefaultAction();
            $request->setActionName($action);
            $action = $this->formatActionName($action);
        } else {
            $controller = $this->getController();
            $action     = $this->formatActionName($action);
            if (!method_exists($controller, $action)) {
                $action = $this->getDefaultAction();
                $request->setActionName($action);
                $action = $this->formatActionName($action);
            }
        }

        return $action;
    }
}
```

The above code checks to see that the requested action exists in the controller class; if not, it resets the action to the default action.

This method is nice because you can transparently alter the action prior to final dispatch. However, it also means that typos in the URL may still dispatch correctly, which is not great for search engine optimization.

- Use `Zend_Controller_Action::preDispatch()` or `Zend_Controller_Plugin_Abstract::preDispatch()` to identify invalid actions.

By subclassing `Zend_Controller_Action` and modifying `preDispatch()`, you can modify all of your controllers to forward to another action or redirect prior to actually dispatching the action. The code for this will look similar to the code for overriding `__call()`, above.

Alternatively, you can check this information in a global plugin. This has the advantage of being action controller independent; if your application consists of a variety of action controllers, and not all of them inherit from the same class, this method can add consistency in handling your various classes.

As an example:

```
class My_Controller_PreDispatchPlugin extends Zend_Controller_Plugin_Abstract
{
    public function preDispatch(Zend_Controller_Request_Abstract $request)
    {
        $front      = Zend_Controller_Front::getInstance();
        $dispatcher = $front->getDispatcher();
        $class      = $dispatcher->getControllerClass($request);
        if (!$controller) {
            $class = $dispatcher->getDefaultControllerClass($request);
        }

        $r      = new ReflectionClass($class);
        $action = $dispatcher->getActionMethod($request);

        if (!$r->hasMethod($action)) {
            $defaultAction  = $dispatcher->getDefaultAction();
            $controllerName = $request->getControllerName();
            $response       = $front->getResponse();
            $response->setRedirect('/' . $controllerName
                                   . '/' . $defaultAction);
            $response->sendHeaders();
            exit;
        }
    }
}
```

In this example, we check to see if the action requested is available in the controller. If not, we redirect to the default action in the controller, and exit script execution immediately.

Zend_Currency

1. Introduction to Zend_Currency

Zend_Currency is part of the strong support for I18n in Zend Framework. It handles all issues related to currency, money representation, formatting, exchange services and calculation.

1.1. Why should you use Zend_Currency?

Zend_Currency offers you the following benefit:

- *Complete Locale support*

  This component works with all available locales and therefore knows about more than 100 different localized currencies. This includes informations like currency names, abbreviations, money signs and much more.

- *Reusable Currency Definitions*

  Zend_Currency has the advantage that already defined currency representations can be reused. You could also have 2 different representations for the same currency.

- *Currency calculation*

  Zend_Currency allows you also to calculate with currency values. Therefor it provides you a interface to exchange services.

- *Additional Methods*

  Zend_Currency includes several additional methods that offer informations about details to currencies like: which currency is used within a defined region, or what are the known abbreviations of a currency.

2. Using Zend_Currency

2.1. Generic usage

The simplest usecase within an application is to use the clients locale. When you create a instance of Zend_Currency without giving any options, your clients locale will be used to set the proper currency.

Example 106. Creating a currency with client settings

Let's assume that your client has set "en_US" as wished language within his browser. In this case Zend_Currency will automatically detect the currency which has to be used.

```
$currency = new Zend_Currency();

// See the default settings which are depending on the client
// var_dump($currency);
```

The created object would now contain the currency "US Dollar" as this is the actual assigned currency for US (United States). It has also other options set, like "$" for the currency sign or "USD" for the abbreviation.

Automatic locale detection does not always work

You should note that this automatic locale detection does not always work properly. The reason for this behaviour is that `Zend_Currency` must have a locale which includes a region. When the client would only set "en" as locale `Zend_Currency` would not know which of the more than 30 countries was meant. In this case an exception would be raised.

A client could also omit the locale settings within his browser. This would lead to the problem that your environment settings will be used as fallback and could also lead to an exception.

2.2. Currency creation based on a locale

To prevent the problems with your client you could simply set the wished locale manually.

```
$currency = new Zend_Currency('en_US');

// You can also use the 'locale' option
// $currency = new Zend_Currency(array('locale' => 'en_US'));

// See the actual settings which are fixed to 'en_US'
// var_dump($currency);
```

As within our first example the used currency will be "US Dollar". But now we are no longer dependend on the clients settings.

`Zend_Currency` also supports the usage of an application-wide locale. You can set a `Zend_Locale` instance in the registry as shown below. With this notation you can avoid setting the locale manually for each instance when you want to use the same locale throughout the application.

```
// in your bootstrap file
$locale = new Zend_Locale('de_AT');
Zend_Registry::set('Zend_Locale', $locale);

// somewhere in your application
$currency = new Zend_Currency();
```

3. Options for currencies

Depending on your needs, several options can be specified at instantiation. All of this options have default values. But sometimes it is necessary to define how your currencies will be rendered. This includes for example:

- *Currency symbol, shortname or name*:

  `Zend_Currency` knows all currency names, abbreviations and signs. But sometimes you could be in need to define the string which has to be displayed as replacement for a currency.

- *Currency position*:

  The position of the currency symbol is automatically defined. But sometimes you could be in need to define it manually.

- *Script*:

You could define the script which shall be used to display digits. Detailed information about scripts and their usage can be found in `Zend_Locale`'s chapter Numeral System Conversion.

- *Number formatting*:

  The amount of currency (generally known as money value) is formatted by usaging the formatting rules defined by the locale. For example is the ',' sign in english used as separator for thousands, but in german as precision sign.

The following list mentions all options which could be set. They can eighter be set at initiation or by using the `setFormat()` method. In any case you have to give this options as array.

- *currency*: Defines the abbreviation which can be displayed.
- *display*: Defines which part of the currency should be used for displaying the currency representation. There are 4 representations which can be used and which are all described in this table.
- *format*: Defines the format which should be used for displaying numbers. This number-format includes for example the thousand separator. You can either use a default format by giving a locale identifier, or define the number-format manually. If no format is set the locale from the `Zend_Currency` object will be used. See the chapter about number formatting for details.
- *locale*: Defines a locale for this currency. It will be used for detecting the default values when other settings are omitted. Note that when you don't set a locale yourself, it will be detected automatically which could lead to problems.
- *name*: Defines the full currency name which can be displayed.
- *position*: Defines the position at which the currency description should be displayed. The supported positions are described this section.
- *precision*: Defines the precision which should be used for the currency representation. The default value depends on the locale and is for most locales *2*.
- *script*: Defined which script should be used for displaying digits. The default script for most locales is *'Latn'*, which includes the digits 0 to 9. Other scripts such as 'Arab' (Arabian) are using other digits. See the chapter about numbering systems for details and available options.
- *service*: Defines the exchange service which has to be used when calculating with different currencies.
- *symbol*: Defines the currency symbol which can be displayed.
- *value*: Defines the currency amount (money value). Using this option you should also set the service option.

As you can see there is much which could be changed. Still as already mentioned the default values for this settings conform the official standard for currency represenation in every country.

4. What makes a currency?

The currency consists of several informations. A name, a abbreviation and a sign. Each of these could be relevant to be displayed, but only one at the same time. It would not be a good practice to display something like "USD 1.000 $".

Therefor Zend_Currency supports the definition of the currency information which has to be rendered. The following constants can be used:

Table 31. Rendered informations for a currency

Constant	Description
NO_SYMBOL	No currency representation will be rendered at all
USE_SYMBOL	The currency symbol will be rendered. For US Dollar this would be '$'
USE_SHORTNAME	The abbreviation for this currency will be rendered. For US Dollar this would be 'USD'. Most abbreviations consist of 3 characters
USE_NAME	The full name for this currency will be rendered. For US Dollar the full name would be "US Dollar"

Example 107. Selecting the currency description

Let's assume that your client has again set "en_US" as locale. Using no option the returned value could look like this:

```
$currency = new Zend_Currency(
    array(
        'value' => 100,
    )
);

print $currency; // Could return '$ 100'
```

By giving the proper option you can define what information which has to be rendered.

```
$currency = new Zend_Currency(
    array(
        'value'   => 100,
        'display' => Zend_Currency::USE_SHORTNAME,
    )
);

print $currency; // Could return 'USD 100'
```

Without providing the display the currency sign will be used when rendering the object. When the currency has no sign, the abbreviation will be used as replacement.

Not all currencies have signs

You should note that not all currencies have default currency signs. This means, that when there is no default sign, and you set the sign to be rendered, you will not have a rendered currency at all because the sign is an empty string.

Sometimes it is necessary to change the default informations. You can set each of the three currency informations independently by giving the proper option. See the following example.

Example 108. Changing the currency description

Let's assume that your client has again set "en_US" as locale. But now we don't want to use the default settings but set our own description. This can simply be done providing the proper option:

```
$currency = new Zend_Currency(
    array(
        'value' => 100,
        'name'  => 'Dollar',
    )
);

print $currency; // Could return 'Dollar 100'
```

You could also set a sign or an abbreviations yourself.

```
$currency = new Zend_Currency(
    array(
        'value'    => 100,
        'symbol' => '$$$',
    )
);

print $currency; // Could return '$$$ 100'
```

Automatic display settings

When you set a name, abbreviation or sign yourself, than this new information will automatically be set to be rendered. This simplification prevents you from setting the proper display option when you set a information.

So using the sign option you can omit display and don't neet to setting it to 'USE_SYMBOL'.

5. Where is the currency?

The position where the currency sign or name will be displayed depends on the locale. Still, when you want to define this setting yourself you have to use the display option and provide one of the following constants:

Table 32. Available positions for the currency

Constant	Description
STANDARD	Sets the standard position as defined within the locale
RIGHT	Displays the currency representation at the right side of the value
LEFT	Displays the currency representation at the left side of the value

Example 109. Setting the currency position

Let's assume that your client has again set "en_US" as locale. Using no option the returned value could look like this:

```
$currency = new Zend_Currency(
    array(
        'value' => 100,
    )
);

print $currency; // Could return '$ 100'
```

So by using the default setting the currency (in our case $) could eighter be rendered left or right from the value. Now let's define a fixed position:

```
$currency = new Zend_Currency(
    array(
        'value'    => 100,
        'position' => Zend_Currency::RIGHT,
    )
);

print $currency; // Could return '100 $';
```

Note that in the second snippet the position of USD is fixed regardless of the used locale or currency.

6. How does the currency look like?

How the value of a currency will be rendered depends mainly on the used locale. There are several informations which are set by the locale. Each of them can manually be overridden by using the proper option.

For example, most locales are using the latin script for rendering numbers. But there are languages like "Arabic" which are using other digits. And when you have an arabic website you may also want to render other currencies by using the arabic script. See the following example:

Example 110. Using a custom script

Let's expect that we are again using our "Dollar" currency. But now we want to render our currency by using the arabic script.

```
$currency = new Zend_Currency(
    array(
        'value'  => 1000,
        'script' => 'Arab',
    )
);

print $currency; // Could return '$ ########'
```

For more informations about available scripts look into `Zend_Locale`'s chapter about numbering systems.

But also the formatting of a currency number (money value) can be changed. Per default it depends on the used locale. It includes the separator which will be used between thousands, which sign will be used as decimal point, and also the used precision.

```
$currency = new Zend_Currency(
    array(
        'value'    => 1000,
        'currency' => 'USD'
        'format'   => 'de',
    )
);

print $currency; // Could return '$ 1.000'
```

There are two ways to define the format which will be used. You can eighter give a locale or define a format manually.

When you are using a locale for defining the format all is done automatically. The locale 'de', for example, defines '.' as separator for thousands and ',' as decimal point. Within english this is reversed.

```
$currency_1 = new Zend_Currency(
    array(
        'value'    => 1000,
        'currency' => 'USD'
        'format'   => 'de',
    )
);

$currency_2 = new Zend_Currency(
    array(
        'value'    => 1000,
        'currency' => 'USD'
        'format'   => 'en',
    )
);

print $currency_1; // Could return '$ 1.000'
print $currency_2; // Could return '$ 1,000'
```

When you define it manually then you must conform the format as described in this chapter about localizing. See the following:

```
$currency = new Zend_Currency(
    array(
        'value'    => 1000,
        'currency' => 'USD'
        'format'   => '#0',
    )
);

print $currency; // Could return '$ 1000'
```

In the above snippet we deleted the separator and also the precision.

7. How much is my currency?

When you are working with currencies then you normally want to display an amount of money. And when you work with different currencies then you have to do this with three different things. The amount you want to display, the precision you want to use, and probably the exchange rate.

7.1. Working with currency values

The currency value, a.k.a. the money, you want to use can easily be set by using the value option.

```
$currency = new Zend_Currency(
    array(
        'value'    => 1000,
        'currency' => 'USD',
    )
);

print $currency; // Could return '$ 1.000'
```

Using the `setFormat()` method with this array option, and also by using the `setValue()` method you can set the value afterwards.

```
$currency = new Zend_Currency(
    array(
        'value'    => 1000,
        'currency' => 'USD',
    )
);

print $currency->setValue(2000); // Could return '$ 2.000'
```

With the `getValue()` method you will get the actual set value.

7.2. Using precision on currencies

When working with currencies they you probably also have to handle precision. Most currencies use a precision of 2. This means that when you have 100 US dollars you could also have 50 cents. The related value is simply a floating value.

```
$currency = new Zend_Currency(
    array(
        'value'    => 1000.50,
        'currency' => 'USD',
    )
);

print $currency; // Could return '$ 1.000,50'
```

Of course, as the default precision is 2, you will get '00' for the decimal value when there is no precision to display.

```
$currency = new Zend_Currency(
    array(
        'value'    => 1000,
        'currency' => 'USD',
    )
);

print $currency; // Could return '$ 1.000,00'
```

To get rid of this default precision you could simply use the precision option and set it to '0'. And you can set any other precision you want to use between 0 and 9. All values will be rounded or streched when they don't fit the set precision.

```
$currency = new Zend_Currency(
    array(
        'value'     => 1000,30,
        'currency'  => 'USD',
        'precision' => 0
    )
);

print $currency; // Could return '$ 1.000'
```

8. Calculating with currencies

When working with currencies you will sometimes also have to calculate with them. `Zend_Currency` allows you to do this with some simple methods. The following methods are supported for calculation:

- *`add()`*: This method adds the given currency to the existing currency object.
- *`sub()`*: This method substracts the given currency from the existing currency object.
- *`div()`*: This method divides the given currency from the existing currency object.
- *`mul()`*: This method multiplies the given currency with the existing currency object.
- *`mod()`*: This method calculates the remaining value (modulo) from dividing the given currency from the existing currency object.
- *`compare()`*: This method compares the given currency with the existing currency object. When both values are equal it returns '0'. When the existing currency value is greater than the given, this method will return 1. Otherwise you will get '-1' returned.
- *`equals()`*: This method compares the given currency with the existing currency object. When both values are equal it returns TRUE, otherwise FALSE.
- *`isMore()`*: This method compares the given currency with the existing currency object. When the existing currency is greater than the given one, you will get TRUE in return, otherwise FALSE.
- *`isLess()`*: This method compares the given currency with the existing currency object. When the existing currency is less than the given one, you will get TRUE in return, otherwise FALSE.

As you can see the multiple methods allow any kind of calculation with `Zend_Currency`. See the next snippets as example:

```
$currency = new Zend_Currency(
    array(
        'value'    => 1000,
        'currency' => 'USD',
    )
);

print $currency; // Could return '$ 1.000,00'

$currency->add(500);
print $currency; // Could return '$ 1.500,00'
```

```
$currency_2 = new Zend_Currency(
```

```
    array(
        'value'    => 500,
        'currency' => 'USD',
    )
);

if ($currency->isMore($currency_2)) {
    print "First is more";
}

$currency->div(5);
print $currency; // Could return '$ 200,00'
```

9. Exchanging currencies

Within the previous section we discussed currency calculations. But as you can imaging calculating currencies does often mean to calculate different currencies from different countries.

In this case the currencies have to be exchanged so that both use the same currency. Within real live this information is available by banks or by daily papers. But as we are in web, we should use available exchange services. `Zend_Currency` allows their usage with a simple callback.

First let's write a simple exchange service.

```
class SimpleExchange implements Zend_Currency_CurrencyInterface
{
    public function getRate($from, $to)
    {
        if ($from !== "USD") {
            throw new Exception('We can only exchange USD');
        }

        switch ($to) {
            case 'EUR':
                return 0.5;
            case 'JPE':
                return 0.7;
       }

       throw new Exception('Unable to exchange $to');
    }
}
```

We have now created a manual exchange service. It will not fit the real live, but it shows you how currency exchange works.

Your exchange class must implement the `Zend_Currency_CurrencyInterface` interface. This interface requires the single method `getRate()` to be implemented. This method has two parameters it will receive. Both are the short names for the given currencies. `Zend_Currency` on the other side needs the exchange rate to be returned.

In a living exchange class you would probably ask the service provider for the correct exchange rates. For our example the manual rate will be ok.

Now we will simply attach our exchange class with `Zend_Currency`. There are two ways to do this. Eigher by attaching a instance of the Exchange class, or by simply giving a string with the classname.

```
$currency = new Zend_Currency(
    array(
        'value'    => 1000,
        'currency' => 'USD',
    )
);

$service  = new SimpleExchange();

// attach the exchange service
$currency->setService($service);

$currency_2 = new Zend_Currency(
    array(
        'value'    => 1000,
        'currency' => 'EUR',
    )
);

print $currency->add($currency2);
```

The above example will return '$ 3.000' because the 1.000 EUR will be converted by a rate of 2 to 2.000 USD.

Calculation without exchange service

When you try to calculate two currency objects which do not use the same currency and have no exchange service attached, you will get an exception. The reason is that `Zend_Currency` is then not able to switch between the different currencies.

10. Additional informations on Zend_Currency

10.1. Currency informations

Sometimes it is necessary to get informations which are related to a currency. `Zend_Currency` provides you with several methods to get this informations. Available methods include the following:

- *`getCurrencyList()`*: Returns a list of all currencies which are used in the given region as array. Defaults to the objects locale when no region has been given.
- *`getLocale()`*: Returns the set locale for the actual currency.
- *`getName()`*: Returns the full name for the actual currency. When there is no full name available for the actual currency, it will return the abbreviation for it.
- *`getRegionList()`*: Returns a list of all regions where this currency is used as array. Defaults to the objects currency when no currency has been given.
- *`getService()`*: Returns the set exchange service object for the actual currency.
- *`getShortName()`*: Returns the abbreviation for the actual currency.
- *`getSymbol()`*: Returns the currency sign for the currency. When the currency has no symbol, then it will return the abbreviation for it.

- *getValue()*: Returns the set value for the actual currency.

Let's see some code snippets as example:

```
$currency = new Zend_Currency();

var_dump($currency->getValue());
// returns 0

var_dump($currency->getRegionList());
// could return an array with all regions where USD is used

var_dump($currency->getRegionList('EUR'));
// returns an array with all regions where EUR is used

var_dump($currency->getName());
// could return 'US Dollar'

var_dump($currency->getName('EUR'));
// returns 'Euro'
```

As you can see, several methods allow to use additional parameters to override the actual object to get informations for other currencies. Omitting this parameters will return informations from the actual set currency.

10.2. Currency Performance Optimization

`Zend_Currency`'s performance can be optimized using `Zend_Cache`. The static method `Zend_Currency::setCache($cache)` accepts one option: a `Zend_Cache` adapter. If the cache adapter is set, the localization data which is used by `Zend_Currency` will be cached. Additionally there are some static methods for manipulating the cache: `getCache()`, `hasCache()`, `clearCache()` and `removeCache()`.

Example 111. Caching currencies

```
// creating a cache object
$cache = Zend_Cache::factory('Core',
                             'File',
                             array('lifetime' => 120,
                                   'automatic_serialization' => true),
                             array('cache_dir'
                                       => dirname(__FILE__) . '/_files/'));
Zend_Currency::setCache($cache);
```

Zend_Date

1. Introduction

The Zend_Date component offers a detailed, but simple API for manipulating dates and times. Its methods accept a wide variety of types of information, including date parts, in numerous combinations yielding many features and possibilities above and beyond the existing PHP date related functions. For the very latest manual updates, please see our online manual (frequently synced to Subversion).

Although simplicity remains the goal, working with localized dates and times while modifying, combining, and comparing parts involves some unavoidable complexity. Dates, as well as times, are often written differently in different locales. For example, some place the month first, while other write the year first when expressing calendar dates. For more information about handling localization and normalization, please refer to Zend_Locale.

Zend_Date also supports abbreviated names of months in many languages. Zend_Locale facilitates the normalization of localized month and weekday names to timestamps, which may, in turn, be shown localized to other regions.

1.1. Always Set a Default Timezone

Before using any date related functions in PHP or Zend Framework, first make certain your application has a correct default timezone, by either setting the TZ environment variable, using the date.timezone php.ini setting, or using date_default_timezone_set(). In PHP, we can adjust all date and time related functions to work for a particular user by setting a default timezone according to the user's expectations. For a complete list of timezone settings, see the CLDR Timezone Identifier List.

Example 112. Setting a Default Timezone

```
// timezone for an American in California
date_default_timezone_set('America/Los_Angeles');
// timezone for a German in Germany
date_default_timezone_set('Europe/Berlin');
```

When creating Zend_Date instances, their timezone will automatically become the current default timezone! Thus, the timezone setting will account for any Daylight Savings Time (DST) in effect, eliminating the need to explicitly specify DST.

Keep in mind that the timezones *UTC* and *GMT* do not include Daylight Saving Time. This means that even if you define per hand that Zend_Date should work with DST, it would automatically be switched back for the instances of Zend_Date which have been set to UTC or GMT.

1.2. Why Use Zend_Date?

Zend_Date offers the following features, which extend the scope of PHP date functions:

- Simple API

  Zend_Date offers a very simple API, which combines the best of date and time functionality from four programming languages. It is possible, for example, to add or compare two times within a single row.

- Completely internationalized

  All full and abbreviated names of months and weekdays are supported for more than 130 languages. Methods support both input and the output of dates using the localized names of months and weekdays, in the conventional format associated with each locale.

- Unlimited timestamps

  Although PHP 5.2 docs state, "The valid range of a timestamp is typically from Fri, 13 Dec 1901 20:45:54 GMT to Tue, 19 Jan 2038 03:14:07 GMT," `Zend_Date` supports a nearly unlimited range, with the help of the BCMath extension. If BCMath is not available, then `Zend_Date` will have reduced support only for timestamps within the range of the float type supported by your server. "The size of a float is platform-dependent, although a maximum of **~1.8e308** with a precision of roughly 14 decimal digits is a common value (that's 64 bit IEEE format)." [http://www.php.net/float]. Additionally, inherent limitations of float data types, and rounding error of float numbers may introduce errors into calculations. To avoid these problems, Zend Framework's I18n components use BCMath extension, if available.

- Support for ISO-8601 date specifications

  ISO-8601 date specifications are supported. Even partially compliant ISO-8601 date specifications will be identified. These date formats are particularly useful when working with databases. for example, even though MsSQL and MySQL differ a little from each other, both are supported by `Zend_Date` using the Zend_Date::ISO_8601 format specification constant. When date strings conform to "**Y/m/d**" or "**Y-m-d H:i:s**", according to PHP `date()` format tokens, use `Zend_Date`'s built-in support for ISO-8601 formatted dates.

- Calculate sunrise and sunset

  For any place and day, the times for sunrise and sunset can be displayed, so that you won't miss a single daylight second for working on your favorite PHP project :)

2. Theory of Operation

Why is there only one class `Zend_Date` for handling dates and times in Zend Framework?

Many languages split the handling of times and calendar dates into two classes. However, Zend Framework strives for extreme simplicity, and forcing the developer to manage different objects with different methods for times and dates becomes a burden in many situations. Since `Zend_Date` methods support working with ambiguous dates that might not include all parts (era, year, month, day, hour, minute, second, timezone), developers enjoy the flexibility and ease of using the same class and the same methods to perform the same manipulations (e.g. addition, subtraction, comparison, merging of date parts, etc.). Splitting the handling of these date fragments into multiple classes would create complications when smooth interoperation is desired with a small learning curve. A single class reduces code duplication for similar operations, without the need for a complex inheritance hierarchy.

2.1. Internals

- UNIX Timestamp

  All dates and times, even ambiguous ones (e.g. no year), are represented internally as absolute moments in time, represented as a UNIX timestamp expressing the difference between the desired time and January 1st, 1970 00:00:00 GMT. This was only possible, because `Zend_Date` is not limited to UNIX timestamps nor integer values. The BCMath extension is required to support extremely large dates outside of the range Fri, 13 Dec 1901

20:45:54 GMT to Tue, 19 Jan 2038 03:14:07 GMT. Additional, tiny math errors may arise due to the inherent limitations of float data types and rounding, unless using the BCMath extension.

- Date parts as timestamp offsets

  Thus, an instance object representing three hours would be expressed as three hours after January 1st, 1970 00:00:00 GMT -i.e. 0 + 3 * 60 * 60 = 10800.

- PHP functions

  Where possible, `Zend_Date` usually uses PHP functions to improve performance.

3. Basic Methods

The following sections show basic usage of `Zend_Date` primarily by example. For this manual, "dates" always imply a calendar date with a time, even when not explicitly mentioned, and vice-versa. The part not specified defaults to an internal representation of "zero". Thus, adding a date having no calendar date and a time value of 12 hours to another date consisting only of a calendar date would result in a date having that calendar date and a time of "noon".

Setting only a specific date, with no time part, implies a time set to 00:00:00. Conversely, setting only a specific time implies a date internally set to 01.01.1970 plus the number of seconds equal to the elapsed hours, minutes, and seconds identified by the time. Normally, people measure things from a starting point, such as the year 0 A.D. However, many software systems use the first second of the year 1970 as the starting point, and denote times as a timestamp offset counting the number of seconds elapsed from this starting point.

3.1. Current Date

Without any arguments, constructing an instance returns an object in the default locale with the current, local date using PHP's `time()` function to obtain the UNIX timestamp for the object. Make sure your PHP environment has the correct default timezone .

Example 113. Creating the Current Date

```
$date = new Zend_Date();

// Output of the current timestamp
print $date;
```

3.2. Zend_Date by Example

Reviewing basic methods of `Zend_Date` is a good place to start for those unfamiliar with date objects in other languages or frameworks. A small example will be provided for each method below.

3.2.1. Output a Date

The date in a `Zend_Date` object may be obtained as a localized integer or string using the `get()` method. There are many available options, which will be explained in later sections.

Example 114. get() - Output a Date

```
$date = new Zend_Date();

// Output of the desired date
print $date->get();
```

3.2.2. Setting a Date

The `set()` method alters the date stored in the object, and returns the final date value as a timestamp (not an object). Again, there are many options which will be explored in later sections.

Example 115. set() - Set a Date

```
$date = new Zend_Date();

// Setting of a new time
$date->set('13:00:00',Zend_Date::TIMES);
print $date->get(Zend_Date::W3C);
```

3.2.3. Adding and Subtracting Dates

Adding two dates with `add()` usually involves adding a real date in time with an artificial timestramp representing a date part, such as 12 hours, as shown in the example below. Both `add()` and `sub()` use the same set of options as `set()`, which will be explained later.

Example 116. add() - Adding Dates

```
$date = new Zend_Date();

// changes $date by adding 12 hours
$date->add('12:00:00', Zend_Date::TIMES);

echo "Date via get() = ", $date->get(Zend_Date::W3C), "\n";

// use magic __toString() method to call Zend_Date's toString()
echo "Date via toString() = ", $date, "\n";
```

3.2.4. Comparison of Dates

All basic `Zend_Date` methods can operate on entire dates contained in the objects, or can operate on date parts, such as comparing the minutes value in a date to an absolute value. For example, the current minutes in the current time may be compared with a specific number of minutes using `compare()`, as in the example below.

Example 117. compare() - Compare Dates

```
$date = new Zend_Date();

// Comparation of both times
if ($date->compare(10, Zend_Date::MINUTE) == -1) {
    print "This hour is less than 10 minutes old";
} else {
    print "This hour is at least 10 minutes old";
}
```

For simple equality comparisons, use `equals()`, which returns a boolean.

Example 118. equals() - Identify a Date or Date Part

```
$date = new Zend_Date();

// Comparation of the two dates
if ($date->equals(10, Zend_Date::HOUR)) {
    print "It's 10 o'clock. Time to get to work.";
} else {
    print "It is not 10 o'clock. You can keep sleeping.";
}
```

4. Zend_Date API Overview

While the Zend_Date API remains simplistic and unitary, its design remains flexible and powerful through the rich permutations of operations and operands.

4.1. Zend_Date Options

4.1.1. Selecting the Date Format Type

Several methods use date format strings, in a way similar to PHP's date(). If you are more comfortable with PHP's date format specifier than with ISO format specifiers, then you can use Zend_Date::setOptions(array('format_type' => 'php')). Afterward, use PHP's date format specifiers for all functions which accept a $format parameter. Use Zend_Date::setOptions(array('format_type' => 'iso')) to switch back to the default mode of supporting only ISO date format tokens. For a list of supported format codes, see Self-Defined OUTPUT Formats Using PHP's date() Format Specifiers

4.1.2. DST and Date Math

When dates are manipulated, sometimes they cross over a DST change, normally resulting in the date losing or gaining an hour. For exmaple, when adding months to a date before a DST change, if the resulting date is after the DST change, then the resulting date will appear to lose or gain an hour, resulting in the time value of the date changing. For boundary dates, such as midnight of the first or last day of a month, adding enough months to cross a date boundary results in the date losing an hour and becoming the last hour of the preceding month, giving the appearance of an "off by 1" error. To avoid this situation, the DST change ignored by using the fix_dst option. When crossing the Summer or Winter DST boundary, normally an hour is substracted or added depending on the date. For example, date math crossing the Spring DST leads to a date having a day value one less than expected, if the time part of the date was originally 00:00:00. Since Zend_Date is based on timestamps, and not calendar dates with a time component, the timestamp loses an hour, resulting in the date having a calendar day value one less than expected. To prevent such problems use the option fix_dst, which defaults to TRUE, causing DST to have no effect on date "math" (addMonth(), subMonth()). Use Zend_Date::setOptions(array('fix_dst' => false)) to enable the subtraction or addition of the DST adjustment when performing date "math".

If your actual timezone within the instance of Zend_Date is set to UTC or GMT the option 'fix_dst' will not be used because these two timezones do not work with DST. When you change the timezone for this instance again to a timezone which is not UTC or GMT the previous set 'fix_dst' option will be used again for date "math".

4.1.3. Month Calculations

When adding or substracting months from an existing date, the resulting value for the day of the month might be unexpected, if the original date fell on a day close to the end of the month.

For example, when adding one month to January 31st, people familiar with SQL will expect February 28th as the result. On the other side, people familiar with Excel and OpenOffice will expect March 3rd as the result. The problem only occurs, if the resulting month does not have the day, which is set in the original date. For Zend Framework developers, the desired behavior is selectable using the extend_month option to choose either the SQL behaviour, if set to FALSE, or the spreadsheet behaviour when set to TRUE. The default behaviour for extend_month is FALSE, providing behavior compatible to SQL. By default, `Zend_Date` computes month calculations by truncating dates to the end of the month (if necessary), without wrapping into the next month when the original date designates a day of the month exceeding the number of days in the resulting month. Use `Zend_Date::setOptions(array('extend_month' => true))` to make month calculations work like popular spreadsheet programs.

4.1.4. Speed up Date Localization and Normalization with Zend_Cache

You can speed up `Zend_Date` by using an `Zend_Cache` adapter. This speeds up all methods of `Zend_Date` when you are using localized data. For example all methods which accept `Zend_Date::DATE` and `Zend_Date::TIME` constants would benefit from this. To set an `Zend_Cache` adapter to `Zend_Date` just use `Zend_Date::setOptions(array('cache' => $adapter))`.

4.1.5. Receiving Syncronised Timestamps with Zend_TimeSync

Normally the clocks from servers and computers differ from each other. `Zend_Date` is able to handle such problems with the help of `Zend_TimeSync`. You can set a timeserver with `Zend_Date::setOptions(array('timesync' => $timeserver))` which will set the offset between the own actual timestamp and the real actual timestamp for all instances of `Zend_Date`. Using this option does not change the timestamp of existing instances. So best usage is to set it within the bootstrap file.

4.2. Working with Date Values

Once input has been normalized via the creation of a `Zend_Date` object, it will have an associated timezone, but an internal representation using standard UNIX timestamps. In order for a date to be rendered in a localized manner, a timezone must be known first. The default timezone is always GMT or UTC. To examine an object's timezone use `getTimeZone()`. To change an object's timezone, use `setTimeZone()`. All manipulations of these objects are assumed to be relative to this timezone.

Beware of mixing and matching operations with date parts between date objects for different timezones, which generally produce undesireable results, unless the manipulations are only related to the timestamp. Operating on `Zend_Date` objects having different timezones generally works, except as just noted, since dates are normalized to UNIX timestamps on instantiation of `Zend_Date`.

Most methods expect a constant selecting the desired `$part` of a date, such as `Zend_Date::HOUR`. These constants are valid for all of the functions below. A list of all available constants is provided in list of all constants. If no `$part` is specified, then `Zend_Date::TIMESTAMP` is assumed. Alternatively, a user-specified format may be used for `$part`, using the same underlying mechanism and format codes as `Zend_Locale_Format::getDate()`. If a date object is constructed using an obviously invalid date (e.g. a month number greater than 12), then `Zend_Date` will throw an exception, unless no specific date format has been selected -i.e. `$part` is either `NULL` or `Zend_Date::DATES` (a "loose" format).

Example 119. User-Specified Input Date Format

```
$date1 = new Zend_Date('Feb 31, 2007', null, 'en_US');
echo $date1, "\n"; // outputs "Mar 3, 2007 12:00:00 AM"

$date2 = new Zend_Date('Feb 31, 2007', Zend_Date::DATES, 'en_US');
echo $date2, "\n"; // outputs "Mar 3, 2007 12:00:00 AM"

// strictly restricts interpretation to specified format
$date3 = new Zend_Date('Feb 31, 2007', 'MM.dd.yyyy');
echo $date3, "\n"; // outputs "Mar 3, 2007 12:00:00 AM"
```

If the optional `$locale` parameter is provided, then the `$locale` disambiguates the `$date` operand by replacing month and weekday names for string `$date` operands, and even parsing date strings expressed according to the conventions of that locale (see Zend_Locale_Format::getDate()). The automatic normalization of localized `$date` operands of a string type occurs when `$part` is one of the `Zend_Date::DATE*` or `Zend_Date::TIME*` constants. The locale identifies which language should be used to parse month names and weekday names, if the `$date` is a string containing a date. If there is no `$date` input parameter, then the `$locale` parameter specifies the locale to use for localizing output (e.g. the date format for a string representation). Note that the `$date` input parameter might actually have a type name instead (e.g. `$hour` for `addHour()`), although that does not prevent the use of `Zend_Date` objects as arguments for that parameter. If no `$locale` was specified, then the locale of the current object is used to interpret `$date`, or select the localized format for output.

Since Zend Framework 1.7.0 `Zend_Date` does also support the usage of an application wide locale. You can simply set a `Zend_Locale` instance to the registry like shown below. With this notation you can forget about setting the locale manually with each instance when you want to use the same locale multiple times.

```
// in your bootstrap file
$locale = new Zend_Locale('de_AT');
Zend_Registry::set('Zend_Locale', $locale);

// somewhere in your application
$date = new Zend_Date('31.Feb.2007');
```

4.3. Basic Zend_Date Operations Common to Many Date Parts

The methods `add()`, `sub()`, `compare()`, `get()`, and `set()` operate generically on dates. In each case, the operation is performed on the date held in the instance object. The `$date` operand is required for all of these methods, except `get()`, and may be a `Zend_Date` instance object, a numeric string, or an integer. These methods assume `$date` is a timestamp, if it is not an object. However, the `$part` operand controls which logical part of the two dates are operated on, allowing operations on parts of the object's date, such as year or minute, even when `$date` contains a long form date string, such as, "December 31, 2007 23:59:59". The result of the operation changes the date in the object, except for `compare()`, and `get()`.

Example 120. Operating on Parts of Dates

```
$date = new Zend_Date(); // $date's timestamp === time()

// changes $date by adding 12 hours
$date->add('12', Zend_Date::HOUR);
print $date;
```

Convenience methods exist for each combination of the basic operations and several common date parts as shown in the tables below. These convenience methods help us lazy programmers avoid having to type out the date part constants when using the general methods above. Conveniently, they are named by combining a prefix (name of a basic operation) with a suffix (type of date part), such as `addYear()`. In the list below, all combinations of "Date Parts" and "Basic Operations" exist. For example, the operation "add" exists for each of these date parts, including `addDay()`, `addYear()`, etc.

These convenience methods have the same equivalent functionality as the basic operation methods, but expect string and integer `$date` operands containing only the values representing the type indicated by the suffix of the convenience method. Thus, the names of these methods (e.g. "Year" or "Minute") identify the units of the `$date` operand, when `$date` is a string or integer.

4.3.1. List of Date Parts

Table 33. Date Parts

Date Part	Explanation
Timestamp	UNIX timestamp, expressed in seconds elapsed since January 1st, 1970 00:00:00 GMT.
Year	Gregorian calendar year (e.g. 2006)
Month	Gregorian calendar month (1-12, localized names supported)
24 hour clock	Hours of the day (0-23) denote the hours elapsed, since the start of the day.
minute	Minutes of the hour (0-59) denote minutes elapsed, since the start of the hour.
Second	Seconds of the minute (0-59) denote the elapsed seconds, since the start of the minute.
millisecond	Milliseconds denote thousandths of a second (0-999). `Zend_Date` supports two additional methods for working with time units smaller than seconds. By default, `Zend_Date` instances use a precision defaulting to milliseconds, as seen using `getFractionalPrecision()`. To change the precision use `setFractionalPrecision($precision)`. However, precision is limited practically to microseconds, since `Zend_Date` uses microtime().
Day	`Zend_Date::DAY_SHORT` is extracted from `$date` if the `$date` operand is an instance of `Zend_Date` or a numeric string. Otherwise, an attempt is made to extract the day according to the conventions documented for these constants: `Zend_Date::WEEKDAY_NARROW`, `Zend_Date::WEEKDAY_NAME`, `Zend_Date::WEEKDAY_SHORT`,

Date Part	Explanation
	`Zend_Date::WEEKDAY` (Gregorian calendar assumed)
Week	`Zend_Date::WEEK` is extracted from `$date` if the `$date` operand is an instance of `Zend_Date` or a numeric string. Otherwise an exception is raised. (Gregorian calendar assumed)
Date	`Zend_Date::DAY_MEDIUM` is extracted from `$date` if the `$date` operand is an instance of `Zend_Date`. Otherwise, an attempt is made to normalize the `$date` string into a `Zend_Date::DATE_MEDIUM` formatted date. The format of `Zend_Date::DAY_MEDIUM` depends on the object's locale.
Weekday	Weekdays are represented numerically as 0 (for Sunday) through 6 (for Saturday). `Zend_Date::WEEKDAY_DIGIT` is extracted from `$date`, if the `$date` operand is an instance of `Zend_Date` or a numeric string. Otherwise, an attempt is made to extract the day according to the conventions documented for these constants: `Zend_Date::WEEKDAY_NARROW`, `Zend_Date::WEEKDAY_NAME`, `Zend_Date::WEEKDAY_SHORT`, `Zend_Date::WEEKDAY` (Gregorian calendar assumed)
DayOfYear	In `Zend_Date`, the day of the year represents the number of calendar days elapsed since the start of the year (0-365). As with other units above, fractions are rounded down to the nearest whole number. (Gregorian calendar assumed)
Arpa	Arpa dates (i.e. RFC 822 formatted dates) are supported. Output uses either a "GMT" or "Local differential hours+min" format (see section 5 of RFC 822). Before PHP 5.2.2, using the `DATE_RFC822` constant with PHP date functions sometimes produces incorrect results. `Zend_Date`'s results are correct. Example: Mon, 31 Dec 06 23:59:59 GMT
Iso	Only complete ISO 8601 dates are supported for output. Example: 2009-02-14T00:31:30+01:00

4.3.2. List of Date Operations

The basic operations below can be used instead of the convenience operations for specific date parts, if the appropriate constant is used for the `$part` parameter.

Table 34. Basic Operations

Basic Operation	Explanation
`get()`	*`get($part = null, $locale = null)`* Use `get($part)` to retrieve the date `$part` of this object's date localized to `$locale` as a formatted string or integer. When using the BCMath extension, numeric strings might be returned instead of integers for large values.  **Behaviour of get()** Unlike `get()`, the other get*() convenience methods only return instances of `Zend_Date` containing a date representing the selected or computed date or time.
`set()`	*`set($date, $part = null, $locale = null)`* Sets the `$part` of the current object to the corresponding value for that part found in the input `$date` having a locale `$locale`.
`add()`	*`add($date, $part = null, $locale = null)`* Adds the `$part` of `$date` having a locale `$locale` to the current object's date.
`sub()`	*`sub($date, $part = null, $locale = null)`* Subtracts the `$part` of `$date` having a locale `$locale` from the current object's date.
`copyPart()`	*`copyPart($part, $locale = null)`* Returns a cloned object, with only `$part` of the object's date copied to the clone, with the clone have its locale arbitrarily set to `$locale` (if specified).
`compare()`	*`compare($date, $part = null, $locale = null)`* compares `$part` of `$date` to this object's timestamp, returning 0 if they are equal, 1 if this object's part was more recent than `$date`'s part, otherwise -1.

4.4. Comparing Dates

The following basic operations do not have corresponding convenience methods for the date parts listed in Zend_Date API Overview.

Table 35. Date Comparison Methods

Method	Explanation
equals()	*equals($date, $part = null, $locale = null)* returns TRUE, if $part of $date having locale $locale is the same as this object's date $part, otherwise FALSE
isEarlier()	*isEarlier($date, $part = null, $locale = null)* returns TRUE, if $part of this object's date is earlier than $part of $date having a locale $locale
isLater()	*isLater($date, $part = null, $locale = null)* returns TRUE, if $part of this object's date is later than $part of $date having a locale $locale
isToday()	*isToday()* Tests if today's year, month, and day match this object's date value, using this object's timezone.
isTomorrow()	*isTomorrow()* Tests if tomorrow's year, month, and day match this object's date value, using this object's timezone.
isYesterday()	*isYesterday()* Tests if yesterday's year, month, and day match this object's date value, using this object's timezone.
isLeapYear()	*isLeapYear()* Use isLeapYear() to determine if the current object is a leap year, or use Zend_Date::checkLeapYear($year) to check $year, which can be a string, integer, or instance of Zend_Date. Is the year a leap year?
isDate()	*isDate($date, $format = null, $locale = null)* This method checks if a given date is a real date and returns TRUE if all checks are ok. It works like PHP's checkdate() function but can also check for localized month names and for dates extending the range of checkdate()

4.5. Getting Dates and Date Parts

Several methods support retrieving values related to a `Zend_Date` instance.

Table 36. Date Output Methods

Method	Explanation
`toString()`	*`toString($format = null, $locale = null)`* Invoke directly or via the magic method `__toString()`. The `toString()` method automatically formats the date object's value according to the conventions of the object's locale, or an optionally specified `$locale`. For a list of supported format codes, see Self-Defined OUTPUT Formats with ISO.
`toArray()`	*`toArray()`* Returns an array representation of the selected date according to the conventions of the object's locale. The returned array is equivalent to PHP's getdate() function and includes: • Number of day as '*day*' (`Zend_Date::DAY_SHORT`) • Number of month as '*month*' (`Zend_Date::MONTH_SHORT`) • Year as '*year*' (`Zend_Date::YEAR`) • Hour as '*hour*' (`Zend_Date::HOUR_SHORT`) • Minute as '*minute*' (`Zend_Date::MINUTE_SHORT`) • Second as '*second*' (`Zend_Date::SECOND_SHORT`) • Abbreviated timezone as '*timezone*' (`Zend_Date::TIMEZONE`) • Unix timestamp as '*timestamp*' (`Zend_Date::TIMESTAMP`) • Number of weekday as '*weekday*' (`Zend_Date::WEEKDAY_DIGIT`) • Day of year as '*dayofyear*' (`Zend_Date::DAY_OF_YEAR`) • Week as '*week*' (`Zend_Date::WEEK`) • Delay of timezone to GMT as '*gmtsecs*' (`Zend_Date::GMT_SECS`)

Method	Explanation
`toValue()`	*`toValue($part = null)`* Returns an integer representation of the selected date `$part` according to the conventions of the object's locale. Returns `FALSE` when `$part` selects a non-numeric value, such as `Zend_Date::MONTH_NAME_SHORT`.  **Limitation of toValue()** This method calls `get()` and casts the result to a PHP integer, which will give unpredictable results, if `get()` returns a numeric string containing a number too large for a PHP integer on your system. Use `get()` instead.
get()	*`get($part = null, $locale = null)`* This method returns the `$part` of object's date localized to `$locale` as a formatted string or integer. See get() for more information.
`now()`	*`now($locale = null)`* This convenience function is equivalent to **new Zend_Date()**. It returns the current date as a `Zend_Date` object, having `$locale`

4.6. Working with Fractions of Seconds

Several methods support retrieving values related to a `Zend_Date` instance.

Table 37. Date Output Methods

Method	Explanation
`getFractionalPrecision()`	Return the precision of the part seconds
`setFractionalPrecision()`	Set the precision of the part seconds

4.7. Sunrise / Sunset

Three methods provide access to geographically localized information about the Sun, including the time of sunrise and sunset.

Table 38. Miscellaneous Methods

Method	Explanation
`getSunrise($location)`	Return the date's time of sunrise
`getSunset($location)`	Return the date's time of sunset
`getSunInfo($location)`	Return an array with the date's sun dates

5. Creation of Dates

`Zend_Date` provides several different ways to create a new instance of itself. As there are different needs the most convenient ways will be shown in this chapter.

5.1. Create the Actual Date

The simplest way of creating a date object is to create the actual date. You can either create a new instance with **new Zend_Date()** or use the convenient static method `Zend_Date::now()` which both will return the actual date as new instance of `Zend_Date`. The actual date always include the actual date and time for the actual set timezone.

Example 121. Date Creation by Instance

Date creation by creating a new instance means that you do not need to give an parameter. Of course there are several parameters which will be described later but normally this is the simplest and most used way to get the actual date as `Zend_Date` instance.

```
$date = new Zend_Date();
```

Example 122. Static Date Creation

Sometimes it is easier to use a static method for date creation. Therefor you can use the *now()* method. It returns a new instance of `Zend_Date` the same way as if you would use **new Zend_Date()**. But it will always return the actual date and can not be changed by giving optional parameters.

```
$date = Zend_Date::now();
```

5.2. Create a Date from Database

Databases are often used to store date values. But the problem is, that every database outputs its date values in a different way. *MsSQL* databases use a quite different standard date output than *MySQL* databases. But for simplification `Zend_Date` makes it very easy to create a date from database date values.

Of course each database can be said to convert the output of a defined column to a special value. For example you could convert a *datetime* value to output a minute value. But this is time expensive and often you are in need of handling dates in an other way than expected when creating the database query.

So we have one quick and one convenient way of creating dates from database values.

Example 123. Quick Creation of Dates from Database Date Values

All databases are known to handle queries as fast as possible. They are built to act and respond quick. The quickest way for handling dates is to get unix timestamps from the database. All databases store date values internal as timestamp (not unix timestamp). This means that the time for creating a timestamp through a query is much smaller than converting it to a specified format.

```
// SELECT UNIX_TIMESTAMP(my_datetime_column) FROM my_table
$date = new Zend_Date($unixtimestamp, Zend_Date::TIMESTAMP);
```

Example 124. Convenient Creation of Dates from Database Date Values

The standard output of all databases is quite different even if it looks the same on the first eyecatch. But all are part of the ISO Standard and explained through it. So the easiest way of date creation is the usage of `Zend_Date::ISO_8601`. Databases which are known to be recognised by `Zend_Date::ISO_8601` are *MySQL*, *MsSQL* for example. But all databases are also able to return a ISO-8601 representation of a date column. ISO-8601 has the big advantage that it is human readable. The disadvantage is that ISO-8601 needs more time for computation than a simple unix timestamp. But it should also be mentioned that unix timestamps are only supported for dates after 1 January 1970.

```
// SELECT datecolumn FROM my_table
$date = new Zend_Date($datecolumn, Zend_Date::ISO_8601);
```

5.3. Create Dates from an Array

Dates can also be created by the usage of an array. This is a simple and easy way. The used array keys are:

- *day*: day of the date as number
- *month*: month of the date as number
- *year*: full year of the date
- *hour*: hour of the date
- *minute*: minute of the date
- *second*: second of the date

Example 125. Date Creation by Array

Normally you will give a complete date array for creation of a new date instance. But when you do not give all values, the not given array values are zeroed. This means that if f.e. no hour is given the hour *0* is used.

```
$datearray = array('year' => 2006,
                   'month' => 4,
                   'day' => 18,
                   'hour' => 12,
                   'minute' => 3,
                   'second' => 10);
$date = new Zend_Date($datearray);
```

```
$datearray = array('year' => 2006, 'month' => 4, 'day' => 18);
$date = new Zend_Date($datearray);
```

6. Constants for General Date Functions

Whenever a `Zend_Date` method has a `$parts` parameter, one of the constants below can be used as the argument for that parameter, in order to select a specific part of a date or indicate the date format used or desired (e.g. RFC 822).

6.1. Using Constants

For example, the constant `Zend_Date::HOUR` can be used in the ways shown below. When working with days of the week, calendar dates, hours, minutes, seconds, and any other date

parts that are expressed differently when in different parts of the world, the object's timezone will automatically be used to compute the correct value, even though the internal timestamp is the same for the same moment in time, regardless of the user's physical location in the world. Regardless of the units involved, output must be expressed either as GMT or UTC or localized to a locale. The example output below reflects localization to Europe/GMT+1 hour (e.g. Germany, Austria, France).

Table 39. Operations Involving Zend_Date::HOUR

Method	Description	Original date	Result
`get(Zend_Date::HOUR)`	Output of the hour	2009-02-13T14:53:27+01:00	14
`set(12, Zend_Date::HOUR)`	Set new hour	2009-02-13T14:53:27+01:00	2009-02-13T12:53:27+01:00
`add(12, Zend_Date::HOUR)`	Add hours	2009-02-13T14:53:27+01:00	2009-02-14T02:53:27+01:00
`sub(12, Zend_Date::HOUR)`	Subtract hours	2009-02-13T14:53:27+01:00	2009-02-13T02:53:27+01:00
`compare(12, Zend_Date::HOUR)`	Compare hour, returns 0, 1 or -1	2009-02-13T14:53:27+01:00	1 (for object > argument)
`copy(Zend_Date::HOUR)`	Copies only the hour part	2009-02-13T14:53:27+01:00	1970-01-01T14:00:00+01:00
`equals(14, Zend_Date::HOUR)`	Compares the hour, returns TRUE or FALSE	2009-02-13T14:53:27+01:00	TRUE
`isEarlier(12, Zend_Date::HOUR)`	Compares the hour, returns TRUE or FALSE	2009-02-13T14:53:27+01:00	TRUE
`isLater(12, Zend_Date::HOUR)`	Compares the hour, returns TRUE or FALSE	2009-02-13T14:53:27+01:00	FALSE

6.2. List of All Constants

Each part of a date or time has a unique constant in `Zend_Date`. All constants supported by `Zend_Date` are listed below.

Table 40. Day Constants

Constant	Description	Date	Result
`Zend_Date::DAY`	Day (as number, two digits)	2009-02-13T14:53:27+01:00	13
`Zend_Date::DAY_SHORT`	Day (as number, one or two digits)	2009-02-06T14:53:27+01:00	6
`Zend_Date::WEEKDAY`	Weekday (Name of the day, localized, complete)	2009-02-13T14:53:27+01:00	*Friday*
`Zend_Date::WEEKDAY_SHORT`	Weekday (Name of the day, localized, abbreviated, two to four chars)	2009-02-13T14:53:27+01:00	Fri for Friday
`Zend_Date::WEEKDAY_NAME`	Weekday (Name of the day, localized,	2009-02-13T14:53:27+01:00	Fri for Friday

Constant	Description	Date	Result
	abbreviated, one or two chars)		
Zend_Date::WEEKDAY_NARROW	Weekday (Name of the day, localized, abbreviated, one char)	2009-02-13T14:53:27+01:00	F Friday
Zend_Date::WEEKDAY_DIGIT	Weekday (0 = Sunday, 6 = Saturday)	2009-02-13T14:53:27+01:00	5 Friday
Zend_Date::WEEKDAY_8601	Weekday according to ISO 8601 (1 = Monday, 7 = Sunday)	2009-02-13T14:53:27+01:00	5 Friday
Zend_Date::DAY_OF_YEAR	Day (as a number, one or two digits)	2009-02-13T14:53:27+01:00	43
Zend_Date::DAY_SUFFIX	English addendum for the day (st, nd, rd, th)	2009-02-13T14:53:27+01:00	th

Table 41. Week Constants

Constant	Description	Date	Result
Zend_Date::WEEK	Week (as number, 1-53)	2009-02-13T14:53:27+01:00	7

Table 42. Month Constants

Constant	Description	Date	Result
Zend_Date::MONTH_NAME	Month (Name of the month, localized, complete)	2009-02-13T14:53:27+01:00	*February*
Zend_Date::MONTH_NAME_SHORT	Name of the month, localized, abbreviated, two to four chars)	2009-02-13T14:53:27+01:00	Feb
Zend_Date::MONTH_NAME_NARROW	Name of the month, localized, abbreviated, one or two chars)	2009-02-13T14:53:27+01:00	F
Zend_Date::MONTH	Month (Number of the month, two digits)	2009-02-13T14:53:27+01:00	02
Zend_Date::MONTH_SHORT	Month (Number of the month, one or two digits)	2009-02-13T14:53:27+01:00	2
Zend_Date::MONTH_DAYS	Number of days for this month (number)	2009-02-13T14:53:27+01:00	28

Table 43. Year Constants

Constant	Description	Date	Result
Zend_Date::YEAR	Year (number)	2009-02-13T14:53:27+01:00	2009
Zend_Date::YEAR_8601	Year according to ISO 8601 (number)	2009-02-13T14:53:27+01:00	2009

Constant	Description	Date	Result
Zend_Date::YEAR_SHORT	Year (number, two digits)	2009-02-13T14:53:27+01:00	09
Zend_Date::YEAR_SHORT_8601	Year according to ISO 8601 (number, two digits)	2009-02-13T14:53:27+01:00	09
Zend_Date::LEAPYEAR	Is the year a leap year? (TRUE or FALSE)	2009-02-13T14:53:27+01:00	FALSE

Table 44. Time Constants

Constant	Description	Date	Result
Zend_Date::HOUR	Hour (00-23, two digits)	2009-02-13T14:53:27+01:00	14
Zend_Date::HOUR_SHORT	Hour (0-23, one or two digits)	2009-02-13T14:53:27+01:00	14
Zend_Date::HOUR_SHORT_AM	Hour (1-12, one or two digits)	2009-02-13T14:53:27+01:00	2
Zend_Date::HOUR_AM	Hour (01-12, two digits)	2009-02-13T14:53:27+01:00	02
Zend_Date::MINUTE	Minute (00-59, two digits)	2009-02-13T14:53:27+01:00	53
Zend_Date::MINUTE_SHORT	Minute (0-59, one or two digits)	2009-02-13T14:03:27+01:00	3
Zend_Date::SECOND	Second (00-59, two digits)	2009-02-13T14:53:27+01:00	27
Zend_Date::SECOND_SHORT	Second (0-59, one or two digits)	2009-02-13T14:53:07+01:00	7
Zend_Date::MILLISECOND	Millisecond (theoretically infinite)	**2009-02-06T14:53:27.205546**	*205546*
Zend_Date::MERIDIEM	Time of day (forenoon or afternoon)	2009-02-13T14:53:27+01:00	*afternoon*
Zend_Date::SWATCH	Swatch Internet Time	2009-02-13T14:53:27+01:00	620

Table 45. Timezone Constants

Constant	Description	Date	Result
Zend_Date::TIMEZONE	Name der time zone (string, abbreviated)	2009-02-13T14:53:27+01:00	CET
Zend_Date::TIMEZONE_NAME	Name of the time zone (string, complete)	2009-02-13T14:53:27+01:00	***Europe/Paris***
Zend_Date::TIMEZONE_SECS	Difference of the time zone to GMT in seconds (integer)	2009-02-13T14:53:27+01:00	*3600* (seconds to GMT)
Zend_Date::GMT_DIFF	Difference to GMT in seconds (string)	2009-02-13T14:53:27+01:00	*+0100*

Constant	Description	Date	Result
`Zend_Date::GMT_DIFF_SEP`	Difference to GMT in seconds (string, separated)	2009-02-13T14:53:27+0100	*+01:00*
`Zend_Date::DAYLIGHT`	Summer time or Winter time? (TRUE or FALSE)	2009-02-13T14:53:27+0100	FALSE

Table 46. Date Format Constants (formats include timezone)

Constant	Description	Date	Result
`Zend_Date::ISO_8601`	Date according to ISO 8601 (string, complete)	2009-02-13T14:53:27+0100	*2009-02-13T14:53:27+01:00*
`Zend_Date::RFC_2822`	Date according to RFC 2822 (string)	2009-02-13T14:53:27+0100	*Fri, 13 Feb 2009 14:53:27 +0100*
`Zend_Date::TIMESTAMP`	Unix time (seconds since 1.1.1970, mixed)	2009-02-13T14:53:27+0100	*1234533207*
`Zend_Date::ATOM`	Date according to ATOM (string)	2009-02-13T14:53:27+0100	*2009-02-13T14:53:27+01:00*
`Zend_Date::COOKIE`	Date for Cookies (string, for Cookies)	2009-02-13T14:53:27+0100	***Friday, 13-Feb-09 14:53:27 Europe/Paris***
`Zend_Date::RFC_822`	Date according to RFC 822 (string)	2009-02-13T14:53:27+0100	*Fri, 13 Feb 09 14:53:27 +0100*
`Zend_Date::RFC_850`	Date according to RFC 850 (string)	2009-02-13T14:53:27+0100	***Friday, 13-Feb-09 14:53:27 Europe/Paris***
`Zend_Date::RFC_1036`	Date according to RFC 1036 (string)	2009-02-13T14:53:27+0100	*Fri, 13 Feb 09 14:53:27 +0100*
`Zend_Date::RFC_1123`	Date according to RFC 1123 (string)	2009-02-13T14:53:27+0100	*Fri, 13 Feb 2009 14:53:27 +0100*
`Zend_Date::RSS`	Date for RSS Feeds (string)	2009-02-13T14:53:27+0100	*Fri, 13 Feb 2009 14:53:27 +0100*
`Zend_Date::W3C`	Date for HTML or HTTP according to W3C (string)	2009-02-13T14:53:27+0100	*2009-02-13T14:53:27+01:00*

Especially note `Zend_Date::DATES`, since this format specifier has a unique property within `Zend_Date` as an *input* format specifier. When used as an input format for `$part`, this constant provides the most flexible acceptance of a variety of similar date formats. Heuristics are used to automatically extract dates from an input string and then "fix" simple errors in dates (if any), such as swapping of years, months, and days, when possible.

Table 47. Date and Time Formats (format varies by locale)

Constant	Description	Date	Result
`Zend_Date::ERA`	Epoch (string, localized, abbreviated)	2009-02-13T14:53:27+0100	*AD* (anno Domini)
`Zend_Date::ERA_NAME`	Epoch (string, localized, complete)	2009-02-13T14:53:27+0100	*anno domini* (anno Domini)

Constant	Description	Date	Result
Zend_Date::DATES	Standard date (string, localized, default value).	2009-02-13T14:53:27+01:00	*13.02.2009*
Zend_Date::DATE_FULL	Complete date (string, localized, complete)	2009-02-13T14:53:27+01:00	*Friday, 13. February 2009*
Zend_Date::DATE_LONG	Long date (string, localized, long)	2009-02-13T14:53:27+01:00	*13. February 2009*
Zend_Date::DATE_MEDIUM	Normal date (string, localized, normal)	2009-02-13T14:53:27+01:00	*13.02.2009*
Zend_Date::DATE_SHORT	Abbreviated Date (string, localized, abbreviated)	2009-02-13T14:53:27+01:00	*13.02.09*
Zend_Date::TIMES	Standard time (string, localized, default value)	2009-02-13T14:53:27+01:00	*14:53:27*
Zend_Date::TIME_FULL	Complete time (string, localized, complete)	2009-02-13T14:53:27+01:00	***14:53 Uhr CET***
Zend_Date::TIME_LONG	Long time (string, localized, Long)	2009-02-13T14:53:27+01:00	***14:53:27 CET***
Zend_Date::TIME_MEDIUM	Normal time (string, localized, normal)	2009-02-13T14:53:27+01:00	*14:53:27*
Zend_Date::TIME_SHORT	Abbreviated time (string, localized, abbreviated)	2009-02-13T14:53:27+01:00	*14:53*
Zend_Date::DATETIME	Standard date with time (string, localized, default value).	2009-02-13T14:53:27+01:00	*13.02.2009 14:53:27*
Zend_Date::DATETIME_FULL	Complete date with time (string, localized, complete)	2009-02-13T14:53:27+01:00	***Friday, 13. February 2009 14:53 Uhr CET***
Zend_Date::DATETIME_LONG	Long date with time (string, localized, long)	2009-02-13T14:53:27+01:00	***13. February 2009 14:53:27 CET***
Zend_Date::DATETIME_MEDIUM	Normal date with time (string, localized, normal)	2009-02-13T14:53:27+01:00	*13.02.2009 14:53:27*
Zend_Date::DATETIME_SHORT	Abbreviated date with time (string, localized, abbreviated)	2009-02-13T14:53:27+01:00	*13.02.09 14:53*

6.3. Self-Defined OUTPUT Formats with ISO

If you need a date format not shown above, then use a self-defined format composed from the ISO format token specifiers below. The following examples illustrate the usage of constants from the table below to create self-defined ISO formats. The format length is unlimited. Also, multiple usage of format constants is allowed.

The accepted format specifiers can be changed from ISO Format to PHP's date format if you are more comfortable with it. However, not all formats defined in the ISO norm are supported with PHP's date format specifiers. Use the `Zend_Date::setOptions(array('format_type' => 'php'))` method to switch `Zend_Date` methods from supporting ISO format specifiers to PHP `date()` type specifiers (see Self-Defined OUTPUT Formats Using PHP's date() Format Specifiers below).

Example 126. Self-Defined ISO Formats

```
$locale = new Zend_Locale('de_AT');
$date = new Zend_Date(1234567890, false, $locale);
print $date->toString("'Era:GGGG='GGGG, ' Date:yy.MMMM.dd'yy.MMMM.dd");
```

Table 48. Constants for ISO 8601 Date Output

Constant	Description	Corresponds best to	Result
G	Epoch, localized, abbreviated	`Zend_Date::ERA`	*AD*
GG	Epoch, localized, abbreviated	`Zend_Date::ERA`	*AD*
GGG	Epoch, localized, abbreviated	`Zend_Date::ERA`	*AD*
GGGG	Epoch, localized, complete	`Zend_Date::ERA_NAME`	*Anno domini*
GGGGG	Epoch, localized, abbreviated	`Zend_Date::ERA`	*a*
y	Year, at least one digit	`Zend_Date::YEAR`	*9*
yy	Year, at least two digit	`Zend_Date::YEAR_SHORT`	*09*
yyy	Year, at least three digit	`Zend_Date::YEAR`	*2009*
yyyy	Year, at least four digit	`Zend_Date::YEAR`	*2009*
yyyyy	Year, at least five digit	`Zend_Date::YEAR`	*02009*
Y	Year according to ISO 8601, at least one digit	`Zend_Date::YEAR_8601`	*9*
YY	Year according to ISO 8601, at least two digit	`Zend_Date::YEAR_SHORT_8601`	*09*
YYY	Year according to ISO 8601, at least three digit	`Zend_Date::YEAR_8601`	*2009*
YYYY	Year according to ISO 8601, at least four digit	`Zend_Date::YEAR_8601`	*2009*
YYYYY	Year according to ISO 8601, at least five digit	`Zend_Date::YEAR_8601`	*02009*
M	Month, one or two digit	`Zend_Date::MONTH_SHORT`	*2*
MM	Month, two digit	`Zend_Date::MONTH`	*02*
MMM	Month, localized, abbreviated	`Zend_Date::MONTH_NAME_SHORT`	*Feb*

Constant	Description	Corresponds best to	Result
MMMM	Month, localized, complete	Zend_Date::MONTH_NAME	*February*
MMMMM	Month, localized, abbreviated, one digit	Zend_Date::MONTH_NAME_NARROW	F
w	Week, one or two digit	Zend_Date::WEEK	*5*
ww	Week, two digit	Zend_Date::WEEK	*05*
d	Day of the month, one or two digit	Zend_Date::DAY_SHORT	*9*
dd	Day of the month, two digit	Zend_Date::DAY	*09*
D	Day of the year, one, two or three digit	Zend_Date::DAY_OF_YEAR	*7*
DD	Day of the year, two or three digit	Zend_Date::DAY_OF_YEAR	*07*
DDD	Day of the year, three digit	Zend_Date::DAY_OF_YEAR	*007*
E	Day of the week, localized, abbreviated, one char	Zend_Date::WEEKDAY_NARROW	M
EE	Day of the week, localized, abbreviated, two or more chars	Zend_Date::WEEKDAY_NAME	Mo
EEE	Day of the week, localized, abbreviated, three chars	Zend_Date::WEEKDAY_SHORT	Mon
EEEE	Day of the week, localized, complete	Zend_Date::WEEKDAY	*Monday*
EEEEE	Day of the week, localized, abbreviated, one digit	Zend_Date::WEEKDAY_NARROW	M
e	Number of the day, one digit	Zend_Date::WEEKDAY_DIGIT	*1*
ee	Number of the day, two digit	Zend_Date::WEEKDAY_NARROW	*01*
a	Time of day, localized	Zend_Date::MERIDIEM	*vorm.*
h	Hour, (1-12), one or two digit	Zend_Date::HOUR_SHORT_AM	*2*
hh	Hour, (01-12), two digit	Zend_Date::HOUR_AM	*02*
H	Hour, (0-23), one or two digit	Zend_Date::HOUR_SHORT	*2*
HH	Hour, (00-23), two digit	Zend_Date::HOUR	*02*
m	Minute, (0-59), one or two digit	Zend_Date::MINUTE_SHORT	*2*

Constant	Description	Corresponds best to	Result
mm	Minute, (00-59), two digit	`Zend_Date::MINUTE`	*02*
s	Second, (0-59), one or two digit	`Zend_Date::SECOND_SHORT`	*2*
ss	Second, (00-59), two digit	`Zend_Date::SECOND`	*02*
S	Millisecond	`Zend_Date::MILLISECOND`	*208*
z	Time zone, localized, abbreviated	`Zend_Date::TIMEZONE_NAME`	*CET*
zz	Time zone, localized, abbreviated	`Zend_Date::TIMEZONE_NAME`	*CET*
zzz	Time zone, localized, abbreviated	`Zend_Date::TIMEZONE_NAME`	*CET*
zzzz	Time zone, localized, complete	`Zend_Date::TIMEZONE_NAME`	*Europe/Paris*
Z	Difference of time zone	`Zend_Date::GMT_DIFF`	*+0100*
ZZ	Difference of time zone	`Zend_Date::GMT_DIFF`	*+0100*
ZZZ	Difference of time zone	`Zend_Date::GMT_DIFF`	*+0100*
ZZZZ	Difference of time zone, separated	`Zend_Date::GMT_DIFF_SEP`	*+01:00*
A	Millisecond	`Zend_Date::MILLISECOND`	*208*

Note that the default ISO format differs from PHP's format which can be irritating if you have not used in previous. Especially the format specifiers for *Year and Minute* are often not used in the intended way.

For *year* there are two specifiers available which are often mistaken. The *Y* specifier for the ISO year and the *y* specifier for the real year. The difference is small but significant. *Y* calculates the ISO year, which is often used for calendar formats. See for example the 31. December 2007. The real year is 2007, but it is the first day of the first week in the week 1 of the year 2008. So, if you are using '**dd.MM.yyyy**' you will get '**31.December.2007**' but if you use '**dd.MM.YYYY**' you will get '**31.December.2008**'. As you see this is no bug but a expected behaviour depending on the used specifiers.

For *minute* the difference is not so big. ISO uses the specifier *m* for the minute, unlike PHP which uses *i*. So if you are getting no minute in your format check if you have used the right specifier.

6.4. Self-Defined OUTPUT Formats Using PHP's date() Format Specifiers

If you are more comfortable with PHP's date format specifier than with ISO format specifiers, then you can use the `Zend_Date::setOptions(array('format_type' => 'php'))` method to switch `Zend_Date` methods from supporting ISO format specifiers to PHP `date()` type specifiers. Afterwards, all format parameters must be given with PHP's `date()` format specifiers. The PHP date format lacks some of the formats supported by the ISO Format, and

vice-versa. If you are not already comfortable with it, then use the standard ISO format instead. Also, if you have legacy code using PHP's date format, then either manually convert it to the ISO format using Zend_Locale_Format::convertPhpToIsoFormat(), or use `setOptions()`. The following examples illustrate the usage of constants from the table below to create self-defined formats.

Example 127. Self-Defined Formats with PHP Specifier

```
$locale = new Zend_Locale('de_AT');
Zend_Date::setOptions(array('format_type' => 'php'));
$date = new Zend_Date(1234567890, false, $locale);

// outputs something like 'February 16, 2007, 3:36 am'
print $date->toString('F j, Y, g:i a');

print $date->toString("'Format:D M j G:i:s T Y='D M j G:i:s T Y");
```

PHP Date format and using constants

It is important to note that `Zend_Date`'s constants are using the ISO notation. This means, that when you set `Zend_Date` to use the PHP notation, you should not use `Zend_Date`'s constants, but define the wished format manually. If you don't follow this recommendation, you can get unexpected results.

The following table shows the list of PHP date format specifiers with their equivalent `Zend_Date` constants and CLDR and ISO equivalent format specifiers. In most cases, when the CLDR and ISO format does not have an equivalent format specifier, the PHP format specifier is not altered by `Zend_Locale_Format::convertPhpToIsoFormat()`, and the `Zend_Date` methods then recognize these "peculiar" PHP format specifiers, even when in the default "ISO" format mode.

Table 49. Constants for PHP Date Output

Constant	Description	Corresponds best to	closest CLDR equivalent	Result
d	Day of the month, two digit	`Zend_Date::DAY`	dd	*09*
D	Day of the week, localized, abbreviated, three digit	`Zend_Date::WEEKDAY`	EEE	*Mon*
j	Day of the month, one or two digit	`Zend_Date::DAY_SHORT`	d	*9*
l (lowercase L)	Day of the week, localized, complete	`Zend_Date::WEEKDAY`	EEEE	*Monday*
N	Number of the weekday, one digit	`Zend_Date::WEEKDAY_8601`	e	*4*
S	English suffixes for day of month, two chars	no equivalent	no equivalent	*st*
w	Number of the weekday,	`Zend_Date::WEEKDAY_DIGIT`	e	*4*

Constant	Description	Corresponds best to	closest CLDR equivalent	Result
	0=sunday, 6=saturday			
z	Day of the year, one, two or three digit	Zend_Date::DAY_OF_YEAR	D	*7*
W	Week, one or two digit	Zend_Date::WEEK	w	*5*
F	Month, localized, complete	Zend_Date::MONTH_NAME	MMMM	*February*
m	Month, two digit	Zend_Date::MONTH	MM	*02*
M	Month, localized, abbreviated	Zend_Date::MONTH_NAME_SHORT	MMM	*Feb*
n	Month, one or two digit	Zend_Date::MONTH_SHORT	M	*2*
t	Number of days per month, one or two digits	Zend_Date::MONTH_DAYS	ddd	*30*
L	Leapyear, boolean	Zend_Date::LEAPYEAR	no equivalent	*TRUE*
o	Year according to ISO 8601, at least four digit	Zend_Date::YEAR_8601	YYYY	*2009*
Y	Year, at least four digit	Zend_Date::YEAR	yyyy	*2009*
y	Year, at least two digit	Zend_Date::YEAR_SHORT	yy	*09*
a	Time of day, localized	Zend_Date::MERIDIEM	a (sort of, but likely to be uppercase)	*vorm.*
A	Time of day, localized	Zend_Date::MERIDIEM	a (sort of, but no guarantee that the format is uppercase)	***VORM.***
B	Swatch internet time	Zend_Date::SWATCH	no equivalent	*1463*
g	Hour, (1-12), one or two digit	Zend_Date::HOUR_SHORT_AM	h	*2*
G	Hour, (0-23), one or two digit	Zend_Date::HOUR_SHORT	H	*2*
h	Hour, (01-12), two digit	Zend_Date::HOUR_AM	hh	*02*
H	Hour, (00-23), two digit	Zend_Date::HOUR	HH	*02*

Constant	Description	Corresponds best to	closest CLDR equivalent	Result
i	Minute, (00-59), two digit	Zend_Date::MINUTE	mm	*02*
s	Second, (00-59), two digit	Zend_Date::SECOND	ss	*02*
e	Time zone, localized, complete	Zend_Date::TIMEZONE_NAME	zzzz	***Europe/Paris***
I	Daylight	Zend_Date::DAYLIGHT	no equivalent	*1*
O	Difference of time zone	Zend_Date::GMT_DIFF	Z or ZZZ	*+0100*
P	Difference of time zone, separated	Zend_Date::GMT_DIFF_SEP	ZZZZ	*+01:00*
T	Time zone, localized, abbreviated	Zend_Date::TIMEZONE	z or zzz	*CET*
Z	Time zone offset in seconds	Zend_Date::TIMEZONE_SECS	no equivalent	*3600*
c	Standard Iso format output	Zend_Date::ISO_8601	no equivalent	*2004-02-13T15:19:21+00:00*
r	Standard Rfc 2822 format output	Zend_Date::RFC_2822	no equivalent	*Thu, 21 Dec 2000 16:01:07 +0200*
U	Unix timestamp	Zend_Date::TIMESTAMP	no equivalent	*15275422364*

7. Working Examples

Within this chapter, we will describe several additional functions which are also available through `Zend_Date`. Of course all described functions have additional examples to show the expected working and the simple API for the proper using of them.

7.1. Checking Dates

Probably most dates you will get as input are strings. But the problem with strings is that you can not be sure if the string is a real date. Therefor `Zend_Date` has spent an own static function to check date strings. `Zend_Locale` has an own function `getDate($date, $locale)` which parses a date and returns the proper and normalized date parts. A monthname for example will be recognised and returned just a month number. But as `Zend_Locale` does not know anything about dates because it is a normalizing and localizing class we have integrated an own function `isDate($date)` which checks this.

`isDate($date, $format, $locale)` can take up to 3 parameters and needs minimum one parameter. So what we need to verify a date is, of course, the date itself as string. The second parameter can be the format which the date is expected to have. If no format is given the standard date format from your locale is used. For details about how formats should look like see the chapter about self defined formats.

The third parameter is also optional as the second parameter and can be used to give a locale. We need the locale to normalize monthnames and daynames. So with the third parameter we

are able to recognise dates like '**01.Jänner.2000**' or '**01.January.2000**' depending on the given locale.

`isDate()` of course checks if a date does exist. `Zend_Date` itself does not check a date. So it is possible to create a date like '**31.February.2000**' with `Zend_Date` because `Zend_Date` will automatically correct the date and return the proper date. In our case '**03.March.2000**'. `isDate()` on the other side does this check and will return `FALSE` on '**31.February.2000**' because it knows that this date is impossible.

Example 128. Checking Dates

```
// Checking dates
$date = '01.03.2000';
if (Zend_Date::isDate($date)) {
    print "String $date is a date";
} else {
    print "String $date is NO date";
}

// Checking localized dates
$date = '01 February 2000';
if (Zend_Date::isDate($date,'dd MMMM yyyy', 'en')) {
    print "String $date is a date";
} else {
    print "String $date is NO date";
}

// Checking impossible dates
$date = '30 February 2000';
if (Zend_Date::isDate($date,'dd MMMM yyyy', 'en')) {
    print "String $date is a date";
} else {
    print "String $date is NO date";
}
```

7.2. Sunrise and Sunset

`Zend_Date` has also functions integrated for getting informations from the sun. Often it is necessary to get the time for sunrise or sunset within a particularly day. This is quite easy with `Zend_Date` as just the expected day has to be given and additionally location for which the sunrise or sunset has to be calculated.

As most people do not know the location of their city we have also spent a helper class which provides the location data for about 250 capital and other big cities around the whole world. Most people could use cities near themself as the difference for locations situated to each other can only be measured within some seconds.

For creating a listbox and choosing a special city the function `Zend_Date_Cities::getCityList()` can be used. It returns the names of all available predefined cities for the helper class.

Example 129. Getting all Available Cities

```
// Output the complete list of available cities
print_r (Zend_Date_Cities::getCityList());
```

The location itself can be received with the `Zend_Date_Cities::city()` function. It accepts the name of the city as returned by the `Zend_Date_Cities::getCityList()` function and optional as second parameter the horizon to set.

There are 4 defined horizons which can be used with locations to receive the exact time of sunset and sunrise. The '`$horizon`' parameter is always optional in all functions. If it is not set, the 'effective' horizon is used.

Table 50. Types of Supported Horizons for Sunset and Sunrise

Horizon	Description	Usage
effective	Standard horizon	Expects the world to be a ball. This horizon is always used if non is defined.
civil	Common horizon	Often used in common medias like TV or radio
nautic	Nautic horizon	Often used in sea navigation
astronomic	Astronomic horizon	Often used for calculation with stars

Of course also a self-defined location can be given and calculated with. Therefor a 'latitude' and a 'longitude' has to be given and optional the 'horizon'.

Example 130. Getting the Location for a City

```
// Get the location for a defined city
// uses the effective horizon as no horizon is defined
print_r (Zend_Date_Cities::city('Vienna'));

// use the nautic horizon
print_r (Zend_Date_Cities::city('Vienna', 'nautic'));

// self definition of a location
$mylocation = array('latitude' => 41.5, 'longitude' => 13.2446);
```

As now all needed data can be set the next is to create a `Zend_Date` object with the day where sunset or sunrise should be calculated. For the calculation there are 3 functions available. It is possible to calculate sunset with '`getSunset()`', sunrise with '`getSunrise()`' and all available informations related to the sun with '`getSunInfo()`'. After the calculation the `Zend_Date` object will be returned with the calculated time.

Example 131. Calculating Sun Information

```
// Get the location for a defined city
$city = Zend_Date_Cities::city('Vienna');

// create a date object for the day for which the sun has to be calculated
$date = new Zend_Date('10.03.2007', Zend_Date::ISO_8601, 'de');

// calculate sunset
$sunset = $date->getSunset($city);
print $sunset->get(Zend_Date::ISO_8601);

// calculate all sun informations
$info = $date->getSunInfo($city);
foreach ($info as $sun) {
    print "\n" . $sun->get(Zend_Date::ISO_8601);
}
```

7.3. Time Zones

Time zones are as important as dates themselves. There are several time zones depending on where in the world a user lives. So working with dates also means to set the proper timezone. This may sound complicated but it's easier as expected. As already mentioned in the first chapter of `Zend_Date` the default timezone has to be set. Either by `php.ini` or by definition within the bootstrap file.

A `Zend_Date` object of course also stores the actual timezone. Even if the timezone is changed after the creation of the object it remembers the original timezone and works with it. It is also not necessary to change the timezone within the code with PHP functions. `Zend_Date` has two built-in functions which makes it possible to handle this.

`getTimezone()` returns the actual set timezone of within the `Zend_Date` object. Remember that `Zend_Date` is not coupled with PHP internals. So the returned timezone is not the timezone of the PHP script but the timezone of the object. `setTimezone($zone)` is the second function and makes it possible to set new timezone for `Zend_Date`. A given timezone is always checked. If it does not exist an exception will be thrown. Additionally the actual scripts or systems timezone can be set to the date object by calling `setTimezone()` without the zone parameter. This is also done automatically when the date object is created.

Example 132. Working with Time Zones

```
// Set a default timezone... this has to be done within the bootstrap
// file or php.ini.
// We do this here just for having a complete example.
date_default_timezone_set('Europe/Vienna');

// create a date object
$date = new Zend_Date('10.03.2007', Zend_Date::DATES, 'de');

// view our date object
print $date->getIso();

// what timezone do we have ?
print $date->getTimezone();

// set another timezone
$date->setTimezone('America/Chicago');

// what timezone do we now have ?
print $date->getTimezone();

// see the changed date object
print $date->getIso();
```

`Zend_Date` always takes the actual timezone for object creation as shown in the first lines of the example. Changing the timezone within the created object also has an effect to the date itself. Dates are always related to a timezone. Changing the timezone for a `Zend_Date` object does not change the time of `Zend_Date`. Remember that internally dates are always stored as timestamps and in GMT. So the timezone means how much hours should be substracted or added to get the actual global time for the own timezone and region.

Having the timezone coupled within `Zend_Date` has another positive effect. It is possible to have several dates with different timezones.

Example 133. Multiple Time Zones

```
// Set a default timezone... this has to be done within the bootstrap
// file or php.ini.
// We do this here just for having a complete example.
date_default_timezone_set('Europe/Vienna');

// create a date object
$date = new Zend_Date('10.03.2007 00:00:00', Zend_Date::ISO_8601, 'de');

// view our date object
print $date->getIso();

// the date stays unchanged even after changeing the timezone
date_default_timezone_set('America/Chicago');
print $date->getIso();

$otherdate = clone $date;
$otherdate->setTimezone('Brazil/Acre');

// view our date object
print $otherdate->getIso();

// set the object to the actual systems timezone
$lastdate = clone $date;
$lastdate->setTimezone();

// view our date object
print $lastdate->getIso();
```

Zend_Db

1. Zend_Db_Adapter

`Zend_Db` and its related classes provide a simple SQL database interface for Zend Framework. The `Zend_Db_Adapter` is the basic class you use to connect your PHP application to an RDBMS. There is a different Adapter class for each brand of RDBMS.

The `Zend_Db` adapters create a bridge from the vendor-specific PHP extensions to a common interface to help you write PHP applications once and deploy with multiple brands of RDBMS with very little effort.

The interface of the adapter class is similar to the interface of the PHP Data Objects extension. `Zend_Db` provides Adapter classes to PDO drivers for the following RDBMS brands:

- IBM DB2 and Informix Dynamic Server (IDS), using the pdo_ibm PHP extension
- MySQL, using the pdo_mysql PHP extension
- Microsoft SQL Server, using the pdo_dblib PHP extension
- Oracle, using the pdo_oci PHP extension
- PostgreSQL, using the pdo_pgsql PHP extension
- SQLite, using the pdo_sqlite PHP extension

In addition, `Zend_Db` provides Adapter classes that utilize PHP database extensions for the following RDBMS brands:

- MySQL, using the mysqli PHP extension
- Oracle, using the oci8 PHP extension
- IBM DB2 and DB2/i5, using the ibm_db2 PHP extension
- Firebird/Interbase, using the php_interbase PHP extension

Each `Zend_Db` Adapter uses a PHP extension. You must have the respective PHP extension enabled in your PHP environment to use a `Zend_Db` Adapter. For example, if you use any of the PDO `Zend_Db` Adapters, you need to enable both the PDO extension and the PDO driver for the brand of RDBMS you use.

1.1. Connecting to a Database Using an Adapter

This section describes how to create an instance of a database Adapter. This corresponds to making a connection to your RDBMS server from your PHP application.

1.1.1. Using a Zend_Db Adapter Constructor

You can create an instance of an adapter using its constructor. An adapter constructor takes one argument, which is an array of parameters used to declare the connection.

Example 134. Using an Adapter Constructor

```
$db = new Zend_Db_Adapter_Pdo_Mysql(array(
    'host'     => '127.0.0.1',
    'username' => 'webuser',
    'password' => 'xxxxxxxx',
    'dbname'   => 'test'
));
```

1.1.2. Using the Zend_Db Factory

As an alternative to using an adapter constructor directly, you can create an instance of an adapter using the static method `Zend_Db::factory()`. This method dynamically loads the adapter class file on demand using Zend_Loader::loadClass().

The first argument is a string that names the base name of the adapter class. For example the string '`Pdo_Mysql`' corresponds to the class `Zend_Db_Adapter_Pdo_Mysql`. The second argument is the same array of parameters you would have given to the adapter constructor.

Example 135. Using the Adapter Factory Method

```
// We don't need the following statement because the
// Zend_Db_Adapter_Pdo_Mysql file will be loaded for us by the Zend_Db
// factory method.

// require_once 'Zend/Db/Adapter/Pdo/Mysql.php';

// Automatically load class Zend_Db_Adapter_Pdo_Mysql
// and create an instance of it.
$db = Zend_Db::factory('Pdo_Mysql', array(
    'host'     => '127.0.0.1',
    'username' => 'webuser',
    'password' => 'xxxxxxxx',
    'dbname'   => 'test'
));
```

If you create your own class that extends `Zend_Db_Adapter_Abstract`, but you do not name your class with the "`Zend_Db_Adapter`" package prefix, you can use the `factory()` method to load your adapter if you specify the leading portion of the adapter class with the 'adapterNamespace' key in the parameters array.

Example 136. Using the Adapter Factory Method for a Custom Adapter Class

```
// We don't need to load the adapter class file
// because it will be loaded for us by the Zend_Db factory method.

// Automatically load class MyProject_Db_Adapter_Pdo_Mysql and create
// an instance of it.
$db = Zend_Db::factory('Pdo_Mysql', array(
    'host'             => '127.0.0.1',
    'username'         => 'webuser',
    'password'         => 'xxxxxxxx',
    'dbname'           => 'test',
    'adapterNamespace' => 'MyProject_Db_Adapter'
));
```

1.1.3. Using Zend_Config with the Zend_Db Factory

Optionally, you may specify either argument of the `factory()` method as an object of type Zend_Config.

If the first argument is a config object, it is expected to contain a property named adapter, containing the string naming the adapter class name base. Optionally, the object may contain a property named params, with subproperties corresponding to adapter parameter names. This is used only if the second argument of the `factory()` method is absent.

Example 137. Using the Adapter Factory Method with a Zend_Config Object

In the example below, a `Zend_Config` object is created from an array. You can also load data from an external file using classes such as Zend_Config_Ini and Zend_Config_Xml.

```
$config = new Zend_Config(
    array(
        'database' => array(
            'adapter' => 'Mysqli',
            'params'  => array(
                'host'     => '127.0.0.1',
                'dbname'   => 'test',
                'username' => 'webuser',
                'password' => 'secret',
            )
        )
    )
);

$db = Zend_Db::factory($config->database);
```

The second argument of the `factory()` method may be an associative array containing entries corresponding to adapter parameters. This argument is optional. If the first argument is of type `Zend_Config`, it is assumed to contain all parameters, and the second argument is ignored.

1.1.4. Adapter Parameters

The following list explains common parameters recognized by `Zend_Db` Adapter classes.

- *host*: a string containing a hostname or IP address of the database server. If the database is running on the same host as the PHP application, you may use 'localhost' or '127.0.0.1'.
- *username*: account identifier for authenticating a connection to the RDBMS server.
- *password*: account password credential for authenticating a connection to the RDBMS server.
- *dbname*: database instance name on the RDBMS server.
- *port*: some RDBMS servers can accept network connections on a administrator-specified port number. The port parameter allow you to specify the port to which your PHP application connects, to match the port configured on the RDBMS server.
- *charset*: specify the charset used for the connection.
- *options*: this parameter is an associative array of options that are generic to all `Zend_Db_Adapter` classes.

- *driver_options*: this parameter is an associative array of additional options that are specific to a given database extension. One typical use of this parameter is to set attributes of a PDO driver.

- *adapterNamespace*: names the initial part of the class name for the adapter, instead of 'Zend_Db_Adapter'. Use this if you need to use the factory() method to load a non-Zend database adapter class.

Example 138. Passing the Case-Folding Option to the Factory

You can specify this option by the constant Zend_Db::CASE_FOLDING. This corresponds to the ATTR_CASE attribute in PDO and IBM DB2 database drivers, adjusting the case of string keys in query result sets. The option takes values Zend_Db::CASE_NATURAL (the default), Zend_Db::CASE_UPPER, and Zend_Db::CASE_LOWER.

```
$options = array(
    Zend_Db::CASE_FOLDING => Zend_Db::CASE_UPPER
);

$params = array(
    'host'           => '127.0.0.1',
    'username'       => 'webuser',
    'password'       => 'xxxxxxxx',
    'dbname'         => 'test',
    'options'        => $options
);

$db = Zend_Db::factory('Db2', $params);
```

Example 139. Passing the Auto-Quoting Option to the Factory

You can specify this option by the constant Zend_Db::AUTO_QUOTE_IDENTIFIERS. If the value is TRUE (the default), identifiers like table names, column names, and even aliases are delimited in all SQL syntax generated by the Adapter object. This makes it simple to use identifiers that contain SQL keywords, or special characters. If the value is FALSE, identifiers are not delimited automatically. If you need to delimit identifiers, you must do so yourself using the quoteIdentifier() method.

```
$options = array(
    Zend_Db::AUTO_QUOTE_IDENTIFIERS => false
);

$params = array(
    'host'           => '127.0.0.1',
    'username'       => 'webuser',
    'password'       => 'xxxxxxxx',
    'dbname'         => 'test',
    'options'        => $options
);

$db = Zend_Db::factory('Pdo_Mysql', $params);
```

Example 140. Passing PDO Driver Options to the Factory

```
$pdoParams = array(
    PDO::MYSQL_ATTR_USE_BUFFERED_QUERY => true
);

$params = array(
    'host'           => '127.0.0.1',
    'username'       => 'webuser',
    'password'       => 'xxxxxxxx',
    'dbname'         => 'test',
    'driver_options' => $pdoParams
);

$db = Zend_Db::factory('Pdo_Mysql', $params);

echo $db->getConnection()
        ->getAttribute(PDO::MYSQL_ATTR_USE_BUFFERED_QUERY);
```

Example 141. Passing Serialization Options to the Factory

```
$options = array(
    Zend_Db::ALLOW_SERIALIZATION => false
);

$params = array(
    'host'           => '127.0.0.1',
    'username'       => 'webuser',
    'password'       => 'xxxxxxxx',
    'dbname'         => 'test',
    'options'        => $options
);

$db = Zend_Db::factory('Pdo_Mysql', $params);
```

1.1.5. Managing Lazy Connections

Creating an instance of an Adapter class does not immediately connect to the RDBMS server. The Adapter saves the connection parameters, and makes the actual connection on demand, the first time you need to execute a query. This ensures that creating an Adapter object is quick and inexpensive. You can create an instance of an Adapter even if you are not certain that you need to run any database queries during the current request your application is serving.

If you need to force the Adapter to connect to the RDBMS, use the `getConnection()` method. This method returns an object for the connection as represented by the respective PHP database extension. For example, if you use any of the Adapter classes for PDO drivers, then `getConnection()` returns the PDO object, after initiating it as a live connection to the specific database.

It can be useful to force the connection if you want to catch any exceptions it throws as a result of invalid account credentials, or other failure to connect to the RDBMS server. These exceptions are not thrown until the connection is made, so it can help simplify your application code if you handle the exceptions in one place, instead of at the time of the first query against the database.

Additionally, an adapter can get serialized to store it, for example, in a session variable. This can be very useful not only for the adapter itself, but for other objects that aggregate it, like a

`Zend_Db_Select` object. By default, adapters are allowed to be serialized, if you don't want it, you should consider passing the `Zend_Db::ALLOW_SERIALIZATION` option with `FALSE`, see the example above. To respect lazy connections principle, the adapter won't reconnect itself after being unserialized. You must then call `getConnection()` yourself. You can make the adapter auto-reconnect by passing the `Zend_Db::AUTO_RECONNECT_ON_UNSERIALIZE` with `TRUE` as an adapter option.

Example 142. Handling Connection Exceptions

```
try {
    $db = Zend_Db::factory('Pdo_Mysql', $parameters);
    $db->getConnection();
} catch (Zend_Db_Adapter_Exception $e) {
    // perhaps a failed login credential, or perhaps the RDBMS is not running
} catch (Zend_Exception $e) {
    // perhaps factory() failed to load the specified Adapter class
}
```

1.2. Example Database

In the documentation for `Zend_Db` classes, we use a set of simple tables to illustrate usage of the classes and methods. These example tables could store information for tracking bugs in a software development project. The database contains four tables:

- *accounts* stores information about each user of the bug-tracking database.
- *products* stores information about each product for which a bug can be logged.
- *bugs* stores information about bugs, including that current state of the bug, the person who reported the bug, the person who is assigned to fix the bug, and the person who is assigned to verify the fix.
- *bugs_products* stores a relationship between bugs and products. This implements a many-to-many relationship, because a given bug may be relevant to multiple products, and of course a given product can have multiple bugs.

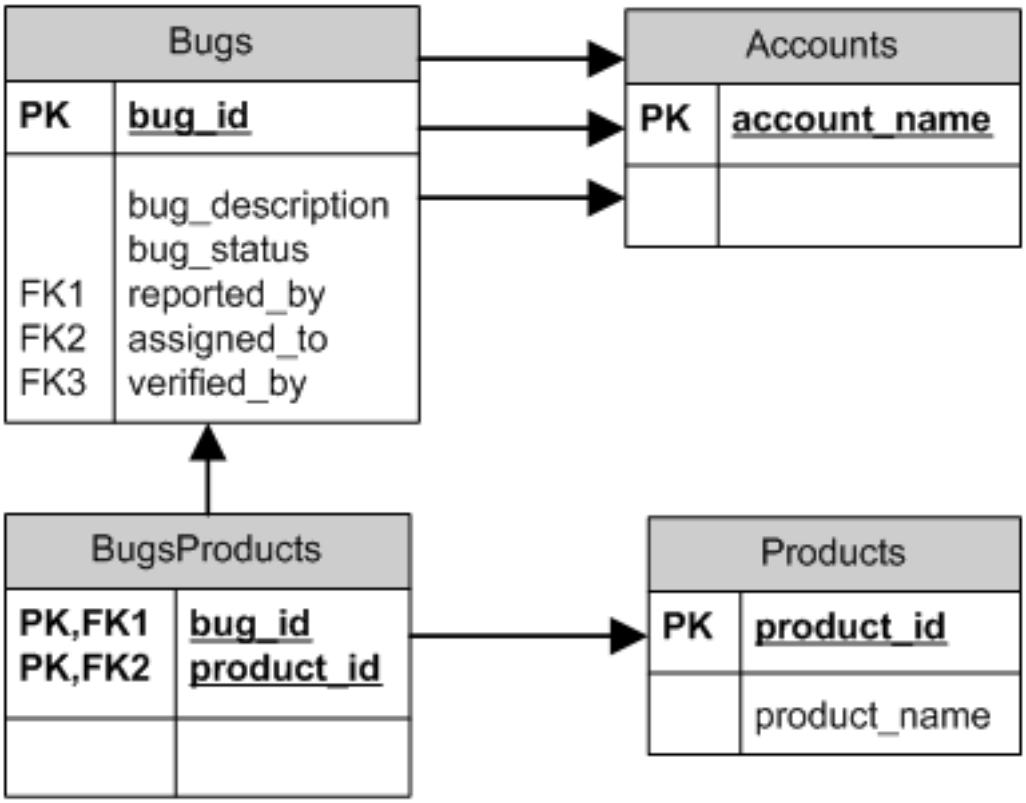
The following SQL data definition language pseudocode describes the tables in this example database. These example tables are used extensively by the automated unit tests for `Zend_Db`.

```
CREATE TABLE accounts (
  account_name      VARCHAR(100) NOT NULL PRIMARY KEY
);

CREATE TABLE products (
  product_id        INTEGER NOT NULL PRIMARY KEY,
  product_name      VARCHAR(100)
);

CREATE TABLE bugs (
  bug_id            INTEGER NOT NULL PRIMARY KEY,
  bug_description   VARCHAR(100),
  bug_status        VARCHAR(20),
  reported_by       VARCHAR(100) REFERENCES accounts(account_name),
  assigned_to       VARCHAR(100) REFERENCES accounts(account_name),
  verified_by       VARCHAR(100) REFERENCES accounts(account_name)
);
```

```
CREATE TABLE bugs_products (
  bug_id            INTEGER NOT NULL REFERENCES bugs,
  product_id        INTEGER NOT NULL REFERENCES products,
  PRIMARY KEY       (bug_id, product_id)
);
```

Also notice that the 'bugs' table contains multiple foreign key references to the 'accounts' table. Each of these foreign keys may reference a different row in the 'accounts' table for a given bug.

The diagram below illustrates the physical data model of the example database.



1.3. Reading Query Results

This section describes methods of the Adapter class with which you can run SELECT queries and retrieve the query results.

1.3.1. Fetching a Complete Result Set

You can run a SQL SELECT query and retrieve its results in one step using the `fetchAll()` method.

The first argument to this method is a string containing a SELECT statement. Alternatively, the first argument can be an object of class Zend_Db_Select. The Adapter automatically converts this object to a string representation of the SELECT statement.

The second argument to `fetchAll()` is an array of values to substitute for parameter placeholders in the SQL statement.

Example 143. Using fetchAll()

```
$sql = 'SELECT * FROM bugs WHERE bug_id = ?';

$result = $db->fetchAll($sql, 2);
```

1.3.2. Changing the Fetch Mode

By default, `fetchAll()` returns an array of rows, each of which is an associative array. The keys of the associative array are the columns or column aliases named in the select query.

You can specify a different style of fetching results using the `setFetchMode()` method. The modes supported are identified by constants:

- *Zend_Db::FETCH_ASSOC*: return data in an array of associative arrays. The array keys are column names, as strings. This is the default fetch mode for `Zend_Db_Adapter` classes.

  Note that if your select-list contains more than one column with the same name, for example if they are from two different tables in a JOIN, there can be only one entry in the associative array for a given name. If you use the FETCH_ASSOC mode, you should specify column aliases in your SELECT query to ensure that the names result in unique array keys.

  By default, these strings are returned as they are returned by the database driver. This is typically the spelling of the column in the RDBMS server. You can specify the case for these strings, using the `Zend_Db::CASE_FOLDING` option. Specify this when instantiating the Adapter. See Example 138, “Passing the Case-Folding Option to the Factory”.

- *Zend_Db::FETCH_NUM*: return data in an array of arrays. The arrays are indexed by integers, corresponding to the position of the respective field in the select-list of the query.

- *Zend_Db::FETCH_BOTH*: return data in an array of arrays. The array keys are both strings as used in the FETCH_ASSOC mode, and integers as used in the FETCH_NUM mode. Note that the number of elements in the array is double that which would be in the array if you used either FETCH_ASSOC or FETCH_NUM.

- *Zend_Db::FETCH_COLUMN*: return data in an array of values. The value in each array is the value returned by one column of the result set. By default, this is the first column, indexed by 0.

- *Zend_Db::FETCH_OBJ*: return data in an array of objects. The default class is the PHP built-in class stdClass. Columns of the result set are available as public properties of the object.

Example 144. Using setFetchMode()

```
$db->setFetchMode(Zend_Db::FETCH_OBJ);

$result = $db->fetchAll('SELECT * FROM bugs WHERE bug_id = ?', 2);

// $result is an array of objects
echo $result[0]->bug_description;
```

1.3.3. Fetching a Result Set as an Associative Array

The `fetchAssoc()` method returns data in an array of associative arrays, regardless of what value you have set for the fetch mode.

Example 145. Using fetchAssoc()

```
$db->setFetchMode(Zend_Db::FETCH_OBJ);

$result = $db->fetchAssoc('SELECT * FROM bugs WHERE bug_id = ?', 2);

// $result is an array of associative arrays, in spite of the fetch mode
echo $result[0]['bug_description'];
```

1.3.4. Fetching a Single Column from a Result Set

The `fetchCol()` method returns data in an array of values, regardless of the value you have set for the fetch mode. This only returns the first column returned by the query. Any other columns returned by the query are discarded. If you need to return a column other than the first, see Section 2.3.4, “Fetching a Single Column from a Result Set”.

Example 146. Using fetchCol()

```
$db->setFetchMode(Zend_Db::FETCH_OBJ);

$result = $db->fetchCol(
    'SELECT bug_description, bug_id FROM bugs WHERE bug_id = ?', 2);

// contains bug_description; bug_id is not returned
echo $result[0];
```

1.3.5. Fetching Key-Value Pairs from a Result Set

The `fetchPairs()` method returns data in an array of key-value pairs, as an associative array with a single entry per row. The key of this associative array is taken from the first column returned by the SELECT query. The value is taken from the second column returned by the SELECT query. Any other columns returned by the query are discarded.

You should design the SELECT query so that the first column returned has unique values. If there are duplicates values in the first column, entries in the associative array will be overwritten.

Example 147. Using fetchPairs()

```
$db->setFetchMode(Zend_Db::FETCH_OBJ);

$result = $db->fetchPairs('SELECT bug_id, bug_status FROM bugs');

echo $result[2];
```

1.3.6. Fetching a Single Row from a Result Set

The `fetchRow()` method returns data using the current fetch mode, but it returns only the first row fetched from the result set.

Example 148. Using fetchRow()

```
$db->setFetchMode(Zend_Db::FETCH_OBJ);

$result = $db->fetchRow('SELECT * FROM bugs WHERE bug_id = 2');

// note that $result is a single object, not an array of objects
echo $result->bug_description;
```

1.3.7. Fetching a Single Scalar from a Result Set

The `fetchOne()` method is like a combination of `fetchRow()` with `fetchCol()`, in that it returns data only for the first row fetched from the result set, and it returns only the value of the first column in that row. Therefore it returns only a single scalar value, not an array or an object.

Example 149. Using fetchOne()

```
$result = $db->fetchOne('SELECT bug_status FROM bugs WHERE bug_id = 2');

// this is a single string value
echo $result;
```

1.4. Writing Changes to the Database

You can use the Adapter class to write new data or change existing data in your database. This section describes methods to do these operations.

1.4.1. Inserting Data

You can add new rows to a table in your database using the `insert()` method. The first argument is a string that names the table, and the second argument is an associative array, mapping column names to data values.

Example 150. Inserting in a Table

```
$data = array(
    'created_on'      => '2007-03-22',
    'bug_description' => 'Something wrong',
    'bug_status'      => 'NEW'
);

$db->insert('bugs', $data);
```

Columns you exclude from the array of data are not specified to the database. Therefore, they follow the same rules that an SQL INSERT statement follows: if the column has a DEFAULT clause, the column takes that value in the row created, otherwise the column is left in a `NULL` state.

By default, the values in your data array are inserted using parameters. This reduces risk of some types of security issues. You don't need to apply escaping or quoting to values in the data array.

You might need values in the data array to be treated as SQL expressions, in which case they should not be quoted. By default, all data values passed as strings are treated as string literals. To specify that the value is an SQL expression and therefore should not be quoted, pass the value in the data array as an object of type `Zend_Db_Expr` instead of a plain string.

Example 151. Inserting Expressions in a Table

```
$data = array(
    'created_on'      => new Zend_Db_Expr('CURDATE()'),
    'bug_description' => 'Something wrong',
    'bug_status'      => 'NEW'
);

$db->insert('bugs', $data);
```

1.4.2. Retrieving a Generated Value

Some RDBMS brands support auto-incrementing primary keys. A table defined this way generates a primary key value automatically during an INSERT of a new row. The return value of the `insert()` method is *not* the last inserted ID, because the table might not have an auto-incremented column. Instead, the return value is the number of rows affected (usually 1).

If your table is defined with an auto-incrementing primary key, you can call the `lastInsertId()` method after the insert. This method returns the last value generated in the scope of the current database connection.

Example 152. Using lastInsertId() for an Auto-Increment Key

```
$db->insert('bugs', $data);

// return the last value generated by an auto-increment column
$id = $db->lastInsertId();
```

Some RDBMS brands support a sequence object, which generates unique values to serve as primary key values. To support sequences, the `lastInsertId()` method accepts two optional string arguments. These arguments name the table and the column, assuming you have followed the convention that a sequence is named using the table and column names for which the sequence generates values, and a suffix "_seq". This is based on the convention used by PostgreSQL when naming sequences for SERIAL columns. For example, a table "bugs" with primary key column "bug_id" would use a sequence named "bugs_bug_id_seq".

Example 153. Using lastInsertId() for a Sequence

```
$db->insert('bugs', $data);

// return the last value generated by sequence 'bugs_bug_id_seq'.
$id = $db->lastInsertId('bugs', 'bug_id');

// alternatively, return the last value generated by sequence 'bugs_seq'.
$id = $db->lastInsertId('bugs');
```

If the name of your sequence object does not follow this naming convention, use the `lastSequenceId()` method instead. This method takes a single string argument, naming the sequence literally.

Example 154. Using lastSequenceId()

```
$db->insert('bugs', $data);

// return the last value generated by sequence 'bugs_id_gen'.
$id = $db->lastSequenceId('bugs_id_gen');
```

For RDBMS brands that don't support sequences, including MySQL, Microsoft SQL Server, and SQLite, the arguments to the `lastInsertId()` method are ignored, and the value returned is the most recent value generated for any table by INSERT operations during the current connection. For these RDBMS brands, the `lastSequenceId()` method always returns `NULL`.

Why Not Use "SELECT MAX(id) FROM table"?

Sometimes this query returns the most recent primary key value inserted into the table. However, this technique is not safe to use in an environment where

multiple clients are inserting records to the database. It is possible, and therefore is bound to happen eventually, that another client inserts another row in the instant between the insert performed by your client application and your query for the `MAX(id)` value. Thus the value returned does not identify the row you inserted, it identifies the row inserted by some other client. There is no way to know when this has happened.

Using a strong transaction isolation mode such as "repeatable read" can mitigate this risk, but some RDBMS brands don't support the transaction isolation required for this, or else your application may use a lower transaction isolation mode by design.

Furthermore, using an expression like "MAX(id)+1" to generate a new value for a primary key is not safe, because two clients could do this query simultaneously, and then both use the same calculated value for their next INSERT operation.

All RDBMS brands provide mechanisms to generate unique values, and to return the last value generated. These mechanisms necessarily work outside of the scope of transaction isolation, so there is no chance of two clients generating the same value, and there is no chance that the value generated by another client could be reported to your client's connection as the last value generated.

1.4.3. Updating Data

You can update rows in a database table using the `update()` method of an Adapter. This method takes three arguments: the first is the name of the table; the second is an associative array mapping columns to change to new values to assign to these columns.

The values in the data array are treated as string literals. See Section 1.4.1, “Inserting Data” for information on using SQL expressions in the data array.

The third argument is a string containing an SQL expression that is used as criteria for the rows to change. The values and identifiers in this argument are not quoted or escaped. You are responsible for ensuring that any dynamic content is interpolated into this string safely. See Section 1.5, “Quoting Values and Identifiers” for methods to help you do this.

The return value is the number of rows affected by the update operation.

Example 155. Updating Rows

```
$data = array(
    'updated_on'      => '2007-03-23',
    'bug_status'      => 'FIXED'
);

$n = $db->update('bugs', $data, 'bug_id = 2');
```

If you omit the third argument, then all rows in the database table are updated with the values specified in the data array.

If you provide an array of strings as the third argument, these strings are joined together as terms in an expression separated by `AND` operators.

If you provide an array of arrays as the third argument, the values will be automatically quoted into the keys. These will then be joined together as terms, separated by `AND` operators.

Example 156. Updating Rows Using an Array of Expressions

```
$data = array(
    'updated_on'      => '2007-03-23',
    'bug_status'      => 'FIXED'
);

$where[] = "reported_by = 'goofy'";
$where[] = "bug_status = 'OPEN'";

$n = $db->update('bugs', $data, $where);

// Resulting SQL is:
//  UPDATE "bugs" SET "update_on" = '2007-03-23', "bug_status" = 'FIXED'
//  WHERE ("reported_by" = 'goofy') AND ("bug_status" = 'OPEN')
```

Example 157. Updating Rows Using an Array of Arrays

```
$data = array(
    'updated_on'      => '2007-03-23',
    'bug_status'      => 'FIXED'
);

$where['reported_by = ?'] = 'goofy';
$where['bug_status = ?']  = 'OPEN';

$n = $db->update('bugs', $data, $where);

// Resulting SQL is:
//  UPDATE "bugs" SET "update_on" = '2007-03-23', "bug_status" = 'FIXED'
//  WHERE ("reported_by" = 'goofy') AND ("bug_status" = 'OPEN')
```

1.4.4. Deleting Data

You can delete rows from a database table using the `delete()` method. This method takes two arguments: the first is a string naming the table.

The second argument is a string containing an SQL expression that is used as criteria for the rows to delete. The values and identifiers in this argument are not quoted or escaped. You are responsible for ensuring that any dynamic content is interpolated into this string safely. See Section 1.5, “Quoting Values and Identifiers” for methods to help you do this.

The return value is the number of rows affected by the delete operation.

Example 158. Deleting Rows

```
$n = $db->delete('bugs', 'bug_id = 3');
```

If you omit the second argument, the result is that all rows in the database table are deleted.

If you provide an array of strings as the second argument, these strings are joined together as terms in an expression separated by `AND` operators.

If you provide an array of arrays as the second argument, the values will be automatically quoted into the keys. These will then be joined together as terms, separated by `AND` operators.

1.5. Quoting Values and Identifiers

When you form SQL queries, often it is the case that you need to include the values of PHP variables in SQL expressions. This is risky, because if the value in a PHP string contains certain symbols, such as the quote symbol, it could result in invalid SQL. For example, notice the imbalanced quote characters in the following query:

```
$name = "O'Reilly";
$sql = "SELECT * FROM bugs WHERE reported_by = '$name'";

echo $sql;
// SELECT * FROM bugs WHERE reported_by = 'O'Reilly'
```

Even worse is the risk that such code mistakes might be exploited deliberately by a person who is trying to manipulate the function of your web application. If they can specify the value of a PHP variable through the use of an HTTP parameter or other mechanism, they might be able to make your SQL queries do things that you didn't intend them to do, such as return data to which the person should not have privilege to read. This is a serious and widespread technique for violating application security, known as "SQL Injection" (see http://en.wikipedia.org/wiki/SQL_Injection).

The `Zend_Db` Adapter class provides convenient functions to help you reduce vulnerabilities to SQL Injection attacks in your PHP code. The solution is to escape special characters such as quotes in PHP values before they are interpolated into your SQL strings. This protects against both accidental and deliberate manipulation of SQL strings by PHP variables that contain special characters.

1.5.1. Using quote()

The `quote()` method accepts a single argument, a scalar string value. It returns the value with special characters escaped in a manner appropriate for the RDBMS you are using, and surrounded by string value delimiters. The standard SQL string value delimiter is the single-quote (').

Example 159. Using quote()

```
$name = $db->quote("O'Reilly");
echo $name;
// 'O\'Reilly'

$sql = "SELECT * FROM bugs WHERE reported_by = $name";

echo $sql;
// SELECT * FROM bugs WHERE reported_by = 'O\'Reilly'
```

Note that the return value of `quote()` includes the quote delimiters around the string. This is different from some functions that escape special characters but do not add the quote delimiters, for example mysql_real_escape_string().

Values may need to be quoted or not quoted according to the SQL datatype context in which they are used. For instance, in some RDBMS brands, an integer value must not be quoted as a string if it is compared to an integer-type column or expression. In other words, the following is an error in some SQL implementations, assuming intColumn has a SQL datatype of `INTEGER`

```
SELECT * FROM atable WHERE intColumn = '123'
```

You can use the optional second argument to the `quote()` method to apply quoting selectively for the SQL datatype you specify.

Example 160. Using quote() with a SQL Type

```
$value = '1234';
$sql = 'SELECT * FROM atable WHERE intColumn = '
     . $db->quote($value, 'INTEGER');
```

Each `Zend_Db_Adapter` class has encoded the names of numeric SQL datatypes for the respective brand of RDBMS. You can also use the constants `Zend_Db::INT_TYPE`, `Zend_Db::BIGINT_TYPE`, and `Zend_Db::FLOAT_TYPE` to write code in a more RDBMS-independent way.

`Zend_Db_Table` specifies SQL types to `quote()` automatically when generating SQL queries that reference a table's key columns.

1.5.2. Using quoteInto()

The most typical usage of quoting is to interpolate a PHP variable into a SQL expression or statement. You can use the `quoteInto()` method to do this in one step. This method takes two arguments: the first argument is a string containing a placeholder symbol (?), and the second argument is a value or PHP variable that should be substituted for that placeholder.

The placeholder symbol is the same symbol used by many RDBMS brands for positional parameters, but the `quoteInto()` method only emulates query parameters. The method simply interpolates the value into the string, escapes special characters, and applies quotes around it. True query parameters maintain the separation between the SQL string and the parameters as the statement is parsed in the RDBMS server.

Example 161. Using quoteInto()

```
$sql = $db->quoteInto("SELECT * FROM bugs WHERE reported_by = ?", "O'Reilly");

echo $sql;
// SELECT * FROM bugs WHERE reported_by = 'O\'Reilly'
```

You can use the optional third parameter of `quoteInto()` to specify the SQL datatype. Numeric datatypes are not quoted, and other types are quoted.

Example 162. Using quoteInto() with a SQL Type

```
$sql = $db
    ->quoteInto("SELECT * FROM bugs WHERE bug_id = ?", '1234', 'INTEGER');

echo $sql;
// SELECT * FROM bugs WHERE reported_by = 1234
```

1.5.3. Using quoteIdentifier()

Values are not the only part of SQL syntax that might need to be variable. If you use PHP variables to name tables, columns, or other identifiers in your SQL statements, you might need to quote these strings too. By default, SQL identifiers have syntax rules like PHP and most other programming languages. For example, identifiers should not contain spaces, certain punctuation

or special characters, or international characters. Also certain words are reserved for SQL syntax, and should not be used as identifiers.

However, SQL has a feature called *delimited identifiers*, which allows broader choices for the spelling of identifiers. If you enclose a SQL identifier in the proper types of quotes, you can use identifiers with spellings that would be invalid without the quotes. Delimited identifiers can contain spaces, punctuation, or international characters. You can also use SQL reserved words if you enclose them in identifier delimiters.

The `quoteIdentifier()` method works like `quote()`, but it applies the identifier delimiter characters to the string according to the type of Adapter you use. For example, standard SQL uses double-quotes (") for identifier delimiters, and most RDBMS brands use that symbol. MySQL uses back-quotes (`) by default. The `quoteIdentifier()` method also escapes special characters within the string argument.

Example 163. Using quoteIdentifier()

```
// we might have a table name that is an SQL reserved word
$tableName = $db->quoteIdentifier("order");

$sql = "SELECT * FROM $tableName";

echo $sql
// SELECT * FROM "order"
```

SQL delimited identifiers are case-sensitive, unlike unquoted identifiers. Therefore, if you use delimited identifiers, you must use the spelling of the identifier exactly as it is stored in your schema, including the case of the letters.

In most cases where SQL is generated within `Zend_Db` classes, the default is that all identifiers are delimited automatically. You can change this behavior with the option `Zend_Db::AUTO_QUOTE_IDENTIFIERS`. Specify this when instantiating the Adapter. See Example 139, “Passing the Auto-Quoting Option to the Factory”.

1.6. Controlling Database Transactions

Databases define transactions as logical units of work that can be committed or rolled back as a single change, even if they operate on multiple tables. All queries to a database are executed within the context of a transaction, even if the database driver manages them implicitly. This is called *auto-commit* mode, in which the database driver creates a transaction for every statement you execute, and commits that transaction after your SQL statement has been executed. By default, all `Zend_Db` Adapter classes operate in auto-commit mode.

Alternatively, you can specify the beginning and resolution of a transaction, and thus control how many SQL queries are included in a single group that is committed (or rolled back) as a single operation. Use the `beginTransaction()` method to initiate a transaction. Subsequent SQL statements are executed in the context of the same transaction until you resolve it explicitly.

To resolve the transaction, use either the `commit()` or `rollBack()` methods. The `commit()` method marks changes made during your transaction as committed, which means the effects of these changes are shown in queries run in other transactions.

The `rollBack()` method does the opposite: it discards the changes made during your transaction. The changes are effectively undone, and the state of the data returns to how it was before you began your transaction. However, rolling back your transaction has no effect on changes made by other transactions running concurrently.

After you resolve this transaction, `Zend_Db_Adapter` returns to auto-commit mode until you call `beginTransaction()` again.

Example 164. Managing a Transaction to Ensure Consistency

```
// Start a transaction explicitly.
$db->beginTransaction();

try {
    // Attempt to execute one or more queries:
    $db->query(...);
    $db->query(...);
    $db->query(...);

    // If all succeed, commit the transaction and all changes
    // are committed at once.
    $db->commit();

} catch (Exception $e) {
    // If any of the queries failed and threw an exception,
    // we want to roll back the whole transaction, reversing
    // changes made in the transaction, even those that succeeded.
    // Thus all changes are committed together, or none are.
    $db->rollBack();
    echo $e->getMessage();
}
```

1.7. Listing and Describing Tables

The `listTables()` method returns an array of strings, naming all tables in the current database.

The `describeTable()` method returns an associative array of metadata about a table. Specify the name of the table as a string in the first argument to this method. The second argument is optional, and names the schema in which the table exists.

The keys of the associative array returned are the column names of the table. The value corresponding to each column is also an associative array, with the following keys and values:

Table 51. Metadata Fields Returned by describeTable()

Key	Type	Description
SCHEMA_NAME	(string)	Name of the database schema in which this table exists.
TABLE_NAME	(string)	Name of the table to which this column belongs.
COLUMN_NAME	(string)	Name of the column.
COLUMN_POSITION	(integer)	Ordinal position of the column in the table.
DATA_TYPE	(string)	RDBMS name of the datatype of the column.
DEFAULT	(string)	Default value for the column, if any.

Key	Type	Description
NULLABLE	(boolean)	TRUE if the column accepts SQL NULL's, FALSE if the column has a NOT NULL constraint.
LENGTH	(integer)	Length or size of the column as reported by the RDBMS.
SCALE	(integer)	Scale of SQL NUMERIC or DECIMAL type.
PRECISION	(integer)	Precision of SQL NUMERIC or DECIMAL type.
UNSIGNED	(boolean)	TRUE if an integer-based type is reported as UNSIGNED.
PRIMARY	(boolean)	TRUE if the column is part of the primary key of this table.
PRIMARY_POSITION	(integer)	Ordinal position (1-based) of the column in the primary key.
IDENTITY	(boolean)	TRUE if the column uses an auto-generated value.

How the IDENTITY Metadata Field Relates to Specific RDBMSs

The IDENTITY metadata field was chosen as an 'idiomatic' term to represent a relation to surrogate keys. This field can be commonly known by the following values:-

- IDENTITY - DB2, MSSQL
- AUTO_INCREMENT - MySQL
- SERIAL - PostgreSQL
- SEQUENCE - Oracle

If no table exists matching the table name and optional schema name specified, then `describeTable()` returns an empty array.

1.8. Closing a Connection

Normally it is not necessary to close a database connection. PHP automatically cleans up all resources and the end of a request. Database extensions are designed to close the connection as the reference to the resource object is cleaned up.

However, if you have a long-duration PHP script that initiates many database connections, you might need to close the connection, to avoid exhausting the capacity of your RDBMS server. You can use the Adapter's `closeConnection()` method to explicitly close the underlying database connection.

Since release 1.7.2, you could check you are currently connected to the RDBMS server with the method `isConnected()`. This means that a connection resource has been initiated and wasn't closed. This function is not currently able to test for example a server side closing of the

connection. This is internally use to close the connection. It allow you to close the connection multiple times without errors. It was already the case before 1.7.2 for PDO adapters but not for the others.

Example 165. Closing a Database Connection

```
$db->closeConnection();
```

Does Zend_Db Support Persistent Connections?

Yes, persistence is supported through the addition of the persistent flag set to TRUE in the configuration (not driver_configuration) of an adapter in `Zend_Db`.

Example 166. Using the Persitence Flag with the Oracle Adapter

```
$db = Zend_Db::factory('Oracle', array(
    'host'       => '127.0.0.1',
    'username'   => 'webuser',
    'password'   => 'xxxxxxxx',
    'dbname'     => 'test',
    'persistent' => true
));
```

Please note that using persistent connections can cause an excess of idle connections on the RDBMS server, which causes more problems than any performance gain you might achieve by reducing the overhead of making connections.

Database connections have state. That is, some objects in the RDBMS server exist in session scope. Examples are locks, user variables, temporary tables, and information about the most recently executed query, such as rows affected, and last generated id value. If you use persistent connections, your application could access invalid or privileged data that were created in a previous PHP request.

Currently, only Oracle, DB2, and the PDO adapters (where specified by PHP) support persistence in `Zend_Db`.

1.9. Running Other Database Statements

There might be cases in which you need to access the connection object directly, as provided by the PHP database extension. Some of these extensions may offer features that are not surfaced by methods of `Zend_Db_Adapter_Abstract`.

For example, all SQL statements run by `Zend_Db` are prepared, then executed. However, some database features are incompatible with prepared statements. DDL statements like CREATE and ALTER cannot be prepared in MySQL. Also, SQL statements don't benefit from the MySQL Query Cache, prior to MySQL 5.1.17.

Most PHP database extensions provide a method to execute SQL statements without preparing them. For example, in PDO, this method is `exec()`. You can access the connection object in the PHP extension directly using `getConnection()`.

Example 167. Running a Non-Prepared Statement in a PDO Adapter

```
$result = $db->getConnection()->exec('DROP TABLE bugs');
```

Similarly, you can access other methods or properties that are specific to PHP database extensions. Be aware, though, that by doing this you might constrain your application to the interface provided by the extension for a specific brand of RDBMS.

In future versions of `Zend_Db`, there will be opportunities to add method entry points for functionality that is common to the supported PHP database extensions. This will not affect backward compatibility.

1.10. Retrieving Server Version

Since release 1.7.2, you could retrieve the server version in PHP syntax style to be able to use `version_compare()`. If the information isn't available, you will receive `NULL`.

Example 168. Verifying server version before running a query

```
$version = $db->getServerVersion();
if (!is_null($version)) {
    if (version_compare($version, '5.0.0', '>=')) {
        // do something
    } else {
        // do something else
    }
} else {
    // impossible to read server version
}
```

1.11. Notes on Specific Adapters

This section lists differences between the Adapter classes of which you should be aware.

1.11.1. IBM DB2

- Specify this Adapter to the `factory()` method with the name 'Db2'.
- This Adapter uses the PHP extension ibm_db2.
- IBM DB2 supports both sequences and auto-incrementing keys. Therefore the arguments to `lastInsertId()` are optional. If you give no arguments, the Adapter returns the last value generated for an auto-increment key. If you give arguments, the Adapter returns the last value generated by the sequence named according to the convention '*table_column*_seq'.

1.11.2. MySQLi

- Specify this Adapter to the `factory()` method with the name 'Mysqli'.
- This Adapter utilizes the PHP extension mysqli.
- MySQL does not support sequences, so `lastInsertId()` ignores its arguments and always returns the last value generated for an auto-increment key. The `lastSequenceId()` method returns `NULL`.

1.11.3. Oracle

- Specify this Adapter to the `factory()` method with the name 'Oracle'.

- This Adapter uses the PHP extension oci8.
- Oracle does not support auto-incrementing keys, so you should specify the name of a sequence to `lastInsertId()` or `lastSequenceId()`.
- The Oracle extension does not support positional parameters. You must use named parameters.
- Currently the `Zend_Db::CASE_FOLDING` option is not supported by the Oracle adapter. To use this option with Oracle, you must use the PDO OCI adapter.
- By default, LOB fields are returned as OCI-Lob objects. You could retrieve them as string for all requests by using driver options 'lob_as_string' or for particular request by using `setLobAsString(boolean)` on adapter or on statement.

1.11.4. Microsoft SQL Server

- Specify this Adapter to the `factory()` method with the name 'Sqlsrv'.
- This Adapter uses the PHP extension sqlsrv
- Microsoft SQL Server does not support sequences, so `lastInsertId()` ignores primary key argument and returns the last value generated for an auto-increment key if a table name is specified or a last insert query returned id. The `lastSequenceId()` method returns `NULL`.
- `Zend_Db_Adapter_Sqlsrv` sets `QUOTED_IDENTIFIER ON` immediately after connecting to a SQL Server database. This makes the driver use the standard SQL identifier delimiter symbol (") instead of the proprietary square-brackets syntax SQL Server uses for delimiting identifiers.
- You can specify driver_options as a key in the options array. The value can be a anything from here http://msdn.microsoft.com/en-us/library/cc296161(SQL.90).aspx.
- You can use `setTransactionIsolationLevel()` to set isolation level for current connection. The value can be `SQLSRV_TXN_READ_UNCOMMITTED`, `SQLSRV_TXN_READ_COMMITTED`, `SQLSRV_TXN_REPEATABLE_READ`, `SQLSRV_TXN_SNAPSHOT` or `SQLSRV_TXN_SERIALIZABLE`.
- As of Zend Framework 1.9, the minimal supported build of the PHP SQL Server extension from Microsoft is 1.0.1924.0. and the MSSQL Server Native Client version 9.00.3042.00.

1.11.5. PDO for IBM DB2 and Informix Dynamic Server (IDS)

- Specify this Adapter to the `factory()` method with the name '`Pdo_Ibm`'.
- This Adapter uses the PHP extensions pdo and pdo_ibm.
- You must use at least PDO_IBM extension version 1.2.2. If you have an earlier version of this extension, you must upgrade the PDO_IBM extension from PECL.

1.11.6. PDO Microsoft SQL Server

- Specify this Adapter to the `factory()` method with the name '`Pdo_Mssql`'.
- This Adapter uses the PHP extensions pdo and pdo_dblib.

- Microsoft SQL Server does not support sequences, so `lastInsertId()` ignores its arguments and always returns the last value generated for an auto-increment key. The `lastSequenceId()` method returns `NULL`.

- If you are working with unicode strings in an encoding other than UCS-2 (such as UTF-8), you may have to perform a conversion in your application code or store the data in a binary column. Please refer to Microsoft's Knowledge Base for more information.

- `Zend_Db_Adapter_Pdo_Mssql` sets `QUOTED_IDENTIFIER ON` immediately after connecting to a SQL Server database. This makes the driver use the standard SQL identifier delimiter symbol (") instead of the proprietary square-brackets syntax SQL Server uses for delimiting identifiers.

- You can specify pdoType as a key in the options array. The value can be "mssql" (the default), "dblib", "freetds", or "sybase". This option affects the DSN prefix the adapter uses when constructing the DSN string. Both "freetds" and "sybase" imply a prefix of "sybase:", which is used for the FreeTDS set of libraries. See also http://www.php.net/manual/en/ref.pdo-dblib.connection.php for more information on the DSN prefixes used in this driver.

1.11.7. PDO MySQL

- Specify this Adapter to the `factory()` method with the name '`Pdo_Mysql`'.

- This Adapter uses the PHP extensions pdo and pdo_mysql.

- MySQL does not support sequences, so `lastInsertId()` ignores its arguments and always returns the last value generated for an auto-increment key. The `lastSequenceId()` method returns `NULL`.

1.11.8. PDO Oracle

- Specify this Adapter to the `factory()` method with the name '`Pdo_Oci`'.

- This Adapter uses the PHP extensions pdo and pdo_oci.

- Oracle does not support auto-incrementing keys, so you should specify the name of a sequence to `lastInsertId()` or `lastSequenceId()`.

1.11.9. PDO PostgreSQL

- Specify this Adapter to the `factory()` method with the name '`Pdo_Pgsql`'.

- This Adapter uses the PHP extensions pdo and pdo_pgsql.

- PostgreSQL supports both sequences and auto-incrementing keys. Therefore the arguments to `lastInsertId()` are optional. If you give no arguments, the Adapter returns the last value generated for an auto-increment key. If you give arguments, the Adapter returns the last value generated by the sequence named according to the convention '*table_column*_seq'.

1.11.10. PDO SQLite

- Specify this Adapter to the `factory()` method with the name '`Pdo_Sqlite`'.

- This Adapter uses the PHP extensions pdo and pdo_sqlite.

- SQLite does not support sequences, so `lastInsertId()` ignores its arguments and always returns the last value generated for an auto-increment key. The `lastSequenceId()` method returns `NULL`.
- To connect to an SQLite2 database, specify **'sqlite2' => true** in the array of parameters when creating an instance of the `Pdo_Sqlite` Adapter.
- To connect to an in-memory SQLite database, specify **'dbname' => ':memory:'** in the array of parameters when creating an instance of the `Pdo_Sqlite` Adapter.
- Older versions of the SQLite driver for PHP do not seem to support the PRAGMA commands necessary to ensure that short column names are used in result sets. If you have problems that your result sets are returned with keys of the form "tablename.columnname" when you do a join query, then you should upgrade to the current version of PHP.

1.11.11. Firebird/Interbase

- This Adapter uses the PHP extension php_interbase.
- Firebird/interbase does not support auto-incrementing keys, so you should specify the name of a sequence to `lastInsertId()` or `lastSequenceId()`.
- Currently the `Zend_Db::CASE_FOLDING` option is not supported by the Firebird/interbase adapter. Unquoted identifiers are automatically returned in upper case.
- Adapter name is `ZendX_Db_Adapter_Firebird`.

  Remember to use the param adapterNamespace with value `ZendX_Db_Adapter`.

  We recommend to update the `gds32.dll` (or linux equivalent) bundled with php, to the same version of the server. For Firebird the equivalent `gds32.dll` is `fbclient.dll`.

  By default all identifiers (tables names, fields) are returned in upper case.

2. Zend_Db_Statement

In addition to convenient methods such as `fetchAll()` and `insert()` documented in Section 1, “Zend_Db_Adapter”, you can use a statement object to gain more options for running queries and fetching result sets. This section describes how to get an instance of a statement object, and how to use its methods.

`Zend_Db_Statement` is based on the PDOStatement object in the PHP Data Objects extension.

2.1. Creating a Statement

Typically, a statement object is returned by the `query()` method of the database Adapter class. This method is a general way to prepare any SQL statement. The first argument is a string containing an SQL statement. The optional second argument is an array of values to bind to parameter placeholders in the SQL string.

Example 169. Creating a SQL statement object with query()

```
$stmt = $db->query(
            'SELECT * FROM bugs WHERE reported_by = ? AND bug_status = ?',
            array('goofy', 'FIXED')
        );
```

The statement object corresponds to a SQL statement that has been prepared, and executed once with the bind-values specified. If the statement was a SELECT query or other type of statement that returns a result set, it is now ready to fetch results.

You can create a statement with its constructor, but this is less typical usage. There is no factory method to create this object, so you need to load the specific statement class and call its constructor. Pass the Adapter object as the first argument, and a string containing an SQL statement as the second argument. The statement is prepared, but not executed.

Example 170. Using a SQL statement constructor

```
$sql = 'SELECT * FROM bugs WHERE reported_by = ? AND bug_status = ?';

$stmt = new Zend_Db_Statement_Mysqli($db, $sql);
```

2.2. Executing a Statement

You need to execute a statement object if you create it using its constructor, or if you want to execute the same statement multiple times. Use the `execute()` method of the statement object. The single argument is an array of value to bind to parameter placeholders in the statement.

If you use *positional parameters*, or those that are marked with a question mark symbol ('*?*'), pass the bind values in a plain array.

Example 171. Executing a statement with positional parameters

```
$sql = 'SELECT * FROM bugs WHERE reported_by = ? AND bug_status = ?';

$stmt = new Zend_Db_Statement_Mysqli($db, $sql);

$stmt->execute(array('goofy', 'FIXED'));
```

If you use *named parameters*, or those that are indicated by a string identifier preceded by a colon character ('*:*'), pass the bind values in an associative array. The keys of this array should match the parameter names.

Example 172. Executing a statement with named parameters

```
$sql = 'SELECT * FROM bugs WHERE ' .
       'reported_by = :reporter AND bug_status = :status';

$stmt = new Zend_Db_Statement_Mysqli($db, $sql);

$stmt->execute(array(':reporter' => 'goofy', ':status' => 'FIXED'));
```

PDO statements support both positional parameters and named parameters, but not both types in a single SQL statement. Some of the `Zend_Db_Statement` classes for non-PDO extensions may support only one type of parameter or the other.

2.3. Fetching Results from a SELECT Statement

You can call methods on the statement object to retrieve rows from SQL statements that produce result set. SELECT, SHOW, DESCRIBE and EXPLAIN are examples of statements that produce a result set. INSERT, UPDATE, and DELETE are examples of statements that don't produce a result set. You can execute the latter SQL statements using `Zend_Db_Statement`, but you cannot call methods to fetch rows of results from them.

2.3.1. Fetching a Single Row from a Result Set

To retrieve one row from the result set, use the `fetch()` method of the statement object. All three arguments of this method are optional:

- *Fetch style* is the first argument. This controls the structure in which the row is returned. See Section 1.3.2, “Changing the Fetch Mode” for a description of the valid values and the corresponding data formats.
- *Cursor orientation* is the second argument. The default is Zend_Db::FETCH_ORI_NEXT, which simply means that each call to `fetch()` returns the next row in the result set, in the order returned by the RDBMS.
- *Offset* is the third argument. If the cursor orientation is Zend_Db::FETCH_ORI_ABS, then the offset number is the ordinal number of the row to return. If the cursor orientation is Zend_Db::FETCH_ORI_REL, then the offset number is relative to the cursor position before `fetch()` was called.

`fetch()` returns FALSE if all rows of the result set have been fetched.

Example 173. Using fetch() in a loop

```
$stmt = $db->query('SELECT * FROM bugs');

while ($row = $stmt->fetch()) {
    echo $row['bug_description'];
}
```

See also PDOStatement::fetch().

2.3.2. Fetching a Complete Result Set

To retrieve all the rows of the result set in one step, use the `fetchAll()` method. This is equivalent to calling the `fetch()` method in a loop and returning all the rows in an array. The `fetchAll()` method accepts two arguments. The first is the fetch style, as described above, and the second indicates the number of the column to return, when the fetch style is Zend_Db::FETCH_COLUMN.

Example 174. Using fetchAll()

```
$stmt = $db->query('SELECT * FROM bugs');

$rows = $stmt->fetchAll();

echo $rows[0]['bug_description'];
```

See also PDOStatement::fetchAll().

2.3.3. Changing the Fetch Mode

By default, the statement object returns rows of the result set as associative arrays, mapping column names to column values. You can specify a different format for the statement class to return rows, just as you can in the Adapter class. You can use the `setFetchMode()` method of the statement object to specify the fetch mode. Specify the fetch mode using `Zend_Db` class constants FETCH_ASSOC, FETCH_NUM, FETCH_BOTH, FETCH_COLUMN, and FETCH_OBJ. See Section 1.3.2, “Changing the Fetch Mode” for more information on these modes. Subsequent calls to the statement methods `fetch()` or `fetchAll()` use the fetch mode that you specify.

Example 175. Setting the fetch mode

```
$stmt = $db->query('SELECT * FROM bugs');

$stmt->setFetchMode(Zend_Db::FETCH_NUM);

$rows = $stmt->fetchAll();

echo $rows[0][0];
```

See also PDOStatement::setFetchMode().

2.3.4. Fetching a Single Column from a Result Set

To return a single column from the next row of the result set, use `fetchColumn()`. The optional argument is the integer index of the column, and it defaults to 0. This method returns a scalar value, or FALSE if all rows of the result set have been fetched.

Note this method operates differently than the `fetchCol()` method of the Adapter class. The `fetchColumn()` method of a statement returns a single value from one row. The `fetchCol()` method of an adapter returns an array of values, taken from the first column of all rows of the result set.

Example 176. Using fetchColumn()

```
$stmt = $db->query('SELECT bug_id, bug_description, bug_status FROM bugs');

$bug_status = $stmt->fetchColumn(2);
```

See also PDOStatement::fetchColumn().

2.3.5. Fetching a Row as an Object

To retrieve a row from the result set structured as an object, use the `fetchObject()`. This method takes two optional arguments. The first argument is a string that names the class name of the object to return; the default is 'stdClass'. The second argument is an array of values that will be passed to the constructor of that class.

Example 177. Using fetchObject()

```
$stmt = $db->query('SELECT bug_id, bug_description, bug_status FROM bugs');

$obj = $stmt->fetchObject();

echo $obj->bug_description;
```

See also PDOStatement::fetchObject().

3. Zend_Db_Profiler

3.1. Introduction

`Zend_Db_Profiler` can be enabled to allow profiling of queries. Profiles include the queries processed by the adapter as well as elapsed time to run the queries, allowing inspection of the queries that have been performed without needing to add extra debugging code to classes. Advanced usage also allows the developer to filter which queries are profiled.

Enable the profiler by either passing a directive to the adapter constructor, or by asking the adapter to enable it later.

```
$params = array(
    'host'     => '127.0.0.1',
    'username' => 'webuser',
    'password' => 'xxxxxxxx',
    'dbname'   => 'test'
    'profiler' => true  // turn on profiler
                        // set to false to disable (disabled by default)
);

$db = Zend_Db::factory('PDO_MYSQL', $params);

// turn off profiler:
$db->getProfiler()->setEnabled(false);

// turn on profiler:
$db->getProfiler()->setEnabled(true);
```

The value of the 'profiler' option is flexible. It is interpreted differently depending on its type. Most often, you should use a simple boolean value, but other types enable you to customize the profiler behavior.

A boolean argument sets the profiler to enabled if it is a TRUE value, or disabled if FALSE. The profiler class is the adapter's default profiler class, `Zend_Db_Profiler`.

```
$params['profiler'] = true;
$db = Zend_Db::factory('PDO_MYSQL', $params);
```

An instance of a profiler object makes the adapter use that object. The object type must be `Zend_Db_Profiler` or a subclass thereof. Enabling the profiler is done separately.

```
$profiler = MyProject_Db_Profiler();
$profiler->setEnabled(true);
$params['profiler'] = $profiler;
$db = Zend_Db::factory('PDO_MYSQL', $params);
```

The argument can be an associative array containing any or all of the keys 'enabled', 'instance', and 'class'. The 'enabled' and 'instance' keys correspond to the boolean and instance types documented above. The 'class' key is used to name a class to use for a custom profiler. The class must be `Zend_Db_Profiler` or a subclass. The class is instantiated with no constructor arguments. The 'class' option is ignored when the 'instance' option is supplied.

```
$params['profiler'] = array(
    'enabled' => true,
    'class'   => 'MyProject_Db_Profiler'
);
$db = Zend_Db::factory('PDO_MYSQL', $params);
```

Finally, the argument can be an object of type `Zend_Config` containing properties, which are treated as the array keys described above. For example, a file "`config.ini`" might contain the following data:

```
[main]
db.profiler.class   = "MyProject_Db_Profiler"
db.profiler.enabled = true
```

This configuration can be applied by the following PHP code:

```
$config = new Zend_Config_Ini('config.ini', 'main');
$params['profiler'] = $config->db->profiler;
$db = Zend_Db::factory('PDO_MYSQL', $params);
```

The 'instance' property may be used as in the following:

```
$profiler = new MyProject_Db_Profiler();
$profiler->setEnabled(true);
$configData = array(
    'instance' => $profiler
    );
$config = new Zend_Config($configData);
$params['profiler'] = $config;
$db = Zend_Db::factory('PDO_MYSQL', $params);
```

3.2. Using the Profiler

At any point, grab the profiler using the adapter's `getProfiler()` method:

```
$profiler = $db->getProfiler();
```

This returns a `Zend_Db_Profiler` object instance. With that instance, the developer can examine your queries using a variety of methods:

- `getTotalNumQueries()` returns the total number of queries that have been profiled.
- `getTotalElapsedSecs()` returns the total number of seconds elapsed for all profiled queries.
- `getQueryProfiles()` returns an array of all query profiles.
- `getLastQueryProfile()` returns the last (most recent) query profile, regardless of whether or not the query has finished (if it hasn't, the end time will be `NULL`)
- `clear()` clears any past query profiles from the stack.

The return value of `getLastQueryProfile()` and the individual elements of `getQueryProfiles()` are `Zend_Db_Profiler_Query` objects, which provide the ability to inspect the individual queries themselves:

- `getQuery()` returns the SQL text of the query. The SQL text of a prepared statement with parameters is the text at the time the query was prepared, so it contains parameter placeholders, not the values used when the statement is executed.
- `getQueryParams()` returns an array of parameter values used when executing a prepared query. This includes both bound parameters and arguments to the statement's `execute()` method. The keys of the array are the positional (1-based) or named (string) parameter indices.
- `getElapsedSecs()` returns the number of seconds the query ran.

The information `Zend_Db_Profiler` provides is useful for profiling bottlenecks in applications, and for debugging queries that have been run. For instance, to see the exact query that was last run:

```
$query = $profiler->getLastQueryProfile();
```

```
echo $query->getQuery();
```

Perhaps a page is generating slowly; use the profiler to determine first the total number of seconds of all queries, and then step through the queries to find the one that ran longest:

```
$totalTime    = $profiler->getTotalElapsedSecs();
$queryCount   = $profiler->getTotalNumQueries();
$longestTime  = 0;
$longestQuery = null;

foreach ($profiler->getQueryProfiles() as $query) {
    if ($query->getElapsedSecs() > $longestTime) {
        $longestTime  = $query->getElapsedSecs();
        $longestQuery = $query->getQuery();
    }
}

echo 'Executed ' . $queryCount . ' queries in ' . $totalTime .
     ' seconds' . "\n";
echo 'Average query length: ' . $totalTime / $queryCount .
     ' seconds' . "\n";
echo 'Queries per second: ' . $queryCount / $totalTime . "\n";
echo 'Longest query length: ' . $longestTime . "\n";
echo "Longest query: \n" . $longestQuery . "\n";
```

3.3. Advanced Profiler Usage

In addition to query inspection, the profiler also allows the developer to filter which queries get profiled. The following methods operate on a `Zend_Db_Profiler` instance:

3.3.1. Filter by query elapsed time

`setFilterElapsedSecs()` allows the developer to set a minimum query time before a query is profiled. To remove the filter, pass the method a `NULL` value.

```
// Only profile queries that take at least 5 seconds:
$profiler->setFilterElapsedSecs(5);

// Profile all queries regardless of length:
$profiler->setFilterElapsedSecs(null);
```

3.3.2. Filter by query type

`setFilterQueryType()` allows the developer to set which types of queries should be profiled; to profile multiple types, logical OR them. Query types are defined as the following `Zend_Db_Profiler` constants:

- `Zend_Db_Profiler::CONNECT`: connection operations, or selecting a database.
- `Zend_Db_Profiler::QUERY`: general database queries that do not match other types.
- `Zend_Db_Profiler::INSERT`: any query that adds new data to the database, generally SQL INSERT.
- `Zend_Db_Profiler::UPDATE`: any query that updates existing data, usually SQL UPDATE.
- `Zend_Db_Profiler::DELETE`: any query that deletes existing data, usually SQL `DELETE`.

- `Zend_Db_Profiler::SELECT`: any query that retrieves existing data, usually SQL SELECT.
- `Zend_Db_Profiler::TRANSACTION`: any transactional operation, such as start transaction, commit, or rollback.

As with `setFilterElapsedSecs()`, you can remove any existing filters by passing `NULL` as the sole argument.

```
// profile only SELECT queries
$profiler->setFilterQueryType(Zend_Db_Profiler::SELECT);

// profile SELECT, INSERT, and UPDATE queries
$profiler->setFilterQueryType(Zend_Db_Profiler::SELECT |
                              Zend_Db_Profiler::INSERT |
                              Zend_Db_Profiler::UPDATE);

// profile DELETE queries
$profiler->setFilterQueryType(Zend_Db_Profiler::DELETE);

// Remove all filters
$profiler->setFilterQueryType(null);
```

3.3.3. Retrieve profiles by query type

Using `setFilterQueryType()` can cut down on the profiles generated. However, sometimes it can be more useful to keep all profiles, but view only those you need at a given moment. Another feature of `getQueryProfiles()` is that it can do this filtering on-the-fly, by passing a query type (or logical combination of query types) as its first argument; see this section for a list of the query type constants.

```
// Retrieve only SELECT query profiles
$profiles = $profiler->getQueryProfiles(Zend_Db_Profiler::SELECT);

// Retrieve only SELECT, INSERT, and UPDATE query profiles
$profiles = $profiler->getQueryProfiles(Zend_Db_Profiler::SELECT |
                                        Zend_Db_Profiler::INSERT |
                                        Zend_Db_Profiler::UPDATE);

// Retrieve DELETE query profiles
$profiles = $profiler->getQueryProfiles(Zend_Db_Profiler::DELETE);
```

3.4. Specialized Profilers

A Specialized Profiler is an object that inherits from `Zend_Db_Profiler`. Specialized Profilers treat profiling information in specific ways.

3.4.1. Profiling with Firebug

`Zend_Db_Profiler_Firebug` sends profiling infomation to the Firebug Console.

All data is sent via the `Zend_Wildfire_Channel_HttpHeaders` component which uses HTTP headers to ensure the page content is not disturbed. Debugging AJAX requests that require clean JSON and XML responses is possible with this approach.

Requirements:

- Firefox Browser ideally version 3 but version 2 is also supported.

- Firebug Firefox Extension which you can download from https://addons.mozilla.org/en-US/firefox/addon/1843.
- FirePHP Firefox Extension which you can download from https://addons.mozilla.org/en-US/firefox/addon/6149.

Example 178. DB Profiling with Zend_Controller_Front

```
// In your bootstrap file

$profiler = new Zend_Db_Profiler_Firebug('All DB Queries');
$profiler->setEnabled(true);

// Attach the profiler to your db adapter
$db->setProfiler($profiler)

// Dispatch your front controller

// All DB queries in your model, view and controller
// files will now be profiled and sent to Firebug
```

Example 179. DB Profiling without Zend_Controller_Front

```
$profiler = new Zend_Db_Profiler_Firebug('All DB Queries');
$profiler->setEnabled(true);

// Attach the profiler to your db adapter
$db->setProfiler($profiler)

$request  = new Zend_Controller_Request_Http();
$response = new Zend_Controller_Response_Http();
$channel  = Zend_Wildfire_Channel_HttpHeaders::getInstance();
$channel->setRequest($request);
$channel->setResponse($response);

// Start output buffering
ob_start();

// Now you can run your DB queries to be profiled

// Flush profiling data to browser
$channel->flush();
$response->sendHeaders();
```

4. Zend_Db_Select

4.1. Introduction

The `Zend_Db_Select` object represents a SQL SELECT query statement. The class has methods for adding individual parts to the query. You can specify some parts of the query using PHP methods and data structures, and the class forms the correct SQL syntax for you. After you build a query, you can execute the query as if you had written it as a string.

The value offered by `Zend_Db_Select` includes:

- Object-oriented methods for specifying SQL queries in a piece-by-piece manner;
- Database-independent abstraction of some parts of the SQL query;

- Automatic quoting of metadata identifiers in most cases, to support identifiers containing SQL reserved words and special characters;
- Quoting identifiers and values, to help reduce risk of SQL injection attacks.

Using `Zend_Db_Select` is not mandatory. For very simple SELECT queries, it is usually simpler to specify the entire SQL query as a string and execute it using Adapter methods like `query()` or `fetchAll()`. Using `Zend_Db_Select` is helpful if you need to assemble a SELECT query procedurally, or based on conditional logic in your application.

4.2. Creating a Select Object

You can create an instance of a `Zend_Db_Select` object using the `select()` method of a `Zend_Db_Adapter_Abstract` object.

Example 180. Example of the database adapter's select() method

```
$db = Zend_Db::factory( ...options... );
$select = $db->select();
```

Another way to create a `Zend_Db_Select` object is with its constructor, specifying the database adapter as an argument.

Example 181. Example of creating a new Select object

```
$db = Zend_Db::factory( ...options... );
$select = new Zend_Db_Select($db);
```

4.3. Building Select queries

When building the query, you can add clauses of the query one by one. There is a separate method to add each clause to the `Zend_Db_Select` object.

Example 182. Example of the using methods to add clauses

```
// Create the Zend_Db_Select object
$select = $db->select();

// Add a FROM clause
$select->from( ...specify table and columns... )

// Add a WHERE clause
$select->where( ...specify search criteria... )

// Add an ORDER BY clause
$select->order( ...specify sorting criteria... );
```

You also can use most methods of the `Zend_Db_Select` object with a convenient fluent interface. A fluent interface means that each method returns a reference to the object on which it was called, so you can immediately call another method.

Example 183. Example of the using the fluent interface

```
$select = $db->select()
    ->from( ...specify table and columns... )
    ->where( ...specify search criteria... )
    ->order( ...specify sorting criteria... );
```

The examples in this section show usage of the fluent interface, but you can use the non-fluent interface in all cases. It is often necessary to use the non-fluent interface, for example, if your application needs to perform some logic before adding a clause to a query.

4.3.1. Adding a FROM clause

Specify the table for this query using the `from()` method. You can specify the table name as a simple string. `Zend_Db_Select` applies identifier quoting around the table name, so you can use special characters.

Example 184. Example of the from() method

```
// Build this query:
//   SELECT *
//   FROM "products"

$select = $db->select()
             ->from( 'products' );
```

You can also specify the correlation name (sometimes called the "table alias") for a table. Instead of a simple string, use an associative array mapping the correlation name to the table name. In other clauses of the SQL query, use this correlation name. If your query joins more than one table, `Zend_Db_Select` generates unique correlation names based on the table names, for any tables for which you don't specify the correlation name.

Example 185. Example of specifying a table correlation name

```
// Build this query:
//   SELECT p.*
//   FROM "products" AS p

$select = $db->select()
             ->from( array('p' => 'products') );
```

Some RDBMS brands support a leading schema specifier for a table. You can specify the table name as "`schemaName.tableName`", where `Zend_Db_Select` quotes each part individually, or you may specify the schema name separately. A schema name specified in the table name takes precedence over a schema provided separately in the event that both are provided.

Example 186. Example of specifying a schema name

```
// Build this query:
//   SELECT *
//   FROM "myschema"."products"

$select = $db->select()
             ->from( 'myschema.products' );

// or

$select = $db->select()
             ->from('products', '*', 'myschema');
```

4.3.2. Adding Columns

In the second argument of the `from()` method, you can specify the columns to select from the respective table. If you specify no columns, the default is "*", the SQL wildcard for "all columns".

You can list the columns in a simple array of strings, or as an associative mapping of column alias to column name. If you only have one column to query, and you don't need to specify a column alias, you can list it as a plain string instead of an array.

If you give an empty array as the columns argument, no columns from the respective table are included in the result set. See a code example under the section on the `join()` method.

You can specify the column name as "`correlationName.columnName`". `Zend_Db_Select` quotes each part individually. If you don't specify a correlation name for a column, it uses the correlation name for the table named in the current `from()` method.

Example 187. Examples of specifying columns

```
// Build this query:
//   SELECT p."product_id", p."product_name"
//   FROM "products" AS p

$select = $db->select()
             ->from(array('p' => 'products'),
                    array('product_id', 'product_name'));

// Build the same query, specifying correlation names:
//   SELECT p."product_id", p."product_name"
//   FROM "products" AS p

$select = $db->select()
             ->from(array('p' => 'products'),
                    array('p.product_id', 'p.product_name'));

// Build this query with an alias for one column:
//   SELECT p."product_id" AS prodno, p."product_name"
//   FROM "products" AS p

$select = $db->select()
             ->from(array('p' => 'products'),
                    array('prodno' => 'product_id', 'product_name'));
```

4.3.3. Adding Expression Columns

Columns in SQL queries are sometimes expressions, not simply column names from a table. Expressions should not have correlation names or quoting applied. If your column string contains parentheses, `Zend_Db_Select` recognizes it as an expression.

You also can create an object of type `Zend_Db_Expr` explicitly, to prevent a string from being treated as a column name. `Zend_Db_Expr` is a minimal class that contains a single string. `Zend_Db_Select` recognizes objects of type `Zend_Db_Expr` and converts them back to string, but does not apply any alterations, such as quoting or correlation names.

Using `Zend_Db_Expr` for column names is not necessary if your column expression contains parentheses; `Zend_Db_Select` recognizes parentheses and treats the string as an expression, skipping quoting and correlation names.

Example 188. Examples of specifying columns containing expressions

```
// Build this query:
//   SELECT p."product_id", LOWER(product_name)
//   FROM "products" AS p
// An expression with parentheses implicitly becomes
// a Zend_Db_Expr.

$select = $db->select()
             ->from(array('p' => 'products'),
                    array('product_id', 'LOWER(product_name)'));

// Build this query:
//   SELECT p."product_id", (p.cost * 1.08) AS cost_plus_tax
//   FROM "products" AS p

$select = $db->select()
             ->from(array('p' => 'products'),
                    array('product_id',
                          'cost_plus_tax' => '(p.cost * 1.08)')
                   );

// Build this query using Zend_Db_Expr explicitly:
//   SELECT p."product_id", p.cost * 1.08 AS cost_plus_tax
//   FROM "products" AS p

$select = $db->select()
             ->from(array('p' => 'products'),
                    array('product_id',
                          'cost_plus_tax' =>
                              new Zend_Db_Expr('p.cost * 1.08'))
                    );
```

In the cases above, `Zend_Db_Select` does not alter the string to apply correlation names or identifier quoting. If those changes are necessary to resolve ambiguity, you must make the changes manually in the string.

If your column names are SQL keywords or contain special characters, you should use the Adapter's `quoteIdentifier()` method and interpolate the result into the string. The `quoteIdentifier()` method uses SQL quoting to delimit the identifier, which makes it clear that it is an identifier for a table or a column, and not any other part of SQL syntax.

Your code is more database-independent if you use the `quoteIdentifier()` method instead of typing quotes literally in your string, because some RDBMS brands use nonstandard symbols for quoting identifiers. The `quoteIdentifier()` method is designed to use the appropriate quoting symbols based on the adapter type. The `quoteIdentifier()` method also escapes any quote characters that appear within the identifier name itself.

Example 189. Examples of quoting columns in an expression

```
// Build this query,
// quoting the special column name "from" in the expression:
//   SELECT p."from" + 10 AS origin
//   FROM "products" AS p

$select = $db->select()
             ->from(array('p' => 'products'),
                    array('origin' =>
                              '(p.' . $db->quoteIdentifier('from') . ' + 10)')
                   );
```

4.3.4. Adding columns to an existing FROM or JOIN table

There may be cases where you wish to add columns to an existing FROM or JOIN table after those methods have been called. The `columns()` method allows you to add specific columns at any point before the query is executed. You can supply the columns as either a string or `Zend_Db_Expr` or as an array of these elements. The second argument to this method can be omitted, implying that the columns are to be added to the FROM table, otherwise an existing correlation name must be used.

Example 190. Examples of adding columns with the columns() method

```
// Build this query:
//   SELECT p."product_id", p."product_name"
//   FROM "products" AS p

$select = $db->select()
             ->from(array('p' => 'products'), 'product_id')
             ->columns('product_name');

// Build the same query, specifying correlation names:
//   SELECT p."product_id", p."product_name"
//   FROM "products" AS p

$select = $db->select()
             ->from(array('p' => 'products'), 'p.product_id')
             ->columns('product_name', 'p');
             // Alternatively use columns('p.product_name')
```

4.3.5. Adding Another Table to the Query with JOIN

Many useful queries involve using a JOIN to combine rows from multiple tables. You can add tables to a `Zend_Db_Select` query using the `join()` method. Using this method is similar to the `from()` method, except you can also specify a join condition in most cases.

Example 191. Example of the join() method

```
// Build this query:
//   SELECT p."product_id", p."product_name", l.*
//   FROM "products" AS p JOIN "line_items" AS l
//     ON p.product_id = l.product_id

$select = $db->select()
             ->from(array('p' => 'products'),
                    array('product_id', 'product_name'))
             ->join(array('l' => 'line_items'),
                    'p.product_id = l.product_id');
```

The second argument to `join()` is a string that is the join condition. This is an expression that declares the criteria by which rows in one table match rows in the other table. You can use correlation names in this expression.

No quoting is applied to the expression you specify for the join condition; if you have column names that need to be quoted, you must use `quoteIdentifier()` as you form the string for the join condition.

The third argument to `join()` is an array of column names, like that used in the `from()` method. It defaults to "*", supports correlation names, expressions, and `Zend_Db_Expr` in the same way as the array of column names in the `from()` method.

To select no columns from a table, use an empty array for the list of columns. This usage works in the `from()` method too, but typically you want some columns from the primary table in your queries, whereas you might want no columns from a joined table.

Example 192. Example of specifying no columns

```
// Build this query:
//   SELECT p."product_id", p."product_name"
//   FROM "products" AS p JOIN "line_items" AS l
//     ON p.product_id = l.product_id

$select = $db->select()
             ->from(array('p' => 'products'),
                    array('product_id', 'product_name'))
             ->join(array('l' => 'line_items'),
                    'p.product_id = l.product_id',
                    array() ); // empty list of columns
```

Note the empty `array()` in the above example in place of a list of columns from the joined table.

SQL has several types of joins. See the list below for the methods to support different join types in `Zend_Db_Select`.

- **INNER JOIN** with the `join(table, join, [columns])` or `joinInner(table, join, [columns])` methods.

  This may be the most common type of join. Rows from each table are compared using the join condition you specify. The result set includes only the rows that satisfy the join condition. The result set can be empty if no rows satisfy this condition.

All RDBMS brands support this join type.

- **LEFT JOIN** with the `joinLeft(table, condition, [columns])` method.

  All rows from the left operand table are included, matching rows from the right operand table included, and the columns from the right operand table are filled with `NULL` if no row exists matching the left table.

  All RDBMS brands support this join type.

- **RIGHT JOIN** with the `joinRight(table, condition, [columns])` method.

  Right outer join is the complement of left outer join. All rows from the right operand table are included, matching rows from the left operand table included, and the columns from the left operand table are filled with `NULL`'s if no row exists matching the right table.

  Some RDBMS brands don't support this join type, but in general any right join can be represented as a left join by reversing the order of the tables.

- **FULL JOIN** with the `joinFull(table, condition, [columns])` method.

  A full outer join is like combining a left outer join and a right outer join. All rows from both tables are included, paired with each other on the same row of the result set if they satisfy the join condition, and otherwise paired with `NULL`'s in place of columns from the other table.

  Some RDBMS brands don't support this join type.

- **CROSS JOIN** with the `joinCross(table, [columns])` method.

  A cross join is a Cartesian product. Every row in the first table is matched to every row in the second table. Therefore the number of rows in the result set is equal to the product of the number of rows in each table. You can filter the result set using conditions in a WHERE clause; in this way a cross join is similar to the old SQL-89 join syntax.

  The `joinCross()` method has no parameter to specify the join condition. Some RDBMS brands don't support this join type.

- **NATURAL JOIN** with the `joinNatural(table, [columns])` method.

  A natural join compares any column(s) that appear with the same name in both tables. The comparison is equality of all the column(s); comparing the columns using inequality is not a natural join. Only natural inner joins are supported by this API, even though SQL permits natural outer joins as well.

  The `joinNatural()` method has no parameter to specify the join condition.

In addition to these join methods, you can simplify your queries by using the JoinUsing methods. Instead of supplying a full condition to your join, you simply pass the column name on which to join and the `Zend_Db_Select` object completes the condition for you.

Example 193. Example of the joinUsing() method

```
// Build this query:
//   SELECT *
//   FROM "table1"
//   JOIN "table2"
//   ON "table1".column1 = "table2".column1
//   WHERE column2 = 'foo'

$select = $db->select()
             ->from('table1')
             ->joinUsing('table2', 'column1')
             ->where('column2 = ?', 'foo');
```

Each of the applicable join methods in the `Zend_Db_Select` component has a corresponding 'using' method.

- `joinUsing(table, join, [columns])` and `joinInnerUsing(table, join, [columns])`
- `joinLeftUsing(table, join, [columns])`
- `joinRightUsing(table, join, [columns])`
- `joinFullUsing(table, join, [columns])`

4.3.6. Adding a WHERE Clause

You can specify criteria for restricting rows of the result set using the `where()` method. The first argument of this method is a SQL expression, and this expression is used in a SQL WHERE clause in the query.

Example 194. Example of the where() method

```
// Build this query:
//   SELECT product_id, product_name, price
//   FROM "products"
//   WHERE price > 100.00

$select = $db->select()
             ->from('products',
                    array('product_id', 'product_name', 'price'))
             ->where('price > 100.00');
```

No quoting is applied to expressions given to the `where()` or `orWhere()` methods. If you have column names that need to be quoted, you must use `quoteIdentifier()` as you form the string for the condition.

The second argument to the `where()` method is optional. It is a value to substitute into the expression. `Zend_Db_Select` quotes the value and substitutes it for a question-mark ("?") symbol in the expression.

Example 195. Example of a parameter in the where() method

```
// Build this query:
//   SELECT product_id, product_name, price
//   FROM "products"
//   WHERE (price > 100.00)

$minimumPrice = 100;

$select = $db->select()
             ->from('products',
                    array('product_id', 'product_name', 'price'))
             ->where('price > ?', $minimumPrice);
```

You can pass an array as the second parameter to the `where()` method when using the SQL IN operator.

Example 196. Example of an array parameter in the where() method

```
// Build this query:
//   SELECT product_id, product_name, price
//   FROM "products"
//   WHERE (product_id IN (1, 2, 3))

$productIds = array(1, 2, 3);

$select = $db->select()
             ->from('products',
                    array('product_id', 'product_name', 'price'))
             ->where('product_id IN (?)', $productIds);
```

You can invoke the `where()` method multiple times on the same `Zend_Db_Select` object. The resulting query combines the multiple terms together using AND between them.

Example 197. Example of multiple where() methods

```
// Build this query:
//   SELECT product_id, product_name, price
//   FROM "products"
//   WHERE (price > 100.00)
//     AND (price < 500.00)

$minimumPrice = 100;
$maximumPrice = 500;

$select = $db->select()
             ->from('products',
                    array('product_id', 'product_name', 'price'))
             ->where('price > ?', $minimumPrice)
             ->where('price < ?', $maximumPrice);
```

If you need to combine terms together using OR, use the `orWhere()` method. This method is used in the same way as the `where()` method, except that the term specified is preceded by OR, instead of AND.

Example 198. Example of the orWhere() method

```
// Build this query:
//   SELECT product_id, product_name, price
//   FROM "products"
//   WHERE (price < 100.00)
//     OR (price > 500.00)

$minimumPrice = 100;
$maximumPrice = 500;

$select = $db->select()
             ->from('products',
                    array('product_id', 'product_name', 'price'))
             ->where('price < ?', $minimumPrice)
             ->orWhere('price > ?', $maximumPrice);
```

`Zend_Db_Select` automatically puts parentheses around each expression you specify using the `where()` or `orWhere()` methods. This helps to ensure that Boolean operator precedence does not cause unexpected results.

Example 199. Example of parenthesizing Boolean expressions

```
// Build this query:
//   SELECT product_id, product_name, price
//   FROM "products"
//   WHERE (price < 100.00 OR price > 500.00)
//     AND (product_name = 'Apple')

$minimumPrice = 100;
$maximumPrice = 500;
$prod = 'Apple';

$select = $db->select()
             ->from('products',
                    array('product_id', 'product_name', 'price'))
             ->where("price < $minimumPrice OR price > $maximumPrice")
             ->where('product_name = ?', $prod);
```

In the example above, the results would be quite different without the parentheses, because AND has higher precedence than OR. `Zend_Db_Select` applies the parentheses so the effect is that each expression in successive calls to the `where()` bind more tightly than the AND that combines the expressions.

4.3.7. Adding a GROUP BY Clause

In SQL, the **GROUP BY** clause allows you to reduce the rows of a query result set to one row per unique value found in the column(s) named in the **GROUP BY** clause.

In `Zend_Db_Select`, you can specify the column(s) to use for calculating the groups of rows using the `group()` method. The argument to this method is a column or an array of columns to use in the **GROUP BY** clause.

Example 200. Example of the group() method

```
// Build this query:
//   SELECT p."product_id", COUNT(*) AS line_items_per_product
//   FROM "products" AS p JOIN "line_items" AS l
//     ON p.product_id = l.product_id
//   GROUP BY p.product_id

$select = $db->select()
             ->from(array('p' => 'products'),
                    array('product_id'))
             ->join(array('l' => 'line_items'),
                    'p.product_id = l.product_id',
                    array('line_items_per_product' => 'COUNT(*)'))
             ->group('p.product_id');
```

Like the columns array in the `from()` method, you can use correlation names in the column name strings, and the column is quoted as an identifier unless the string contains parentheses or is an object of type `Zend_Db_Expr`.

4.3.8. Adding a HAVING Clause

In SQL, the `HAVING` clause applies a restriction condition on groups of rows. This is similar to how a `WHERE` clause applies a restriction condition on rows. But the two clauses are different because `WHERE` conditions are applied before groups are defined, whereas `HAVING` conditions are applied after groups are defined.

In `Zend_Db_Select`, you can specify conditions for restricting groups using the `having()` method. Its usage is similar to that of the `where()` method. The first argument is a string containing a SQL expression. The optional second argument is a value that is used to replace a positional parameter placeholder in the SQL expression. Expressions given in multiple invocations of the `having()` method are combined using the Boolean `AND` operator, or the `OR` operator if you use the `orHaving()` method.

Example 201. Example of the having() method

```
// Build this query:
//   SELECT p."product_id", COUNT(*) AS line_items_per_product
//   FROM "products" AS p JOIN "line_items" AS l
//     ON p.product_id = l.product_id
//   GROUP BY p.product_id
//   HAVING line_items_per_product > 10

$select = $db->select()
             ->from(array('p' => 'products'),
                    array('product_id'))
             ->join(array('l' => 'line_items'),
                    'p.product_id = l.product_id',
                    array('line_items_per_product' => 'COUNT(*)'))
             ->group('p.product_id')
             ->having('line_items_per_product > 10');
```

No quoting is applied to expressions given to the `having()` or `orHaving()` methods. If you have column names that need to be quoted, you must use `quoteIdentifier()` as you form the string for the condition.

4.3.9. Adding an ORDER BY Clause

In SQL, the `ORDER BY` clause specifies one or more columns or expressions by which the result set of a query is sorted. If multiple columns are listed, the secondary columns are used to resolve ties; the sort order is determined by the secondary columns if the preceding columns contain identical values. The default sorting is from least value to greatest value. You can also sort by greatest value to least value for a given column in the list by specifying the keyword `DESC` after that column.

In `Zend_Db_Select`, you can use the `order()` method to specify a column or an array of columns by which to sort. Each element of the array is a string naming a column. Optionally with the `ASC DESC` keyword following it, separated by a space.

Like in the `from()` and `group()` methods, column names are quoted as identifiers, unless they contain parentheses or are an object of type `Zend_Db_Expr`.

Example 202. Example of the order() method

```
// Build this query:
//   SELECT p."product_id", COUNT(*) AS line_items_per_product
//   FROM "products" AS p JOIN "line_items" AS l
//     ON p.product_id = l.product_id
//   GROUP BY p.product_id
//   ORDER BY "line_items_per_product" DESC, "product_id"

$select = $db->select()
             ->from(array('p' => 'products'),
                    array('product_id'))
             ->join(array('l' => 'line_items'),
                    'p.product_id = l.product_id',
                    array('line_items_per_product' => 'COUNT(*)'))
             ->group('p.product_id')
             ->order(array('line_items_per_product DESC',
                           'product_id'));
```

4.3.10. Adding a LIMIT Clause

Some RDBMS brands extend SQL with a query clause known as the `LIMIT` clause. This clause reduces the number of rows in the result set to at most a number you specify. You can also specify to skip a number of rows before starting to output. This feature makes it easy to take a subset of a result set, for example when displaying query results on progressive pages of output.

In `Zend_Db_Select`, you can use the `limit()` method to specify the count of rows and the number of rows to skip. The *first* argument to this method is the desired count of rows. The *second* argument is the number of rows to skip.

Example 203. Example of the limit() method

```
// Build this query:
//   SELECT p."product_id", p."product_name"
//   FROM "products" AS p
//   LIMIT 10, 20
// Equivalent to:
//   SELECT p."product_id", p."product_name"
//   FROM "products" AS p
//   LIMIT 20 OFFSET 10

$select = $db->select()
              ->from(array('p' => 'products'),
                     array('product_id', 'product_name'))
              ->limit(20, 10);
```

The `LIMIT` syntax is not supported by all RDBMS brands. Some RDBMS require different syntax to support similar functionality. Each `Zend_Db_Adapter_Abstract` class includes a method to produce SQL appropriate for that RDBMS.

Use the `limitPage()` method for an alternative way to specify row count and offset. This method allows you to limit the result set to one of a series of fixed-length subsets of rows from the query's total result set. In other words, you specify the length of a "page" of results, and the ordinal number of the single page of results you want the query to return. The page number is the first argument of the `limitPage()` method, and the page length is the second argument. Both arguments are required; they have no default values.

Example 204. Example of the limitPage() method

```
// Build this query:
//   SELECT p."product_id", p."product_name"
//   FROM "products" AS p
//   LIMIT 10, 20

$select = $db->select()
              ->from(array('p' => 'products'),
                     array('product_id', 'product_name'))
              ->limitPage(2, 10);
```

4.3.11. Adding the DISTINCT Query Modifier

The `distinct()` method enables you to add the `DISTINCT` keyword to your SQL query.

Example 205. Example of the distinct() method

```
// Build this query:
//   SELECT DISTINCT p."product_name"
//   FROM "products" AS p

$select = $db->select()
              ->distinct()
              ->from(array('p' => 'products'), 'product_name');
```

4.3.12. Adding the FOR UPDATE Query Modifier

The `forUpdate()` method enables you to add the `FOR UPDATE` modifier to your SQL query.

Example 206. Example of forUpdate() method

```
// Build this query:
//   SELECT FOR UPDATE p.*
//   FROM "products" AS p

$select = $db->select()
             ->forUpdate()
             ->from(array('p' => 'products'));
```

4.3.13. Building a UNION Query

You can build union queries with `Zend_Db_Select` by passing an array of `Zend_Db_Select` or SQL Query strings into the `union()` method. As second parameter you can pass the `Zend_Db_Select::SQL_UNION` or `Zend_Db_Select::SQL_UNION_ALL` constants to specify which type of union you want to perform.

Example 207. Example of union() method

```
$sql1 = $db->select();
$sql2 = "SELECT ...";

$select = $db->select()
    ->union(array($sql1, $sql2))
    ->order("id");
```

4.4. Executing Select Queries

This section describes how to execute the query represented by a `Zend_Db_Select` object.

4.4.1. Executing Select Queries from the Db Adapter

You can execute the query represented by the `Zend_Db_Select` object by passing it as the first argument to the `query()` method of a `Zend_Db_Adapter_Abstract` object. Use the `Zend_Db_Select` objects instead of a string query.

The `query()` method returns an object of type `Zend_Db_Statement` or PDOStatement, depending on the adapter type.

Example 208. Example using the Db adapter's query() method

```
$select = $db->select()
             ->from('products');

$stmt = $db->query($select);
$result = $stmt->fetchAll();
```

4.4.2. Executing Select Queries from the Object

As an alternative to using the `query()` method of the adapter object, you can use the `query()` method of the `Zend_Db_Select` object. Both methods return an object of type `Zend_Db_Statement` or PDOStatement, depending on the adapter type.

Example 209. Example using the Select object's query method

```
$select = $db->select()
             ->from('products');

$stmt = $select->query();
$result = $stmt->fetchAll();
```

4.4.3. Converting a Select Object to a SQL String

If you need access to a string representation of the SQL query corresponding to the `Zend_Db_Select` object, use the `__toString()` method.

Example 210. Example of the __toString() method

```
$select = $db->select()
             ->from('products');

$sql = $select->__toString();
echo "$sql\n";

// The output is the string:
//   SELECT * FROM "products"
```

4.5. Other methods

This section describes other methods of the `Zend_Db_Select` class that are not covered above: `getPart()` and `reset()`.

4.5.1. Retrieving Parts of the Select Object

The `getPart()` method returns a representation of one part of your SQL query. For example, you can use this method to return the array of expressions for the `WHERE` clause, or the array of columns (or column expressions) that are in the `SELECT` list, or the values of the count and offset for the `LIMIT` clause.

The return value is not a string containing a fragment of SQL syntax. The return value is an internal representation, which is typically an array structure containing values and expressions. Each part of the query has a different structure.

The single argument to the `getPart()` method is a string that identifies which part of the Select query to return. For example, the string `'from'` identifies the part of the Select object that stores information about the tables in the `FROM` clause, including joined tables.

The `Zend_Db_Select` class defines constants you can use for parts of the SQL query. You can use these constant definitions, or you can the literal strings.

Table 52. Constants used by getPart() and reset()

Constant	String value
`Zend_Db_Select::DISTINCT`	`'distinct'`
`Zend_Db_Select::FOR_UPDATE`	`'forupdate'`
`Zend_Db_Select::COLUMNS`	`'columns'`

Constant	String value
Zend_Db_Select::FROM	'from'
Zend_Db_Select::WHERE	'where'
Zend_Db_Select::GROUP	'group'
Zend_Db_Select::HAVING	'having'
Zend_Db_Select::ORDER	'order'
Zend_Db_Select::LIMIT_COUNT	'limitcount'
Zend_Db_Select::LIMIT_OFFSET	'limitoffset'

Example 211. Example of the getPart() method

```
$select = $db->select()
              ->from('products')
              ->order('product_id');

// You can use a string literal to specify the part
$orderData = $select->getPart( 'order' );

// You can use a constant to specify the same part
$orderData = $select->getPart( Zend_Db_Select::ORDER );

// The return value may be an array structure, not a string.
// Each part has a different structure.
print_r( $orderData );
```

4.5.2. Resetting Parts of the Select Object

The `reset()` method enables you to clear one specified part of the SQL query, or else clear all parts of the SQL query if you omit the argument.

The single argument is optional. You can specify the part of the query to clear, using the same strings you used in the argument to the `getPart()` method. The part of the query you specify is reset to a default state.

If you omit the parameter, `reset()` changes all parts of the query to their default state. This makes the `Zend_Db_Select` object equivalent to a new object, as though you had just instantiated it.

Example 212. Example of the reset() method

```
// Build this query:
//   SELECT p.*
//   FROM "products" AS p
//   ORDER BY "product_name"

$select = $db->select()
             ->from(array('p' => 'products')
             ->order('product_name');

// Changed requirement, instead order by a different columns:
//   SELECT p.*
//   FROM "products" AS p
//   ORDER BY "product_id"

// Clear one part so we can redefine it
$select->reset( Zend_Db_Select::ORDER );

// And specify a different column
$select->order('product_id');

// Clear all parts of the query
$select->reset();
```

5. Zend_Db_Table

5.1. Introduction

The `Zend_Db_Table` class is an object-oriented interface to database tables. It provides methods for many common operations on tables. The base class is extensible, so you can add custom logic.

The `Zend_Db_Table` solution is an implementation of the Table Data Gateway pattern. The solution also includes a class that implements the Row Data Gateway pattern.

5.2. Using Zend_Db_Table as a concrete class

As of Zend Framework 1.9, you can instantiate `Zend_Db_Table`. This added benefit is that you do not have to extend a base class and configure it to do simple operations such as selecting, inserting, updating and deleteing on a single table. below is an example of the simplest of use cases.

Example 213. Declaring a table class with just the string name

```
Zend_Db_Table::setDefaultAdapter($dbAdapter);
$bugTable = new Zend_Db_Table('bug');
```

The above example represents the simplest of use cases. Make not of all the options describe below for configuring `Zend_Db_Table` tables. If you want to be able to use the concrete usage case, in addition to the more complex relationhip features, see the `Zend_Db_Table_Definition` documentation.

5.3. Defining a Table Class

For each table in your database that you want to access, define a class that extends `Zend_Db_Table_Abstract`.

5.3.1. Defining the Table Name and Schema

Declare the database table for which this class is defined, using the protected variable $_name. This is a string, and must contain the name of the table spelled as it appears in the database.

Example 214. Declaring a table class with explicit table name

```
class Bugs extends Zend_Db_Table_Abstract
{
    protected $_name = 'bugs';
}
```

If you don't specify the table name, it defaults to the name of the class. If you rely on this default, the class name must match the spelling of the table name as it appears in the database.

Example 215. Declaring a table class with implicit table name

```
class bugs extends Zend_Db_Table_Abstract
{
    // table name matches class name
}
```

You can also declare the schema for the table, either with the protected variable $_schema, or with the schema prepended to the table name in the $_name property. Any schema specified with the $_name property takes precedence over a schema specified with the $_schema property. In some RDBMS brands, the term for schema is "database" or "tablespace," but it is used similarly.

Example 216. Declaring a table class with schema

```
// First alternative:
class Bugs extends Zend_Db_Table_Abstract
{
    protected $_schema = 'bug_db';
    protected $_name   = 'bugs';
}

// Second alternative:
class Bugs extends Zend_Db_Table_Abstract
{
    protected $_name = 'bug_db.bugs';
}

// If schemas are specified in both $_name and $_schema, the one
// specified in $_name takes precedence:

class Bugs extends Zend_Db_Table_Abstract
{
    protected $_name   = 'bug_db.bugs';
    protected $_schema = 'ignored';
}
```

The schema and table names may also be specified via constructor configuration directives, which override any default values specified with the $_name and $_schema properties. A schema specification given with the name directive overrides any value provided with the schema option.

Example 217. Declaring table and schema names upon instantiation

```
class Bugs extends Zend_Db_Table_Abstract
{
}

// First alternative:

$tableBugs = new Bugs(array('name' => 'bugs', 'schema' => 'bug_db'));

// Second alternative:

$tableBugs = new Bugs(array('name' => 'bug_db.bugs'));

// If schemas are specified in both 'name' and 'schema', the one
// specified in 'name' takes precedence:

$tableBugs = new Bugs(array('name' => 'bug_db.bugs',
                            'schema' => 'ignored'));
```

If you don't specify the schema name, it defaults to the schema to which your database adapter instance is connected.

5.3.2. Defining the Table Primary Key

Every table must have a primary key. You can declare the column for the primary key using the protected variable `$_primary`. This is either a string that names the single column for the primary key, or else it is an array of column names if your primary key is a compound key.

Example 218. Example of specifying the primary key

```
class Bugs extends Zend_Db_Table_Abstract
{
    protected $_name = 'bugs';
    protected $_primary = 'bug_id';
}
```

If you don't specify the primary key, `Zend_Db_Table_Abstract` tries to discover the primary key based on the information provided by the `describeTable()´` method.

Every table class must know which column(s) can be used to address rows uniquely. If no primary key column(s) are specified in the table class definition or the table constructor arguments, or discovered in the table metadata provided by `describeTable()`, then the table cannot be used with `Zend_Db_Table`.

5.3.3. Overriding Table Setup Methods

When you create an instance of a Table class, the constructor calls a set of protected methods that initialize metadata for the table. You can extend any of these methods to define metadata explicitly. Remember to call the method of the same name in the parent class at the end of your method.

Example 219. Example of overriding the _setupTableName() method

```
class Bugs extends Zend_Db_Table_Abstract
{
    protected function _setupTableName()
    {
        $this->_name = 'bugs';
        parent::_setupTableName();
    }
}
```

The setup methods you can override are the following:

- `_setupDatabaseAdapter()` checks that an adapter has been provided; gets a default adapter from the registry if needed. By overriding this method, you can set a database adapter from some other source.
- `_setupTableName()` defaults the table name to the name of the class. By overriding this method, you can set the table name before this default behavior runs.
- `_setupMetadata()` sets the schema if the table name contains the pattern "schema.table"; calls `describeTable()` to get metadata information; defaults the `$_cols` array to the columns reported by `describeTable()`. By overriding this method, you can specify the columns.
- `_setupPrimaryKey()` defaults the primary key columns to those reported by `describeTable()`; checks that the primary key columns are included in the `$_cols` array. By overriding this method, you can specify the primary key columns.

5.3.4. Table initialization

If application-specific logic needs to be initialized when a Table class is constructed, you can select to move your tasks to the `init()` method, which is called after all Table metadata has been processed. This is recommended over the `__construct` method if you do not need to alter the metadata in any programmatic way.

Example 220. Example usage of init() method

```
class Bugs extends Zend_Db_Table_Abstract
{
    protected $_observer;

    public function init()
    {
        $this->_observer = new MyObserverClass();
    }
}
```

5.4. Creating an Instance of a Table

Before you use a Table class, create an instance using its constructor. The constructor's argument is an array of options. The most important option to a Table constructor is the database adapter instance, representing a live connection to an RDBMS. There are three ways of specifying the database adapter to a Table class, and these three ways are described below:

5.4.1. Specifying a Database Adapter

The first way to provide a database adapter to a Table class is by passing it as an object of type `Zend_Db_Adapter_Abstract` in the options array, identified by the key `'db'`.

Example 221. Example of constructing a Table using an Adapter object

```
$db = Zend_Db::factory('PDO_MYSQL', $options);

$table = new Bugs(array('db' => $db));
```

5.4.2. Setting a Default Database Adapter

The second way to provide a database adapter to a Table class is by declaring an object of type `Zend_Db_Adapter_Abstract` to be a default database adapter for all subsequent instances of Tables in your application. You can do this with the static method `Zend_Db_Table_Abstract::setDefaultAdapter()`. The argument is an object of type `Zend_Db_Adapter_Abstract`.

Example 222. Example of constructing a Table using a the Default Adapter

```
$db = Zend_Db::factory('PDO_MYSQL', $options);
Zend_Db_Table_Abstract::setDefaultAdapter($db);

// Later...

$table = new Bugs();
```

It can be convenient to create the database adapter object in a central place of your application, such as the bootstrap, and then store it as the default adapter. This gives you a means to ensure that the adapter instance is the same throughout your application. However, setting a default adapter is limited to a single adapter instance.

5.4.3. Storing a Database Adapter in the Registry

The third way to provide a database adapter to a Table class is by passing a string in the options array, also identified by the `'db'` key. The string is used as a key to the static `Zend_Registry` instance, where the entry at that key is an object of type `Zend_Db_Adapter_Abstract`.

Example 223. Example of constructing a Table using a Registry key

```
$db = Zend_Db::factory('PDO_MYSQL', $options);
Zend_Registry::set('my_db', $db);

// Later...

$table = new Bugs(array('db' => 'my_db'));
```

Like setting the default adapter, this gives you the means to ensure that the same adapter instance is used throughout your application. Using the registry is more flexible, because you can store more than one adapter instance. A given adapter instance is specific to a certain RDBMS brand and database instance. If your application needs access to multiple databases or even multiple database brands, then you need to use multiple adapters.

5.5. Inserting Rows to a Table

You can use the Table object to insert rows into the database table on which the Table object is based. Use the `insert()` method of your Table object. The argument is an associative array, mapping column names to values.

Example 224. Example of inserting to a Table

```
$table = new Bugs();

$data = array(
    'created_on'      => '2007-03-22',
    'bug_description' => 'Something wrong',
    'bug_status'      => 'NEW'
);

$table->insert($data);
```

By default, the values in your data array are inserted as literal values, using parameters. If you need them to be treated as SQL expressions, you must make sure they are distinct from plain strings. Use an object of type `Zend_Db_Expr` to do this.

Example 225. Example of inserting expressions to a Table

```
$table = new Bugs();

$data = array(
    'created_on'      => new Zend_Db_Expr('CURDATE()'),
    'bug_description' => 'Something wrong',
    'bug_status'      => 'NEW'
);
```

In the examples of inserting rows above, it is assumed that the table has an auto-incrementing primary key. This is the default behavior of `Zend_Db_Table_Abstract`, but there are other types of primary keys as well. The following sections describe how to support different types of primary keys.

5.5.1. Using a Table with an Auto-incrementing Key

An auto-incrementing primary key generates a unique integer value for you if you omit the primary key column from your SQL `INSERT` statement.

In `Zend_Db_Table_Abstract`, if you define the protected variable `$_sequence` to be the Boolean value `TRUE`, then the class assumes that the table has an auto-incrementing primary key.

Example 226. Example of declaring a Table with auto-incrementing primary key

```
class Bugs extends Zend_Db_Table_Abstract
{
    protected $_name = 'bugs';

    // This is the default in the Zend_Db_Table_Abstract class;
    // you do not need to define this.
    protected $_sequence = true;
}
```

MySQL, Microsoft SQL Server, and SQLite are examples of RDBMS brands that support auto-incrementing primary keys.

PostgreSQL has a `SERIAL` notation that implicitly defines a sequence based on the table and column name, and uses the sequence to generate key values for new rows. IBM DB2 has an `IDENTITY` notation that works similarly. If you use either of these notations, treat your `Zend_Db_Table` class as having an auto-incrementing column with respect to declaring the `$_sequence` member as `TRUE`.

5.5.2. Using a Table with a Sequence

A sequence is a database object that generates a unique value, which can be used as a primary key value in one or more tables of the database.

If you define `$_sequence` to be a string, then `Zend_Db_Table_Abstract` assumes the string to name a sequence object in the database. The sequence is invoked to generate a new value, and this value is used in the `INSERT` operation.

Example 227. Example of declaring a Table with a sequence

```
class Bugs extends Zend_Db_Table_Abstract
{
    protected $_name = 'bugs';

    protected $_sequence = 'bug_sequence';
}
```

Oracle, PostgreSQL, and IBM DB2 are examples of RDBMS brands that support sequence objects in the database.

PostgreSQL and IBM DB2 also have syntax that defines sequences implicitly and associated with columns. If you use this notation, treat the table as having an auto-incrementing key column. Define the sequence name as a string only in cases where you would invoke the sequence explicitly to get the next key value.

5.5.3. Using a Table with a Natural Key

Some tables have a natural key. This means that the key is not automatically generated by the table or by a sequence. You must specify the value for the primary key in this case.

If you define the `$_sequence` to be the Boolean value `FALSE`, then `Zend_Db_Table_Abstract` assumes that the table has a natural primary key. You must provide values for the primary key columns in the array of data to the `insert()` method, or else this method throws a `Zend_Db_Table_Exception`.

Example 228. Example of declaring a Table with a natural key

```
class BugStatus extends Zend_Db_Table_Abstract
{
    protected $_name = 'bug_status';

    protected $_sequence = false;
}
```

All RDBMS brands support tables with natural keys. Examples of tables that are often declared as having natural keys are lookup tables, intersection tables in many-to-many relationships, or most tables with compound primary keys.

5.6. Updating Rows in a Table

You can update rows in a database table using the `update` method of a Table class. This method takes two arguments: an associative array of columns to change and new values to assign to these columns; and an SQL expression that is used in a `WHERE` clause, as criteria for the rows to change in the `UPDATE` operation.

Example 229. Example of updating rows in a Table

```
$table = new Bugs();

$data = array(
    'updated_on'      => '2007-03-23',
    'bug_status'      => 'FIXED'
);

$where = $table->getAdapter()->quoteInto('bug_id = ?', 1234);

$table->update($data, $where);
```

Since the table `update()` method proxies to the database adapter `update()` method, the second argument can be an array of SQL expressions. The expressions are combined as Boolean terms using an `AND` operator.

The values and identifiers in the SQL expression are not quoted for you. If you have values or identifiers that require quoting, you are responsible for doing this. Use the `quote()`, `quoteInto()`, and `quoteIdentifier()` methods of the database adapter.

5.7. Deleting Rows from a Table

You can delete rows from a database table using the `delete()` method. This method takes one argument, which is an SQL expression that is used in a `WHERE` clause, as criteria for the rows to delete.

Example 230. Example of deleting rows from a Table

```
$table = new Bugs();

$where = $table->getAdapter()->quoteInto('bug_id = ?', 1235);

$table->delete($where);
```

Since the table `delete()` method proxies to the database adapter `delete()` method, the argument can also be an array of SQL expressions. The expressions are combined as Boolean terms using an `AND` operator.

The values and identifiers in the SQL expression are not quoted for you. If you have values or identifiers that require quoting, you are responsible for doing this. Use the `quote()`, `quoteInto()`, and `quoteIdentifier()` methods of the database adapter.

5.8. Finding Rows by Primary Key

You can query the database table for rows matching specific values in the primary key, using the `find()` method. The first argument of this method is either a single value or an array of values to match against the primary key of the table.

Example 231. Example of finding rows by primary key values

```
$table = new Bugs();

// Find a single row
// Returns a Rowset
$rows = $table->find(1234);

// Find multiple rows
// Also returns a Rowset
$rows = $table->find(array(1234, 5678));
```

If you specify a single value, the method returns at most one row, because a primary key cannot have duplicate values and there is at most one row in the database table matching the value you specify. If you specify multiple values in an array, the method returns at most as many rows as the number of distinct values you specify.

The `find()` method might return fewer rows than the number of values you specify for the primary key, if some of the values don't match any rows in the database table. The method even may return zero rows. Because the number of rows returned is variable, the `find()` method returns an object of type `Zend_Db_Table_Rowset_Abstract`.

If the primary key is a compound key, that is, it consists of multiple columns, you can specify the additional columns as additional arguments to the `find()` method. You must provide as many arguments as the number of columns in the table's primary key.

To find multiple rows from a table with a compound primary key, provide an array for each of the arguments. All of these arrays must have the same number of elements. The values in each array are formed into tuples in order; for example, the first element in all the array arguments define the first compound primary key value, then the second elements of all the arrays define the second compound primary key value, and so on.

Example 232. Example of finding rows by compound primary key values

The call to `find()` below to match multiple rows can match two rows in the database. The first row must have primary key value (1234, 'ABC'), and the second row must have primary key value (5678, 'DEF').

```
class BugsProducts extends Zend_Db_Table_Abstract
{
    protected $_name = 'bugs_products';
    protected $_primary = array('bug_id', 'product_id');
}

$table = new BugsProducts();

// Find a single row with a compound primary key
// Returns a Rowset
$rows = $table->find(1234, 'ABC');

// Find multiple rows with compound primary keys
// Also returns a Rowset
$rows = $table->find(array(1234, 5678), array('ABC', 'DEF'));
```

5.9. Querying for a Set of Rows

5.9.1. Select API

The API for fetch operations has been superseded to allow a `Zend_Db_Table_Select` object to modify the query. However, the deprecated usage of the `fetchRow()` and `fetchAll()` methods will continue to work without modification.

The following statements are all legal and functionally identical, however it is recommended to update your code to take advantage of the new usage where possible.

```
/**
 * Fetching a rowset
 */
$rows = $table->fetchAll(
    'bug_status = "NEW"',
    'bug_id ASC',
    10,
    0
    );
$rows = $table->fetchAll(
    $table->select()
        ->where('bug_status = ?', 'NEW')
        ->order('bug_id ASC')
        ->limit(10, 0)
    );
// or with binding
$rows = $table->fetchAll(
    $table->select()
        ->where('bug_status = :status')
        ->bind(array(':status'=>'NEW')
        ->order('bug_id ASC')
        ->limit(10, 0)
```

```
    );

/**
 * Fetching a single row
 */
$row = $table->fetchRow(
    'bug_status = "NEW"',
    'bug_id ASC'
    );
$row = $table->fetchRow(
    $table->select()
        ->where('bug_status = ?', 'NEW')
        ->order('bug_id ASC')
    );
// or with binding
$row = $table->fetchRow(
    $table->select()
        ->where('bug_status = :status')
        ->bind(array(':status'=>'NEW')
        ->order('bug_id ASC')
    );
```

The `Zend_Db_Table_Select` object is an extension of the `Zend_Db_Select` object that applies specific restrictions to a query. The enhancements and restrictions are:

- You *can* elect to return a subset of columns within a fetchRow or fetchAll query. This can provide optimization benefits where returning a large set of results for all columns is not desirable.
- You *can* specify columns that evaluate expressions from within the selected table. However this will mean that the returned row or rowset will be readOnly and cannot be used for save() operations. A `Zend_Db_Table_Row` with readOnly status will throw an exception if a `save()` operation is attempted.
- You *can* allow JOIN clauses on a select to allow multi-table lookups.
- You *can not* specify columns from a JOINed tabled to be returned in a row/rowset. Doing so will trigger a PHP error. This was done to ensure the integrity of the `Zend_Db_Table` is retained. i.e. A `Zend_Db_Table_Row` should only reference columns derived from its parent table.

Example 233. Simple usage

```
$table = new Bugs();

$select = $table->select();
$select->where('bug_status = ?', 'NEW');

$rows = $table->fetchAll($select);
```

Fluent interfaces are implemented across the component, so this can be rewritten this in a more abbreviated form.

Example 234. Example of fluent interface

```
$table = new Bugs();

$rows =
    $table->fetchAll($table->select()->where('bug_status = ?', 'NEW'));
```

5.9.2. Fetching a rowset

You can query for a set of rows using any criteria other than the primary key values, using the `fetchAll()` method of the Table class. This method returns an object of type `Zend_Db_Table_Rowset_Abstract`.

Example 235. Example of finding rows by an expression

```
$table = new Bugs();

$select = $table->select()->where('bug_status = ?', 'NEW');

$rows = $table->fetchAll($select);
```

You may also pass sorting criteria in an `ORDER BY` clause, as well as count and offset integer values, used to make the query return a specific subset of rows. These values are used in a `LIMIT` clause, or in equivalent logic for RDBMS brands that do not support the `LIMIT` syntax.

Example 236. Example of finding rows by an expression

```
$table = new Bugs();

$order  = 'bug_id';

// Return the 21st through 30th rows
$count  = 10;
$offset = 20;

$select = $table->select()->where('bug_status = ?', 'NEW')
                          ->order($order)
                          ->limit($count, $offset);

$rows = $table->fetchAll($select);
```

All of the arguments above are optional. If you omit the ORDER clause, the result set includes rows from the table in an unpredictable order. If no LIMIT clause is set, you retrieve every row in the table that matches the WHERE clause.

5.9.3. Advanced usage

For more specific and optimized requests, you may wish to limit the number of columns returned in a row/rowset. This can be achieved by passing a FROM clause to the select object. The first argument in the FROM clause is identical to that of a `Zend_Db_Select` object with the addition of being able to pass an instance of `Zend_Db_Table_Abstract` and have it automatically determine the table name.

Example 237. Retrieving specific columns

```
$table = new Bugs();

$select = $table->select();
$select->from($table, array('bug_id', 'bug_description'))
       ->where('bug_status = ?', 'NEW');

$rows = $table->fetchAll($select);
```

The rowset contains rows that are still 'valid' - they simply contain a subset of the columns of a table. If a save() method is called on a partial row then only the fields available will be modified.

You can also specify expressions within a FROM clause and have these returned as a readOnly row/rowset. In this example we will return a rows from the bugs table that show an aggregate of the number of new bugs reported by individuals. Note the GROUP clause. The 'count' column will be made available to the row for evaluation and can be accessed as if it were part of the schema.

Example 238. Retrieving expressions as columns

```
$table = new Bugs();

$select = $table->select();
$select->from($table,
              array('COUNT(reported_by) as `count`', 'reported_by'))
       ->where('bug_status = ?', 'NEW')
       ->group('reported_by');

$rows = $table->fetchAll($select);
```

You can also use a lookup as part of your query to further refine your fetch operations. In this example the accounts table is queried as part of a search for all new bugs reported by 'Bob'.

Example 239. Using a lookup table to refine the results of fetchAll()

```
$table = new Bugs();

// retrieve with from part set, important when joining
$select = $table->select(Zend_Db_Table::SELECT_WITH_FROM_PART);
$select->setIntegrityCheck(false)
       ->where('bug_status = ?', 'NEW')
       ->join('accounts', 'accounts.account_name = bugs.reported_by')
       ->where('accounts.account_name = ?', 'Bob');

$rows = $table->fetchAll($select);
```

The `Zend_Db_Table_Select` is primarily used to constrain and validate so that it may enforce the criteria for a legal SELECT query. However there may be certain cases where you require the flexibility of the `Zend_Db_Table_Row` component and do not require a writable or deletable row. for this specific user case, it is possible to retrieve a row/rowset by passing a FALSE value to setIntegrityCheck. The resulting row/rowset will be returned as a 'locked' row (meaning the save(), delete() and any field-setting methods will throw an exception).

Example 240. Removing the integrity check on Zend_Db_Table_Select to allow JOINed rows

```
$table = new Bugs();

$select = $table->select(Zend_Db_Table::SELECT_WITH_FROM_PART)
                ->setIntegrityCheck(false);
$select->where('bug_status = ?', 'NEW')
       ->join('accounts',
              'accounts.account_name = bugs.reported_by',
              'account_name')
       ->where('accounts.account_name = ?', 'Bob');

$rows = $table->fetchAll($select);
```

5.10. Querying for a Single Row

You can query for a single row using criteria similar to that of the `fetchAll()` method.

Example 241. Example of finding a single row by an expression

```
$table = new Bugs();

$select  = $table->select()->where('bug_status = ?', 'NEW')
                           ->order('bug_id');

$row = $table->fetchRow($select);
```

This method returns an object of type `Zend_Db_Table_Row_Abstract`. If the search criteria you specified match no rows in the database table, then `fetchRow()` returns PHP's `NULL` value.

5.11. Retrieving Table Metadata Information

The `Zend_Db_Table_Abstract` class provides some information about its metadata. The `info()` method returns an array structure with information about the table, its columns and primary key, and other metadata.

Example 242. Example of getting the table name

```
$table = new Bugs();

$info = $table->info();

echo "The table name is " . $info['name'] . "\n";
```

The keys of the array returned by the `info()` method are described below:

- *name* => the name of the table.
- *cols* => an array, naming the column(s) of the table.
- *primary* => an array, naming the column(s) in the primary key.
- *metadata* => an associative array, mapping column names to information about the columns. This is the information returned by the `describeTable()` method.

- *rowClass* => the name of the concrete class used for Row objects returned by methods of this table instance. This defaults to `Zend_Db_Table_Row`.
- *rowsetClass* => the name of the concrete class used for Rowset objects returned by methods of this table instance. This defaults to `Zend_Db_Table_Rowset`.
- *referenceMap* => an associative array, with information about references from this table to any parent tables. See Section 8.2, “Defining Relationships”.
- *dependentTables* => an array of class names of tables that reference this table. See Section 8.2, “Defining Relationships”.
- *schema* => the name of the schema (or database or tablespace) for this table.

5.12. Caching Table Metadata

By default, `Zend_Db_Table_Abstract` queries the underlying database for table metadata whenever that data is needed to perform table operations. The table object fetches the table metadata from the database using the adapter's `describeTable()` method. Operations requiring this introspection include:

- `insert()`
- `find()`
- `info()`

In some circumstances, particularly when many table objects are instantiated against the same database table, querying the database for the table metadata for each instance may be undesirable from a performance standpoint. In such cases, users may benefit by caching the table metadata retrieved from the database.

There are two primary ways in which a user may take advantage of table metadata caching:

- *Call `Zend_Db_Table_Abstract::setDefaultMetadataCache()`* - This allows a developer to once set the default cache object to be used for all table classes.
- *Configure `Zend_Db_Table_Abstract::__construct()`* - This allows a developer to set the cache object to be used for a particular table class instance.

In both cases, the cache specification must be either `NULL` (i.e., no cache used) or an instance of `Zend_Cache_Core`. The methods may be used in conjunction when it is desirable to have both a default metadata cache and the ability to change the cache for individual table objects.

Example 243. Using a Default Metadata Cache for all Table Objects

The following code demonstrates how to set a default metadata cache to be used for all table objects:

```
// First, set up the Cache
$frontendOptions = array(
    'automatic_serialization' => true
    );

$backendOptions  = array(
    'cache_dir'               => 'cacheDir'
    );

$cache = Zend_Cache::factory('Core',
                             'File',
                             $frontendOptions,
                             $backendOptions);

// Next, set the cache to be used with all table objects
Zend_Db_Table_Abstract::setDefaultMetadataCache($cache);

// A table class is also needed
class Bugs extends Zend_Db_Table_Abstract
{
    // ...
}

// Each instance of Bugs now uses the default metadata cache
$bugs = new Bugs();
```

Example 244. Using a Metadata Cache for a Specific Table Object

The following code demonstrates how to set a metadata cache for a specific table object instance:

```
// First, set up the Cache
$frontendOptions = array(
    'automatic_serialization' => true
    );

$backendOptions  = array(
    'cache_dir'               => 'cacheDir'
    );

$cache = Zend_Cache::factory('Core',
                             'File',
                             $frontendOptions,
                             $backendOptions);

// A table class is also needed
class Bugs extends Zend_Db_Table_Abstract
{
    // ...
}

// Configure an instance upon instantiation
$bugs = new Bugs(array('metadataCache' => $cache));
```

Automatic Serialization with the Cache Frontend

Since the information returned from the adapter's describeTable() method is an array, ensure that the `automatic_serialization` option is set to `TRUE` for the `Zend_Cache_Core` frontend.

Though the above examples use `Zend_Cache_Backend_File`, developers may use whatever cache backend is appropriate for the situation. Please see Zend_Cache for more information.

5.12.1. Hardcoding Table Metadata

To take metadata caching a step further, you can also choose to hardcode metadata. In this particular case, however, any changes to the table schema will require a change in your code. As such, it is only recommended for those who are optimizing for production usage.

The metadata structure is as follows:

```
protected $_metadata = array(
    '<column_name>' => array(
        'SCHEMA_NAME'      => <string>,
        'TABLE_NAME'       => <string>,
        'COLUMN_NAME'      => <string>,
        'COLUMN_POSITION'  => <int>,
        'DATA_TYPE'        => <string>,
        'DEFAULT'          => NULL|<value>,
        'NULLABLE'         => <bool>,
        'LENGTH'           => <string - length>,
        'SCALE'            => NULL|<value>,
        'PRECISION'        => NULL|<value>,
        'UNSIGNED'         => NULL|<bool>,
        'PRIMARY'          => <bool>,
        'PRIMARY_POSITION' => <int>,
        'IDENTITY'         => <bool>,
    ),
    // additional columns...
);
```

An easy way to get the appropriate values is to use the metadata cache, and then to deserialize values stored in the cache.

You can disable this optimization by turning of the `metadataCacheInClass` flag:

```
// At instantiation:
$bugs = new Bugs(array('metadataCacheInClass' => false));

// Or later:
$bugs->setMetadataCacheInClass(false);
```

The flag is enabled by default, which ensures that the `$_metadata` array is only populated once per instance.

5.13. Customizing and Extending a Table Class

5.13.1. Using Custom Row or Rowset Classes

By default, methods of the Table class return a Rowset in instances of the concrete class `Zend_Db_Table_Rowset`, and Rowsets contain a collection of instances of the concrete

class `Zend_Db_Table_Row` You can specify an alternative class to use for either of these, but they must be classes that extend `Zend_Db_Table_Rowset_Abstract` and `Zend_Db_Table_Row_Abstract`, respectively.

You can specify Row and Rowset classes using the Table constructor's options array, in keys `'rowClass'` and `'rowsetClass'` respectively. Specify the names of the classes using strings.

Example 245. Example of specifying the Row and Rowset classes

```
class My_Row extends Zend_Db_Table_Row_Abstract
{
    ...
}

class My_Rowset extends Zend_Db_Table_Rowset_Abstract
{
    ...
}

$table = new Bugs(
    array(
        'rowClass'    => 'My_Row',
        'rowsetClass' => 'My_Rowset'
    )
);

$where = $table->getAdapter()->quoteInto('bug_status = ?', 'NEW')

// Returns an object of type My_Rowset,
// containing an array of objects of type My_Row.
$rows = $table->fetchAll($where);
```

You can change the classes by specifying them with the `setRowClass()` and `setRowsetClass()` methods. This applies to rows and rowsets created subsequently; it does not change the class of any row or rowset objects you have created previously.

Example 246. Example of changing the Row and Rowset classes

```
$table = new Bugs();

$where = $table->getAdapter()->quoteInto('bug_status = ?', 'NEW')

// Returns an object of type Zend_Db_Table_Rowset
// containing an array of objects of type Zend_Db_Table_Row.
$rowsStandard = $table->fetchAll($where);

$table->setRowClass('My_Row');
$table->setRowsetClass('My_Rowset');

// Returns an object of type My_Rowset,
// containing an array of objects of type My_Row.
$rowsCustom = $table->fetchAll($where);

// The $rowsStandard object still exists, and it is unchanged.
```

For more information on the Row and Rowset classes, see Section 6, “Zend_Db_Table_Row” and Section 7, “Zend_Db_Table_Rowset”.

5.13.2. Defining Custom Logic for Insert, Update, and Delete

You can override the `insert()` and `update()` methods in your Table class. This gives you the opportunity to implement custom code that is executed before performing the database operation. Be sure to call the parent class method when you are done.

Example 247. Custom logic to manage timestamps

```
class Bugs extends Zend_Db_Table_Abstract
{
    protected $_name = 'bugs';

    public function insert(array $data)
    {
        // add a timestamp
        if (empty($data['created_on'])) {
            $data['created_on'] = time();
        }
        return parent::insert($data);
    }

    public function update(array $data, $where)
    {
        // add a timestamp
        if (empty($data['updated_on'])) {
            $data['updated_on'] = time();
        }
        return parent::update($data, $where);
    }
}
```

You can also override the `delete()` method.

5.13.3. Define Custom Search Methods in Zend_Db_Table

You can implement custom query methods in your Table class, if you have frequent need to do queries against this table with specific criteria. Most queries can be written using `fetchAll()`, but this requires that you duplicate code to form the query conditions if you need to run the query in several places in your application. Therefore it can be convenient to implement a method in the Table class to perform frequently-used queries against this table.

Example 248. Custom method to find bugs by status

```
class Bugs extends Zend_Db_Table_Abstract
{
    protected $_name = 'bugs';

    public function findByStatus($status)
    {
        $where = $this->getAdapter()->quoteInto('bug_status = ?', $status);
        return $this->fetchAll($where, 'bug_id');
    }
}
```

5.13.4. Define Inflection in Zend_Db_Table

Some people prefer that the table class name match a table name in the RDBMS by using a string transformation called *inflection*.

For example, if your table class name is "BugsProducts", it would match the physical table in the database called "bugs_products," if you omit the explicit declaration of the $_name class property. In this inflection mapping, the class name spelled in "CamelCase" format would be transformed to lower case, and words are separated with an underscore.

You can specify the database table name independently from the class name by declaring the table name with the $_name class property in each of your table classes.

Zend_Db_Table_Abstract performs no inflection to map the class name to the table name. If you omit the declaration of $_name in your table class, the class maps to a database table that matches the spelling of the class name exactly.

It is inappropriate to transform identifiers from the database, because this can lead to ambiguity or make some identifiers inaccessible. Using the SQL identifiers exactly as they appear in the database makes Zend_Db_Table_Abstract both simpler and more flexible.

If you prefer to use inflection, then you must implement the transformation yourself, by overriding the _setupTableName() method in your Table classes. One way to do this is to define an abstract class that extends Zend_Db_Table_Abstract, and then the rest of your tables extend your new abstract class.

Example 249. Example of an abstract table class that implements inflection

```
abstract class MyAbstractTable extends Zend_Db_Table_Abstract
{
    protected function _setupTableName()
    {
        if (!$this->_name) {
            $this->_name = myCustomInflector(get_class($this));
        }
        parent::_setupTableName();
    }
}

class BugsProducts extends MyAbstractTable
{
}
```

You are responsible for writing the functions to perform inflection transformation. Zend Framework does not provide such a function.

6. Zend_Db_Table_Row

6.1. Introduction

Zend_Db_Table_Row is a class that contains an individual row of a Zend_Db_Table object. When you run a query against a Table class, the result is returned in a set of Zend_Db_Table_Row objects. You can also use this object to create new rows and add them to the database table.

Zend_Db_Table_Row is an implementation of the Row Data Gateway pattern.

6.2. Fetching a Row

Zend_Db_Table_Abstract provides methods find() and fetchAll(), which each return an object of type Zend_Db_Table_Rowset, and the method fetchRow(), which returns an object of type Zend_Db_Table_Row.

Example 250. Example of fetching a row

```
$bugs = new Bugs();
$row = $bugs->fetchRow($bugs->select()->where('bug_id = ?', 1));
```

A `Zend_Db_Table_Rowset` object contains a collection of `Zend_Db_Table_Row` objects. See the chapter about table rowset for details.

Example 251. Example of reading a row in a rowset

```
$bugs = new Bugs();
$rowset = $bugs->fetchAll($bugs->select()->where('bug_status = ?', 1));
$row = $rowset->current();
```

6.2.1. Reading column values from a row

`Zend_Db_Table_Row_Abstract` provides accessor methods so you can reference columns in the row as object properties.

Example 252. Example of reading a column in a row

```
$bugs = new Bugs();
$row = $bugs->fetchRow($bugs->select()->where('bug_id = ?', 1));

// Echo the value of the bug_description column
echo $row->bug_description;
```

Earlier versions of `Zend_Db_Table_Row` mapped these column accessors to the database column names using a string transformation called *inflection*.

Currently, `Zend_Db_Table_Row` does not implement inflection. Accessed property names need to match the spelling of the column names as they appear in your database.

6.2.2. Retrieving Row Data as an Array

You can access the row's data as an array using the `toArray()` method of the Row object. This returns an associative array of the column names to the column values.

Example 253. Example of using the toArray() method

```
$bugs = new Bugs();
$row = $bugs->fetchRow($bugs->select()->where('bug_id = ?', 1));

// Get the column/value associative array from the Row object
$rowArray = $row->toArray();

// Now use it as a normal array
foreach ($rowArray as $column => $value) {
    echo "Column: $column\n";
    echo "Value:  $value\n";
}
```

The array returned from `toArray()` is not updateable. You can modify values in the array as you can with any array, but you cannot save changes to this array to the database directly.

6.2.3. Fetching data from related tables

The `Zend_Db_Table_Row_Abstract` class provides methods for fetching rows and rowsets from related tables. See the relationship chapter for more information on table relationships.

6.3. Writing rows to the database

6.3.1. Changing column values in a row

You can set individual column values using column accessors, similar to how the columns are read as object properties in the example above.

Using a column accessor to set a value changes the column value of the row object in your application, but it does not commit the change to the database yet. You can do that with the `save()` method.

Example 254. Example of changing a column in a row

```
$bugs = new Bugs();
$row = $bugs->fetchRow($bugs->select()->where('bug_id = ?', 1));

// Change the value of one or more columns
$row->bug_status = 'FIXED';

// UPDATE the row in the database with new values
$row->save();
```

6.3.2. Inserting a new row

You can create a new row for a given table with the `createRow()` method of the table class. You can access fields of this row with the object-oriented interface, but the row is not stored in the database until you call the `save()` method.

Example 255. Example of creating a new row for a table

```
$bugs = new Bugs();
$newRow = $bugs->createRow();

// Set column values as appropriate for your application
$newRow->bug_description = '...description...';
$newRow->bug_status = 'NEW';

// INSERT the new row to the database
$newRow->save();
```

The optional argument to the `createRow()` method is an associative array, with which you can populate fields of the new row.

Example 256. Example of populating a new row for a table

```
$data = array(
    'bug_description' => '...description...',
    'bug_status'      => 'NEW'
);

$bugs = new Bugs();
$newRow = $bugs->createRow($data);

// INSERT the new row to the database
$newRow->save();
```

The `createRow()` method was called `fetchNew()` in earlier releases of `Zend_Db_Table`. You are encouraged to use the new method name, even though the old name continues to work for the sake of backward compatibility.

6.3.3. Changing values in multiple columns

`Zend_Db_Table_Row_Abstract` provides the `setFromArray()` method to enable you to set several columns in a single row at once, specified in an associative array that maps the column names to values. You may find this method convenient for setting values both for new rows and for rows you need to update.

Example 257. Example of using setFromArray() to set values in a new Row

```
$bugs = new Bugs();
$newRow = $bugs->createRow();

// Data are arranged in an associative array
$data = array(
    'bug_description' => '...description...',
    'bug_status'      => 'NEW'
);

// Set all the column values at once
$newRow->setFromArray($data);

// INSERT the new row to the database
$newRow->save();
```

6.3.4. Deleting a row

You can call the `delete()` method on a Row object. This deletes rows in the database matching the primary key in the Row object.

Example 258. Example of deleting a row

```
$bugs = new Bugs();
$row = $bugs->fetchRow('bug_id = 1');

// DELETE this row
$row->delete();
```

You do not have to call `save()` to apply the delete; it is executed against the database immediately.

6.4. Serializing and unserializing rows

It is often convenient to save the contents of a database row to be used later. *Serialization* is the name for the operation that converts an object into a form that is easy to save in offline storage (for example, a file). Objects of type `Zend_Db_Table_Row_Abstract` are serializable.

6.4.1. Serializing a Row

Simply use PHP's `serialize()` function to create a string containing a byte-stream representation of the Row object argument.

Example 259. Example of serializing a row

```
$bugs = new Bugs();
$row = $bugs->fetchRow('bug_id = 1');

// Convert object to serialized form
$serializedRow = serialize($row);

// Now you can write $serializedRow to a file, etc.
```

6.4.2. Unserializing Row Data

Use PHP's `unserialize()` function to restore a string containing a byte-stream representation of an object. The function returns the original object.

Note that the Row object returned is in a *disconnected* state. You can read the Row object and its properties, but you cannot change values in the Row or execute other methods that require a database connection (for example, queries against related tables).

Example 260. Example of unserializing a serialized row

```
$rowClone = unserialize($serializedRow);

// Now you can use object properties, but read-only
echo $rowClone->bug_description;
```

Why do Rows unserialize in a disconnected state?

A serialized object is a string that is readable to anyone who possesses it. It could be a security risk to store parameters such as database account and password in plain, unencrypted text in the serialized string. You would not want to store such data to a text file that is not protected, or send it in an email or other medium that is easily read by potential attackers. The reader of the serialized object should not be able to use it to gain access to your database without knowing valid credentials.

6.4.3. Reactivating a Row as Live Data

You can reactivate a disconnected Row, using the `setTable()` method. The argument to this method is a valid object of type `Zend_Db_Table_Abstract`, which you create. Creating a Table object requires a live connection to the database, so by reassociating the Table with the Row, the Row gains access to the database. Subsequently, you can change values in the Row object and save the changes to the database.

Example 261. Example of reactivating a row

```
$rowClone = unserialize($serializedRow);

$bugs = new Bugs();

// Reconnect the row to a table, and
// thus to a live database connection
$rowClone->setTable($bugs);

// Now you can make changes to the row and save them
$rowClone->bug_status = 'FIXED';
$rowClone->save();
```

6.5. Extending the Row class

`Zend_Db_Table_Row` is the default concrete class that extends `Zend_Db_Table_Row_Abstract`. You can define your own concrete class for instances of Row by extending `Zend_Db_Table_Row_Abstract`. To use your new Row class to store results of Table queries, specify the custom Row class by name either in the `$_rowClass` protected member of a Table class, or in the array argument of the constructor of a Table object.

Example 262. Specifying a custom Row class

```
class MyRow extends Zend_Db_Table_Row_Abstract
{
    // ...customizations
}

// Specify a custom Row to be used by default
// in all instances of a Table class.
class Products extends Zend_Db_Table_Abstract
{
    protected $_name = 'products';
    protected $_rowClass = 'MyRow';
}

// Or specify a custom Row to be used in one
// instance of a Table class.
$bugs = new Bugs(array('rowClass' => 'MyRow'));
```

6.5.1. Row initialization

If application-specific logic needs to be initialized when a row is constructed, you can select to move your tasks to the `init()` method, which is called after all row metadata has been processed. This is recommended over the `__construct()` method if you do not need to alter the metadata in any programmatic way.

Example 263. Example usage of init() method

```
class MyApplicationRow extends Zend_Db_Table_Row_Abstract
{
    protected $_role;

    public function init()
    {
        $this->_role = new MyRoleClass();
    }
}
```

6.5.2. Defining Custom Logic for Insert, Update, and Delete in Zend_Db_Table_Row

The Row class calls protected methods `_insert()`, `_update()`, and `_delete()` before performing the corresponding operations `INSERT`, `UPDATE`, and `DELETE`. You can add logic to these methods in your custom Row subclass.

If you need to do custom logic in a specific table, and the custom logic must occur for every operation on that table, it may make more sense to implement your custom code in the `insert()`, `update()` and `delete()` methods of your Table class. However, sometimes it may be necessary to do custom logic in the Row class.

Below are some example cases where it might make sense to implement custom logic in a Row class instead of in the Table class:

Example 264. Example of custom logic in a Row class

The custom logic may not apply in all cases of operations on the respective Table. You can provide custom logic on demand by implementing it in a Row class and creating an instance of the Table class with that custom Row class specified. Otherwise, the Table uses the default Row class.

You need data operations on this table to record the operation to a `Zend_Log` object, but only if the application configuration has enabled this behavior.

```
class MyLoggingRow extends Zend_Db_Table_Row_Abstract
{
    protected function _insert()
    {
        $log = Zend_Registry::get('database_log');
        $log->info(Zend_Debug::dump($this->_data,
                                    "INSERT: $this->_tableClass",
                                    false)
                   );
    }
}

// $loggingEnabled is an example property that depends
// on your application configuration
if ($loggingEnabled) {
    $bugs = new Bugs(array('rowClass' => 'MyLoggingRow'));
} else {
    $bugs = new Bugs();
}
```

Example 265. Example of a Row class that logs insert data for multiple tables

The custom logic may be common to multiple tables. Instead of implementing the same custom logic in every one of your Table classes, you can implement the code for such actions in the definition of a Row class, and use this Row in each of your Table classes.

In this example, the logging code is identical in all table classes.

```
class MyLoggingRow extends Zend_Db_Table_Row_Abstract
{
    protected function _insert()
    {
        $log = Zend_Registry::get('database_log');
        $log->info(Zend_Debug::dump($this->_data,
                                    "INSERT: $this->_tableClass",
                                    false)
                   );
    }
}

class Bugs extends Zend_Db_Table_Abstract
{
    protected $_name = 'bugs';
    protected $_rowClass = 'MyLoggingRow';
}

class Products extends Zend_Db_Table_Abstract
{
    protected $_name = 'products';
    protected $_rowClass = 'MyLoggingRow';
}
```

6.5.3. Define Inflection in Zend_Db_Table_Row

Some people prefer that the table class name match a table name in the RDBMS by using a string transformation called *inflection*.

`Zend_Db` classes do not implement inflection by default. See the chapter about extending inflection for an explanation of this policy.

If you prefer to use inflection, then you must implement the transformation yourself, by overriding the `_transformColumn()` method in a custom Row class, and using that custom Row class when you perform queries against your Table class.

Example 266. Example of defining an inflection transformation

This allows you to use an inflected version of the column name in the accessors. The Row class uses the `_transformColumn()` method to change the name you use to the native column name in the database table.

```
class MyInflectedRow extends Zend_Db_Table_Row_Abstract
{
    protected function _transformColumn($columnName)
    {
        $nativeColumnName = myCustomInflector($columnName);
        return $nativeColumnName;
    }
}

class Bugs extends Zend_Db_Table_Abstract
{
    protected $_name = 'bugs';
    protected $_rowClass = 'MyInflectedRow';
}

$bugs = new Bugs();
$row = $bugs->fetchNew();

// Use camelcase column names, and rely on the
// transformation function to change it into the
// native representation.
$row->bugDescription = 'New description';
```

You are responsible for writing the functions to perform inflection transformation. Zend Framework does not provide such a function.

7. Zend_Db_Table_Rowset

7.1. Introduction

When you run a query against a Table class using the `find()` or `fetchAll()` methods, the result is returned in an object of type `Zend_Db_Table_Rowset_Abstract`. A Rowset contains a collection of objects descending from `Zend_Db_Table_Row_Abstract`. You can iterate through the Rowset and access individual Row objects, reading or modifying data in the Rows.

7.2. Fetching a Rowset

`Zend_Db_Table_Abstract` provides methods `find()` and `fetchAll()`, each of which returns an object of type `Zend_Db_Table_Rowset_Abstract`.

Example 267. Example of fetching a rowset

```
$bugs   = new Bugs();
$rowset = $bugs->fetchAll("bug_status = 'NEW'");
```

7.3. Retrieving Rows from a Rowset

The Rowset itself is usually less interesting than the Rows that it contains. This section illustrates how to get the Rows that comprise the Rowset.

A legitimate query returns zero rows when no rows in the database match the query conditions. Therefore, a Rowset object might contain zero Row objects. Since `Zend_Db_Table_Rowset_Abstract` implements the `Countable` interface, you can use `count()` to determine the number of Rows in the Rowset.

Example 268. Counting the Rows in a Rowset

```
$rowset   = $bugs->fetchAll("bug_status = 'FIXED'");

$rowCount = count($rowset);

if ($rowCount > 0) {
    echo "found $rowCount rows";
} else {
    echo 'no rows matched the query';
}
```

Example 269. Reading a Single Row from a Rowset

The simplest way to access a Row from a Rowset is to use the `current()` method. This is particularly appropriate when the Rowset contains exactly one Row.

```
$bugs   = new Bugs();
$rowset = $bugs->fetchAll("bug_id = 1");
$row    = $rowset->current();
```

If the Rowset contains zero rows, `current()` returns PHP's `NULL` value.

Example 270. Iterating through a Rowset

Objects descending from `Zend_Db_Table_Rowset_Abstract` implement the `SeekableIterator` interface, which means you can loop through them using the `foreach()` construct. Each value you retrieve this way is a `Zend_Db_Table_Row_Abstract` object that corresponds to one record from the table.

```
$bugs = new Bugs();

// fetch all records from the table
$rowset = $bugs->fetchAll();

foreach ($rowset as $row) {

    // output 'Zend_Db_Table_Row' or similar
    echo get_class($row) . "\n";

    // read a column in the row
    $status = $row->bug_status;

    // modify a column in the current row
    $row->assigned_to = 'mmouse';

    // write the change to the database
    $row->save();
}
```

Example 271. Seeking to a known position into a Rowset

`SeekableIterator` allows you to seek to a position that you would like the iterator to jump to. Simply use the `seek()` method for that. Pass it an integer representing the number of the Row you would like your Rowset to point to next, don't forget that it starts with index 0. If the index is wrong, ie doesn't exist, an exception will be thrown. You should use `count()` to check the number of results before seeking to a position.

```
$bugs = new Bugs();

// fetch all records from the table
$rowset = $bugs->fetchAll();

// takes the iterator to the 9th element (zero is one element) :
$rowset->seek(8);

// retrieve it
$row9 = $rowset->current();

// and use it
$row9->assigned_to = 'mmouse';
$row9->save();
```

`getRow()` allows you to get a specific row in the Rowset, knowing its position; don't forget however that positions start with index zero. The first parameter for `getRow()` is an integer for the position asked. The second optional parameter is a boolean; it tells the Rowset iterator if it must seek to that position in the same time, or not (default is `FALSE`). This method returns a `Zend_Db_Table_Row` object by default. If the position requested does not exist, an exception will be thrown. Here is an example:

```
$bugs = new Bugs();

// fetch all records from the table
$rowset = $bugs->fetchAll();

// retrieve the 9th element immediately:
$row9->getRow(8);

// and use it:
$row9->assigned_to = 'mmouse';
$row9->save();
```

After you have access to an individual Row object, you can manipulate the Row using methods described in Zend_Db_Table_Row.

7.4. Retrieving a Rowset as an Array

You can access all the data in the Rowset as an array using the `toArray()` method of the Rowset object. This returns an array containing one entry per Row. Each entry is an associative array having keys that correspond to column names and elements that correspond to the respective column values.

Example 272. Using toArray()

```
$bugs   = new Bugs();
$rowset = $bugs->fetchAll();

$rowsetArray = $rowset->toArray();

$rowCount = 1;
foreach ($rowsetArray as $rowArray) {
    echo "row #$rowCount:\n";
    foreach ($rowArray as $column => $value) {
        echo "\t$column => $value\n";
    }
    ++$rowCount;
    echo "\n";
}
```

The array returned from `toArray()` is not updateable. That is, you can modify values in the array as you can with any array, but changes to the array data are not propagated to the database.

7.5. Serializing and Unserializing a Rowset

Objects of type `Zend_Db_Table_Rowset_Abstract` are serializable. In a similar fashion to serializing an individual Row object, you can serialize a Rowset and unserialize it later.

Example 273. Serializing a Rowset

Simply use PHP's `serialize()` function to create a string containing a byte-stream representation of the Rowset object argument.

```
$bugs   = new Bugs();
$rowset = $bugs->fetchAll();

// Convert object to serialized form
$serializedRowset = serialize($rowset);

// Now you can write $serializedRowset to a file, etc.
```

Example 274. Unserializing a Serialized Rowset

Use PHP's `unserialize()` function to restore a string containing a byte-stream representation of an object. The function returns the original object.

Note that the Rowset object returned is in a *disconnected* state. You can iterate through the Rowset and read the Row objects and their properties, but you cannot change values in the Rows or execute other methods that require a database connection (for example, queries against related tables).

```
$rowsetDisconnected = unserialize($serializedRowset);

// Now you can use object methods and properties, but read-only
$row = $rowsetDisconnected->current();
echo $row->bug_description;
```

Why do Rowsets unserialize in a disconnected state?

A serialized object is a string that is readable to anyone who possesses it. It could be a security risk to store parameters such as database account and password in

plain, unencrypted text in the serialized string. You would not want to store such data to a text file that is not protected, or send it in an email or other medium that is easily read by potential attackers. The reader of the serialized object should not be able to use it to gain access to your database without knowing valid credentials.

You can reactivate a disconnected Rowset using the `setTable()` method. The argument to this method is a valid object of type `Zend_Db_Table_Abstract`, which you create. Creating a Table object requires a live connection to the database, so by reassociating the Table with the Rowset, the Rowset gains access to the database. Subsequently, you can change values in the Row objects contained in the Rowset and save the changes to the database.

Example 275. Reactivating a Rowset as Live Data

```
$rowset = unserialize($serializedRowset);

$bugs = new Bugs();

// Reconnect the rowset to a table, and
// thus to a live database connection
$rowset->setTable($bugs);

$row = $rowset->current();

// Now you can make changes to the row and save them
$row->bug_status = 'FIXED';
$row->save();
```

Reactivating a Rowset with `setTable()` also reactivates all the Row objects contained in that Rowset.

7.6. Extending the Rowset class

You can use an alternative concrete class for instances of Rowsets by extending `Zend_Db_Table_Rowset_Abstract`. Specify the custom Rowset class by name either in the `$_rowsetClass` protected member of a Table class, or in the array argument of the constructor of a Table object.

Example 276. Specifying a custom Rowset class

```
class MyRowset extends Zend_Db_Table_Rowset_Abstract
{
    // ...customizations
}

// Specify a custom Rowset to be used by default
// in all instances of a Table class.
class Products extends Zend_Db_Table_Abstract
{
    protected $_name = 'products';
    protected $_rowsetClass = 'MyRowset';
}

// Or specify a custom Rowset to be used in one
// instance of a Table class.
$bugs = new Bugs(array('rowsetClass' => 'MyRowset'));
```

Typically, the standard `Zend_Db_Rowset` concrete class is sufficient for most usage. However, you might find it useful to add new logic to a Rowset, specific to a given Table. For example, a new method could calculate an aggregate over all the Rows in the Rowset.

Example 277. Example of Rowset class with a new method

```
class MyBugsRowset extends Zend_Db_Table_Rowset_Abstract
{
    /**
     * Find the Row in the current Rowset with the
     * greatest value in its 'updated_at' column.
     */
    public function getLatestUpdatedRow()
    {
        $max_updated_at = 0;
        $latestRow = null;
        foreach ($this as $row) {
            if ($row->updated_at > $max_updated_at) {
                $latestRow = $row;
            }
        }
        return $latestRow;
    }
}

class Bugs extends Zend_Db_Table_Abstract
{
    protected $_name = 'bugs';
    protected $_rowsetClass = 'MyBugsRowset';
}
```

8. Zend_Db_Table Relationships

8.1. Introduction

Tables have relationships to each other in a relational database. An entity in one table can be linked to one or more entities in another table by using referential integrity constraints defined in the database schema.

The `Zend_Db_Table_Row` class has methods for querying related rows in other tables.

8.2. Defining Relationships

Define classes for each of your tables, extending the abstract class `Zend_Db_Table_Abstract`, as described in Section 5.3, “Defining a Table Class”. Also see Section 1.2, “Example Database” for a description of the example database for which the following example code is designed.

Below are the PHP class definitions for these tables:

```
class Accounts extends Zend_Db_Table_Abstract
{
    protected $_name            = 'accounts';
    protected $_dependentTables = array('Bugs');
}

class Products extends Zend_Db_Table_Abstract
{
```

```
    protected $_name            = 'products';
    protected $_dependentTables = array('BugsProducts');
}

class Bugs extends Zend_Db_Table_Abstract
{
    protected $_name            = 'bugs';

    protected $_dependentTables = array('BugsProducts');

    protected $_referenceMap    = array(
        'Reporter' => array(
            'columns'           => 'reported_by',
            'refTableClass'     => 'Accounts',
            'refColumns'        => 'account_name'
        ),
        'Engineer' => array(
            'columns'           => 'assigned_to',
            'refTableClass'     => 'Accounts',
            'refColumns'        => 'account_name'
        ),
        'Verifier' => array(
            'columns'           => array('verified_by'),
            'refTableClass'     => 'Accounts',
            'refColumns'        => array('account_name')
        )
    );
}

class BugsProducts extends Zend_Db_Table_Abstract
{
    protected $_name = 'bugs_products';

    protected $_referenceMap    = array(
        'Bug' => array(
            'columns'           => array('bug_id'),
            'refTableClass'     => 'Bugs',
            'refColumns'        => array('bug_id')
        ),
        'Product' => array(
            'columns'           => array('product_id'),
            'refTableClass'     => 'Products',
            'refColumns'        => array('product_id')
        )
    );

}
```

If you use `Zend_Db_Table` to emulate cascading UPDATE and DELETE operations, declare the `$_dependentTables` array in the class for the parent table. List the class name for each dependent table. Use the class name, not the physical name of the SQL table.

Skip declaration of `$_dependentTables` if you use referential integrity constraints in the RDBMS server to implement cascading operations. See Section 8.6, “Cascading Write Operations” for more information.

Declare the `$_referenceMap` array in the class for each dependent table. This is an associative array of reference "rules". A reference rule identifies which table is the parent table in the

relationship, and also lists which columns in the dependent table reference which columns in the parent table.

The rule key is a string used as an index to the `$_referenceMap` array. This rule key is used to identify each reference relationship. Choose a descriptive name for this rule key. It's best to use a string that can be part of a PHP method name, as you will see later.

In the example PHP code above, the rule keys in the Bugs table class are: `'Reporter'`, `'Engineer'`, `'Verifier'`, and `'Product'`.

The value of each rule entry in the `$_referenceMap` array is also an associative array. The elements of this rule entry are described below:

- *columns* => A string or an array of strings naming the foreign key column name(s) in the dependent table.

  It's common for this to be a single column, but some tables have multi-column keys.

- *refTableClass* => The class name of the parent table. Use the class name, not the physical name of the SQL table.

  It's common for a dependent table to have only one reference to its parent table, but some tables have multiple references to the same parent table. In the example database, there is one reference from the `bugs` table to the `products` table, but three references from the `bugs` table to the `accounts` table. Put each reference in a separate entry in the `$_referenceMap` array.

- *refColumns* => A string or an array of strings naming the primary key column name(s) in the parent table.

  It's common for this to be a single column, but some tables have multi-column keys. If the reference uses a multi-column key, the order of columns in the `'columns'` entry must match the order of columns in the `'refColumns'` entry.

  It is optional to specify this element. If you don't specify the `refColumns`, the column(s) reported as the primary key columns of the parent table are used by default.

- *onDelete* => The rule for an action to execute if a row is deleted in the parent table. See Section 8.6, “Cascading Write Operations” for more information.

- *onUpdate* => The rule for an action to execute if values in primary key columns are updated in the parent table. See Section 8.6, “Cascading Write Operations” for more information.

8.3. Fetching a Dependent Rowset

If you have a Row object as the result of a query on a parent table, you can fetch rows from dependent tables that reference the current row. Use the method:

```
$row->findDependentRowset($table, [$rule]);
```

This method returns a `Zend_Db_Table_Rowset_Abstract` object, containing a set of rows from the dependent table `$table` that refer to the row identified by the `$row` object.

The first argument `$table` can be a string that specifies the dependent table by its class name. You can also specify the dependent table by using an object of that table class.

Example 278. Fetching a Dependent Rowset

This example shows getting a Row object from the table `Accounts`, and finding the `Bugs` reported by that account.

```
$accountsTable = new Accounts();
$accountsRowset = $accountsTable->find(1234);
$user1234 = $accountsRowset->current();

$bugsReportedByUser = $user1234->findDependentRowset('Bugs');
```

The second argument `$rule` is optional. It is a string that names the rule key in the `$_referenceMap` array of the dependent table class. If you don't specify a rule, the first rule in the array that references the parent table is used. If you need to use a rule other than the first, you need to specify the key.

In the example code above, the rule key is not specified, so the rule used by default is the first one that matches the parent table. This is the rule `'Reporter'`.

Example 279. Fetching a Dependent Rowset By a Specific Rule

This example shows getting a Row object from the table `Accounts`, and finding the `Bugs` assigned to be fixed by the user of that account. The rule key string that corresponds to this reference relationship in this example is `'Engineer'`.

```
$accountsTable = new Accounts();
$accountsRowset = $accountsTable->find(1234);
$user1234 = $accountsRowset->current();

$bugsAssignedToUser = $user1234->findDependentRowset('Bugs', 'Engineer');
```

You can also add criteria, ordering and limits to your relationships using the parent row's select object.

Example 280. Fetching a Dependent Rowset using a Zend_Db_Table_Select

This example shows getting a Row object from the table `Accounts`, and finding the `Bugs` assigned to be fixed by the user of that account, limited only to 3 rows and ordered by name.

```
$accountsTable = new Accounts();
$accountsRowset = $accountsTable->find(1234);
$user1234 = $accountsRowset->current();
$select = $accountsTable->select()->order('name ASC')
                                  ->limit(3);

$bugsAssignedToUser = $user1234->findDependentRowset('Bugs',
                                                     'Engineer',
                                                     $select);
```

Alternatively, you can query rows from a dependent table using a special mechanism called a "magic method". `Zend_Db_Table_Row_Abstract` invokes the method: `findDependentRowset('<TableClass>', '<Rule>')` if you invoke a method on the Row object matching either of the following patterns:

- `$row->find<TableClass>()`
- `$row->find<TableClass>By<Rule>()`

In the patterns above, `<TableClass>` and `<Rule>` are strings that correspond to the class name of the dependent table, and the dependent table's rule key that references the parent table.

Some application frameworks, such as Ruby on Rails, use a mechanism called "inflection" to allow the spelling of identifiers to change depending on usage. For simplicity, `Zend_Db_Table_Row` does not provide any inflection mechanism. The table identity and the rule key named in the method call must match the spelling of the class and rule key exactly.

Example 281. Fetching Dependent Rowsets using the Magic Method

This example shows finding dependent Rowsets equivalent to those in the previous examples. In this case, the application uses the magic method invocation instead of specifying the table and rule as strings.

```
$accountsTable = new Accounts();
$accountsRowset = $accountsTable->find(1234);
$user1234 = $accountsRowset->current();

// Use the default reference rule
$bugsReportedBy = $user1234->findBugs();

// Specify the reference rule
$bugsAssignedTo = $user1234->findBugsByEngineer();
```

8.4. Fetching a Parent Row

If you have a Row object as the result of a query on a dependent table, you can fetch the row in the parent to which the dependent row refers. Use the method:

```
$row->findParentRow($table, [$rule]);
```

There always should be exactly one row in the parent table referenced by a dependent row, therefore this method returns a Row object, not a Rowset object.

The first argument `$table` can be a string that specifies the parent table by its class name. You can also specify the parent table by using an object of that table class.

Example 282. Fetching the Parent Row

This example shows getting a Row object from the table `Bugs` (for example one of those bugs with status 'NEW'), and finding the row in the `Accounts` table for the user who reported the bug.

```
$bugsTable = new Bugs();
$bugsRowset = $bugsTable->fetchAll(array('bug_status = ?' => 'NEW'));
$bug1 = $bugsRowset->current();

$reporter = $bug1->findParentRow('Accounts');
```

The second argument `$rule` is optional. It is a string that names the rule key in the `$_referenceMap` array of the dependent table class. If you don't specify a rule, the first rule in the array that references the parent table is used. If you need to use a rule other than the first, you need to specify the key.

In the example above, the rule key is not specified, so the rule used by default is the first one that matches the parent table. This is the rule `'Reporter'`.

Example 283. Fetching a Parent Row By a Specific Rule

This example shows getting a Row object from the table `Bugs`, and finding the account for the engineer assigned to fix that bug. The rule key string that corresponds to this reference relationship in this example is `'Engineer'`.

```
$bugsTable = new Bugs();
$bugsRowset = $bugsTable->fetchAll(array('bug_status = ?', 'NEW'));
$bug1 = $bugsRowset->current();

$engineer = $bug1->findParentRow('Accounts', 'Engineer');
```

Alternatively, you can query rows from a parent table using a "magic method". `Zend_Db_Table_Row_Abstract` invokes the method: `findParentRow('<TableClass>', '<Rule>')` if you invoke a method on the Row object matching either of the following patterns:

- `$row->findParent<TableClass>([Zend_Db_Table_Select $select])`
- `$row->findParent<TableClass>By<Rule>([Zend_Db_Table_Select $select])`

In the patterns above, `<TableClass>` and `<Rule>` are strings that correspond to the class name of the parent table, and the dependent table's rule key that references the parent table.

The table identity and the rule key named in the method call must match the spelling of the class and rule key exactly.

Example 284. Fetching the Parent Row using the Magic Method

This example shows finding parent Rows equivalent to those in the previous examples. In this case, the application uses the magic method invocation instead of specifying the table and rule as strings.

```
$bugsTable = new Bugs();
$bugsRowset = $bugsTable->fetchAll(array('bug_status = ?', 'NEW'));
$bug1 = $bugsRowset->current();

// Use the default reference rule
$reporter = $bug1->findParentAccounts();

// Specify the reference rule
$engineer = $bug1->findParentAccountsByEngineer();
```

8.5. Fetching a Rowset via a Many-to-many Relationship

If you have a Row object as the result of a query on one table in a many-to-many relationship (for purposes of the example, call this the "origin" table), you can fetch corresponding rows in the other table (call this the "destination" table) via an intersection table. Use the method:

```
$row->findManyToManyRowset($table,
                           $intersectionTable,
                           [$rule1,
                               [$rule2,
```

```
                                    [Zend_Db_Table_Select $select]
                                ]
                            ]);
```

This method returns a `Zend_Db_Table_Rowset_Abstract` containing rows from the table `$table`, satisfying the many-to-many relationship. The current Row object `$row` from the origin table is used to find rows in the intersection table, and that is joined to the destination table.

The first argument `$table` can be a string that specifies the destination table in the many-to-many relationship by its class name. You can also specify the destination table by using an object of that table class.

The second argument `$intersectionTable` can be a string that specifies the intersection table between the two tables in the many-to-many relationship by its class name. You can also specify the intersection table by using an object of that table class.

Example 285. Fetching a Rowset with the Many-to-many Method

This example shows getting a Row object from the origin table `Bugs`, and finding rows from the destination table `Products`, representing products related to that bug.

```
$bugsTable = new Bugs();
$bugsRowset = $bugsTable->find(1234);
$bug1234 = $bugsRowset->current();

$productsRowset = $bug1234->findManyToManyRowset('Products',
                                                 'BugsProducts');
```

The third and fourth arguments `$rule1` and `$rule2` are optional. These are strings that name the rule keys in the `$_referenceMap` array of the intersection table.

The `$rule1` key names the rule for the relationship from the intersection table to the origin table. In this example, this is the relationship from `BugsProducts` to `Bugs`.

The `$rule2` key names the rule for the relationship from the intersection table to the destination table. In this example, this is the relationship from `Bugs` to `Products`.

Similarly to the methods for finding parent and dependent rows, if you don't specify a rule, the method uses the first rule in the `$_referenceMap` array that matches the tables in the relationship. If you need to use a rule other than the first, you need to specify the key.

In the example code above, the rule key is not specified, so the rules used by default are the first ones that match. In this case, `$rule1` is `'Reporter'` and `$rule2` is `'Product'`.

Example 286. Fetching a Rowset with the Many-to-many Method By a Specific Rule

This example shows geting a Row object from the origin table `Bugs`, and finding rows from the destination table `Products`, representing products related to that bug.

```
$bugsTable = new Bugs();
$bugsRowset = $bugsTable->find(1234);
$bug1234 = $bugsRowset->current();

$productsRowset = $bug1234->findManyToManyRowset('Products',
                                                 'BugsProducts',
                                                 'Bug');
```

Alternatively, you can query rows from the destination table in a many-to-many relationship using a "magic method." `Zend_Db_Table_Row_Abstract` invokes the method: `findManyToManyRowset('<TableClass>', '<IntersectionTableClass>', '<Rule1>', '<Rule2>')` if you invoke a method matching any of the following patterns:

- `$row->find<TableClass>Via<IntersectionTableClass>([Zend_Db_Table_Select $select])`
- `$row->find<TableClass>Via<IntersectionTableClass>By<Rule1>([Zend_Db_Table_Select $select])`
- `$row->find<TableClass>Via<IntersectionTableClass>By<Rule1>And<Rule2>([Zend_Db_Table_Select $select])`

In the patterns above, `<TableClass>` and `<IntersectionTableClass>` are strings that correspond to the class names of the destination table and the intersection table, respectively. `<Rule1>` and `<Rule2>` are strings that correspond to the rule keys in the intersection table that reference the origin table and the destination table, respectively.

The table identities and the rule keys named in the method call must match the spelling of the class and rule key exactly.

Example 287. Fetching Rowsets using the Magic Many-to-many Method

This example shows finding rows in the destination table of a many-to-many relationship representing products related to a given bug.

```
$bugsTable = new Bugs();
$bugsRowset = $bugsTable->find(1234);
$bug1234 = $bugsRowset->current();

// Use the default reference rule
$products = $bug1234->findProductsViaBugsProducts();

// Specify the reference rule
$products = $bug1234->findProductsViaBugsProductsByBug();
```

8.6. Cascading Write Operations

Declare DRI in the database:

Declaring cascading operations in `Zend_Db_Table` is intended *only* for RDBMS brands that do not support declarative referential integrity (DRI).

For example, if you use MySQL's MyISAM storage engine, or SQLite, these solutions do not support DRI. You may find it helpful to declare the cascading operations with `Zend_Db_Table`.

If your RDBMS implements DRI and the `ON DELETE` and `ON UPDATE` clauses, you should declare these clauses in your database schema, instead of using the cascading feature in `Zend_Db_Table`. Declaring cascading DRI rules in the RDBMS is better for database performance, consistency, and integrity.

Most importantly, do not declare cascading operations both in the RDBMS and in your `Zend_Db_Table` class.

You can declare cascading operations to execute against a dependent table when you apply an UPDATE or a DELETE to a row in a parent table.

Example 288. Example of a Cascading Delete

This example shows deleting a row in the Products table, which is configured to automatically delete dependent rows in the Bugs table.

```
$productsTable = new Products();
$productsRowset = $productsTable->find(1234);
$product1234 = $productsRowset->current();

$product1234->delete();
// Automatically cascades to Bugs table
// and deletes dependent rows.
```

Similarly, if you use UPDATE to change the value of a primary key in a parent table, you may want the value in foreign keys of dependent tables to be updated automatically to match the new value, so that such references are kept up to date.

It's usually not necessary to update the value of a primary key that was generated by a sequence or other mechanism. But if you use a *natural key* that may change value occasionally, it is more likely that you need to apply cascading updates to dependent tables.

To declare a cascading relationship in the Zend_Db_Table, edit the rules in the $_referenceMap. Set the associative array keys 'onDelete' and 'onUpdate' to the string 'cascade' (or the constant self::CASCADE). Before a row is deleted from the parent table, or its primary key values updated, any rows in the dependent table that refer to the parent's row are deleted or updated first.

Example 289. Example Declaration of Cascading Operations

In the example below, rows in the Bugs table are automatically deleted if the row in the Products table to which they refer is deleted. The 'onDelete' element of the reference map entry is set to self::CASCADE.

No cascading update is done in the example below if the primary key value in the parent class is changed. The 'onUpdate' element of the reference map entry is self::RESTRICT. You can get the same result by omitting the 'onUpdate' entry.

```
class BugsProducts extends Zend_Db_Table_Abstract
{
    ...
    protected $_referenceMap = array(
        'Product' => array(
            'columns'           => array('product_id'),
            'refTableClass'     => 'Products',
            'refColumns'        => array('product_id'),
            'onDelete'          => self::CASCADE,
            'onUpdate'          => self::RESTRICT
        ),
        ...
    );
}
```

8.6.1. Notes Regarding Cascading Operations

Cascading operations invoked by Zend_Db_Table are not atomic.

This means that if your database implements and enforces referential integrity constraints, a cascading `UPDATE` executed by a `Zend_Db_Table` class conflicts with the constraint, and results in a referential integrity violation. You can use cascading `UPDATE` in `Zend_Db_Table` *only* if your database does not enforce that referential integrity constraint.

Cascading `DELETE` suffers less from the problem of referential integrity violations. You can delete dependent rows as a non-atomic action before deleting the parent row that they reference.

However, for both `UPDATE` and `DELETE`, changing the database in a non-atomic way also creates the risk that another database user can see the data in an inconsistent state. For example, if you delete a row and all its dependent rows, there is a small chance that another database client program can query the database after you have deleted the dependent rows, but before you delete the parent row. That client program may see the parent row with no dependent rows, and assume this is the intended state of the data. There is no way for that client to know that its query read the database in the middle of a change.

The issue of non-atomic change can be mitigated by using transactions to isolate your change. But some RDBMS brands don't support transactions, or allow clients to read "dirty" changes that have not been committed yet.

Cascading operations in `Zend_Db_Table` are invoked only by `Zend_Db_Table`.

Cascading deletes and updates defined in your `Zend_Db_Table` classes are applied if you execute the `save()` or `delete()` methods on the Row class. However, if you update or delete data using another interface, such as a query tool or another application, the cascading operations are not applied. Even when using `update()` and `delete()` methods in the `Zend_Db_Adapter` class, cascading operations defined in your `Zend_Db_Table` classes are not executed.

No Cascading `INSERT`.

There is no support for a cascading `INSERT`. You must insert a row to a parent table in one operation, and insert row(s) to a dependent table in a separate operation.

9. Zend_Db_Table_Definition

9.1. Introduction

`Zend_Db_Table_Definition` is a class that can be used to describe the relationships and configuration options that should be used when `Zend_Db_Table` is used via concrete instantiation.

9.2. Basic Usage

For all of the same options that are available when configuring an extended `Zend_Db_Table_Abstract` class, those options are also available when describing a definition file. This definition file should be passed to the class at instantiation time so that it can know the full definition of all tables in said definition.

Below is a definition that will describe the table names and relationships between table objects. Note: if 'name' is left out of the definition, it will be taken as the key of the defined table (an example of this is in the 'genre' section in the example below.)

Example 290. Describing the Definition of a Database Data Model

```
$definition = new Zend_Db_Table_Definition(array(
    'author' => array(
        'name' => 'author',
        'dependentTables' => array('book')
        ),
    'book' => array(
        'name' => 'book',
        'referenceMap' => array(
            'author' => array(
                'columns' => 'author_id',
                'refTableClass' => 'author',
                'refColumns' => 'id'
                )
            )
        ),
    'genre' => null,
    'book_to_genre' => array(
        'referenceMap' => array(
            'book' => array(
                'columns' => 'book_id',
                'refTableClass' => 'book',
                'refColumns' => 'id'
                ),
            'genre' => array(
                'columns' => 'genre_id',
                'refTableClass' => 'genre',
                'refColumns' => 'id'
                )
            )
        )
    ));
```

As you can see, the same options you'd generally see inside of an extended `Zend_Db_Table_Abstract` class are documented in this array as well. When passed into `Zend_Db_Table` constructor, this definition is *persisted* to any tables it will need to create in order to return the proper rows.

Below is an example of the primary table instantiation as well as the `findDependentRowset()` and `findManyToManyRowset()` calls that will correspond to the data model described above:

Example 291. Interacting with the described definition

```
$authorTable = new Zend_Db_Table('author', $definition);
$authors = $authorTable->fetchAll();

foreach ($authors as $author) {
    echo $author->id
       . ': '
       . $author->first_name
       . ' '
       . $author->last_name
       . PHP_EOL;
    $books = $author->findDependentRowset('book');
    foreach ($books as $book) {
        echo '    Book: ' . $book->title . PHP_EOL;
        $genreOutputArray = array();
        $genres = $book->findManyToManyRowset('genre', 'book_to_genre');
        foreach ($genres as $genreRow) {
            $genreOutputArray[] = $genreRow->name;
        }
        echo '        Genre: ' . implode(', ', $genreOutputArray) . PHP_EOL;
    }
}
```

9.3. Advanced Usage

Sometimes you want to use both paradigms for defining and using the table gateway: both by extension and concrete instantiation. To do this simply leave out any table configurations out of the definition. This will allow `Zend_Db_Table` to look for the actual refered class instead of the definition key.

Building on the example above, we will allow for one of the table configurations to be a `Zend_Db_Table_Abstract` extended class, while keeping the rest of the tables as part of the definition. We will also show how one would interact with this new definition.

Example 292. Interacting A Mixed Use Zend_Db_Table Definition

```
class MyBook extends Zend_Db_Table_Abstract
{
    protected $_name = 'book';
    protected $_referenceMap = array(
        'author' => array(
            'columns' => 'author_id',
            'refTableClass' => 'author',
            'refColumns' => 'id'
            )
        );
}

$definition = new Zend_Db_Table_Definition(array(
    'author' => array(
        'name' => 'author',
        'dependentTables' => array('MyBook')
        ),
    'genre' => null,
    'book_to_genre' => array(
        'referenceMap' => array(
            'book' => array(
                'columns' => 'book_id',
                'refTableClass' => 'MyBook',
                'refColumns' => 'id'
                ),
            'genre' => array(
                'columns' => 'genre_id',
                'refTableClass' => 'genre',
                'refColumns' => 'id'
                )
            )
        )
    ));

$authorTable = new Zend_Db_Table('author', $definition);
$authors = $authorTable->fetchAll();

foreach ($authors as $author) {
    echo $author->id
       . ': '
       . $author->first_name
       . ' '
       . $author->last_name
       . PHP_EOL;
    $books = $author->findDependentRowset(new MyBook());
    foreach ($books as $book) {
        echo '    Book: ' . $book->title . PHP_EOL;
        $genreOutputArray = array();
        $genres = $book->findManyToManyRowset('genre', 'book_to_genre');
        foreach ($genres as $genreRow) {
            $genreOutputArray[] = $genreRow->name;
        }
        echo '        Genre: ' . implode(', ', $genreOutputArray) . PHP_EOL;
    }
}
```

Zend_Debug

1. Dumping Variables

The static method `Zend_Debug::dump()` prints or returns information about an expression. This simple technique of debugging is common because it is easy to use in an ad hoc fashion and requires no initialization, special tools, or debugging environment.

Example 293. Example of dump() method

```
Zend_Debug::dump($var, $label = null, $echo = true);
```

The `$var` argument specifies the expression or variable about which the `Zend_Debug::dump()` method outputs information.

The `$label` argument is a string to be prepended to the output of `Zend_Debug::dump()`. It may be useful, for example, to use labels if you are dumping information about multiple variables on a given screen.

The boolean `$echo` argument specifies whether the output of `Zend_Debug::dump()` is echoed or not. If `TRUE`, the output is echoed. Regardless of the value of the `$echo` argument, the return value of this method contains the output.

It may be helpful to understand that `Zend_Debug::dump()` method wraps the PHP function `var_dump()`. If the output stream is detected as a web presentation, the output of `var_dump()` is escaped using `htmlspecialchars()` and wrapped with (X)HTML **<pre>** tags.

Debugging with Zend_Log

Using `Zend_Debug::dump()` is best for ad hoc debugging during software development. You can add code to dump a variable and then remove the code very quickly.

Also consider the Zend_Log component when writing more permanent debugging code. For example, you can use the `DEBUG` log level and the stream log writer to output the string returned by `Zend_Debug::dump()`.

Zend_Dojo

1. Introduction

Zend Framework ships Dojo Toolkit to support out-of-the-box rich internet application development. Integration points with Dojo include:

- JSON-RPC support
- dojo.data compatibility
- View helper to help setup the Dojo environment
- Dijit-specific `Zend_View` helpers
- Dijit-specific `Zend_Form` elements and decorators

The Dojo distribution itself may be found in the `externals/dojo/` directory of the Zend Framework distribution. This is a source distribution, which includes Dojo's full javascript source, unit tests, and build tools. You can symlink this into your javascript directory, copy it, or use the build tool to create your own custom build to include in your project. Alternatively, you can use one of the Content Delivery Networks that offer Dojo (Zend Framework supports both the official AOL CDN as well as the Google CDN).

2. Zend_Dojo_Data: dojo.data Envelopes

Dojo provides data abstractions for data-enabled widgets via its dojo.data component. This component provides the ability to attach a data store, provide some metadata regarding the identity field and optionally a label field, and an API for querying, sorting, and retrieving records and sets of records from the datastore.

dojo.data is often used with XmlHttpRequest to pull dynamic data from the server. The primary mechanism for this is to extend the QueryReadStore to point at a URL and specify the query information. The server side then returns data in the following JSON format:

```
{
    identifier: '<name>',
    <label: '<label>',>
    items: [
        { name: '...', label: '...', someKey: '...' },
        ...
    ]
}
```

`Zend_Dojo_Data` provides a simple interface for building such structures programmatically, interacting with them, and serializing them to an array or JSON.

2.1. Zend_Dojo_Data Usage

At its simplest, dojo.data requires that you provide the name of the identifier field in each item, and a set of items (data). You can either pass these in via the constructor, or via mutators:

Example 294. Zend_Dojo_Data initialization via constructor

```
$data = new Zend_Dojo_Data('id', $items);
```

Example 295. Zend_Dojo_Data initialization via mutators

```
$data = new Zend_Dojo_Data();
$data->setIdentifier('id')
     ->addItems($items);
```

You can also add a single item at a time, or append items, using `addItem()` and `addItems()`.

Example 296. Appending data to Zend_Dojo_Data

```
$data = new Zend_Dojo_Data($identifier, $items);
$data->addItem($someItem);

$data->addItems($someMoreItems);
```

Always use an identifier!

Every dojo.data datastore requires that the identifier column be provided as metadata, including `Zend_Dojo_Data`. In fact, if you attempt to add items without an identifier, it will raise an exception.

Individual items may be one of the following:

- Associative arrays
- Objects implementing a `toArray()` method
- Any other objects (will serialize via get_object_vars())

You can attach collections of the above items via `addItems()` or `setItems()` (overwrites all previously set items); when doing so, you may pass a single argument:

- Arrays
- Objects implementing the `Traversable` interface ,which includes the interfaces `Iterator` and `ArrayAccess`.

If you want to specify a field that will act as a label for the item, call `setLabel()`:

Example 297. Specifying a label field in Zend_Dojo_Data

```
$data->setLabel('name');
```

Finally, you can also load a `Zend_Dojo_Data` item from a dojo.data JSON array, using the `fromJson()` method.

Example 298. Populating Zend_Dojo_Data from JSON

```
$data->fromJson($json);
```

2.2. Adding metadata to your containers

Some Dojo components require additional metadata along with the dojo.data payload. As an example, **dojox.grid.Grid** can pull data dynamically from a **dojox.data.QueryReadStore**. For pagination to work correctly, each return payload should contain a numRows key with the total number of rows that could be returned by the query. With this data, the grid knows when to continue making small requests to the server for subsets of data and when to stop making more requests (i.e., it has reached the last page of data). This technique is useful for serving large sets of data in your grids without loading the entire set at once.

`Zend_Dojo_Data` allows assigning metadata properties as to the object. The following illustrates usage:

```
// Set the "numRows" to 100
$data->setMetadata('numRows', 100);

// Set several items at once:
$data->setMetadata(array(
    'numRows' => 100,
    'sort'    => 'name',
));

// Inspect a single metadata value:
$numRows = $data->getMetadata('numRows');

// Inspect all metadata:
$metadata = $data->getMetadata();

// Remove a metadata item:
$data->clearMetadata('numRows');

// Remove all metadata:
$data->clearMetadata();
```

2.3. Advanced Use Cases

Besides acting as a serializable data container, `Zend_Dojo_Data` also provides the ability to manipulate and traverse the data in a variety of ways.

`Zend_Dojo_Data` implements the interfaces `ArrayAccess`, `Iterator`, and `Countable`. You can therefore use the data collection almost as if it were an array.

All items are referenced by the identifier field. Since identifiers must be unique, you can use the values of this field to pull individual records. There are two ways to do this: with the `getItem()` method, or via array notation.

```
// Using getItem():
$item = $data->getItem('foo');

// Or use array notation:
$item = $data['foo'];
```

If you know the identifier, you can use it to retrieve an item, update it, delete it, create it, or test for it:

```
// Update or create an item:
```

```
$data['foo'] = array('title' => 'Foo', 'email' => 'foo@foo.com');

// Delete an item:
unset($data['foo']);

// Test for an item:
if (isset($data[foo])) {
}
```

You can loop over all items as well. Internally, all items are stored as arrays.

```
foreach ($data as $item) {
    echo $item['title'] . ': ' . $item['description'] . "\n";
}
```

Or even count to see how many items you have:

```
echo count($data), " items found!";
```

Finally, as the class implements `__toString()`, you can also cast it to JSON simply by echoing it or casting to string:

```
echo $data; // echo as JSON string

$json = (string) $data; // cast to string == cast to JSON
```

2.3.1. Available Methods

Besides the methods necessary for implementing the interfaces listed above, the following methods are available.

- `setItems($items)`: set multiple items at once, overwriting any items that were previously set in the object. `$items` should be an array or a `Traversable` object.
- `setItem($item, $id = null)`: set an individual item, optionally passing an explicit identifier. Overwrites the item if it is already in the collection. Valid items include associative arrays, objects implementing `toArray()`, or any object with public properties.
- `addItem($item, $id = null)`: add an individual item, optionally passing an explicit identifier. Will raise an exception if the item already exists in the collection. Valid items include associative arrays, objects implementing `toArray()`, or any object with public properties.
- `addItems($items)`: add multiple items at once, appending them to any current items. Will raise an exception if any of the new items have an identifier matching an identifier already in the collection. `$items` should be an array or a `Traversable` object.
- `getItems()`: retrieve all items as an array of arrays.
- `hasItem($id)`: determine whether an item with the given identifier exists in the collection.
- `getItem($id)`: retrieve an item with the given identifier from the collection; the item returned will be an associative array. If no item matches, a `NULL` value is returned.
- `removeItem($id)`: remove an item with the given identifier from the collection.

- `clearItems()`: remove all items from the collection.
- `setIdentifier($identifier)`: set the name of the field that represents the unique identifier for each item in the collection.
- `getIdentifier()`: retrieve the name of the identifier field.
- `setLabel($label)`: set the name of a field to be used as a display label for an item.
- `getLabel()`: retrieve the label field name.
- `toArray()`: cast the object to an array. At a minimum, the array will contain the keys 'identifier', 'items', and 'label' if a label field has been set in the object.
- `toJson()`: cast the object to a JSON representation.

3. Dojo View Helpers

Zend Framework provides the following Dojo-specific view helpers:

- *dojo():* setup the Dojo environment for your page, including dojo configuration values, custom module paths, module require statements, theme stylesheets, CDN usage, and more.

Example 299. Using Dojo View Helpers

To use Dojo view helpers, you will need to tell your view object where to find them. You can do this by calling `addHelperPath()`:

```
$view->addHelperPath('Zend/Dojo/View/Helper/', 'Zend_Dojo_View_Helper');
```

Alternately, you can use `Zend_Dojo`'s `enableView()` method to do the work for you:

```
Zend_Dojo::enableView($view);
```

3.1. dojo() View Helper

The `dojo()` view helper is intended to simplify setting up the Dojo environment, including the following responsibilities:

- Specifying either a CDN or a local path to a Dojo install.
- Specifying paths to custom Dojo modules.
- Specifying dojo.require statements.
- Specifying dijit stylesheet themes to use.
- Specifying dojo.addOnLoad() events.

The `dojo()` view helper implementation is an example of a placeholder implementation. The data set in it persists between view objects and may be directly echoed from your layout script.

Example 300. dojo() View Helper Usage Example

For this example, let's assume the developer will be using Dojo from a local path, will require several dijits, and will be utilizing the Tundra dijit theme.

On many pages, the developer may not utilize Dojo at all. So, we will first focus on a view script where Dojo is needed and then on the layout script, where we will setup some of the Dojo environment and then render it.

First, we need to tell our view object to use the Dojo view helper paths. This can be done in your bootstrap or an early-running plugin; simply grab your view object and execute the following:

```
$view->addHelperPath('Zend/Dojo/View/Helper/', 'Zend_Dojo_View_Helper');
```

Next, the view script. In this case, we're going to specify that we will be using a FilteringSelect -- which will consume a custom store based on QueryReadStore, which we'll call 'PairedStore' and store in our 'custom' module.

```
<?php // setup data store for FilteringSelect ?>
<div dojoType="custom.PairedStore" jsId="stateStore"
    url="/data/autocomplete/type/state/format/ajax"
    requestMethod="get"></div>

<?php // Input element: ?>
State: <input id="state" dojoType="dijit.form.FilteringSelect"
    store="stateStore" pageSize="5" />

<?php // setup required dojo elements:
$this->dojo()->enable()
             ->setDjConfigOption('parseOnLoad', true)
             ->registerModulePath('custom', '../custom/')
             ->requireModule('dijit.form.FilteringSelect')
             ->requireModule('custom.PairedStore'); ?>
```

In our layout script, we'll then check to see if Dojo is enabled, and, if so, we'll do some more general configuration and assemble it:

```
<?php echo $this->doctype() ?>
<html>
<head>
    <?php echo $this->headTitle() ?>
    <?php echo $this->headMeta() ?>
    <?php echo $this->headLink() ?>
    <?php echo $this->headStyle() ?>
<?php if ($this->dojo()->isEnabled()){
    $this->dojo()->setLocalPath('/js/dojo/dojo.js')
                 ->addStyleSheetModule('dijit.themes.tundra');
    echo $this->dojo();
   }
?>
    <?php echo $this->headScript() ?>
</head>
<body class="tundra">
    <?php echo $this->layout()->content ?>
    <?php echo $this->inlineScript() ?>
</body>
</html>
```

At this point, you only need to ensure that your files are in the correct locations and that you've created the end point action for your FilteringSelect!

UTF-8 encoding used by default

By default, Zend Framework uses UTF-8 as its default encoding, and, specific to this case, `Zend_View` does as well. Character encoding can be set differently on the view object itself using the `setEncoding()` method (or the the `encoding` instantiation parameter). However, since `Zend_View_Interface` does not define accessors for encoding, it's possible that if you are using a custom view implementation with the Dojo view helper, you will not have a `getEncoding()` method, which is what the view helper uses internally for determining the character set in which to encode.

If you do not want to utilize UTF-8 in such a situation, you will need to implement a `getEncoding()` method in your custom view implementation.

3.1.1. Programmatic and Declarative Usage of Dojo

Dojo allows both *declarative* and *programmatic* usage of many of its features. *Declarative* usage uses standard HTML elements with non-standard attributes that are parsed when the page is loaded. While this is a powerful and simple syntax to utilize, for many developers this can cause issues with page validation.

Programmatic usage allows the developer to decorate existing elements by pulling them by ID or CSS selectors and passing them to the appropriate object constructors in Dojo. Because no non-standard HTML attributes are used, pages continue to validate.

In practice, both use cases allow for graceful degradation when javascript is disabled or the various Dojo script resources are unreachable. To promote standards and document validation, Zend Framework uses programmatic usage by default; the various view helpers will generate javascript and push it to the `dojo()` view helper for inclusion when rendered.

Developers using this technique may also wish to explore the option of writing their own programmatic decoration of the page. One benefit would be the ability to specify handlers for dijit events.

To allow this, as well as the ability to use declarative syntax, there are a number of static methods available to set this behavior globally.

Example 301. Specifying Declarative and Programmatic Dojo Usage

To specify declarative usage, simply call the static `setUseDeclarative()` method:

```
Zend_Dojo_View_Helper_Dojo::setUseDeclarative();
```

If you decide instead to use programmatic usage, call the static `setUseProgrammatic()` method:

```
Zend_Dojo_View_Helper_Dojo::setUseProgrammatic();
```

Finally, if you want to create your own programmatic rules, you should specify programmatic usage, but pass in the value '-1'; in this situation, no javascript for decorating any dijits used will be created.

```
Zend_Dojo_View_Helper_Dojo::setUseProgrammatic(-1);
```

3.1.2. Themes

Dojo allows the creation of themes for its dijits (widgets). You may select one by passing in a module path:

```
$view->dojo()->addStylesheetModule('dijit.themes.tundra');
```

The module path is discovered by using the character '.' as a directory separator and using the last value in the list as the name of the CSS file in that theme directory to use; in the example above, Dojo will look for the theme in 'dijit/themes/tundra/tundra.css'.

When using a theme, it is important to remember to pass the theme class to, at the least, a container surrounding any dijits you are using; the most common use case is to pass it in the body:

```
<body class="tundra">
```

3.1.3. Using Layers (Custom Builds)

By default, when you use a dojo.require statement, dojo will make a request back to the server to grab the appropriate javascript file. If you have many dijits in place, this results in many requests to the server -- which is not optimal.

Dojo's answer to this is to provide the ability to create *custom builds*. Builds do several things:

- Groups required files into *layers*; a layer lumps all required files into a single JS file. (Hence the name of this section.)
- "Interns" non-javascript files used by dijits (typically, template files). These are also grouped in the same JS file as the layer.
- Passes the file through ShrinkSafe, which strips whitespace and comments, as well as shortens variable names.

Some files can not be layered, but the build process will create a special release directory with the layer file and all other files. This allows you to have a slimmed-down distribution customized for your site or application needs.

To use a layer, the `dojo()` view helper has a `addLayer()` method for adding paths to required layers:

```
$view->dojo()->addLayer('/js/foo/foo.js');
```

For more information on creating custom builds, please refer to the Dojo build documentation.

3.1.4. Methods Available

The `dojo()` view helper always returns an instance of the dojo placeholder container. That container object has the following methods available:

- `setView(Zend_View_Interface $view)`: set a view instance in the container.
- `enable()`: explicitly enable Dojo integration.
- `disable()`: disable Dojo integration.

- `isEnabled()`: determine whether or not Dojo integration is enabled.
- `requireModule($module)`: setup a dojo.require statement.
- `getModules()`: determine what modules have been required.
- `registerModulePath($module, $path)`: register a custom Dojo module path.
- `getModulePaths()`: get list of registered module paths.
- `addLayer($path)`: add a layer (custom build) path to use.
- `getLayers()`: get a list of all registered layer paths (custom builds).
- `removeLayer($path)`: remove the layer that matches `$path` from the list of registered layers (custom builds).
- `setCdnBase($url)`: set the base URL for a CDN; typically, one of the `Zend_Dojo::CDN_BASE_AOL` or `Zend_Dojo::CDN_BASE_GOOGLE`, but it only needs to be the URL string prior to the version number.
- `getCdnBase()`: retrieve the base CDN url to utilize.
- `setCdnVersion($version = null)`: set which version of Dojo to utilize from the CDN.
- `getCdnVersion()`: retrieve what version of Dojo from the CDN will be used.
- `setCdnDojoPath($path)`: set the relative path to the dojo.js or dojo.xd.js file on a CDN; typically, one of the `Zend_Dojo::CDN_DOJO_PATH_AOL` or `Zend_Dojo::CDN_DOJO_PATH_GOOGLE`, but it only needs to be the path string following the version number.
- `getCdnDojoPath()`: retrieve the last path segment of the CDN url pointing to the dojo.js file.
- `useCdn()`: tell the container to utilize the CDN; implicitly enables integration.
- `setLocalPath($path)`: tell the container the path to a local Dojo install (should be a path relative to the server, and contain the dojo.js file itself); implicitly enables integration.
- `getLocalPath()`: determine what local path to Dojo is being used.
- `useLocalPath()`: is the integration utilizing a Dojo local path?
- `setDjConfig(array $config)`: set dojo/dijit configuration values (expects assoc array).
- `setDjConfigOption($option, $value)`: set a single dojo/dijit configuration value.
- `getDjConfig()`: get all dojo/dijit configuration values.
- `getDjConfigOption($option, $default = null)`: get a single dojo/dijit configuration value.
- `addStylesheetModule($module)`: add a stylesheet based on a module theme.
- `getStylesheetModules()`: get stylesheets registered as module themes.

- `addStylesheet($path)`: add a local stylesheet for use with Dojo.
- `getStylesheets()`: get local Dojo stylesheets.
- `addOnLoad($spec, $function = null)`: add a lambda for dojo.onLoad to call. If one argument is passed, it is assumed to be either a function name or a javascript closure. If two arguments are passed, the first is assumed to be the name of an object instance variable and the second either a method name in that object or a closure to utilize with that object.
- `prependOnLoad($spec, $function = null)`: exactly like `addOnLoad()`, excepts prepends to beginning of onLoad stack.
- `getOnLoadActions()`: retrieve all dojo.onLoad actions registered with the container. This will be an array of arrays.
- `onLoadCaptureStart($obj = null)`: capture data to be used as a lambda for dojo.onLoad(). If $obj is provided, the captured JS code will be considered a closure to use with that Javascript object.
- `onLoadCaptureEnd($obj = null)`: finish capturing data for use with dojo.onLoad().
- `javascriptCaptureStart()`: capture arbitrary javascript to be included with Dojo JS (onLoad, require, etc. statements).
- `javascriptCaptureEnd()`: finish capturing javascript.
- `__toString()`: cast the container to a string; renders all HTML style and script elements.

3.2. Dijit-Specific View Helpers

From the Dojo manual: "Dijit is a widget system layered on top of dojo." Dijit includes a variety of layout and form widgets designed to provide accessibility features, localization, and standardized (and themeable) look-and-feel.

Zend Framework ships a variety of view helpers that allow you to render and utilize dijits within your view scripts. There are three basic types:

- *Layout Containers*: these are designed to be used within your view scripts or consumed by form decorators for forms, sub forms, and display groups. They wrap the various classes offerred in dijit.layout. Each dijit layout view helper expects the following arguments:
  - `$id`: the container name or DOM ID.
  - `$content`: the content to wrap in the layout container.
  - `$params` (optional): dijit-specific parameters. Basically, any non-HTML attribute that can be used to configure the dijit layout container.
  - `$attribs` (optional): any additional HTML attributes that should be used to render the container div. If the key 'id' is passed in this array, it will be used for the form element DOM id, and `$id` will be used for its name.

  If you pass no arguments to a dijit layout view helper, the helper itself will be returned. This allows you to capture content, which is often an easier way to pass content to the layout container. Examples of this functionality will be shown later in this section.

- *Form Dijit*: the dijit.form.Form dijit, while not completely necessary for use with dijit form elements, will ensure that if an attempt is made to submit a form that does not validate against client-side validations, submission will be halted and validation error messages raised. The form dijit view helper expects the following arguments:

  - `$id`: the container name or DOM ID.

  - `$attribs` (optional): any additional HTML attributes that should be used to render the container div.

  - `$content` (optional): the content to wrap in the form. If none is passed, an empty string will be used.

  The argument order varies from the other dijits in order to keep compatibility with the standard `form()` view helper.

- *Form Elements*: these are designed to be consumed with `Zend_Form`, but can be used standalone within view scripts as well. Each dijit element view helper expects the following arguments:

  - `$id`: the element name or DOM ID.

  - `$value` (optional): the current value of the element.

  - `$params` (optional): dijit-specific parameters. Basically, any non-HTML attribute that can be used to configure a dijit.

  - `$attribs` (optional): any additional HTML attributes that should be used to render the dijit. If the key 'id' is passed in this array, it will be used for the form element DOM id, and `$id` will be used for its name.

  Some elements require more arguments; these will be noted with the individual element helper descriptions.

In order to utilize these view helpers, you need to register the path to the dojo view helpers with your view object.

Example 302. Registering the Dojo View Helper Prefix Path

```
$view->addHelperPath('Zend/Dojo/View/Helper', 'Zend_Dojo_View_Helper');
```

3.2.1. Dijit Layout Elements

The dijit.layout family of elements are for creating custom, predictable layouts for your site. For any questions on general usage, read more about them in the Dojo manual.

All dijit layout elements have the signature **string ($id = null, $content = '', array $params = array(), array $attribs = array())**. In all caess, if you pass no arguments, the helper object itself will be returned. This gives you access to the `captureStart()` and `captureEnd()` methods, which allow you to capture content instead of passing it to the layout container.

- *AccordionContainer*: dijit.layout.AccordionContainer. Stack all panes together vertically; clicking on a pane titlebar will expand and display that particular pane.

```
<?php echo $view->accordionContainer(
```

```
    'foo',
    $content,
    array(
        'duration' => 200,
    ),
    array(
        'style' => 'width: 200px; height: 300px;',
    ),
); ?>
```

- *AccordionPane*: dijit.layout.AccordionPane. For use within AccordionContainer.

```
<?php echo $view->accordionPane(
    'foo',
    $content,
    array(
        'title' => 'Pane Title',
    ),
    array(
        'style' => 'background-color: lightgray;',
    ),
); ?>
```

- *BorderContainer*: dijit.layout.BorderContainer. Achieve layouts with optionally resizable panes such as you might see in a traditional application.

```
<?php echo $view->borderContainer(
    'foo',
    $content,
    array(
        'design' => 'headline',
    ),
    array(
        'style' => 'width: 100%; height: 100%',
    ),
); ?>
```

- *ContentPane*: dijit.layout.ContentPane. Use inside any container except AccordionContainer.

```
<?php echo $view->contentPane(
    'foo',
    $content,
    array(
        'title'  => 'Pane Title',
        'region' => 'left',
    ),
    array(
        'style' => 'width: 120px; background-color: lightgray;',
    ),
); ?>
```

- *SplitContainer*: dijit.layout.SplitContainer. Allows resizable content panes; deprecated in Dojo in favor of BorderContainer.

```
<?php echo $view->splitContainer(
    'foo',
    $content,
```

```
    array(
        'orientation'  => 'horizontal',
        'sizerWidth'   => 7,
        'activeSizing' => true,
    ),
    array(
        'style' => 'width: 400px; height: 500px;',
    ),
); ?>
```

- *StackContainer*: dijit.layout.StackContainer. All panes within a StackContainer are placed in a stack; build buttons or functionality to reveal one at a time.

```
<?php echo $view->stackContainer(
    'foo',
    $content,
    array(),
    array(
        'style' => 'width: 400px; height: 500px; border: 1px;',
    ),
); ?>
```

- *TabContainer*: dijit.layout.TabContainer. All panes within a TabContainer are placed in a stack, with tabs positioned on one side for switching between them.

```
<?php echo $view->tabContainer(
    'foo',
    $content,
    array(),
    array(
        'style' => 'width: 400px; height: 500px; border: 1px;',
    ),
); ?>
```

The following capture methods are available for all layout containers:

- **captureStart($id, array $params = array(), array $attribs = array())**: begin capturing content to include in a container. `$params` refers to the dijit params to use with the container, while `$attribs` refer to any general HTML attributes to use.

  Containers may be nested when capturing, *so long as no ids are duplicated*.

- `captureEnd($id)`: finish capturing content to include in a container. `$id` should refer to an id previously used with a `captureStart()` call. Returns a string representing the container and its contents, just as if you'd simply passed content to the helper itself.

Example 303. BorderContainer layout dijit example

BorderContainers, particularly when coupled with the ability to capture content, are especially useful for achieving complex layout effects.

```
$view->borderContainer()->captureStart('masterLayout',
                                       array('design' => 'headline'));

echo $view->contentPane(
    'menuPane',
    'This is the menu pane',
    array('region' => 'top'),
    array('style' => 'background-color: darkblue;')
);

echo  $view->contentPane(
    'navPane',
    'This is the navigation pane',
    array('region' => 'left'),
    array('style' => 'width: 200px; background-color: lightblue;')
);

echo $view->contentPane(
    'mainPane',
    'This is the main content pane area',
    array('region' => 'center'),
    array('style' => 'background-color: white;')
);

echo $view->contentPane(
    'statusPane',
    'Status area',
    array('region' => 'bottom'),
    array('style' => 'background-color: lightgray;')
);

echo $view->borderContainer()->captureEnd('masterLayout');
```

3.2.2. Dijit Form Elements

Dojo's form validation and input dijits are in the dijit.form tree. For more information on general usage of these elements, as well as accepted parameters, please visit the dijit.form documentation.

The following dijit form elements are available in Zend Framework. Except where noted, all have the signature **string ($id, $value = '', array $params = array(), array $attribs = array())**.

- *Button*: dijit.form.Button. Display a form button.

```
<?php echo $view->button(
    'foo',
    'Show Me!',
    array('iconClass' => 'myButtons'),
); ?>
```

- *CheckBox*: dijit.form.CheckBox. Display a checkbox. Accepts an optional fifth argument, the array `$checkedOptions`, which may contain either:

- an indexed array with two values, a checked value and unchecked value, in that order; or
- an associative array with the keys 'checkedValue' and 'unCheckedValue'.

If `$checkedOptions` is not provided, 1 and 0 are assumed.

```
<?php echo $view->checkBox(
    'foo',
    'bar',
    array(),
    array(),
    array('checkedValue' => 'foo', 'unCheckedValue' => 'bar')
); ?>
```

- *ComboBox*: dijit.layout.ComboBox. ComboBoxes are a hybrid between a select and a text box with autocompletion. The key difference is that you may type an option that is not in the list of available options, and it will still consider it valid input. It accepts an optional fifth argument, an associative array `$options`; if provided, ComboBox will be rendered as a *select*. Note also that the *label values* of the `$options` array will be returned in the form -- not the values themselves.

  Alternately, you may pass information regarding a dojo.data datastore to use with the element. If provided, the ComboBox will be rendered as a text *input*, and will pull its options via that datastore.

  To specify a datastore, provide one of the following `$params` key combinations:

  - The key 'store', with an array value; the array should contain the keys:
    - *store*: the name of the javascript variable representing the datastore (this could be the name you would like for it to use).
    - *type*: the datastore type to use; e.g., 'dojo.data.ItemFileReadStore'.
    - *params* (optional): an associative array of key/value pairs to use to configure the datastore. The 'url' param is a typical example.
  - The keys:
    - *store*: a string indicating the datastore name to use.
    - *storeType*: a string indicating the datastore dojo.data type to use (e.g., 'dojo.data.ItemFileReadStore').
    - *storeParams*: an associative array of key/value pairs with which to configure the datastore.

```
// As a select element:
echo $view->comboBox(
    'foo',
    'bar',
    array(
        'autocomplete' => false,
    ),
    array(),
    array(
        'foo' => 'Foo',
```

```
        'bar' => 'Bar',
        'baz' => 'Baz',
    )
);

// As a dojo.data-enabled element:
echo $view->comboBox(
    'foo',
    'bar',
    array(
        'autocomplete' => false,
        'store'        => 'stateStore',
        'storeType'    => 'dojo.data.ItemFileReadStore',
        'storeParams'  => array('url' => '/js/states.json'),
    ),
);
```

- *CurrencyTextBox*: dijit.form.CurrencyTextBox. Inherits from ValidationTextBox, and provides client-side validation of currency. It expects that the dijit parameter 'currency' will be provided with an appropriate 3-character currency code. You may also specify any dijit parameters valid for ValidationTextBox and TextBox.

```
echo $view->currencyTextBox(
    'foo',
    '$25.00',
    array('currency' => 'USD'),
    array('maxlength' => 20)
);
```

Issues with Builds

There are currently known issues with using CurrencyTextBox with build layers. A known work-around is to ensure that your document's Content-Type http-equiv meta tag sets the character set to utf-8, which you can do by calling:

```
$view->headMeta()->appendHttpEquiv('Content-Type',
                                   'text/html; charset=utf-8');
```

This will mean, of course, that you will need to ensure that the `headMeta()` placeholder is echoed in your layout script.

- *DateTextBox*: dijit.form.DateTextBox. Inherits from ValidationTextBox, and provides both client-side validation of dates, as well as a dropdown calendar from which to select a date. You may specify any dijit parameters available to ValidationTextBox or TextBox.

```
echo $view->dateTextBox(
    'foo',
    '2008-07-11',
    array('required' => true)
);
```

- *Editor*: dijit.Editor. Provides a WYSIWYG editor via which users may create or edit content. **dijit.Editor** is a pluggable, extensible editor with a variety of parameters you can utilize for customization; see the dijit.Editor documentation for more details.

```
echo $view->editor('foo');
```

Editor Dijit uses div by default

The Editor dijit uses an HTML DIV by default. The `dijit._editor.RichText` documentation indicates that having it built on an HTML TEXTAREA can potentially have security implications.

In order to allow graceful degradation in environments where Javascript is unavailable, `Zend_Dojo_View_Helper_Editor` also wraps a textarea within a noscript tag; the content of this textarea will be properly escaped to avoid security vulnerability vectors.

- *FilteringSelect*: dijit.form.FilteringSelect. Similar to ComboBox, this is a select/text hybrid that can either render a provided list of options or those fetched via a dojo.data datastore. Unlike ComboBox, however, FilteringSelect does not allow typing in an option not in its list. Additionally, it operates like a standard select in that the option values, not the labels, are returned when the form is submitted.

  Please see the information above on ComboBox for examples and available options for defining datastores.

- *HorizontalSlider* and *VerticalSlider*: dijit.form.HorizontalSlider and dijit.form.VerticalSlider. Sliders allow are UI widgets for selecting numbers in a given range; these are horizontal and vertical variants.

  At their most basic, they require the dijit parameters 'minimum', 'maximum', and 'discreteValues'. These define the range of values. Other common options are:

  - 'intermediateChanges' can be set to indicate whether or not to fire onChange events while the handle is being dragged.
  - 'clickSelect' can be set to allow clicking a location on the slider to set the value.
  - 'pageIncrement' can specify the value by which to increase/decrease when pageUp and pageDown are used.
  - 'showButtons' can be set to allow displaying buttons on either end of the slider for manipulating the value.

  The Zend Framework implementation creates a hidden element to store the value of the slider.

  You may optionally desire to show a rule or labels for the slider. To do so, you will assign one or more of the dijit params 'topDecoration' and/or 'bottomDecoration' (HorizontalSlider) or 'leftDecoration' and/or 'rightDecoration' (VerticalSlider). Each of these expects the following options:

  - *container*: name of the container.
  - *labels* (optional): an array of labels to utilize. Use empty strings on either end to provide labels for inner values only. Required when specifying one of the 'Labels' dijit variants.
  - *dijit* (optional): one of HorizontalRule, HorizontalRuleLabels, VerticalRule, or VerticalRuleLabels, Defaults to one of the Rule dijits.
  - *params* (optional): dijit params for configuring the Rule dijit in use. Parameters specific to these dijits include:

- *container* (optional): array of parameters and attributes for the rule container.
- *labels* (optional): array of parameters and attributes for the labels list container.

- *attribs* (optional): HTML attributes to use with the rules/labels. This should follow the params option format and be an associative array with the keys 'container' and 'labels'.

```
echo $view->horizontalSlider(
    'foo',
    1,
    array(
        'minimum'             => -10,
        'maximum'             => 10,
        'discreteValues'      => 11,
        'intermediateChanges' => true,
        'showButtons'         => true,
        'topDecoration'       => array(
            'container' => 'topContainer'
            'dijit'     => 'HorizontalRuleLabels',
            'labels'    => array(
                ' ',
                '20%',
                '40%',
                '60%',
                '80%',
                ' ',
            ),
            'params' => array(
                'container' => array(
                    'style' => 'height:1.2em; font-size=75%;color:gray;',
                ),
                'labels' => array(
                    'style' => 'height:1em; font-size=75%;color:gray;',
                ),
            ),
        ),
        'bottomDecoration'    => array(
            'container' => 'bottomContainer'
            'labels'    => array(
                '0%',
                '50%',
                '100%',
            ),
            'params' => array(
                'container' => array(
                    'style' => 'height:1.2em; font-size=75%;color:gray;',
                ),
                'labels' => array(
                    'style' => 'height:1em; font-size=75%;color:gray;',
                ),
            ),
        ),
    )
);
```

- *NumberSpinner*: dijit.form.NumberSpinner. Text box for numeric entry, with buttons for incrementing and decrementing.

Expects either an associative array for the dijit parameter 'constraints', or simply the keys 'min', 'max', and 'places' (these would be the expected entries of the constraints parameter as well). 'places' can be used to indicate how much the number spinner will increment and decrement.

```
echo $view->numberSpinner(
    'foo',
    5,
    array(
        'min'    => -10,
        'max'    => 10,
        'places' => 2,
    ),
    array(
        'maxlenth' => 3,
    )
);
```

- *NumberTextBox*: dijit.form.NumberTextBox. NumberTextBox provides the ability to format and display number entries in a localized fashion, as well as validate numerical entries, optionally against given constraints.

```
echo $view->numberTextBox(
    'foo',
    5,
    array(
        'places' => 4,
        'type'   => 'percent',
    ),
    array(
        'maxlength' => 20,
    )
);
```

- *PasswordTextBox*: dijit.form.ValidationTextBox tied to a password input. PasswordTextBox provides the ability to create password input that adheres to the current dijit theme, as well as allow for client-side validation.

```
echo $view->passwordTextBox(
    'foo',
    '',
    array(
        'required' => true,
    ),
    array(
        'maxlength' => 20,
    )
);
```

- *RadioButton*: dijit.form.RadioButton. A set of options from which only one may be selected. This behaves in every way like a regular radio, but has a look-and-feel consistent with other dijits.

  RadioButton accepts an optional fifth argument, `$options`, an associative array of value/label pairs used as the radio options. You may also pass these as the `$attribs` key options.

```
echo $view->radioButton(
```

```
    'foo',
    'bar',
    array(),
    array(),
    array(
        'foo' => 'Foo',
        'bar' => 'Bar',
        'baz' => 'Baz',
    )
);
```

- *SimpleTextarea*: dijit.form.SimpleTextarea. These act like normal textareas, but are styled using the current dijit theme. You do not need to specify either the rows or columns attributes; use ems or percentages for the width and height, instead.

```
echo $view->simpleTextarea(
    'foo',
    'Start writing here...',
    array(),
    array('style' => 'width: 90%; height: 5ems;')
);
```

- *SubmitButton*: a dijit.form.Button tied to a submit input element. See the Button view helper for more details; the key difference is that this button can submit a form.

- *Textarea*: dijit.form.Textarea. These act like normal textareas, except that instead of having a set number of rows, they expand as the user types. The width should be specified via a style setting.

```
echo $view->textarea(
    'foo',
    'Start writing here...',
    array(),
    array('style' => 'width: 300px;')
);
```

- *TextBox*: dijit.form.TextBox. This element is primarily present to provide a common look-and-feel between various dijit elements, and to provide base functionality for the other TextBox-derived classes (ValidationTextBox, NumberTextBox, CurrencyTextBox, DateTextBox, and TimeTextBox).

  Common dijit parameter flags include 'lowercase' (cast to lowercase), 'uppercase' (cast to UPPERCASE), 'propercase' (cast to Proper Case), and trim (trim leading and trailing whitespace); all accept boolean values. Additionally, you may specifiy the parameters 'size' and 'maxLength'.

```
echo $view->textBox(
    'foo',
    'some text',
    array(
        'trim'       => true,
        'propercase' => true,
        'maxLength'  => 20,
    ),
    array(
        'size' => 20,
    )
```

```
);
```

- *TimeTextBox*: dijit.form.TimeTextBox. Also in the TextBox family, TimeTextBox provides a scrollable drop down selection of times from which a user may select. Dijit parameters allow you to specify the time increments available in the select as well as the visible range of times available.

```
echo $view->timeTextBox(
    'foo',
    '',
    array(
        'am.pm'            => true,
        'visibleIncrement' => 'T00:05:00', // 5-minute increments
        'visibleRange'     => 'T02:00:00', // show 2 hours of increments
    ),
    array(
        'size' => 20,
    )
);
```

- *ValidationTextBox*: dijit.form.ValidateTextBox. Provide client-side validations for a text element. Inherits from TextBox.

  Common dijit parameters include:

  - *invalidMessage*: a message to display when an invalid entry is detected.
  - *promptMessage*: a tooltip help message to use.
  - *regExp*: a regular expression to use to validate the text. Regular expression does not require boundary markers.
  - *required*: whether or not the element is required. If so, and the element is embedded in a dijit.form.Form, it will be flagged as invalid and prevent submission.

```
echo $view->validationTextBox(
    'foo',
    '',
    array(
        'required' => true,
        'regExp'   => '[\w]+',
        'invalidMessage' => 'No spaces or non-word characters allowed',
        'promptMessage'  => 'Single word consisting of alphanumeric ' .
                            'characters and underscores only',
    ),
    array(
        'maxlength' => 20,
    )
);
```

3.2.3. Custom Dijits

If you delve into Dojo much at all, you'll find yourself writing custom dijits, or using experimental dijits from Dojox. While Zend Framework cannot support every dijit directly, it does provide some rudimentary support for arbitrary dijit types via the `CustomDijit` view helper.

The `CustomDijit` view helper's API is exactly that of any other dijit, with one major difference: the third "params" argument *must* contain the attribute "dojotype". The value of this attribute should be the Dijit class you plan to use.

`CustomDijit` extends the base `DijitContainer` view helper, which also allows it to capture content (using the `captureStart()`/`captureEnd()` pair of methods). `captureStart()` also expects that you pass the "dojoType" attribute to its "params" argument.

Example 304. Using CustomDijit to render a dojox.layout.ContentPane

dojox.layout.ContentPane is a next-generation iteration of **dijit.layout.ContentPane**, and provides a superset of that class's capabilities. Until it's functionality stabilizes, it will continue to live in Dojox. However, if you want to use it in Zend Framework today, you can, using the `CustomDijit` view helper.

At its most basic, you can do the following:

```
<?php echo $this->customDijit(
    'foo',
    $content,
    array(
        'dojoType' => 'dojox.layout.ContentPane',
        'title'    => 'Custom pane',
        'region'   => 'center'
    )
); ?>
```

If you wanted to capture content instead, simply use the `captureStart()` method, and pass the "dojoType" to the "params" argument:

```
<?php $this->customDijit()->captureStart(
    'foo',
    array(
        'dojoType' => 'dojox.layout.ContentPane',
        'title'    => 'Custom pane',
        'region'   => 'center'
    )
); ?>
This is the content of the pane
<?php echo $this->customDijit()->captureEnd('foo'); ?>
```

You can also extend `CustomDijit` easily to create support for your own custom dijits. As an example, if you extended **dijit.layout.ContentPane** to create your own **foo.ContentPane** class, you could create the following helper to support it:

```
class My_View_Helper_FooContentPane
    extends Zend_Dojo_View_Helper_CustomDijit
{
    protected $_defaultDojoType = 'foo.ContentPane';

    public function fooContentPane(
        $id = null, $value = null,
        array $params = array(), array $attribs = array()
    ) {
        return $this->customDijit($id, $value, $params, $attribs);
    }
}
```

As long as your custom dijit follows the same basic API as official dijits, using or extending `CustomDijit` should work correctly.

4. Dojo Form Elements and Decorators

Building on the dijit view helpers, the `Zend_Dojo_Form` family of classes provides the ability to utilize Dijits natively within your forms.

There are three options for utilizing the Dojo form elements with your forms:

- Use `Zend_Dojo::enableForm()`. This will add plugin paths for decorators and elements to all attached form items, recursively. Additionally, it will dojo-enable the view object. Note, however, that any sub forms you attach *after* this call will also need to be passed through `Zend_Dojo::enableForm()`.
- Use the Dojo-specific form and subform implementations, `Zend_Dojo_Form` and `Zend_Dojo_Form_SubForm` respectively. These can be used as drop-in replacements for `Zend_Form` and `Zend_Form_SubForm`, contain all the appropriate decorator and element paths, set a Dojo-specific default DisplayGroup class, and dojo-enable the view.
- Last, and most tedious, you can set the appropriate decorator and element paths yourself, set the default DisplayGroup class, and dojo-enable the view. Since `Zend_Dojo::enableForm()` does this already, there's little reason to go this route.

Example 305. Enabling Dojo in your existing forms

"But wait," you say; "I'm already extending `Zend_Form` with my own custom form class! How can I Dojo-enable it?"'

First, and easiest, simply change from extending `Zend_Form` to extending `Zend_Dojo_Form`, and update any places where you instantiate `Zend_Form_SubForm` to instantiate `Zend_Dojo_Form_SubForm`.

A second approach is to call `Zend_Dojo::enableForm()` within your custom form's `init()` method; when the form definition is complete, loop through all SubForms to dojo-enable them:

```
class My_Form_Custom extends Zend_Form
{
    public function init()
    {
        // Dojo-enable the form:
        Zend_Dojo::enableForm($this);

        // ... continue form definition from here

        // Dojo-enable all sub forms:
        foreach ($this->getSubForms() as $subForm) {
            Zend_Dojo::enableForm($subForm);
        }
    }
}
```

Usage of the dijit-specific form decorators and elements is just like using any other form decorator or element.

4.1. Dijit-Specific Form Decorators

Most form elements can use the DijitElement decorator, which will grab the dijit parameters from the elements, and pass these and other metadata to the view helper specified by the element. For decorating forms, sub forms, and display groups, however, there are a set of decorators corresponding to the various layout dijits.

All dijit decorators look for the dijitParams property of the given element being decorated, and push them as the `$params` array to the dijit view helper being used; these are then separated from any other properties so that no duplication of information occurs.

4.1.1. DijitElement Decorator

Just like the ViewHelper decorator, DijitElement expects a helper property in the element which it will then use as the view helper when rendering. Dijit parameters will typically be pulled directly from the element, but may also be passed in as options via the dijitParams key (the value of that key should be an associative array of options).

It is important that each element have a unique ID (as fetched from the element's `getId()` method). If duplicates are detected within the `dojo()` view helper, the decorator will trigger a notice, but then create a unique ID by appending the return of `uniqid()` to the identifier.

Standard usage is to simply associate this decorator as the first in your decorator chain, with no additional options.

Example 306. DijitElement Decorator Usage

```
$element->setDecorators(array(
    'DijitElement',
    'Errors',
    'Label',
    'ContentPane',
));
```

4.1.2. DijitForm Decorator

The DijitForm decorator is very similar to the Form decorator; in fact, it can be used basically interchangeably with it, as it utilizes the same view helper name ('form').

Since **dijit.form.Form** does not require any dijit parameters for configuration, the main difference is that the dijit form view helper require that a DOM ID is passed to ensure that programmatic dijit creation can work. The decorator ensures this, by passing the form name as the identifier.

4.1.3. DijitContainer-based Decorators

The `DijitContainer` decorator is actually an abstract class from which a variety of other decorators derive. It offers the same functionality of DijitElement, with the addition of title support. Many layout dijits require or can utilize a title; DijitContainer will utilize the element's legend property, if available, and can also utilize either the 'legend' or 'title' decorator option, if passed. The title will be translated if a translation adapter with a corresponding translation is present.

The following is a list of decorators that inherit from `DijitContainer`:

- AccordionContainer
- AccordionPane
- BorderContainer
- ContentPane
- SplitContainer
- StackContainer
- TabContainer

Example 307. DijitContainer Decorator Usage

```
// Use a TabContainer for your form:
$form->setDecorators(array(
    'FormElements',
    array('TabContainer', array(
        'id'          => 'tabContainer',
        'style'       => 'width: 600px; height: 500px;',
        'dijitParams' => array(
            'tabPosition' => 'top'
        ),
    )),
    'DijitForm',
));

// Use a ContentPane in your sub form (which can be used with all but
// AccordionContainer):
$subForm->setDecorators(array(
    'FormElements',
    array('HtmlTag', array('tag' => 'dl')),
    'ContentPane',
));
```

4.2. Dijit-Specific Form Elements

Each form dijit for which a view helper is provided has a corresponding `Zend_Form` element. All of them have the following methods available for manipulating dijit parameters:

- `setDijitParam($key, $value)`: set a single dijit parameter. If the dijit parameter already exists, it will be overwritten.
- `setDijitParams(array $params)`: set several dijit parameters at once. Any passed parameters matching those already present will overwrite.
- `hasDijitParam($key)`: If a given dijit parameter is defined and present, return `TRUE`, otherwise return FALSE.
- `getDijitParam($key)`: retrieve the given dijit parameter. If not available, a `NULL` value is returned.
- `getDijitParams()`: retrieve all dijit parameters.
- `removeDijitParam($key)`: remove the given dijit parameter.
- `clearDijitParams()`: clear all currently defined dijit parameters.

Dijit parameters are stored in the dijitParams public property. Thus, you can dijit-enable an existing form element simply by setting this property on the element; you simply will not have the above accessors to facilitate manipulating the parameters.

Additionally, dijit-specific elements implement a different list of decorators, corresponding to the following:

```
$element->addDecorator('DijitElement')
        ->addDecorator('Errors')
        ->addDecorator('HtmlTag', array('tag' => 'dd'))
```

```
        ->addDecorator('Label', array('tag' => 'dt'));
```

In effect, the DijitElement decorator is used in place of the standard ViewHelper decorator.

Finally, the base Dijit element ensures that the Dojo view helper path is set on the view.

A variant on DijitElement, DijitMulti, provides the functionality of the `Multi` abstract form element, allowing the developer to specify 'multiOptions' -- typically select options or radio options.

The following dijit elements are shipped in the standard Zend Framework distribution.

4.2.1. Button

While not deriving from the standard Button element, it does implement the same functionality, and can be used as a drop-in replacement for it. The following functionality is exposed:

- `getLabel()` will utilize the element name as the button label if no name is provided. Additionally, it will translate the name if a translation adapter with a matching translation message is available.
- `isChecked()` determines if the value submitted matches the label; if so, it returns `TRUE`. This is useful for determining which button was used when a form was submitted.

Additionally, only the decorators `DijitElement` and `DtDdWrapper` are utilized for Button elements.

Example 308. Example Button dijit element usage

```
$form->addElement(
    'Button',
    'foo',
    array(
        'label' => 'Button Label',
    )
);
```

4.2.2. CheckBox

While not deriving from the standard Checkbox element, it does implement the same functionality. This means that the following methods are exposed:

- `setCheckedValue($value)`: set the value to use when the element is checked.
- `getCheckedValue()`: get the value of the item to use when checked.
- `setUncheckedValue($value)`: set the value of the item to use when it is unchecked.
- `getUncheckedValue()`: get the value of the item to use when it is unchecked.
- `setChecked($flag)`: mark the element as checked or unchecked.
- `isChecked()`: determine if the element is currently checked.

Example 309. Example CheckBox dijit element usage

```
$form->addElement(
    'CheckBox',
    'foo',
    array(
        'label'          => 'A check box',
        'checkedValue'   => 'foo',
        'uncheckedValue' => 'bar',
        'checked'        => true,
    )
);
```

4.2.3. ComboBox and FilteringSelect

As noted in the ComboBox dijit view helper documentation, ComboBoxes are a hybrid between select and text input, allowing for autocompletion and the ability to specify an alternate to the options provided. FilteringSelects are the same, but do not allow arbitrary input.

ComboBoxes return the label values

ComboBoxes return the label values, and not the option values, which can lead to a disconnect in expectations. For this reason, ComboBoxes do not auto-register an `InArray` validator (though FilteringSelects do).

The ComboBox and FilteringSelect form elements provide accessors and mutators for examining and setting the select options as well as specifying a dojo.data datastore (if used). They extend from DijitMulti, which allows you to specify select options via the `setMultiOptions()` and `setMultiOption()` methods. In addition, the following methods are available:

- `getStoreInfo()`: get all datastore information currently set. Returns an empty array if no data is currently set.
- `setStoreId($identifier)`: set the store identifier variable (usually referred to by the attribute 'jsId' in Dojo). This should be a valid javascript variable name.
- `getStoreId()`: retrieve the store identifier variable name.
- `setStoreType($dojoType)`: set the datastore class to use; e.g., "dojo.data.ItemFileReadStore".
- `getStoreType()`: get the dojo datastore class to use.
- `setStoreParams(array $params)`: set any parameters used to configure the datastore object. As an example, dojo.data.ItemFileReadStore datastore would expect a 'url' parameter pointing to a location that would return the dojo.data object.
- `getStoreParams()`: get any datastore parameters currently set; if none, an empty array is returned.
- `setAutocomplete($flag)`: indicate whether or not the selected item will be used when the user leaves the element.
- `getAutocomplete()`: get the value of the autocomplete flag.

By default, if no dojo.data store is registered with the element, this element registers an `InArray` validator which validates against the array keys of registered options. You can disable this behavior by either calling `setRegisterInArrayValidator(false)`, or by passing a `FALSE` value to the registerInArrayValidator configuration key.

Example 310. ComboBox dijit element usage as select input

```
$form->addElement(
    'ComboBox',
    'foo',
    array(
        'label'        => 'ComboBox (select)',
        'value'        => 'blue',
        'autocomplete' => false,
        'multiOptions' => array(
            'red'    => 'Rouge',
            'blue'   => 'Bleu',
            'white'  => 'Blanc',
            'orange' => 'Orange',
            'black'  => 'Noir',
            'green'  => 'Vert',
        ),
    )
);
```

Example 311. ComboBox dijit element usage with datastore

```
$form->addElement(
    'ComboBox',
    'foo',
    array(
        'label'       => 'ComboBox (datastore)',
        'storeId'     => 'stateStore',
        'storeType'   => 'dojo.data.ItemFileReadStore',
        'storeParams' => array(
            'url' => '/js/states.txt',
        ),
        'dijitParams' => array(
            'searchAttr' => 'name',
        ),
    )
);
```

The above examples could also utilize `FilteringSelect` instead of `ComboBox`.

4.2.4. CurrencyTextBox

The CurrencyTextBox is primarily for supporting currency input. The currency may be localized, and can support both fractional and non-fractional values.

Internally, CurrencyTextBox derives from NumberTextBox, ValidationTextBox, and TextBox; all methods available to those classes are available. In addition, the following constraint methods can be used:

- `setCurrency($currency)`: set the currency type to use; should follow the ISO-4217 specification.
- `getCurrency()`: retrieve the current currency type.

- `setSymbol($symbol)`: set the 3-letter ISO-4217 currency symbol to use.
- `getSymbol()`: get the current currency symbol.
- `setFractional($flag)`: set whether or not the currency should allow for fractional values.
- `getFractional()`: retrieve the status of the fractional flag.

Example 312. Example CurrencyTextBox dijit element usage

```
$form->addElement(
    'CurrencyTextBox',
    'foo',
    array(
        'label'          => 'Currency:',
        'required'       => true,
        'currency'       => 'USD',
        'invalidMessage' => 'Invalid amount. ' .
                            'Include dollar sign, commas, and cents.',
        'fractional'     => false,
    )
);
```

4.2.5. DateTextBox

DateTextBox provides a calendar drop-down for selecting a date, as well as client-side date validation and formatting.

Internally, DateTextBox derives from ValidationTextBox and TextBox; all methods available to those classes are available. In addition, the following methods can be used to set individual constraints:

- `setAmPm($flag)` and `getAmPm()`: Whether or not to use AM/PM strings in times.
- `setStrict($flag)` and `getStrict()`: whether or not to use strict regular expression matching when validating input. If FALSE, which is the default, it will be lenient about whitespace and some abbreviations.
- `setLocale($locale)` and `getLocale()`: Set and retrieve the locale to use with this specific element.
- `setDatePattern($pattern)` and `getDatePattern()`: provide and retrieve the unicode date format pattern for formatting the date.
- `setFormatLength($formatLength)` and `getFormatLength()`: provide and retrieve the format length type to use; should be one of "long", "short", "medium" or "full".
- `setSelector($selector)` and `getSelector()`: provide and retrieve the style of selector; should be either "date" or "time".

Example 313. Example DateTextBox dijit element usage

```
$form->addElement(
    'DateTextBox',
    'foo',
    array(
        'label'          => 'Date:',
        'required'       => true,
        'invalidMessage' => 'Invalid date specified.',
        'formatLength'   => 'long',
    )
);
```

4.2.6. Editor

Editor provides a WYSIWYG editor that can be used to both create and edit rich HTML content. dijit.Editor is pluggable and may be extended with custom plugins if desired; see the dijit.Editor documentation for more details.

The Editor form element provides a number of accessors and mutators for manipulating various dijit parameters, as follows:

- *captureEvents* are events that connect to the editing area itself. The following accessors and mutators are available for manipulating capture events:
  - `addCaptureEvent($event)`
  - `addCaptureEvents(array $events)`
  - `setCaptureEvents(array $events)`
  - `getCaptureEvents()`
  - `hasCaptureEvent($event)`
  - `removeCaptureEvent($event)`
  - `clearCaptureEvents()`
- *events* are standard DOM events, such as onClick, onKeyUp, etc. The following accessors and mutators are available for manipulating events:
  - `addEvent($event)`
  - `addEvents(array $events)`
  - `setEvents(array $events)`
  - `getEvents()`
  - `hasEvent($event)`
  - `removeEvent($event)`
  - `clearEvents()`
- *plugins* add functionality to the Editor -- additional tools for the toolbar, additional styles to allow, etc. The following accessors and mutators are available for manipulating plugins:

    - `addPlugin($plugin)`
    - `addPlugins(array $plugins)`
    - `setPlugins(array $plugins)`
    - `getPlugins()`
    - `hasPlugin($plugin)`
    - `removePlugin($plugin)`
    - `clearPlugins()`

- *editActionInterval* is used to group events for undo operations. By default, this value is 3 seconds. The method `setEditActionInterval($interval)` may be used to set the value, while `getEditActionInterval()` will retrieve it.

- *focusOnLoad* is used to determine whether this particular editor will receive focus when the page has loaded. By default, this is `FALSE`. The method `setFocusOnLoad($flag)` may be used to set the value, while `getFocusOnLoad()` will retrieve it.

- *height* specifies the height of the editor; by default, this is 300px. The method `setHeight($height)` may be used to set the value, while `getHeight()` will retrieve it.

- *inheritWidth* is used to determine whether the editor will use the parent container's width or simply default to 100% width. By default, this is `FALSE` (i.e., it will fill the width of the window). The method `setInheritWidth($flag)` may be used to set the value, while `getInheritWidth()` will retrieve it.

- *minHeight* indicates the minimum height of the editor; by default, this is 1em. The method `setMinHeight($height)` may be used to set the value, while `getMinHeight()` will retrieve it.

- *styleSheets* indicate what additional CSS stylesheets should be used to affect the display of the Editor. By default, none are registered, and it inherits the page styles. The following accessors and mutators are available for manipulating editor stylesheets:

    - `addStyleSheet($styleSheet)`
    - `addStyleSheets(array $styleSheets)`
    - `setStyleSheets(array $styleSheets)`
    - `getStyleSheets()`
    - `hasStyleSheet($styleSheet)`
    - `removeStyleSheet($styleSheet)`
    - `clearStyleSheets()`

Example 314. Example Editor dijit element usage

```
$form->addElement('editor', 'content', array(
    'plugins'            => array('undo', '|', 'bold', 'italic'),
    'editActionInterval' => 2,
    'focusOnLoad'        => true,
    'height'             => '250px',
    'inheritWidth'       => true,
    'styleSheets'        => array('/js/custom/editor.css'),
));
```

Editor Dijit uses div by default

The Editor dijit uses an HTML DIV by default. The `dijit._editor.RichText` documentation indicates that having it built on an HTML TEXTAREA can potentially have security implications.

That said, there may be times when you want an Editor widget that can gracefully degrade to a TEXTAREA. In such situations, you can do so by setting the `degrade` property to TRUE:

```
// At instantiation:
$editor = new Zend_Dojo_Form_Element_Editor('foo', array(
    'degrade' => true,
));

// Construction via the form:
$form->addElement('editor', 'content', array(
    'degrade' => true,
));

// Or after instantiation:
$editor->degrade = true;
```

4.2.7. HorizontalSlider

HorizontalSlider provides a slider UI widget for selecting a numeric value in a range. Internally, it sets the value of a hidden element which is submitted by the form.

HorizontalSlider derives from the abstract Slider dijit element. Additionally, it has a variety of methods for setting and configuring slider rules and rule labels.

- `setTopDecorationDijit($dijit)` and `setBottomDecorationDijit($dijit)`: set the name of the dijit to use for either the top or bottom of the slider. This should not include the "dijit.form." prefix, but rather only the final name -- one of "HorizontalRule" or "HorizontalRuleLabels".

- `setTopDecorationContainer($container)` and `setBottomDecorationContainer($container)`: specify the name to use for the container element of the rules; e.g. 'topRule', 'topContainer', etc.

- `setTopDecorationLabels(array $labels)` and `setBottomDecorationLabels(array $labels)`: set the labels to use for one of the RuleLabels dijit types. These should be an indexed array; specify a single empty space to skip a given label position (such as the beginning or end).

- `setTopDecorationParams(array $params)` and `setBottomDecorationParams(array $params)`: dijit parameters to use when configuring the given Rule or RuleLabels dijit.

- `setTopDecorationAttribs(array $attribs)` and `setBottomDecorationAttribs(array $attribs)`: HTML attributes to specify for the given Rule or RuleLabels HTML element container.

- `getTopDecoration()` and `getBottomDecoration()`: retrieve all metadata for a given Rule or RuleLabels definition, as provided by the above mutators.

Example 315. Example HorizontalSlider dijit element usage

The following will create a horizontal slider selection with integer values ranging from -10 to 10. The top will have labels at the 20%, 40%, 60%, and 80% marks. The bottom will have rules at 0, 50%, and 100%. Each time the value is changed, the hidden element storing the value will be updated.

```
$form->addElement(
    'HorizontalSlider',
    'horizontal',
    array(
        'label'                     => 'HorizontalSlider',
        'value'                     => 5,
        'minimum'                   => -10,
        'maximum'                   => 10,
        'discreteValues'            => 11,
        'intermediateChanges'       => true,
        'showButtons'               => true,
        'topDecorationDijit'        => 'HorizontalRuleLabels',
        'topDecorationContainer'    => 'topContainer',
        'topDecorationLabels'       => array(
                ' ',
                '20%',
                '40%',
                '60%',
                '80%',
                ' ',
        ),
        'topDecorationParams'      => array(
            'container' => array(
                'style' => 'height:1.2em; font-size=75%;color:gray;',
            ),
            'list' => array(
                'style' => 'height:1em; font-size=75%;color:gray;',
            ),
        ),
        'bottomDecorationDijit'     => 'HorizontalRule',
        'bottomDecorationContainer' => 'bottomContainer',
        'bottomDecorationLabels'    => array(
                '0%',
                '50%',
                '100%',
        ),
        'bottomDecorationParams'   => array(
            'list' => array(
                'style' => 'height:1em; font-size=75%;color:gray;',
            ),
        ),
    )
);
```

4.2.8. NumberSpinner

A number spinner is a text element for entering numeric values; it also includes elements for incrementing and decrementing the value by a set amount.

The following methods are available:

- `setDefaultTimeout($timeout)` and `getDefaultTimeout()`: set and retrieve the default timeout, in milliseconds, between when the button is held pressed and the value is changed.
- `setTimeoutChangeRate($rate)` and `getTimeoutChangeRate()`: set and retrieve the rate, in milliseconds, at which changes will be made when a button is held pressed.
- `setLargeDelta($delta)` and `getLargeDelta()`: set and retrieve the amount by which the numeric value should change when a button is held pressed.
- `setSmallDelta($delta)` and `getSmallDelta()`: set and retrieve the delta by which the number should change when a button is pressed once.
- `setIntermediateChanges($flag)` and `getIntermediateChanges()`: set and retrieve the flag indicating whether or not each value change should be shown when a button is held pressed.
- `setRangeMessage($message)` and `getRangeMessage()`: set and retrieve the message indicating the range of values available.
- `setMin($value)` and `getMin()`: set and retrieve the minimum value possible.
- `setMax($value)` and `getMax()`: set and retrieve the maximum value possible.

Example 316. Example NumberSpinner dijit element usage

```
$form->addElement(
    'NumberSpinner',
    'foo',
    array(
        'value'             => '7',
        'label'             => 'NumberSpinner',
        'smallDelta'        => 5,
        'largeDelta'        => 25,
        'defaultTimeout'    => 500,
        'timeoutChangeRate' => 100,
        'min'               => 9,
        'max'               => 1550,
        'places'            => 0,
        'maxlength'         => 20,
    )
);
```

4.2.9. NumberTextBox

A number text box is a text element for entering numeric values; unlike NumberSpinner, numbers are entered manually. Validations and constraints can be provided to ensure the number stays in a particular range or format.

Internally, NumberTextBox derives from ValidationTextBox and TextBox; all methods available to those classes are available. In addition, the following methods can be used to set individual constraints:

- `setLocale($locale)` and `getLocale()`: specify and retrieve a specific or alternate locale to use with this dijit.
- `setPattern($pattern)` and `getPattern()`: set and retrieve a number pattern format to use to format the number.

- `setType($type)` and `getType()`: set and retrieve the numeric format type to use (should be one of 'decimal', 'percent', or 'currency').
- `setPlaces($places)` and `getPlaces()`: set and retrieve the number of decimal places to support.
- `setStrict($flag)` and `getStrict()`: set and retrieve the value of the strict flag, which indicates how much leniency is allowed in relation to whitespace and non-numeric characters.

Example 317. Example NumberTextBox dijit element usage

```
$form->addElement(
    'NumberTextBox',
    'elevation',
    array(
        'label'          => 'NumberTextBox',
        'required'       => true,
        'invalidMessage' => 'Invalid elevation.',
        'places'         => 0,
        'constraints'    => array(
            'min'    => -20000,
            'max'    => 20000,
        ),
    )
);
```

4.2.10. PasswordTextBox

PasswordTextBox is simply a ValidationTextBox that is tied to a password input; its sole purpose is to allow for a dijit-themed text entry for passwords that also provides client-side validation.

Internally, PasswordTextBox derives from ValidationTextBox and TextBox; all methods available to those classes are available.

Example 318. Example PasswordTextBox dijit element usage

```
$form->addElement(
    'PasswordTextBox',
    'password',
    array(
        'label'          => 'Password',
        'required'       => true,
        'trim'           => true,
        'lowercase'      => true,
        'regExp'         => '^[a-z0-9]{6,}$',
        'invalidMessage' => 'Invalid password; ' .
                            'must be at least 6 alphanumeric characters',
    )
);
```

4.2.11. RadioButton

RadioButton wraps standard radio input elements to provide a consistent look and feel with other dojo dijits.

RadioButton extends from DijitMulti, which allows you to specify select options via the `setMultiOptions()` and `setMultiOption()` methods.

By default, this element registers an `InArray` validator which validates against the array keys of registered options. You can disable this behavior by either calling `setRegisterInArrayValidator(false)`, or by passing a `FALSE` value to the registerInArrayValidator configuration key.

Example 319. Example RadioButton dijit element usage

```
$form->addElement(
    'RadioButton',
    'foo',
    array(
        'label' => 'RadioButton',
        'multiOptions'  => array(
            'foo' => 'Foo',
            'bar' => 'Bar',
            'baz' => 'Baz',
        ),
        'value' => 'bar',
    )
);
```

4.2.12. SimpleTextarea

SimpleTextarea acts primarily like a standard HTML textarea. However, it does not support either the rows or cols settings. Instead, the textarea width should be specified using standard CSS measurements. Unlike Textarea, it will not grow automatically

Example 320. Example SimpleTextarea dijit element usage

```
$form->addElement(
    'SimpleTextarea',
    'simpletextarea',
    array(
        'label'    => 'SimpleTextarea',
        'required' => true,
        'style'    => 'width: 80em; height: 25em;',
    )
);
```

4.2.13. Slider abstract element

Slider is an abstract element from which HorizontalSlider and VerticalSlider both derive. It exposes a number of common methods for configuring your sliders, including:

- `setClickSelect($flag)` and `getClickSelect()`: set and retrieve the flag indicating whether or not clicking the slider changes the value.
- `setIntermediateChanges($flag)` and `getIntermediateChanges()`: set and retrieve the flag indicating whether or not the dijit will send a notification on each slider change event.
- `setShowButtons($flag)` and `getShowButtons()`: set and retrieve the flag indicating whether or not buttons on either end will be displayed; if so, the user can click on these to change the value of the slider.
- `setDiscreteValues($value)` and `getDiscreteValues()`: set and retrieve the number of discrete values represented by the slider.
- `setMaximum($value)` and `getMaximum()`: set the maximum value of the slider.

- `setMinimum($value)` and `getMinimum()`: set the minimum value of the slider.
- `setPageIncrement($value)` and `getPageIncrement()`: set the amount by which the slider will change on keyboard events.

Example usage is provided with each concrete extending class.

4.2.14. SubmitButton

While there is no Dijit named SubmitButton, we include one here to provide a button dijit capable of submitting a form without requiring any additional javascript bindings. It works exactly like the Button dijit.

Example 321. Example SubmitButton dijit element usage

```
$form->addElement(
    'SubmitButton',
    'foo',
    array(
        'required'   => false,
        'ignore'     => true,
        'label'      => 'Submit Button!',
    )
);
```

4.2.15. TextBox

TextBox is included primarily to provide a text input with consistent look-and-feel to the other dijits. However, it also includes some minor filtering and validation capabilities, represented in the following methods:

- `setLowercase($flag)` and `getLowercase()`: set and retrieve the flag indicating whether or not input should be cast to lowercase.
- `setPropercase($flag)` and `getPropercase()`: set and retrieve the flag indicating whether or not the input should be cast to Proper Case.
- `setUppercase($flag)` and `getUppercase()`: set and retrieve the flag indicating whether or not the input should be cast to UPPERCASE.
- `setTrim($flag)` and `getTrim()`: set and retrieve the flag indicating whether or not leading or trailing whitespace should be stripped.
- `setMaxLength($length)` and `getMaxLength()`: set and retrieve the maximum length of input.

Example 322. Example TextBox dijit element usage

```
$form->addElement(
    'TextBox',
    'foo',
    array(
        'value'      => 'some text',
        'label'      => 'TextBox',
        'trim'       => true,
        'propercase' => true,
    )
);
```

4.2.16. Textarea

Textarea acts primarily like a standard HTML textarea. However, it does not support either the rows or cols settings. Instead, the textarea width should be specified using standard CSS measurements; rows should be omitted entirely. The textarea will then grow vertically as text is added to it.

Example 323. Example Textarea dijit element usage

```
$form->addElement(
    'Textarea',
    'textarea',
    array(
        'label'    => 'Textarea',
        'required' => true,
        'style'    => 'width: 200px;',
    )
);
```

4.2.17. TimeTextBox

TimeTextBox is a text input that provides a drop-down for selecting a time. The drop-down may be configured to show a certain window of time, with specified increments.

Internally, TimeTextBox derives from DateTextBox, ValidationTextBox and TextBox; all methods available to those classes are available. In addition, the following methods can be used to set individual constraints:

- `setTimePattern($pattern)` and `getTimePattern()`: set and retrieve the unicode time format pattern for formatting the time.
- `setClickableIncrement($format)` and `getClickableIncrement()`: set the ISO-8601 string representing the amount by which every clickable element in the time picker increases.
- `setVisibleIncrement($format)` and `getVisibleIncrement()`: set the increment visible in the time chooser; must follow ISO-8601 formats.
- `setVisibleRange($format)` and `getVisibleRange()`: set and retrieve the range of time visible in the time chooser at any given moment; must follow ISO-8601 formats.

Example 324. Example TimeTextBox dijit element usage

The following will create a TimeTextBox that displays 2 hours at a time, with increments of 10 minutes.

```
$form->addElement(
    'TimeTextBox',
    'foo',
    array(
        'label'              => 'TimeTextBox',
        'required'           => true,
        'visibleRange'       => 'T04:00:00',
        'visibleIncrement'   => 'T00:10:00',
        'clickableIncrement' => 'T00:10:00',
    )
);
```

4.2.18. ValidationTextBox

ValidationTextBox provides the ability to add validations and constraints to a text input. Internally, it derives from TextBox, and adds the following accessors and mutators for manipulating dijit parameters:

- `setInvalidMessage($message)` and `getInvalidMessage()`: set and retrieve the tooltip message to display when the value does not validate.
- `setPromptMessage($message)` and `getPromptMessage()`: set and retrieve the tooltip message to display for element usage.
- `setRegExp($regexp)` and `getRegExp()`: set and retrieve the regular expression to use for validating the element. The regular expression does not need boundaries (unlike PHP's preg* family of functions).
- `setConstraint($key, $value)` and `getConstraint($key)`: set and retrieve additional constraints to use when validating the element; used primarily with subclasses. Constraints are stored in the 'constraints' key of the dijit parameters.
- `setConstraints(array $constraints)` and `getConstraints()`: set and retrieve individual constraints to use when validating the element; used primarily with subclasses.
- `hasConstraint($key)`: test whether a given constraint exists.
- `removeConstraint($key)` and `clearConstraints()`: remove an individual or all constraints for the element.

Example 325. Example ValidationTextBox dijit element usage

The following will create a ValidationTextBox that requires a single string consisting solely of word characters (i.e., no spaces, most punctuation is invalid).

```
$form->addElement(
    'ValidationTextBox',
    'foo',
    array(
        'label'          => 'ValidationTextBox',
        'required'       => true,
        'regExp'         => '[\w]+',
        'invalidMessage' => 'Invalid non-space text.',
    )
);
```

4.2.19. VerticalSlider

VerticalSlider is the sibling of HorizontalSlider, and operates in every way like that element. The only real difference is that the 'top*' and 'bottom*' methods are replaced by 'left*' and 'right*', and instead of using HorizontalRule and HorizontalRuleLabels, VerticalRule and VerticalRuleLabels should be used.

Example 326. Example VerticalSlider dijit element usage

The following will create a vertical slider selection with integer values ranging from -10 to 10. The left will have labels at the 20%, 40%, 60%, and 80% marks. The right will have rules at 0, 50%, and 100%. Each time the value is changed, the hidden element storing the value will be updated.

```
$form->addElement(
    'VerticalSlider',
    'foo',
    array(
        'label'                    => 'VerticalSlider',
        'value'                    => 5,
        'style'                    => 'height: 200px; width: 3em;',
        'minimum'                  => -10,
        'maximum'                  => 10,
        'discreteValues'           => 11,
        'intermediateChanges'      => true,
        'showButtons'              => true,
        'leftDecorationDijit'      => 'VerticalRuleLabels',
        'leftDecorationContainer'  => 'leftContainer',
        'leftDecorationLabels'     => array(
                ' ',
                '20%',
                '40%',
                '60%',
                '80%',
                ' ',
        ),
        'rightDecorationDijit' => 'VerticalRule',
        'rightDecorationContainer' => 'rightContainer',
        'rightDecorationLabels' => array(
                '0%',
                '50%',
                '100%',
        ),
    )
);
```

4.3. D

```
                        ' ',
                    ),
                    'topDecorationParams' => array(
                        'container' => array(
                            'style' => 'height:1.2em; ' .
                                       'font-size=75%;color:gray;',
                        ),
                        'list' => array(
                            'style' => 'height:1em; ' .
                                       'font-size=75%;color:gray;',
                        ),
                    ),
                    'bottomDecorationDijit' => 'HorizontalRule',
                    'bottomDecorationContainer' => 'bottomContainer',
                    'bottomDecorationLabels' => array(
                            '0%',
                            '50%',
                            '100%',
                    ),
                    'bottomDecorationParams' => array(
                        'list' => array(
                            'style' => 'height:1em; ' .
                                       'font-size=75%;color:gray;',
                        ),
                    ),
                )
            )
            ->addElement(
                'VerticalSlider',
                'vertical',
                array(
                    'label' => 'VerticalSlider',
                    'value' => 5,
                    'style' => 'height: 200px; width: 3em;',
                    'minimum' => -10,
                    'maximum' => 10,
                    'discreteValues' => 11,
                    'intermediateChanges' => true,
                    'showButtons' => true,
                    'leftDecorationDijit' => 'VerticalRuleLabels',
                    'leftDecorationContainer' => 'leftContainer',
                    'leftDecorationLabels' => array(
                            ' ',
                            '20%',
                            '40%',
                            '60%',
                            '80%',
                            ' ',
                    ),
                    'rightDecorationDijit' => 'VerticalRule',
                    'rightDecorationContainer' => 'rightContainer',
                    'rightDecorationLabels' => array(
                            '0%',
                            '50%',
                            '100%',
                    ),
                )
            );

        $this->addSubForm($textForm, 'textboxtab')
             ->addSubForm($editorForm, 'editortab')
             ->addSubForm($toggleForm, 'toggletab')
             ->addSubForm($selectForm, 'selecttab')
             ->addSubForm($sliderForm, 'slidertab');
    }
}
```

Example 328. Modifying an existing form to utilize Dojo

Existing forms can be modified to utilize Dojo as well, by use of the `Zend_Dojo::enableForm()` static method.

This first example shows decorating an existing form instance:

```
$form = new My_Custom_Form();
Zend_Dojo::enableForm($form);
$form->addElement(
'ComboBox',
'query',
array(
    'label'        => 'Color:',
    'value'        => 'blue',
    'autocomplete' => false,
    'multiOptions' => array(
        'red'    => 'Rouge',
        'blue'   => 'Bleu',
        'white'  => 'Blanc',
        'orange' => 'Orange',
        'black'  => 'Noir',
        'green'  => 'Vert',
    ),
)
);
```

Alternately, you can make a slight tweak to your form initialization:

```
class My_Custom_Form extends Zend_Form
{
    public function init()
    {
        Zend_Dojo::enableForm($this);

        // ...
    }
}
```

Of course, if you can do that... you could and should simply alter the class to inherit from `Zend_Dojo_Form`, which is a drop-in replacement of `Zend_Form` that's already Dojo-enabled...

5. Zend_Dojo build layer support

5.1. Introduction

Dojo build layers provide a clean path from development to production when using Dojo for your UI layer. In development, you can have load-on-demand, rapid application prototyping; a build layer takes all Dojo dependencies and compiles them to a single file, optionally stripping whitespace and comments, and performing code heuristics to allow further minification of variable names. Additionally, it can do CSS minification.

In order to create a build layer, you would traditionally create a JavaScript file that has **dojo.require** statements for each dependency, and optionally some additional code that might run when the script is loaded. As an example:

```
dojo.provide("custom.main");

dojo.require("dijit.layout.TabContainer");
dojo.require("dijit.layout.ContentPane");
dojo.require("dijit.form.Form");
dojo.require("dijit.form.Button");
dojo.require("dijit.form.TextBox");
```

This script is generally referred to as a "layer" script.

Then, in your application's layout, you'd instruct Dojo to load this module:

```
<html>
<head>
    <script type="text/javascript" src="/js/dojo/dojo.js"></script>
    <script type="text/javascript">
        dojo.registerModulePath("custom", "../custom/");
        dojo.require("custom.main");
    </script>
```

If you use `Zend_Dojo` to do this, you'd do the following:

```
$view->dojo()->registerModulePath('custom', '../custom/')
             ->requireModule('custom.main');
```

But since `Zend_Dojo` aggregates your various **dojo.require** statements, how do you create your layer script? You could open each page and view the generated **dojo.require** statements, and cut and paste them into a layer script file manually.

However, a better solution exists: since `Zend_Dojo` aggregates this information already, you can simply pull that information and build your layer file. This is the purpose of `Zend_Dojo_BuildLayer`.

5.2. Generating Custom Module Layers with Zend_Dojo_BuildLayer

At its simplest, you simply instantiate `Zend_Dojo_BuildLayer`, feed it the view object and the name of your custom module layer, and have it generate the content of the layer file; it is up to you to then write it to disk.

As an example, let's say you wanted to create the module layer "custom.main". Assuming you follow the recommended project directory structure, and that you are storing your JavaScript files under `public/js/`, you could do the following:

```
$build = new Zend_Dojo_BuildLayer(array(
    'view'      => $view,
    'layerName' => 'custom.main',
));

$layerContents = $build->generateLayerScript();
$filename      = APPLICATION_PATH . '/../public/js/custom/main.js';
if (!dir_exists(dirname($filename))) {
    mkdir(dirname($filename));
}
file_put_contents($filename, $layerContents);
```

When should you do the above? For it to work correctly, you need to do it after all view scripts and the layout have been rendered, to ensure that the `dojo()` helper is fully populated. One easy way to do so is using a front controller plugin, with a `dispatchLoopShutdown()` hook:

```
class App_Plugin_DojoLayer extends Zend_Controller_Plugin_Abstract
{
    public $layerScript = APPLICATION_PATH . '/../public/js/custom/main.js';
    protected $_build;

    public function dispatchLoopShutdown()
    {
        if (!file_exists($this->layerScript)) {
            $this->generateDojoLayer();
        }
    }

    public function getBuild()
    {
        $viewRenderer = Zend_Controller_Action_HelperBroker::getStaticHelper(
            'ViewRenderer'
        );
        $viewRenderer->initView();
        if (null === $this->_build) {
            $this->_build = new Zend_Dojo_BuildLayer(array(
                'view'      => $viewRenderer->view,
                'layerName' => 'custom.main',
            ));
        }
        return $this->_build;
    }

    public function generateDojoLayer()
    {
        $build = $this->getBuild();
        $layerContents = $build->generateLayerScript();
        if (!dir_exists(dirname($this->layerScript))) {
            mkdir(dirname($this->layerScript));
        }
        file_put_contents($this->layerScript, $layerContents);
    }
}
```

Do not generate the layer on every page

It's tempting to generate the layer script on each and every page. However, this is resource intensive, as it must write to the disk on each page. Additionally, since the mtime of the file will keep changing, you will get no benefits of client-side caching. Write the file once.

5.2.1. BuildLayer options

The above functionality will suffice for most situations. For those needing more customization, a variety of options may be invoked.

5.2.1.1. Setting the view object

While the view object may be passed during instantiation, you may also pass it in to an instance via the `setView()` method:

```
$build->setView($view);
```

5.2.1.2. Setting the layer name

While the layer name may be passed during instantiation, you may also pass it in to an instance via the `setLayerName()` method:

```
$build->setLayerName("custom.main");
```

5.2.1.3. Including onLoad events in the generated layer

dojo.addOnLoad is a useful utility for specifying actions that should trigger when the DOM has finished loading. The `dojo()` view helper can create these statements via its `addOnLoad()` and `onLoadCapture()` methods. In some cases, it makes sense to push these into your layer file instead of rendering them via your view scripts.

By default, these are not rendered; to enable them, pass the consumeOnLoad configuration key during instantiation:

```
$build = new Zend_Dojo_BuildLayer(array(
    'view'          => $view,
    'layerName'     => 'custom.main',
    'consumeOnLoad' => true,
));
```

Alternately, you can use the `setConsumeOnLoad()` method after instantiation:

```
$build->setConsumeOnLoad(true);
```

5.2.1.4. Including captured JavaScript in the generated layer

The `dojo()` view helper includes methods for capturing arbitrary JavaScript to include in the <script> tag containing the various **dojo.require** and **dojo.addOnLoad** statements. This can be useful when creating default data stores or globally scoped objects used throughout your application.

By default, these are not rendered; to enable them, pass the consumeJavascript configuration key during instantiation:

```
$build = new Zend_Dojo_BuildLayer(array(
    'view'              => $view,
    'layerName'         => 'custom.main',
    'consumeJavascript' => true,
));
```

Alternately, you can use the `setConsumeJavascript()` method after instantiation:

```
$build->setConsumeJavascript(true);
```

5.3. Generating Build Profiles with Zend_Dojo_BuildLayer

One of the chief benefits of a Dojo module layer is that it facilitates the creation of a custom build. `Zend_Dojo_BuildLayer` has functionality for generate build profiles.

The simplest use case is to utilize the `generateBuildProfile()` method and send the output to a file:

```
$build = new Zend_Dojo_BuildLayer(array(
    'view'      => $view,
    'layerName' => 'custom.main',
));

$profile   = $build->generateBuildProfile();
$filename  = APPLICATION_PATH . '/../misc/scripts/custom.profile.js';
file_put_contents($filename, $profile);
```

Just like generating layers, you may want to automate this via a `dispatchLoopShutdown()` plugin hook; you could even simply modify the one shown for generating layers to read as follows:

```
class App_Plugin_DojoLayer extends Zend_Controller_Plugin_Abstract
{
    public $layerScript  = APPLICATION_PATH
                         . '/../public/js/custom/main.js';
    public $buildProfile = APPLICATION_PATH
                         . '/../misc/scripts/custom.profile.js';
    protected $_build;

    public function dispatchLoopShutdown()
    {
        if (!file_exists($this->layerScript)) {
            $this->generateDojoLayer();
        }
        if (!file_exists($this->buildProfile)) {
            $this->generateBuildProfile();
        }
    }

    public function generateDojoLayer() { /* ... */ }

    public function generateBuildProfile()
    {
        $profile = $this->getBuild()->generateBuildProfile();
        file_put_contents($this->buildProfile, $profile);
    }

}
```

As noted, with module layers, you should only create the file once.

5.3.1. Build Profile options

The above functionality will suffice for most situations. The only way to customize build profile generation is to provide additional build profile options to utilize.

As an example, you may want to specify what type of optimizations should be performed, whether or not to optimize CSS files in the layer, whether or not to copy tests into the build, etc. For a listing of available options, you should read the Dojo Build documentation and accompanying documentation.

Setting these options is trivial: use the `addProfileOption()`, `addProfileOptions()`, or `setProfileOptions()` methods. The first method adds a single key and value option pair, the second will add several, and the third will overwrite any options with the list of key and value pairs provided.

By default, the following options are set:

```
{
    action:        "release",
    optimize:      "shrinksafe",
    layerOptimize: "shrinksafe",
    copyTests:     false,
    loader:        "default",
    cssOptimize:   "comments"
}
```

You can pass in whatever key and value pairs you want; the Dojo build script will ignore those it does not understand.

As an example of setting options:

```
// A single option:
$build->addProfileOption('version', 'zend-1.3.1');

// Several options:
$build->addProfileOptions(array(
    'loader'   => 'xdomain',
    'optimize' => 'packer',
));

// Or overwrite options:
$build->setProfileOptions(array(
    'version'  => 'custom-1.3.1',
    'loader'   => 'shrinksafe',
    'optimize' => 'shrinksafe',
));
```

Zend_Dom

1. Introduction

`Zend_Dom` provides tools for working with DOM documents and structures. Currently, we offer `Zend_Dom_Query`, which provides a unified interface for querying DOM documents utilizing both XPath and CSS selectors.

2. Zend_Dom_Query

`Zend_Dom_Query` provides mechanisms for querying XML and (X)HTML documents utilizing either XPath or CSS selectors. It was developed to aid with functional testing of MVC applications, but could also be used for rapid development of screen scrapers.

CSS selector notation is provided as a simpler and more familiar notation for web developers to utilize when querying documents with XML structures. The notation should be familiar to anybody who has developed Cascading Style Sheets or who utilizes Javascript toolkits that provide functionality for selecting nodes utilizing CSS selectors (Prototype's $$() and Dojo's dojo.query were both inspirations for the component).

2.1. Theory of Operation

To use `Zend_Dom_Query`, you instantiate a `Zend_Dom_Query` object, optionally passing a document to query (a string). Once you have a document, you can use either the `query()` or `queryXpath()` methods; each method will return a `Zend_Dom_Query_Result` object with any matching nodes.

The primary difference between `Zend_Dom_Query` and using DOMDocument + DOMXPath is the ability to select against CSS selectors. You can utilize any of the following, in any combination:

- *element types*: provide an element type to match: 'div', 'a', 'span', 'h2', etc.
- *style attributes*: CSS style attributes to match: '.error', 'div.error', 'label.required', etc. If an element defines more than one style, this will match as long as the named style is present anywhere in the style declaration.
- *id attributes*: element ID attributes to match: '#content', 'div#nav', etc.
- *arbitrary attributes*: arbitrary element attributes to match. Three different types of matching are provided:
  - *exact match*: the attribute exactly matches the string: 'div[bar="baz"]' would match a div element with a "bar" attribute that exactly matches the value "baz".
  - *word match*: the attribute contains a word matching the string: 'div[bar~="baz"]' would match a div element with a "bar" attribute that contains the word "baz". ' ' would match, but ' ' would not.
  - *substring match*: the attribute contains the string: 'div[bar*="baz"]' would match a div element with a "bar" attribute that contains the string "baz" anywhere within it.
- *direct descendents*: utilize '>' between selectors to denote direct descendents. 'div > span' would select only 'span' elements that are direct descendents of a 'div'. Can also be used with any of the selectors above.

- *descendents*: string together multiple selectors to indicate a hierarchy along which to search. 'div .foo span #one' would select an element of id 'one' that is a descendent of arbitrary depth beneath a 'span' element, which is in turn a descendent of arbitrary depth beneath an element with a class of 'foo', that is an descendent of arbitrary depth beneath a 'div' element. For example, it would match the link to the word 'One' in the listing below:

```
<div>
<table>
    <tr>
        <td class="foo">
            <div>
                Lorem ipsum <span class="bar">
                    <a href="/foo/bar" id="one">One</a>
                    <a href="/foo/baz" id="two">Two</a>
                    <a href="/foo/bat" id="three">Three</a>
                    <a href="/foo/bla" id="four">Four</a>
                </span>
            </div>
        </td>
    </tr>
</table>
</div>
```

Once you've performed your query, you can then work with the result object to determine information about the nodes, as well as to pull them and/or their content directly for examination and manipulation. `Zend_Dom_Query_Result` implements `Countable` and `Iterator`, and store the results internally as DOMNodes/DOMElements. As an example, consider the following call, that selects against the HTML above:

```
$dom = new Zend_Dom_Query($html);
$results = $dom->query('.foo .bar a');

$count = count($results); // get number of matches: 4
foreach ($results as $result) {
    // $result is a DOMElement
}
```

`Zend_Dom_Query` also allows straight XPath queries utilizing the `queryXpath()` method; you can pass any valid XPath query to this method, and it will return a `Zend_Dom_Query_Result` object.

2.2. Methods Available

The `Zend_Dom_Query` family of classes have the following methods available.

2.2.1. Zend_Dom_Query

The following methods are available to `Zend_Dom_Query`:

- `setDocumentXml($document)`: specify an XML string to query against.
- `setDocumentXhtml($document)`: specify an XHTML string to query against.
- `setDocumentHtml($document)`: specify an HTML string to query against.
- `setDocument($document)`: specify a string to query against; `Zend_Dom_Query` will then attempt to autodetect the document type.

- `getDocument()`: retrieve the original document string provided to the object.
- `getDocumentType()`: retrieve the document type of the document provided to the object; will be one of the `DOC_XML`, `DOC_XHTML`, or `DOC_HTML` class constants.
- `query($query)`: query the document using CSS selector notation.
- `queryXpath($xPathQuery)`: query the document using XPath notation.

2.2.2. Zend_Dom_Query_Result

As mentioned previously, `Zend_Dom_Query_Result` implements both `Iterator` and `Countable`, and as such can be used in a `foreach` loop as well as with the `count()` function. Additionally, it exposes the following methods:

- `getCssQuery()`: return the CSS selector query used to produce the result (if any).
- `getXpathQuery()`: return the XPath query used to produce the result. Internally, `Zend_Dom_Query` converts CSS selector queries to XPath, so this value will always be populated.
- `getDocument()`: retrieve the DOMDocument the selection was made against.

Zend_Exception

1. Using Exceptions

`Zend_Exception` is simply the base class for all exceptions thrown within Zend Framework.

Example 329. Catching an Exception

The following code listing demonstrates how to catch an exception thrown in a Zend Framework class:

```
try {
    // Calling Zend_Loader::loadClass() with a non-existant class will cause
    // an exception to be thrown in Zend_Loader:
    Zend_Loader::loadClass('nonexistantclass');
} catch (Zend_Exception $e) {
    echo "Caught exception: " . get_class($e) . "\n";
    echo "Message: " . $e->getMessage() . "\n";
    // Other code to recover from the error
}
```

`Zend_Exception` can be used as a catch-all exception class in a catch block to trap all exceptions thrown by Zend Framework classes. This can be useful when the program can not recover by catching a specific exception type.

The documentation for each Zend Framework component and class will contain specific information on which methods throw exceptions, the circumstances that cause an exception to be thrown, and the various exception types that may be thrown.

2. Basic usage

`Zend_Exception` is the base class for all exceptions thrown by Zend Framework. This class extends the base `Exception` class of PHP.

Example 330. Catch all Zend Framework exceptions

```
try {
    // your code
} catch (Zend_Exception $e) {
    // do something
}
```

Example 331. Catch exceptions thrown by a specific component of Zend Framework

```
try {
    // your code
} catch (Zend_Db_Exception $e) {
    // do something
}
```

3. Previous Exceptions

Since Zend Framework 1.10, `Zend_Exception` implements the PHP 5.3 support for previous exceptions. Simply put, when in a `catch` block, you can throw a new exception that references

the original exception, which helps provide additional context when debugging. By providing this support in Zend Framework, your code may now be forwards compatible with PHP 5.3.

Previous exceptions are indicated as the third argument to an exception constructor.

Example 332. Previous exceptions

```
try {
    $db->query($sql);
} catch (Zend_Db_Statement_Exception $e) {
    if ($e->getPrevious()) {
        echo '[' . get_class($e)
             . '] has the previous exception of ['
             . get_class($e->getPrevious())
             . ']' . PHP_EOL;
    } else {
        echo '[' . get_class($e)
             . '] does not have a previous exception'
             . PHP_EOL;
    }

    echo $e;
    // displays all exceptions starting by the first thrown
    // exception if available.
}
```

Zend_Feed

1. Introduction

Zend_Feed provides functionality for consuming RSS and Atom feeds. It provides a natural syntax for accessing elements of feeds, feed attributes, and entry attributes. Zend_Feed also has extensive support for modifying feed and entry structure with the same natural syntax, and turning the result back into XML. In the future, this modification support could provide support for the Atom Publishing Protocol.

Programmatically, Zend_Feed consists of a base Zend_Feed class, abstract Zend_Feed_Abstract and Zend_Feed_Entry_Abstract base classes for representing Feeds and Entries, specific implementations of feeds and entries for RSS and Atom, and a behind-the-scenes helper for making the natural syntax magic work.

In the example below, we demonstrate a simple use case of retrieving an RSS feed and saving relevant portions of the feed data to a simple PHP array, which could then be used for printing the data, storing to a database, etc.

Be aware

Many RSS feeds have different channel and item properties available. The RSS specification provides for many optional properties, so be aware of this when writing code to work with RSS data.

Example 333. Putting Zend_Feed to Work on RSS Feed Data

```
// Fetch the latest Slashdot headlines
try {
    $slashdotRss =
        Zend_Feed::import('http://rss.slashdot.org/Slashdot/slashdot');
} catch (Zend_Feed_Exception $e) {
    // feed import failed
    echo "Exception caught importing feed: {$e->getMessage()}\n";
    exit;
}

// Initialize the channel data array
$channel = array(
    'title'       => $slashdotRss->title(),
    'link'        => $slashdotRss->link(),
    'description' => $slashdotRss->description(),
    'items'       => array()
    );

// Loop over each channel item and store relevant data
foreach ($slashdotRss as $item) {
    $channel['items'][] = array(
        'title'       => $item->title(),
        'link'        => $item->link(),
        'description' => $item->description()
        );
}
```

2. Importing Feeds

Zend_Feed enables developers to retrieve feeds very easily. If you know the URI of a feed, simply use the Zend_Feed::import() method:

```
$feed = Zend_Feed::import('http://feeds.example.com/feedName');
```

You can also use Zend_Feed to fetch the contents of a feed from a file or the contents of a PHP string variable:

```
// importing a feed from a text file
$feedFromFile = Zend_Feed::importFile('feed.xml');

// importing a feed from a PHP string variable
$feedFromPHP = Zend_Feed::importString($feedString);
```

In each of the examples above, an object of a class that extends Zend_Feed_Abstract is returned upon success, depending on the type of the feed. If an RSS feed were retrieved via one of the import methods above, then a Zend_Feed_Rss object would be returned. On the other hand, if an Atom feed were imported, then a Zend_Feed_Atom object is returned. The import methods will also throw a Zend_Feed_Exception object upon failure, such as an unreadable or malformed feed.

2.1. Custom feeds

Zend_Feed enables developers to create custom feeds very easily. You just have to create an array and to import it with Zend_Feed. This array can be imported with Zend_Feed::importArray() or with Zend_Feed::importBuilder(). In this last case the array will be computed on the fly by a custom data source implementing Zend_Feed_Builder_Interface.

2.1.1. Importing a custom array

```
// importing a feed from an array
$atomFeedFromArray = Zend_Feed::importArray($array);

// the following line is equivalent to the above;
// by default a Zend_Feed_Atom instance is returned
$atomFeedFromArray = Zend_Feed::importArray($array, 'atom');

// importing a rss feed from an array
$rssFeedFromArray = Zend_Feed::importArray($array, 'rss');
```

The format of the array must conform to this structure:

```
array(
    //required
    'title' => 'title of the feed',
    'link'  => 'canonical url to the feed',

    // optional
    'lastUpdate' => 'timestamp of the update date',
    'published'  => 'timestamp of the publication date',

    // required
    'charset' => 'charset of the textual data',
```

```
    // optional
    'description' => 'short description of the feed',
    'author'      => 'author/publisher of the feed',
    'email'       => 'email of the author',

    // optional, ignored if atom is used
    'webmaster' => 'email address for person responsible '
                .  'for technical issues',

    // optional
    'copyright' => 'copyright notice',
    'image'     => 'url to image',
    'generator' => 'generator',
    'language'  => 'language the feed is written in',

    // optional, ignored if atom is used
    'ttl'    => 'how long in minutes a feed can be cached '
             .  'before refreshing',
    'rating' => 'The PICS rating for the channel.',

    // optional, ignored if atom is used
    // a cloud to be notified of updates
    'cloud'       => array(
        // required
        'domain' => 'domain of the cloud, e.g. rpc.sys.com',

        // optional, defaults to 80
        'port' => 'port to connect to',

        // required
        'path'              => 'path of the cloud, e.g. /RPC2',
        'registerProcedure' => 'procedure to call, e.g. myCloud.rssPlsNotify',
        'protocol'          => 'protocol to use, e.g. soap or xml-rpc'
    ),

    // optional, ignored if atom is used
    // a text input box that can be displayed with the feed
    'textInput'   => array(
        // required
        'title'       => 'label of the Submit button in the text input area',
        'description' => 'explains the text input area',
        'name'        => 'the name of the text object in the text input area',
        'link'        => 'URL of the CGI script processing text input requests'
    ),

    // optional, ignored if atom is used
    // Hint telling aggregators which hours they can skip
    'skipHours' => array(
        // up to 24 rows whose value is a number between 0 and 23
        // e.g 13 (1pm)
        'hour in 24 format'
    ),

    // optional, ignored if atom is used
    // Hint telling aggregators which days they can skip
    'skipDays ' => array(
        // up to 7 rows whose value is
        // Monday, Tuesday, Wednesday, Thursday, Friday, Saturday or Sunday
        // e.g Monday
```

```
        'a day to skip'
    ),

    // optional, ignored if atom is used
    // Itunes extension data
    'itunes' => array(
        // optional, default to the main author value
        'author' => 'Artist column',

        // optional, default to the main author value
        // Owner of the podcast
        'owner' => array(
            'name'  => 'name of the owner',
            'email' => 'email of the owner'
        ),

        // optional, default to the main image value
        'image' => 'album/podcast art',

        // optional, default to the main description value
        'subtitle' => 'short description',
        'summary'  => 'longer description',

        // optional
        'block' => 'Prevent an episode from appearing (yes|no)',

        // required, Category column and in iTunes Music Store Browse
        'category' => array(
            // up to 3 rows
            array(
                // required
                'main' => 'main category',

                // optional
                'sub'  => 'sub category'
            )
        ),

        // optional
        'explicit'     => 'parental advisory graphic (yes|no|clean)',
        'keywords'     => 'a comma separated list of 12 keywords maximum',
        'new-feed-url' => 'used to inform iTunes of new feed URL location'
    ),

    'entries' => array(
        array(
            //required
            'title' => 'title of the feed entry',
            'link'  => 'url to a feed entry',

            // required, only text, no html
            'description' => 'short version of a feed entry',

            // optional
            'guid' => 'id of the article, '
                   .  'if not given link value will used',

            // optional, can contain html
            'content' => 'long version',
```

```
            // optional
            'lastUpdate' => 'timestamp of the publication date',
            'comments'   => 'comments page of the feed entry',
            'commentRss' => 'the feed url of the associated comments',

            // optional, original source of the feed entry
            'source' => array(
                // required
                'title' => 'title of the original source',
                'url'   => 'url of the original source'
            ),

            // optional, list of the attached categories
            'category' => array(
                array(
                    // required
                    'term' => 'first category label',

                    // optional
                    'scheme' => 'url that identifies a categorization scheme'
                ),

                array(
                    // data for the second category and so on
                )
            ),

            // optional, list of the enclosures of the feed entry
            'enclosure'    => array(
                array(
                    // required
                    'url' => 'url of the linked enclosure',

                    // optional
                    'type' => 'mime type of the enclosure',
                    'length' => 'length of the linked content in octets'
                ),

                array(
                    //data for the second enclosure and so on
                )
            )
        ),

        array(
            //data for the second entry and so on
        )
    )
);
```

References:

- RSS 2.0 specification: RSS 2.0
- Atom specification: RFC 4287
- WFW specification: Well Formed Web
- iTunes specification: iTunes Technical Specifications

2.1.2. Importing a custom data source

You can create a `Zeed_Feed` instance from any data source implementing `Zend_Feed_Builder_Interface`. You just have to implement the `getHeader()` and `getEntries()` methods to be able to use your object with `Zend_Feed::importBuilder()`. As a simple reference implementation, you can use `Zend_Feed_Builder`, which takes an array in its constructor, performs some minor validation, and then can be used in the `importBuilder()` method. The `getHeader()` method must return an instance of `Zend_Feed_Builder_Header`, and `getEntries()` must return an array of `Zend_Feed_Builder_Entry` instances.

`Zend_Feed_Builder` serves as a concrete implementation to demonstrate the usage. Users are encouraged to make their own classes to implement `Zend_Feed_Builder_Interface`.

Here is an example of `Zend_Feed::importBuilder()` usage:

```
// importing a feed from a custom builder source
$atomFeedFromArray =
    Zend_Feed::importBuilder(new Zend_Feed_Builder($array));

// the following line is equivalent to the above;
// by default a Zend_Feed_Atom instance is returned
$atomFeedFromArray =
    Zend_Feed::importBuilder(new Zend_Feed_Builder($array), 'atom');

// importing a rss feed from a custom builder array
$rssFeedFromArray =
    Zend_Feed::importBuilder(new Zend_Feed_Builder($array), 'rss');
```

2.1.3. Dumping the contents of a feed

To dump the contents of a `Zend_Feed_Abstract` instance, you may use `send()` or `saveXml()` methods.

```
assert($feed instanceof Zend_Feed_Abstract);

// dump the feed to standard output
print $feed->saveXML();

// send http headers and dump the feed
$feed->send();
```

3. Retrieving Feeds from Web Pages

Web pages often contain *<link>* tags that refer to feeds with content relevant to the particular page. `Zend_Feed` enables you to retrieve all feeds referenced by a web page with one simple method call:

```
$feedArray = Zend_Feed::findFeeds('http://www.example.com/news.html');
```

Here the `findFeeds()` method returns an array of `Zend_Feed_Abstract` objects that are referenced by *<link>* tags on the `news.html` web page. Depending on the type of each feed, each respective entry in the `$feedArray` array may be a `Zend_Feed_Rss` or

`Zend_Feed_Atom` instance. `Zend_Feed` will throw a `Zend_Feed_Exception` upon failure, such as an HTTP 404 response code or a malformed feed.

4. Consuming an RSS Feed

Reading an RSS feed is as simple as instantiating a `Zend_Feed_Rss` object with the URL of the feed:

```
$channel = new Zend_Feed_Rss('http://rss.example.com/channelName');
```

If any errors occur fetching the feed, a `Zend_Feed_Exception` will be thrown.

Once you have a feed object, you can access any of the standard RSS "channel" properties directly on the object:

```
echo $channel->title();
```

Note the function syntax. `Zend_Feed` uses a convention of treating properties as XML object if they are requested with variable "getter" syntax (`$obj->property`) and as strings if they are access with method syntax (`$obj->property()`). This enables access to the full text of any individual node while still allowing full access to all children.

If channel properties have attributes, they are accessible using PHP's array syntax:

```
echo $channel->category['domain'];
```

Since XML attributes cannot have children, method syntax is not necessary for accessing attribute values.

Most commonly you'll want to loop through the feed and do something with its entries. `Zend_Feed_Abstract` implements PHP's `Iterator` interface, so printing all titles of articles in a channel is just a matter of:

```
foreach ($channel as $item) {
    echo $item->title() . "\n";
}
```

If you are not familiar with RSS, here are the standard elements you can expect to be available in an RSS channel and in individual RSS items (entries).

Required channel elements:

- `title` - The name of the channel
- `link` - The URL of the web site corresponding to the channel
- `description` - A sentence or several describing the channel

Common optional channel elements:

- `pubDate` - The publication date of this set of content, in RFC 822 date format
- `language` - The language the channel is written in
- `category` - One or more (specified by multiple tags) categories the channel belongs to

RSS <item> elements do not have any strictly required elements. However, either `title` or `description` must be present.

Common item elements:

- `title` - The title of the item
- `link` - The URL of the item
- `description` - A synopsis of the item
- `author` - The author's email address
- `category` - One more categories that the item belongs to
- `comments` - URL of comments relating to this item
- `pubDate` - The date the item was published, in RFC 822 date format

In your code you can always test to see if an element is non-empty with:

```
if ($item->propname()) {
    // ... proceed.
}
```

If you use `$item->propname` instead, you will always get an empty object which will evaluate to TRUE, so your check will fail.

For further information, the official RSS 2.0 specification is available at: http://blogs.law.harvard.edu/tech/rss

5. Consuming an Atom Feed

`Zend_Feed_Atom` is used in much the same way as `Zend_Feed_Rss`. It provides the same access to feed-level properties and iteration over entries in the feed. The main difference is in the structure of the Atom protocol itself. Atom is a successor to RSS; it is more generalized protocol and it is designed to deal more easily with feeds that provide their full content inside the feed, splitting RSS' `description` tag into two elements, `summary` and `content`, for that purpose.

Example 334. Basic Use of an Atom Feed

Read an Atom feed and print the `title` and `summary` of each entry:

```
$feed = new Zend_Feed_Atom('http://atom.example.com/feed/');
echo 'The feed contains ' . $feed->count() . ' entries.' . "\n\n";
foreach ($feed as $entry) {
    echo 'Title: ' . $entry->title() . "\n";
    echo 'Summary: ' . $entry->summary() . "\n\n";
}
```

In an Atom feed you can expect to find the following feed properties:

- `title` - The feed's title, same as RSS's channel title
- `id` - Every feed and entry in Atom has a unique identifier

- `link` - Feeds can have multiple links, which are distinguished by a `type` attribute

  The equivalent to RSS's channel link would be `type="text/html"`. if the link is to an alternate version of the same content that's in the feed, it would have a `rel="alternate"` attribute.

- `subtitle` - The feed's description, equivalent to RSS' channel description

  `author->name()` - The feed author's name

  `author->email()` - The feed author's email address

Atom entries commonly have the following properties:

- `id` - The entry's unique identifier
- `title` - The entry's title, same as RSS item titles
- `link` - A link to another format or an alternate view of this entry
- `summary` - A summary of this entry's content
- `content` - The full content of the entry; can be skipped if the feed just contains summaries
- `author` - with `name` and `email` sub-tags like feeds have
- `published` - the date the entry was published, in RFC 3339 format
- `updated` - the date the entry was last updated, in RFC 3339 format

For more information on Atom and plenty of resources, see http://www.atomenabled.org/.

6. Consuming a Single Atom Entry

Single Atom `<entry>` elements are also valid by themselves. Usually the URL for an entry is the feed's URL followed by `/<entryId>`, such as `http://atom.example.com/feed/1`, using the example URL we used above.

If you read a single entry, you will still have a `Zend_Feed_Atom` object, but it will automatically create an "anonymous" feed to contain the entry.

Example 335. Reading a Single-Entry Atom Feed

```
$feed = new Zend_Feed_Atom('http://atom.example.com/feed/1');
echo 'The feed has: ' . $feed->count() . ' entry.';

$entry = $feed->current();
```

Alternatively, you could instantiate the entry object directly if you know you are accessing an `<entry>`-only document:

Example 336. Using the Entry Object Directly for a Single-Entry Atom Feed

```
$entry = new Zend_Feed_Entry_Atom('http://atom.example.com/feed/1');
echo $entry->title();
```

7. Modifying Feed and Entry structures

Zend_Feed's natural syntax extends to constructing and modifying feeds and entries as well as reading them. You can easily turn your new or modified objects back into well-formed XML for saving to a file or sending to a server.

Example 337. Modifying an Existing Feed Entry

```
$feed = new Zend_Feed_Atom('http://atom.example.com/feed/1');
$entry = $feed->current();

$entry->title = 'This is a new title';
$entry->author->email = 'my_email@example.com';

echo $entry->saveXML();
```

This will output a full (includes `<?xml ... >` prologue) XML representation of the new entry, including any necessary XML namespaces.

Note that the above will work even if the existing entry does not already have an author tag. You can use as many levels of `->` access as you like before getting to an assignment; all of the intervening levels will be created for you automatically if necessary.

If you want to use a namespace other than `atom:`, `rss:`, or `osrss:` in your entry, you need to register the namespace with `Zend_Feed` using `Zend_Feed::registerNamespace()`. When you are modifying an existing element, it will always maintain its original namespace. When adding a new element, it will go into the default namespace if you do not explicitly specify another namespace.

Example 338. Creating an Atom Entry with Elements of Custom Namespaces

```
$entry = new Zend_Feed_Entry_Atom();
// id is always assigned by the server in Atom
$entry->title = 'my custom entry';
$entry->author->name = 'Example Author';
$entry->author->email = 'me@example.com';

// Now do the custom part.
Zend_Feed::registerNamespace('myns', 'http://www.example.com/myns/1.0');

$entry->{'myns:myelement_one'} = 'my first custom value';
$entry->{'myns:container_elt'}->part1 = 'first nested custom part';
$entry->{'myns:container_elt'}->part2 = 'second nested custom part';

echo $entry->saveXML();
```

8. Custom Feed and Entry Classes

Finally, you can extend the `Zend_Feed` classes if you'd like to provide your own format or niceties like automatic handling of elements that should go into a custom namespace.

Here is an example of a custom Atom entry class that handles its own *myns:* namespace entries. Note that it also makes the `registerNamespace()` call for you, so the end user doesn't need to worry about namespaces at all.

```php
/**
 * The custom entry class automatically knows the feed URI (optional) and
 * can automatically add extra namespaces.
 */
class MyEntry extends Zend_Feed_Entry_Atom
{

    public function __construct($uri = 'http://www.example.com/myfeed/',
                                $xml = null)
    {
        parent::__construct($uri, $xml);

        Zend_Feed::registerNamespace('myns',
                                     'http://www.example.com/myns/1.0');
    }

    public function __get($var)
    {
        switch ($var) {
            case 'myUpdated':
                // Translate myUpdated to myns:updated.
                return parent::__get('myns:updated');

            default:
                return parent::__get($var);
            }
    }

    public function __set($var, $value)
    {
        switch ($var) {
            case 'myUpdated':
                // Translate myUpdated to myns:updated.
                parent::__set('myns:updated', $value);
                break;

            default:
                parent::__set($var, $value);
        }
    }

    public function __call($var, $unused)
    {
        switch ($var) {
            case 'myUpdated':
                // Translate myUpdated to myns:updated.
                return parent::__call('myns:updated', $unused);

            default:
                return parent::__call($var, $unused);
        }
    }
$entry = new MyEntry();
$entry->myUpdated = '2005-04-19T15:30';

// method-style call is handled by __call function
$entry->myUpdated();
// property-style call is handled by __get function
$entry->myUpdated;
```

9. Zend_Feed_Reader

9.1. Introduction

`Zend_Feed_Reader` is a component used to consume RSS and Atom feeds of any version, including RDF/RSS 1.0, RSS 2.0 and Atom 0.3/1.0. The API for retrieving feed data is deliberately simple since `Zend_Feed_Reader` is capable of searching any feed of any type for the information requested through the API. If the typical elements containing this information are not present, it will adapt and fall back on a variety of alternative elements instead. This ability to choose from alternatives removes the need for users to create their own abstraction layer on top of the component to make it useful or have any in-depth knowledge of the underlying standards, current alternatives, and namespaced extensions.

Internally, `Zend_Feed_Reader` works almost entirely on the basis of making XPath queries against the feed XML's Document Object Model. The DOM is not exposed though a chained property API like `Zend_Feed` though the underlying `DOMDocument`, `DOMElement` and `DOMXPath` objects are exposed for external manipulation. This singular approach to parsing is consistent and the component offers a plugin system to add to the Feed and Entry level API by writing Extensions on a similar basis.

Performance is assisted in three ways. First of all, `Zend_Feed_Reader` supports caching using `Zend_Cache` to maintain a copy of the original feed XML. This allows you to skip network requests for a feed URI if the cache is valid. Second, the Feed and Entry level API is backed by an internal cache (non-persistant) so repeat API calls for the same feed will avoid additional DOM/XPath use. Thirdly, importing feeds from a URI can take advantage of HTTP Conditional GET requests which allow servers to issue an empty 304 response when the requested feed has not changed since the last time you requested it. In the final case, an instance of `Zend_Cache` will hold the last received feed along with the ETag and Last-Modified header values sent in the HTTP response.

In relation to `Zend_Feed`, `Zend_Feed_Reader` was formulated as a free standing replacement for `Zend_Feed` but it is not backwards compatible with `Zend_Feed`. Rather it is an alternative following a different ideology focused on being simple to use, flexible, consistent and extendable through the plugin system. `Zend_Feed_Reader` is also not capable of constructing feeds and delegates this responsibility to `Zend_Feed_Writer`, its sibling in arms.

9.2. Importing Feeds

Importing a feed with `Zend_Feed_Reader` is not that much different to `Zend_Feed`. Feeds can be imported from a string, file, URI or an instance of type `Zend_Feed_Abstract`. Importing from a URI can additionally utilise a HTTP Conditional GET request. If importing fails, an exception will be raised. The end result will be an object of type `Zend_Feed_Reader_FeedInterface`, the core implementations of which are `Zend_Feed_Reader_Feed_Rss` and `Zend_Feed_Reader_Feed_Atom` (`Zend_Feed` took all the short names!). Both objects support multiple (all existing) versions of these broad feed types.

In the following example, we import an RDF/RSS 1.0 feed and extract some basic information that can be saved to a database or elsewhere.

```
$feed = Zend_Feed_Reader::import('http://www.planet-php.net/rdf/');
$data = array(
    'title'        => $feed->getTitle(),
    'link'         => $feed->getLink(),
    'dateModified' => $feed->getDateModified(),
    'description'  => $feed->getDescription(),
```

```
    'language'      => $feed->getLanguage(),
    'entries'       => array(),
);

foreach ($feed as $entry) {
    $edata = array(
        'title'        => $entry->getTitle(),
        'description'  => $entry->getDescription(),
        'dateModified' => $entry->getDateModified(),
        'authors'       => $entry->getAuthors(),
        'link'         => $entry->getLink(),
        'content'      => $entry->getContent()
    );
    $data['entries'][] = $edata;
}
```

The example above demonstrates `Zend_Feed_Reader`'s API, and it also demonstrates some of its internal operation. In reality, the RDF feed selected does not have any native date or author elements, however it does utilise the Dublin Core 1.1 module which offers namespaced creator and date elements. `Zend_Feed_Reader` falls back on these and similar options if no relevant native elements exist. If it absolutely cannot find an alternative it will return `NULL`, indicating the information could not be found in the feed. You should note that classes implementing `Zend_Feed_Reader_FeedInterface` also implement the SPL `Iterator` and `Countable` interfaces.

Feeds can also be imported from strings, files, and even objects of type `Zend_Feed_Abstract`.

```
// from a URI
$feed = Zend_Feed_Reader::import('http://www.planet-php.net/rdf/');

// from a String
$feed = Zend_Feed_Reader::importString($feedXmlString);

// from a file
$feed = Zend_Feed_Reader::importFile('./feed.xml');

// from a Zend_Feed_Abstract object
$zfeed = Zend_Feed::import('http://www.planet-php.net/atom/');
$feed  = Zend_Feed_Reader::importFeed($zfeed);
```

9.3. Retrieving Underlying Feed and Entry Sources

`Zend_Feed_Reader` does its best not to stick you in a narrow confine. If you need to work on a feed outside of `Zend_Feed_Reader`, you can extract the base `DOMDocument` or `DOMElement` objects from any class, or even an XML string containing these. Also provided are methods to extract the current `DOMXPath` object (with all core and Extension namespaces registered) and the correct prefix used in all XPath queries for the current Feed or Entry. The basic methods to use (on any object) are `saveXml()`, `getDomDocument()`, `getElement()`, `getXpath()` and `getXpathPrefix()`. These will let you break free of `Zend_Feed_Reader` and do whatever else you want.

- `saveXml()` returns an XML string containing only the element representing the current object.
- `getDomDocument()` returns the `DOMDocument` object representing the entire feed (even if called from an Entry object).
- `getElement()` returns the `DOMElement` of the current object (i.e. the Feed or current Entry).

- `getXpath()` returns the `DOMXPath` object for the current feed (even if called from an Entry object) with the namespaces of the current feed type and all loaded Extensions pre-registered.
- `getXpathPrefix()` returns the query prefix for the current object (i.e. the Feed or current Entry) which includes the correct XPath query path for that specific Feed or Entry.

Here's an example where a feed might include an RSS Extension not supported by `Zend_Feed_Reader` out of the box. Notably, you could write and register an Extension (covered later) to do this, but that's not always warranted for a quick check. You must register any new namespaces on the `DOMXPath` object before use unless they are registered by `Zend_Feed_Reader` or an Extension beforehand.

```
$feed        = Zend_Feed_Reader::import('http://www.planet-php.net/rdf/');
$xpathPrefix = $feed->getXpathPrefix();
$xpath       = $feed->getXpath();
$xpath->registerNamespace('admin', 'http://webns.net/mvcb/');
$reportErrorsTo = $xpath->evaluate('string('
                                 . $xpathPrefix
                                 . '/admin:errorReportsTo)');
```

If you register an already registered namespace with a different prefix name to that used internally by `Zend_Feed_Reader`, it will break the internal operation of this component.

9.4. Cache Support and Intelligent Requests

9.4.1. Adding Cache Support to Zend_Feed_Reader

`Zend_Feed_Reader` supports using an instance of `Zend_Cache` to cache feeds (as XML) to avoid unnecessary network requests. Adding a cache is as simple here as it is for other Zend Framework components, create and configure your cache and then tell `Zend_Feed_Reader` to use it! The cache key used is "`Zend_Feed_Reader_`" followed by the MD5 hash of the feed's URI.

```
$frontendOptions = array(
   'lifetime' => 7200,
   'automatic_serialization' => true
);
$backendOptions = array('cache_dir' => './tmp/');
$cache = Zend_Cache::factory(
    'Core', 'File', $frontendOptions, $backendOptions
);

Zend_Feed_Reader::setCache($cache);
```

While it's a little off track, you should also consider adding a cache to `Zend_Loader_PluginLoader` which is used by `Zend_Feed_Reader` to load Extensions.

9.4.2. HTTP Conditional GET Support

The big question often asked when importing a feed frequently, is if it has even changed. With a cache enabled, you can add HTTP Conditional GET support to your arsenal to answer that question.

Using this method, you can request feeds from URIs and include their last known ETag and Last-Modified response header values with the request (using the If-None-Match and If-Modified-Since headers). If the feed on the server remains unchanged, you should receive a 304 response which tells `Zend_Feed_Reader` to use the cached version. If a full feed is sent in a response with a status code of 200, this means the feed has changed and `Zend_Feed_Reader` will parse the new version and save it to the cache. It will also cache the new ETag and Last-Modified header values for future use.

These "conditional" requests are not guaranteed to be supported by the server you request a URI of, but can be attempted regardless. Most common feed sources like blogs should however have this supported. To enable conditional requests, you will need to provide a cache to `Zend_Feed_Reader`.

```
$frontendOptions = array(
   'lifetime' => 86400,
   'automatic_serialization' => true
);
$backendOptions = array('cache_dir' => './tmp/');
$cache = Zend_Cache::factory(
    'Core', 'File', $frontendOptions, $backendOptions
);

Zend_Feed_Reader::setCache($cache);
Zend_Feed_Reader::useHttpConditionalGet();

$feed = Zend_Feed_Reader::import('http://www.planet-php.net/rdf/');
```

In the example above, with HTTP Conditional GET requests enabled, the response header values for ETag and Last-Modified will be cached along with the feed. For the next 24hrs (the cache lifetime), feeds will only be updated on the cache if a non-304 response is received containing a valid RSS or Atom XML document.

If you intend on managing request headers from outside `Zend_Feed_Reader`, you can set the relevant If-None-Matches and If-Modified-Since request headers via the URI import method.

```
$lastEtagReceived = '5e6cefe7df5a7e95c8b1ba1a2ccaff3d';
$lastModifiedDateReceived = 'Wed, 08 Jul 2009 13:37:22 GMT';
$feed = Zend_Feed_Reader::import(
    $uri, $lastEtagReceived, $lastModifiedDateReceived
);
```

9.5. Locating Feed URIs from Websites

These days, many websites are aware that the location of their XML feeds is not always obvious. A small RDF, RSS or Atom graphic helps when the user is reading the page, but what about when a machine visits trying to identify where your feeds are located? To assist in this, websites may point to their feeds using <link> tags in the <head> section of their HTML. To take advantage of this, you can use `Zend_Feed_Reader` to locate these feeds using the static `findFeedLinks()` method.

This method calls any URI and searches for the location of RSS, RDF and Atom feeds assuming the website's HTML contains the relevant links. It then returns a value object where you can check for the existence of a RSS, RDF or Atom feed URI.

The returned object is an `ArrayObject` subclass called `Zend_Feed_Reader_Collection_FeedLink` so you can cast it to an array, or iterate over it,

to access all the detected links. However, as a simple shortcut, you can just grab the first RSS, RDF or Atom link using its public properties as in the example below. Otherwise, each element of the `ArrayObject` is a simple array with the keys "type" and "uri" where the type is one of "rdf", "rss" or "atom".

```
$links = Zend_Feed_Reader::findFeedLinks('http://www.planet-php.net');

if(isset($links->rdf)) {
    echo $links->rdf, "\n"; // http://www.planet-php.org/rdf/
}
if(isset($links->rss)) {
    echo $links->rss, "\n"; // http://www.planet-php.org/rss/
}
if(isset($links->atom)) {
    echo $links->atom, "\n"; // http://www.planet-php.org/atom/
}
```

Based on these links, you can then import from whichever source you wish in the usual manner.

This quick method only gives you one link for each feed type, but websites may indicate many links of any type. Perhaps it's a news site with a RSS feed for each news category. You can iterate over all links using the ArrayObject's iterator.

```
$links = Zend_Feed_Reader::findFeedLinks('http://www.planet-php.net');

foreach ($links as $link) {
    echo $link['uri'], "\n";
}
```

9.6. Attribute Collections

In an attempt to simplify return types, with Zend Framework 1.10 return types from the various feed and entry level methods may include an object of type `Zend_Feed_Reader_Collection_CollectionAbstract`. Despite the special class name which I'll explain below, this is just a simple subclass of SPL's `ArrayObject`.

The main purpose here is to allow the presentation of as much data as possible from the requested elements, while still allowing access to the most relevant data as a simple array. This also enforces a standard approach to returning such data which previously may have wandered between arrays and objects.

The new class type acts identically to `ArrayObject` with the sole addition being a new method `getValues()` which returns a simple flat array containing the most relevant information.

A simple example of this is `Zend_Feed_Reader_FeedInterface::getCategories()`. When used with any RSS or Atom feed, this method will return category data as a container object called `Zend_Feed_Reader_Collection_Category`. The container object will contain, per category, three fields of data: term, scheme and label. The "term" is the basic category name, often machine readable (i.e. plays nice with URIs). The scheme represents a categorisation scheme (usually a URI identifier) also known as a "domain" in RSS 2.0. The "label" is a human readable category name which supports html entities. In RSS 2.0, there is no label attribute so it is always set to the same value as the term for convenience.

To access category labels by themselves in a simple value array, you might commit to something like:

```
$feed = Zend_Feed_Reader::import('http://www.example.com/atom.xml');
$categories = $feed->getCategories();
$labels = array();
foreach ($categories as $cat) {
    $labels[] = $cat['label']
}
```

It's a contrived example, but the point is that the labels are tied up with other information.

However, the container class allows you to access the "most relevant" data as a simple array using the `getValues()` method. The concept of "most relevant" is obviously a judgement call. For categories it means the category labels (not the terms or schemes) while for authors it would be the authors' names (not their email addresses or URIs). The simple array is flat (just values) and passed through `array_unique()` to remove duplication.

```
$feed = Zend_Feed_Reader::import('http://www.example.com/atom.xml');
$categories = $feed->getCategories();
$labels = $categories->getValues();
```

The above example shows how to extract only labels and nothing else thus giving simple access to the category labels without any additional work to extract that data by itself.

9.7. Retrieving Feed Information

Retrieving information from a feed (we'll cover entries/items in the next section though they follow identical principals) uses a clearly defined API which is exactly the same regardless of whether the feed in question is RSS/RDF/Atom. The same goes for sub-versions of these standards and we've tested every single RSS and Atom version. While the underlying feed XML can differ substantially in terms of the tags and elements they present, they nonetheless are all trying to convey similar information and to reflect this all the differences and wrangling over alternative tags are handled internally by `Zend_Feed_Reader` presenting you with an identical interface for each. Ideally, you should not have to care whether a feed is RSS or Atom so long as you can extract the information you want.

While determining common ground between feed types is itself complex, it should be noted that RSS in particular is a constantly disputed "specification". This has its roots in the original RSS 2.0 document which contains ambiguities and does not detail the correct treatment of all elements. As a result, this component rigorously applies the RSS 2.0.11 Specification published by the RSS Advisory Board and its accompanying RSS Best Practices Profile. No other interpretation of RSS 2.0 will be supported though exceptions may be allowed where it does not directly prevent the application of the two documents mentioned above.

Of course, we don't live in an ideal world so there may be times the API just does not cover what you're looking for. To assist you, `Zend_Feed_Reader` offers a plugin system which allows you to write Extensions to expand the core API and cover any additional data you are trying to extract from feeds. If writing another Extension is too much trouble, you can simply grab the underlying DOM or XPath objects and do it by hand in your application. Of course, we really do encourage writing an Extension simply to make it more portable and reusable, and useful Extensions may be proposed to the Framework for formal addition.

Here's a summary of the Core API for Feeds. You should note it comprises not only the basic RSS and Atom standards, but also accounts for a number of included Extensions bundled with

`Zend_Feed_Reader`. The naming of these Extension sourced methods remain fairly generic - all Extension methods operate at the same level as the Core API though we do allow you to retrieve any specific Extension object separately if required.

Table 53. Feed Level API Methods

`getId()`	Returns a unique ID associated with this feed
`getTitle()`	Returns the title of the feed
`getDescription()`	Returns the text description of the feed.
`getLink()`	Returns a URI to the HTML website containing the same or similar information as this feed (i.e. if the feed is from a blog, it should provide the blog's URI where the HTML version of the entries can be read).
`getFeedLink()`	Returns the URI of this feed, which may be the same as the URI used to import the feed. There are important cases where the feed link may differ because the source URI is being updated and is intended to be removed in the future.
`getAuthors()`	Returns an object of type `Zend_Feed_Reader_Collection_Author` which is an `ArrayObject` whose elements are each simple arrays containing any combination of the keys "name", "email" and "uri". Where irrelevant to the source data, some of these keys may be omitted.
`getAuthor(integer $index = 0)`	Returns either the first author known, or with the optional `$index` parameter any specific index on the array of Authors as described above (returning `NULL` if an invalid index).
`getDateCreated()`	Returns the date on which this feed was created. Generally only applicable to Atom where it represents the date the resource described by an Atom 1.0 document was created. The returned date will be a `Zend_Date` object.
`getDateModified()`	Returns the date on which this feed was last modified. The returned date will be a `Zend_Date` object.
`getLanguage()`	Returns the language of the feed (if defined) or simply the language noted in the XML document.
`getGenerator()`	Returns the generator of the feed, e.g. the software which generated it. This may differ between RSS and Atom since Atom defines a different notation.
`getCopyright()`	Returns any copyright notice associated with the feed.
`getHubs()`	Returns an array of all Hub Server URI endpoints which are advertised by the feed for

	use with the Pubsubhubbub Protocol, allowing subscriptions to the feed for real-time updates.
`getCategories()`	Returns a `Zend_Feed_Reader_Collection_Category` object containing the details of any categories associated with the overall feed. The supported fields include "term" (the machine readable category name), "scheme" (the categorisation scheme/domain for this category), and "label" (a html decoded human readable category name). Where any of the three fields are absent from the field, they are either set to the closest available alternative or, in the case of "scheme", set to `NULL`.

Given the variety of feeds in the wild, some of these methods will undoubtedly return `NULL` indicating the relevant information couldn't be located. Where possible, `Zend_Feed_Reader` will fall back on alternative elements during its search. For example, searching an RSS feed for a modification date is more complicated than it looks. RSS 2.0 feeds should include a `<lastBuildDate>` tag and/or a `<pubDate>` element. But what if it doesn't, maybe this is an RSS 1.0 feed? Perhaps it instead has an `<atom:updated>` element with identical information (Atom may be used to supplement RSS's syntax)? Failing that, we could simply look at the entries, pick the most recent, and use its `<pubDate>` element. Assuming it exists... Many feeds also use Dublin Core 1.0/1.1 `<dc:date>` elements for feeds/entries. Or we could find Atom lurking again.

The point is, `Zend_Feed_Reader` was designed to know this. When you ask for the modification date (or anything else), it will run off and search for all these alternatives until it either gives up and returns `NULL`, or finds an alternative that should have the right answer.

In addition to the above methods, all Feed objects implement methods for retrieving the DOM and XPath objects for the current feeds as described earlier. Feed objects also implement the SPL Iterator and Countable interfaces. The extended API is summarised below.

Table 54. Extended Feed Level API Methods

`getDomDocument()`	Returns the parent `DOMDocument` object for the entire source XML document
`getElement()`	Returns the current feed level `DOMElement` object
`saveXml()`	Returns a string containing an XML document of the entire feed element (this is not the original document but a rebuilt version)
`getXpath()`	Returns the `DOMXPath` object used internally to run queries on the `DOMDocument` object (this includes core and Extension namespaces pre-registered)
`getXpathPrefix()`	Returns the valid DOM path prefix prepended to all XPath queries matching the feed being queried
`getEncoding()`	Returns the encoding of the source XML document (note: this cannot account for errors such as the server sending documents in a

	different encoding). Where not defined, the default UTF-8 encoding of Unicode is applied.
`count()`	Returns a count of the entries or items this feed contains (implements SPL `Countable` interface)
`current()`	Returns either the current entry (using the current index from `key()`)
`key()`	Returns the current entry index
`next()`	Increments the entry index value by one
`rewind()`	Resets the entry index to 0
`valid()`	Checks that the current entry index is valid, i.e. it does fall below 0 and does not exceed the number of entries existing.
`getExtensions()`	Returns an array of all Extension objects loaded for the current feed (note: both feed-level and entry-level Extensions exist, and only feed-level Extensions are returned here). The array keys are of the form {ExtensionName}_Feed.
`getExtension(string $name)`	Returns an Extension object for the feed registered under the provided name. This allows more fine-grained access to Extensions which may otherwise be hidden within the implementation of the standard API methods.
`getType()`	Returns a static class constant (e.g. `Zend_Feed_Reader::TYPE_ATOM_03`, i.e. Atom 0.3) indicating exactly what kind of feed is being consumed.

9.8. Retrieving Entry/Item Information

Retrieving information for specific entries or items (depending on whether you speak Atom or RSS) is identical to feed level data. Accessing entries is simply a matter of iterating over a Feed object or using the SPL `Iterator` interface Feed objects implement and calling the appropriate method on each.

Table 55. Entry Level API Methods

`getId()`	Returns a unique ID for the current entry.
`getTitle()`	Returns the title of the current entry.
`getDescription()`	Returns a description of the current entry.
`getLink()`	Returns a URI to the HTML version of the current entry.
`getPermaLink()`	Returns the permanent link to the current entry. In most cases, this is the same as using `getLink()`.
`getAuthors()`	Returns an object of type `Zend_Feed_Reader_Collection_Author` which is an `ArrayObject` whose elements are

	each simple arrays containing any combination of the keys "name", "email" and "uri". Where irrelevant to the source data, some of these keys may be omitted.
`getAuthor(integer $index = 0)`	Returns either the first author known, or with the optional `$index` parameter any specific index on the array of Authors as described above (returning `NULL` if an invalid index).
`getDateCreated()`	Returns the date on which the current entry was created. Generally only applicable to Atom where it represents the date the resource described by an Atom 1.0 document was created.
`getDateModified()`	Returns the date on which the current entry was last modified
`getContent()`	Returns the content of the current entry (this has any entities reversed if possible assuming the content type is HTML). The description is returned if a separate content element does not exist.
`getEnclosure()`	Returns an array containing the value of all attributes from a multi-media <enclosure> element including as array keys: *url*, *length*, *type*. In accordance with the RSS Best Practices Profile of the RSS Advisory Board, no support is offers for multiple enclosures since such support forms no part of the RSS specification.
`getCommentCount()`	Returns the number of comments made on this entry at the time the feed was last generated
`getCommentLink()`	Returns a URI pointing to the HTML page where comments can be made on this entry
`getCommentFeedLink([string $type = 'atom'|'rss'])`	Returns a URI pointing to a feed of the provided type containing all comments for this entry (type defaults to Atom/RSS depending on current feed type).
`getCategories()`	Returns a `Zend_Feed_Reader_Collection_Category` object containing the details of any categories associated with the entry. The supported fields include "term" (the machine readable category name), "scheme" (the categorisation scheme/ domain for this category), and "label" (a html decoded human readable category name). Where any of the three fields are absent from the field, they are either set to the closest available alternative or, in the case of "scheme", set to `NULL`.

The extended API for entries is identical to that for feeds with the exception of the Iterator methods which are not needed here.

There is often confusion over the concepts of modified and created dates. In Atom, these are two clearly defined concepts (so knock yourself out) but in RSS they are vague. RSS 2.0 defines a single <*pubDate*> element which typically refers to the date this entry was published, i.e. a creation date of sorts. This is not always the case, and it may change with updates or not. As a result, if you really want to check whether an entry has changed, don't rely on the results of `getDateModified()`. Instead, consider tracking the MD5 hash of three other elements concatenated, e.g. using `getTitle()`, `getDescription()` and `getContent()`. If the entry was truly updated, this hash computation will give a different result than previously saved hashes for the same entry. This is obviously content oriented, and will not assist in detecting changes to other relevant elements. Atom feeds should not require such steps.

Further muddying the waters, dates in feeds may follow different standards. Atom and Dublin Core dates should follow ISO 8601, and RSS dates should follow RFC 822 or RFC 2822 which is also common. Date methods will throw an exception if `Zend_Date` cannot load the date string using one of the above standards, or the PHP recognised possibilities for RSS dates.

The values returned from these methods are not validated. This means users must perform validation on all retrieved data including the filtering of any HTML such as from `getContent()` before it is output from your application. Remember that most feeds come from external sources, and therefore the default assumption should be that they cannot be trusted.

Table 56. Extended Entry Level API Methods

`getDomDocument()`	Returns the parent `DOMDocument` object for the entire feed (not just the current entry)
`getElement()`	Returns the current entry level `DOMElement` object
`getXpath()`	Returns the `DOMXPath` object used internally to run queries on the `DOMDocument` object (this includes core and Extension namespaces pre-registered)
`getXpathPrefix()`	Returns the valid DOM path prefix prepended to all XPath queries matching the entry being queried
`getEncoding()`	Returns the encoding of the source XML document (note: this cannot account for errors such as the server sending documents in a different encoding). The default encoding applied in the absence of any other is the UTF-8 encoding of Unicode.
`getExtensions()`	Returns an array of all Extension objects loaded for the current entry (note: both feed-level and entry-level Extensions exist, and only entry-level Extensions are returned

	here). The array keys are in the form {ExtensionName}_Entry.
`getExtension(string $name)`	Returns an Extension object for the entry registered under the provided name. This allows more fine-grained access to Extensions which may otherwise be hidden within the implementation of the standard API methods.
`getType()`	Returns a static class constant (e.g. `Zend_Feed_Reader::TYPE_ATOM_03`, i.e. Atom 0.3) indicating exactly what kind of feed is being consumed.

9.9. Extending Feed and Entry APIs

Extending `Zend_Feed_Reader` allows you to add methods at both the feed and entry level which cover the retrieval of information not already supported by `Zend_Feed_Reader`. Given the number of RSS and Atom extensions that exist, this is a good thing since `Zend_Feed_Reader` couldn't possibly add everything.

There are two types of Extensions possible, those which retrieve information from elements which are immediate children of the root element (e.g. `<channel>` for RSS or `<feed>` for Atom) and those who retrieve information from child elements of an entry (e.g. `<item>` for RSS or `<entry>` for Atom). On the filesystem these are grouped as classes within a namespace based on the extension standard's name. For example, internally we have `Zend_Feed_Reader_Extension_DublinCore_Feed` and `Zend_Feed_Reader_Extension_DublinCore_Entry` classes which are two Extensions implementing Dublin Core 1.0/1.1 support.

Extensions are loaded into `Zend_Feed_Reader` using `Zend_Loader_PluginLoader`, so their operation will be familiar from other Zend Framework components. `Zend_Feed_Reader` already bundles a number of these Extensions, however those which are not used internally and registered by default (so called Core Extensions) must be registered to `Zend_Feed_Reader` before they are used. The bundled Extensions include:

Table 57. Core Extensions (pre-registered)

DublinCore (Feed and Entry)	Implements support for Dublin Core Metadata Element Set 1.0 and 1.1
Content (Entry only)	Implements support for Content 1.0
Atom (Feed and Entry)	Implements support for Atom 0.3 and Atom 1.0
Slash	Implements support for the Slash RSS 1.0 module
WellFormedWeb	Implements support for the Well Formed Web CommentAPI 1.0
Thread	Implements support for Atom Threading Extensions as described in RFC 4685
Podcast	Implements support for the Podcast 1.0 `DTD` from Apple

The Core Extensions are somewhat special since they are extremely common and multi-faceted. For example, we have a Core Extension for Atom. Atom is implemented as an Extension (not

just a base class) because it doubles as a valid RSS module - you can insert Atom elements into RSS feeds. I've even seen RDF feeds which use a lot of Atom in place of more common Extensions like Dublin Core.

Table 58. Non-Core Extensions (must register manually)

Syndication	Implements Syndication 1.0 support for RSS feeds
CreativeCommons	A RSS module that adds an element at the <channel> or <item> level that specifies which Creative Commons license applies.

The additional non-Core Extensions are offered but not registered to `Zend_Feed_Reader` by default. If you want to use them, you'll need to tell `Zend_Feed_Reader` to load them in advance of importing a feed. Additional non-Core Extensions will be included in future iterations of the component.

Registering an Extension with `Zend_Feed_Reader`, so it is loaded and its API is available to Feed and Entry objects, is a simple affair using the `Zend_Loader_PluginLoader`. Here we register the optional Slash Extension, and discover that it can be directly called from the Entry level API without any effort. Note that Extension names are case sensitive and use camel casing for multiple terms.

```
Zend_Feed_Reader::registerExtension('Syndication');
$feed = Zend_Feed_Reader::import('http://rss.slashdot.org/Slashdot/slashdot');
$updatePeriod = $feed->current()->getUpdatePeriod();
```

In the simple example above, we checked how frequently a feed is being updated using the `getUpdatePeriod()` method. Since it's not part of `Zend_Feed_Reader`'s core API, it could only be a method supported by the newly registered Syndication Extension.

As you can also notice, the new methods from Extensions are accessible from the main API using PHP's magic methods. As an alternative, you can also directly access any Extension object for a similar result as seen below.

```
Zend_Feed_Reader::registerExtension('Syndication');
$feed = Zend_Feed_Reader::import('http://rss.slashdot.org/Slashdot/slashdot');
$syndication = $feed->getExtension('Syndication');
$updatePeriod = $syndication->getUpdatePeriod();
```

9.9.1. Writing Zend_Feed_Reader Extensions

Inevitably, there will be times when the `Zend_Feed_Reader` API is just not capable of getting something you need from a feed or entry. You can use the underlying source objects, like `DOMDocument`, to get these by hand however there is a more reusable method available by writing Extensions supporting these new queries.

As an example, let's take the case of a purely fictitious corporation named Jungle Books. Jungle Books have been publishing a lot of reviews on books they sell (from external sources and customers), which are distributed as an RSS 2.0 feed. Their marketing department realises that web applications using this feed cannot currently figure out exactly what book is being reviewed. To make life easier for everyone, they determine that the geek department needs to extend RSS 2.0 to include a new element per entry supplying the ISBN-10 or ISBN-13 number of the publication the entry concerns. They define the new `<isbn>` element quite simply with a standard name and namespace URI:

```
JungleBooks 1.0:
http://example.com/junglebooks/rss/module/1.0/
```

A snippet of RSS containing this extension in practice could be something similar to:

```
<?xml version="1.0" encoding="utf-8" ?>
<rss version="2.0"
   xmlns:content="http://purl.org/rss/1.0/modules/content/"
   xmlns:jungle="http://example.com/junglebooks/rss/module/1.0/">
<channel>
    <title>Jungle Books Customer Reviews</title>
    <link>http://example.com/junglebooks</link>
    <description>Many book reviews!</description>
    <pubDate>Fri, 26 Jun 2009 19:15:10 GMT</pubDate>
    <jungle:dayPopular>
        http://example.com/junglebooks/book/938
    </jungle:dayPopular>
    <item>
        <title>Review Of Flatland: A Romance of Many Dimensions</title>
        <link>http://example.com/junglebooks/review/987</link>
        <author>Confused Physics Student</author>
        <content:encoded>
        A romantic square?!
        </content:encoded>
        <pubDate>Thu, 25 Jun 2009 20:03:28 -0700</pubDate>
        <jungle:isbn>048627263X</jungle:isbn>
    </item>
</channel>
</rss>
```

Implementing this new ISBN element as a simple entry level extension would require the following class (using your own class namespace outside of Zend).

```
class My_FeedReader_Extension_JungleBooks_Entry
    extends Zend_Feed_Reader_Extension_EntryAbstract
{
    public function getIsbn()
    {
        if (isset($this->_data['isbn'])) {
            return $this->_data['isbn'];
        }
        $isbn = $this->_xpath->evaluate(
            'string(' . $this->getXpathPrefix() . '/jungle:isbn)'
        );
        if (!$isbn) {
            $isbn = null;
        }
        $this->_data['isbn'] = $isbn;
        return $this->_data['isbn'];
    }

    protected function _registerNamespaces()
    {
        $this->_xpath->registerNamespace(
            'jungle', 'http://example.com/junglebooks/rss/module/1.0/'
        );
    }
}
```

This extension is easy enough to follow. It creates a new method `getIsbn()` which runs an XPath query on the current entry to extract the ISBN number enclosed by the `<jungle:isbn>` element. It can optionally store this to the internal non-persistent cache (no need to keep querying the DOM if it's called again on the same entry). The value is returned to the caller. At the end we have a protected method (it's abstract so it must exist) which registers the Jungle Books namespace for their custom RSS module. While we call this an RSS module, there's nothing to prevent the same element being used in Atom feeds - and all Extensions which use the prefix provided by `getXpathPrefix()` are actually neutral and work on RSS or Atom feeds with no extra code.

Since this Extension is stored outside of Zend Framework, you'll need to register the path prefix for your Extensions so `Zend_Loader_PluginLoader` can find them. After that, it's merely a matter of registering the Extension, if it's not already loaded, and using it in practice.

```
if(!Zend_Feed_Reader::isRegistered('JungleBooks')) {
    Zend_Feed_Reader::addPrefixPath(
        '/path/to/My/FeedReader/Extension', 'My_FeedReader_Extension'
    );
    Zend_Feed_Reader::registerExtension('JungleBooks');
}
$feed = Zend_Feed_Reader::import('http://example.com/junglebooks/rss');

// ISBN for whatever book the first entry in the feed was concerned with
$firstIsbn = $feed->current()->getIsbn();
```

Writing a feed level Extension is not much different. The example feed from earlier included an unmentioned `<jungle:dayPopular>` element which Jungle Books have added to their standard to include a link to the day's most popular book (in terms of visitor traffic). Here's an Extension which adds a `getDaysPopularBookLink()` method to the feel level API.

```
class My_FeedReader_Extension_JungleBooks_Feed
    extends Zend_Feed_Reader_Extension_FeedAbstract
{
    public function getDaysPopularBookLink()
    {
        if (isset($this->_data['dayPopular'])) {
            return $this->_data['dayPopular'];
        }
        $dayPopular = $this->_xpath->evaluate(
            'string(' . $this->getXpathPrefix() . '/jungle:dayPopular)'
        );
        if (!$dayPopular) {
            $dayPopular = null;
        }
        $this->_data['dayPopular'] = $dayPopular;
        return $this->_data['dayPopular'];
    }

    protected function _registerNamespaces()
    {
        $this->_xpath->registerNamespace(
            'jungle', 'http://example.com/junglebooks/rss/module/1.0/'
        );
    }
}
```

Let's repeat the last example using a custom Extension to show the method being used.

```
if(!Zend_Feed_Reader::isRegistered('JungleBooks')) {
    Zend_Feed_Reader::addPrefixPath(
        '/path/to/My/FeedReader/Extension', 'My_FeedReader_Extension'
    );
    Zend_Feed_Reader::registerExtension('JungleBooks');
}
$feed = Zend_Feed_Reader::import('http://example.com/junglebooks/rss');

// URI to the information page of the day's most popular book with visitors
$daysPopularBookLink = $feed->getDaysPopularBookLink();

// ISBN for whatever book the first entry in the feed was concerned with
$firstIsbn = $feed->current()->getIsbn();
```

Going through these examples, you'll note that we don't register feed and entry Extensions separately. Extensions within the same standard may or may not include both a feed and entry class, so `Zend_Feed_Reader` only requires you to register the overall parent name, e.g. JungleBooks, DublinCore, Slash. Internally, it can check at what level Extensions exist and load them up if found. In our case, we have a full set of Extensions now: `JungleBooks_Feed` and `JungleBooks_Entry`.

10. Zend_Feed_Writer

10.1. Introduction

`Zend_Feed_Writer` is the sibling component to `Zend_Feed_Reader` responsible for generating feeds for output. It supports the Atom 1.0 specification (RFC 4287) and RSS 2.0 as specified by the RSS Advisory Board (RSS 2.0.11). It does not deviate from these standards. It does, however, offer a simple Extension system which allows for any extension/module for either of these two specifications to be implemented if they are not provided out of the box.

In many ways, `Zend_Feed_Writer` is the inverse of `Zend_Feed_Reader`. Where `Zend_Feed_Reader` focused on providing an easy to use architecture fronted by getter methods, `Zend_Feed_Writer` is fronted by similarly named setters or mutators. This ensures the API won't pose a learning curve to anyone familiar with `Zend_Feed_Reader`.

As a result of this design, the rest may even be obvious. Behind the scenes, data set on any `Zend_Feed_Reader` object is translated at render time onto a `DOMDocument` object using the necessary feed elements. For each supported feed type there is both an Atom 1.0 and RSS 2.0 renderer. Using a `DOMDocument` rather a templating solution has numerous advantages, the most obvious being the ability to export the `DOMDocument` for additional processing and relying on PHP DOM for correct and valid rendering.

As with `Zend_Feed_Reader`, `Zend_Feed_Writer` is a standalone replacement for `Zend_Feed`'s Builder architecture and is not compatible with those classes.

10.2. Architecture

The architecture of `Zend_Feed_Writer` is very simple. It has two core sets of classes: containers and renderers.

The containers include the `Zend_Feed_Writer_Feed` and `Zend_Feed_Writer_Entry` classes. The Entry classes can be attached to any Feed class. The sole purpose of these containers is to collect data about the feed to generate using a simple interface of setter methods. These methods perform some data validity testing. For example, it will validate any passed URIs, dates, etc. These checks are not tied to any of the feed standards. The container objects also

contain methods to allow for fast rendering and export of the final feed, and these can be reused at will.

While there are two data containers, there are four renderers - two matching container renderers per support feed type. The renderer accept a container, and based on its content attempt to generate a valid feed. If the renderer is unable to generate a valid feed, perhaps due to the container missing an obligatory data point, it will report this by throwing an `Exception`. While it is possible to ignore `Exceptions`, this removes the default safeguard of ensuring you have sufficient data set to render a wholly valid feed.

Due to the system being divided between data containers and renderers, it can make Extensions somewhat interesting. A typical Extension offering namespaced feed and entry level elements, must itself reflect the exact same architecture, i.e. offer feed and entry level data containers, and matching renderers. There is, fortunately, no complex integration work required since all Extension classes are simply registered and automatically used by the core classes. We'll meet Extensions in more detail at the end of this section.

10.3. Getting Started

Using `Zend_Feed_Reader` is as simple as setting data and triggering the renderer. Here is an example to generate a minimal Atom 1.0 feed.

```
/**
 * Create the parent feed
 */
$feed = new Zend_Feed_Writer_Feed;
$feed->setTitle('Paddy\'s Blog');
$feed->setLink('http://www.example.com');
$feed->setFeedLink('http://www.example.com/atom', 'atom');
$feed->addAuthor(array(
    'name'  => 'Paddy',
    'email' => 'paddy@example.com',
    'uri'   => 'http://www.example.com',
));
$feed->setDateModified(time());
$feed->addHub('http://pubsubhubbub.appspot.com/');

/**
 * Add one or more entries. Note that entries must
 * be manually added once created.
 */
$entry = $feed->createEntry();
$entry->setTitle('All Your Base Are Belong To Us');
$entry->setLink('http://www.example.com/all-your-base-are-belong-to-us');
$entry->addAuthor(array(
    'name'  => 'Paddy',
    'email' => 'paddy@example.com',
    'uri'   => 'http://www.example.com',
));
$entry->setDateModified(time());
$entry->setDateCreated(time());
$entry->setDescription('Exposing the difficultly of porting games to English.');
$entry->setContent('I am not writing the article. The example is long enough as is ;).');
$feed->addEntry($entry);

/**
 * Render the resulting feed to Atom 1.0 and assign to $out.
 * You can substitute "atom" with "rss" to generate an RSS 2.0 feed.
```

```
 */
$out = $feed->export('atom');
```

The output rendered should be as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<feed xmlns="http://www.w3.org/2005/Atom">
    <title type="text">Paddy's Blog</title>
    <subtitle type="text">Writing about PC Games since 176 BC.</subtitle>
    <updated>2009-12-14T20:28:18+00:00</updated>
    <generator uri="http://framework.zend.com" version="1.10.0alpha">
        Zend_Feed_Writer
    </generator>
    <link rel="alternate" type="text/html" href="http://www.example.com"/>
    <link rel="self" type="application/atom+xml" href="http://www.example.com/atom"/>
    <id>http://www.example.com</id>
    <author>
        <name>Paddy</name>
        <email>paddy@example.com</email>
        <uri>http://www.example.com</uri>
    </author>
    <link rel="hub" href="http://pubsubhubbub.appspot.com/"/>
    <entry>
        <title type="html"><![CDATA[All Your Base Are Belong To Us]]></title>
        <summary type="html">
            <![CDATA[Exposing the difficultly of porting games to English.]]>
        </summary>
        <published>2009-12-14T20:28:18+00:00</published>
        <updated>2009-12-14T20:28:18+00:00</updated>
        <link rel="alternate" type="text/html" href="http://www.example.com/all-your-base-a
        <id>http://www.example.com/all-your-base-are-belong-to-us</id>
        <author>
        <name>Paddy</name>
        <email>paddy@example.com</email>
        <uri>http://www.example.com</uri>
        </author>
        <content type="html">
            <![CDATA[I am not writing the article. The example is long enough as is ;).]]>
        </content>
    </entry>
</feed>
```

This is a perfectly valid Atom 1.0 example. It should be noted that omitting an obligatory point of data, such as a title, will trigger an `Exception` when rendering as Atom 1.0. This will differ for RSS 2.0 since a title may be omitted so long as a description is present. This gives rise to Exceptions that differ between the two standards depending on the renderer in use. By design, `Zend_Feed_Writer` will not render an invalid feed for either standard unless the end-user deliberately elects to ignore all Exceptions.

10.4. Setting Feed Data Points

Before you can render a feed, you must first setup the data necessary for the feed being rendered. This utilises a simple setter style API which doubles as an initial method for validating the data being set. By design, the API closely matches that for `Zend_Feed_Reader` to avoid undue confusion and uncertainty.

`Zend_Feed_Writer` offers this API via its data container classes `Zend_Feed_Writer_Feed` and `Zend_Feed_Writer_Entry`. These classes merely store all feed data in type-agnostic

manner, meaning you may reuse any data container with any renderer without requiring additional work. Both classes are also amenable to Extensions, meaning that an Extension may define its own container classes which are registered to the base container classes as extensions, and are checked when any method call triggers the base container's `__call()` method.

Here's a summary of the Core API for Feeds. You should note it comprises not only the basic RSS and Atom standards, but also accounts for a number of included Extensions bundled with `Zend_Feed_Writer`. The naming of these Extension sourced methods remain fairly generic - all Extension methods operate at the same level as the Core API though we do allow you to retrieve any specific Extension object separately if required.

Table 59. Feed Level API Methods

`setId()`	Set a unique ID associated with this feed. For Atom 1.0 this is an atom:id element, whereas for RSS 2.0 it is added as a guid element. These are optional so long as a link is added, i.e. the link is set as the ID.
`setTitle()`	Set the title of the feed.
`setDescription()`	Set the text description of the feed.
`setLink()`	Set a URI to the HTML website containing the same or similar information as this feed (i.e. if the feed is from a blog, it should provide the blog's URI where the HTML version of the entries can be read).
`setFeedLinks()`	Add a link to an XML feed, whether the feed being generated or an alternate URI pointing to the same feed but in a different format. At a minimum, it is recommended to include a link to the feed being generated so it has an identifiable final URI allowing a client to track its location changes without necessitating constant redirects. The parameter is an array of arrays, where each sub-array contains the keys "type" and "uri". The type should be one of "atom", "rss", or "rdf". If a type is omitted, it defaults to the type used when rendering the feed.
`setAuthors()`	Sets the data for authors. The parameter is an array of arrays where each sub-array may contain the keys "name", "email" and "uri". The "uri" value is only applicable for Atom feeds since RSS contains no facility to show it. For RSS 2.0, rendering will create two elements - an author element containing the email reference with the name in brackets, and a Dublin Core creator element only containing the name.
`setAuthor()`	Sets the data for a single author following the same format as described above for a single sub-array.

`setDateCreated()`	Sets the date on which this feed was created. Generally only applicable to Atom where it represents the date the resource described by an Atom 1.0 document was created. The expected parameter may be a UNIX timestamp or a `Zend_Date` object.
`getDateModified()`	Sets the date on which this feed was last modified. The expected parameter may be a UNIX timestamp or a `Zend_Date` object.
`setLanguage()`	Sets the language of the feed. This will be omitted unless set.
`getGenerator()`	Allows the setting of a generator. The parameter should be an array containing the keys "name", "version" and "uri". If omitted a default generator will be added referencing `Zend_Feed_Writer`, the current Zend Framework version and the Framework's URI.
`setCopyright()`	Sets a copyright notice associated with the feed.
`setHubs()`	Accepts an array of Pubsubhubbub Hub Endpoints to be rendered in the feed as Atom links so that PuSH Subscribers may subscribe to your feed. Note that you must implement a Pubsubhubbub Publisher in order for real-time updates to be enabled. A Publisher may be implemented using `Zend_Feed_Pubsubhubbub_Publisher`.
`setCategories()`	Accepts an array of categories for rendering, where each element is itself an array whose possible keys include "term", "label" and "scheme". The "term" is a typically a category name suitable for inclusion in a URI. The "label" may be a human readable category name supporting special characters (it is encoded during rendering) and is a required key. The "scheme" (called the domain in RSS) is optional but must be a valid URI.

10.5. Setting Entry Data Points

Here's a summary of the Core API for Entries/Items. You should note it comprises not only the basic RSS and Atom standards, but also accounts for a number of included Extensions bundled with `Zend_Feed_Writer`. The naming of these Extension sourced methods remain fairly generic - all Extension methods operate at the same level as the Core API though we do allow you to retrieve any specific Extension object separately if required.

Table 60. Entry Level API Methods

`setId()`	Set a unique ID associated with this feed. For Atom 1.0 this is an atom:id element, whereas for RSS 2.0 it is added as a guid element.

	These are optional so long as a link is added, i.e. the link is set as the ID.
`setTitle()`	Set the title of the feed.
`setDescription()`	Set the text description of the feed.
`setLink()`	Set a URI to the HTML website containing the same or similar information as this feed (i.e. if the feed is from a blog, it should provide the blog's URI where the HTML version of the entries can be read).
`setFeedLinks()`	Add a link to an XML feed, whether the feed being generated or an alternate URI pointing to the same feed but in a different format. At a minimum, it is recommended to include a link to the feed being generated so it has an identifiable final URI allowing a client to track its location changes without necessitating constant redirects. The parameter is an array of arrays, where each sub-array contains the keys "type" and "uri". The type should be one of "atom", "rss", or "rdf". If a type is omitted, it defaults to the type used when rendering the feed.
`setAuthors()`	Sets the data for authors. The parameter is an array of arrays where each sub-array may contain the keys "name", "email" and "uri". The "uri" value is only applicable for Atom feeds since RSS contains no facility to show it. For RSS 2.0, rendering will create two elements - an author element containing the email reference with the name in brackets, and a Dublin Core creator element only containing the name.
`setAuthor()`	Sets the data for a single author following the same format as described above for a single sub-array.
`setDateCreated()`	Sets the date on which this feed was created. Generally only applicable to Atom where it represents the date the resource described by an Atom 1.0 document was created. The expected parameter may be a UNIX timestamp or a `Zend_Date` object.
`getDateModified()`	Sets the date on which this feed was last modified. The expected parameter may be a UNIX timestamp or a `Zend_Date` object.
`setLanguage()`	Sets the language of the feed. This will be omitted unless set.
`getGenerator()`	Allows the setting of a generator. The parameter should be an array containing the keys "name", "version" and "uri". If omitted a default generator will be added

	referencing `Zend_Feed_Writer`, the current Zend Framework version and the Framework's URI.
`setCopyright()`	Sets a copyright notice associated with the feed.
`setHubs()`	Accepts an array of Pubsubhubbub Hub Endpoints to be rendered in the feed as Atom links so that PuSH Subscribers may subscribe to your feed. Note that you must implement a Pubsubhubbub Publisher in order for real-time updates to be enabled. A Publisher may be implemented using `Zend_Feed_Pubsubhubbub_Publisher`.
`setCategories()`	Accepts an array of categories for rendering, where each element is itself an array whose possible keys include "term", "label" and "scheme". The "term" is a typically a category name suitable for inclusion in a URI. The "label" may be a human readable category name supporting special characters (it is encoded during rendering) and is a required key. The "scheme" (called the domain in RSS) is optional but must be a valid URI.

11. Zend_Feed_Pubsubhubbub

`Zend_Feed_Pubsubhubbub` is an implementation of the PubSubHubbub Core 0.2 Specification (Working Draft). It offers implementations of a Pubsubhubbub Publisher and Subscriber suited to Zend Framework and other PHP applications.

11.1. What is Pubsubhubbub?

Pubsubhubbub is an open, simple web-scale pubsub protocol. A common use case to enable blogs (Publishers) to "push" updates from their RSS or Atom feeds (Topics) to end Subscribers. These Subscribers will have subscribed to the blog's RSS or Atom feed via a Hub, a central server which is notified of any updates by the Publisher and which then distributes these updates to all Subscribers. Any feed may advertise that it supports one or more Hubs using an Atom namespaced link element with a rel attribute of "hub".

Pubsubhubbub has garnered attention because it is a pubsub protocol which is easy to implement and which operates over HTTP. Its philosophy is to replace the traditional model where blog feeds have been polled at regular intervals to detect and retrieve updates. Depending on the frequency of polling, this can take a lot of time to propagate updates to interested parties from planet aggregators to desktop readers. With a pubsub system in place, updates are not simply polled by Subscribers, they are pushed to Subscribers, elimenating any delay. For this reason, Pubsubhubbub forms part of what has been dubbed the real-time web.

The protocol does not exist in isolation. Pubsub systems have been around for a while, such as the familiar Jabber Publish-Subscribe protocol, XEP-0060, or the less well known rssCloud (described in 2001). However these have not achieved widespread adoption typically due to either their complexity, poor timing or lack of suitability for web applications. rssCloud, which was recently revived as a response to the appearance of Pubsubhubbub, has also seen its usage

increase significantly though it lacks a formal specification and currently does not support Atom 1.0 feeds.

Perhaps surprisingly given its relative early age, Pubsubhubbub is already in use including in Google Reader, Feedburner, and there are plugins available for Wordpress blogs.

11.2. Architecture

`Zend_Feed_Pubsubhubbub` implements two sides of the Pubsubhubbub 0.2 Specification: a Publisher and a Subscriber. It does not currently implement a Hub Server though this is in progress for a future Zend Framework release.

A Publisher is responsible for notifying all supported Hubs (many can be supported to add redundancy to the system) of any updates to its feeds, whether they be Atom or RSS based. This is achieved by pinging the supported Hub Servers with the URL of the updated feed. In Pubsubhubbub terminology, any updatable resource capable of being subscribed to is referred to as a Topic. Once a ping is received, the Hub will request the updated feed, process it for updated items, and forward all updates to all Subscribers subscribed to that feed.

A Subscriber is any party or application which subscribes to one or more Hubs to receive updates from a Topic hosted by a Publisher. The Subscriber never directly communicates with the Publisher since the Hub acts as an intermediary, accepting subscriptions and sending updates to subscribed Subscribers. The Subscriber therefore communicates only with the Hub, either to subscribe/unsubscribe to Topics, or when it receives updates from the Hub. This communication design ("Fat Pings") effectively removes the possibility of a "Thundering Herd" issue. This occurs in a pubsub system where the Hub merely informs Subscribers that an update is available, prompting all Subscribers to immediately retrieve the feed from the Publisher giving rise to a traffic spike. In Pubsubhubbub, the Hub distributes the actual update in a "Fat Ping" so the Publisher is not subjected to any traffic spike.

`Zend_Feed_Pubsubhubbub` implements Pubsubhubbub Publishers and Subscribers with the classes `Zend_Feed_Pubsubhubbub_Publisher` and `Zend_Feed_Pubsubhubbub_Subscriber`. In addition, the Subscriber implementation may handle any feed updates forwarded from a Hub by using `Zend_Feed_Pubsubhubbub_Subscriber_Callback`. These classes, their use cases, and APIs are covered in subsequent sections.

11.3. Zend_Feed_Pubsubhubbub_Publisher

In Pubsubhubbub, the Publisher is the party who publishes a live feed and frequently updates it with new content. This may be a blog, an aggregator, or even a web service with a public feed based API. In order for these updates to be pushed to Subscribers, the Publisher must notify all of its supported Hubs that an update has occured using a simple HTTP POST request containing the URI or the updated Topic (i.e the updated RSS or Atom feed). The Hub will confirm receipt of the notification, fetch the updated feed, and forward any updates to any Subscribers who have subscribed to that Hub for updates from the relevant feed.

By design, this means the Publisher has very little to do except send these Hub pings whenever its feeds change. As a result, the Publisher implementation is extremely simple to use and requires very little work to setup and use when feeds are updated.

`Zend_Feed_Pubsubhubbub_Publisher` implements a full Pubsubhubbub Publisher. Its setup for use is also simple, requiring mainly that it is configured with the URI endpoint for all Hubs to be notified of updates, and the URIs of all Topics to be included in the notifications.

The following example shows a Publisher notifying a collection of Hubs about updates to a pair of local RSS and Atom feeds. The class retains a collection of errors which include the Hub URLs, so the notification can be re-attempted later and/or logged if any notifications happen to fail. Each resulting error array also includes a "response" key containing the related HTTP response object. In the event of any errors, it is strongly recommended to attempt the operation for failed Hub Endpoints at least once more at a future time. This may require the use of either a scheduled task for this purpose or a job queue though such extra steps are optional.

```
$publisher = new Zend_Feed_Pubsubhubbub_Publisher;
$publisher->addHubUrls(array(
    'http://pubsubhubbub.appspot.com/',
    'http://hubbub.example.com',
));
$publisher->addUpdatedTopicUrls(array(
    'http://www.example.net/rss',
    'http://www.example.net/atom',
));
$publisher->notifyAll();

if (!$publisher->isSuccess()) {
    // check for errors
    $errors     = $publisher->getErrors();
    $failedHubs = array()
    foreach ($errors as $error) {
        $failedHubs[] = $error['hubUrl'];
    }
}

// reschedule notifications for the failed Hubs in $failedHubs
```

If you prefer having more concrete control over the Publisher, the methods `addHubUrls()` and `addUpdatedTopicUrls()` pass each array value to the singular `addHubUrl()` and `addUpdatedTopicUrl()` public methods. There are also matching `removeUpdatedTopicUrl()` and `removeHubUrl()` methods.

You can also skip setting Hub URIs, and notify each in turn using the `notifyHub()` method which accepts the URI of a Hub endpoint as its only argument.

There are no other tasks to cover. The Publisher implementation is very simple since most of the feed processing and distribution is handled by the selected Hubs. It is however important to detect errors and reschedule notifications as soon as possible (with a reasonable maximum number of retries) to ensure notifications reach all Subscribers. In many cases as a final alternative, Hubs may frequently poll your feeds to offer some additional tolerance for failures both in terms of their own temporary downtime or Publisher errors/downtime.

11.4. Zend_Feed_Pubsubhubbub_Subscriber

In Pubsubhubbub, the Subscriber is the party who wishes to receive updates to any Topic (RSS or Atom feed). They achieve this by subscribing to one or more of the Hubs advertised by that Topic, usually as a set of one or more Atom 1.0 links with a rel attribute of "hub". The Hub from that point forward will send an Atom or RSS feed containing all updates to that Subscriber's Callback URL when it receives an update notification from the Publisher. In this way, the Subscriber need never actually visit the original feed (though it's still recommended at some level to ensure updates are retrieved if ever a Hub goes offline). All subscription requests must contain the URI of the Topic being subscribed and a Callback URL which the Hub will use to confirm the subscription and to forward updates.

The Subsciber therefore has two roles. To create and manage subscriptions, including subscribing for new Topics with a Hub, unsubscribing (if necessary), and periodically renewing subscriptions since they may have a limited validity as set by the Hub. This is handled by `Zend_Feed_Pubsubhubbub_Subscriber`.

The second role is to accept updates sent by a Hub to the Subscriber's Callback URL, i.e. the URI the Subscriber has assigned to handle updates. The Callback URL also handles events where the Hub contacts the Subscriber to confirm all subscriptions and unsubscriptions. This is handled by using an instance of `Zend_Feed_Pubsubhubbub_Subscriber_Callback` when the Callback URL is accessed.

`Zend_Feed_Pubsubhubbub_Subscriber` implements the Pubsubhubbub 0.2 Specification. As this is a new specification version not all Hubs currently implement it. The new specification allows the Callback URL to include a query string which is used by this class, but not supported by all Hubs. In the interests of maximising compatibility it is therefore recommended that the query string component of the Subscriber Callback URI be presented as a path element, i.e. recognised as a parameter in the route associated with the Callback URI and used by the application's Router.

11.4.1. Subscribing and Unsubscribing

`Zend_Feed_Pubsubhubbub_Subscriber` implements a full Pubsubhubbub Subscriber capable of subscribing to, or unsubscribing from, any Topic via any Hub advertised by that Topic. It operates in conjunction with `Zend_Feed_Pubsubhubbub_Subscriber_Callback` which accepts requests from a Hub to confirm all subscription or unsubscription attempts (to prevent third-party misuse).

Any subscription (or unsubscription) requires the relevant information before proceeding, i.e. the URI of the Topic (Atom or RSS feed) to be subscribed to for updates, and the URI of the endpoint for the Hub which will handle the subscription and forwarding of the updates. The lifetime of a subscription may be determined by the Hub but most Hubs should support automatic subscription refreshes by checking with the Subscriber. This is supported by `Zend_Feed_Pubsubhubbub_Subscriber_Callback` and requires no other work on your part. It is still strongly recommended that you use the Hub sourced subscription time to live (ttl) to schedule the creation of new subscriptions (the process is identical to that for any new subscription) to refresh it with the Hub. While it should not be necessary per se, it covers cases where a Hub may not support automatic subscription refreshing and rules out Hub errors for additional redundancy.

With the relevant information to hand, a subscription can be attempted as demonstrated below:

```
$storage = new Zend_Feed_Pubsubhubbub_Model_Subscription;

    $subscriber = new Zend_Feed_Pubsubhubbub_Subscriber;
    $subscriber->setStorage($storage);
    $subscriber->addHubUrl('http://hubbub.example.com');
    $subscriber->setTopicUrl('http://www.example.net/rss.xml');
    $subscriber->setCallbackUrl('http://www.mydomain.com/hubbub/callback');
    $subscriber->subscribeAll();
```

In order to store subscriptions and offer access to this data for general use, the component requires a database (a schema is provided later in this section). By default, it is assumed the table name is "subscription" and it utilises `Zend_Db_Table_Abstract` in the background

meaning it will use the default adapter you have set for your application. You may also pass a specific custom `Zend_Db_Table_Abstract` instance into the associated model `Zend_Feed_Pubsubhubbub_Model_Subscription`. This custom adapter may be as simple in intent as changing the table name to use or as complex as you deem necessary.

While this Model is offered as a default ready-to-roll solution, you may create your own Model using any other backend or database layer (e.g. Doctrine) so long as the resulting class implements the interface `Zend_Feed_Pubsubhubbub_Model_SubscriptionInterface`.

An example schema (MySQL) for a subscription table accessible by the provided model may look similar to:

```
CREATE TABLE IF NOT EXISTS `subscription` (
  `id` varchar(32) COLLATE utf8_unicode_ci NOT NULL DEFAULT '',
  `topic_url` varchar(255) COLLATE utf8_unicode_ci DEFAULT NULL,
  `hub_url` varchar(255) COLLATE utf8_unicode_ci DEFAULT NULL,
  `created_time` datetime DEFAULT NULL,
  `lease_seconds` bigint(20) DEFAULT NULL,
  `verify_token` varchar(255) COLLATE utf8_unicode_ci DEFAULT NULL,
  `secret` varchar(255) COLLATE utf8_unicode_ci DEFAULT NULL,
  `expiration_time` datetime DEFAULT NULL,
  `subscription_state` varchar(12) COLLATE utf8_unicode_ci DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8 COLLATE=utf8_unicode_ci;
```

Behind the scenes, the Subscriber above will send a request to the Hub endpoint containing the following parameters (based on the previous example):

Table 61. Subscription request parameters

Parameter	Value	Explanation
hub.callback	http://www.mydomain.com/hubbub/callback?xhub.subscription=5536df06b5d[illegible]4cd5621a3c16a8184	The URI used by a Hub to contact the Subscriber and a (un)subscription request or send updates from subscribed feeds. The appended query string contains a custom parameter (hence the xhub designation). It is a query string parameter preserved by the Hub and resent with all Subscriber requests. Its purpose is to allow the Subscriber to identify and look up the subscription associated with any Hub request in a backend storage medium. This is a non-standard parameter used by this component in preference to encoding a subscription key in the URI path which is more difficult to implement in a Zend Framework application.

Parameter	Value	Explanation
		Nevertheless, since not all Hubs support query string parameters, we still strongly recommend adding the subscription key as a path component in the form http://www.mydomain.com/hubbub/callback/5536df06b5dcb966edab3a4c4d56213c To accomplish this, it requires defining a route capable of parsing out the final value of the key and then retrieving the value and passing it to the Subscriber Callback object. The value would be passed into the method `Zend_Pubsubhubbub_Subscriber_Callba` A detailed example is offered later.
hub.lease_seconds	2592000	The number of seconds for which the Subscriber would like a new subscription to remain valid for (i.e. a TTL). Hubs may enforce their own maximum subscription period. All subscriptions should be renewed by simply re-subscribing before the subscription period ends to ensure continuity of updates. Hubs should additionally attempt to automatically refresh subscriptions before they expire by contacting Subscribers (handled automatically by the Callback class).
hub.mode	subscribe	Simple value indicating this is a subscription request. Unsubscription requests would use the "unsubscribe" value.
hub.topic	http://www.example.net/rss.xml	The URI of the topic (i.e. Atom or RSS feed) which the Subscriber wishes to subscribe to for updates.
hub.verify	sync	Indicates to the Hub the preferred mode of verifying subscriptions or unsubscriptions. It is repeated twice in order of preference. Technically this component

Parameter	Value	Explanation
		does not distinguish between the two modes and treats both equally.
hub.verify	async	Indicates to the Hub the preferred mode of verifying subscriptions or unsubscriptions. It is repeated twice in order of preference. Technically this component does not distinguish between the two modes and treats both equally.
hub.verify_token	3065919804abcaa7212ae89.87[illegible]	An optional token returned to the Subscriber by the Hub when it is confirming a subscription or unsubscription. Offers a measure of reliance that the confirmation request originates from the correct Hub to prevent misuse.

You can modify several of these parameters to indicate a different preference. For example, you can set a different lease seconds value using `Zend_Pubsubhubbub_Subscriber::setLeaseSeconds()` or show a preference for the async verify mode by using `setPreferredVerificationMode(Zend_Feed_Pubsubhubbub::VERIFICATION_MODE_ASYNC)`. However the Hubs retain the capability to enforce their own preferences and for this reason the component is deliberately designed to work across almost any set of options with minimum end-user configuration required. Conventions are great when they work!

While Hubs may require the use of a specific verification mode (both are supported by `Zend_Pubsubhubbub`), you may indicate a specific preference using the `setPreferredVerificationMode()` method. In "sync" (synchronous) mode, the Hub attempts to confirm a subscription as soon as it is received, and before responding to the subscription request. In "async" (asynchronous) mode, the Hub will return a response to the subscription request immediately, and its verification request may occur at a later time. Since `Zend_Pubsubhubbub` implements the Subscriber verification role as a separate callback class and requires the use of a backend storage medium, it actually supports both transparently though in terms of end-user performance, asynchronous verification is very much preferred to eliminate the impact of a poorly performing Hub tying up end-user server resources and connections for too long.

Unsubscribing from a Topic follows the exact same pattern as the previous example, with the exception that we should call `unsubscribeAll()` instead. The parameters included are identical to a subscription request with the exception that "hub.mode" is set to "unsubscribe".

By default, a new instance of `Zend_Pubsubhubbub_Subscriber` will attempt to use a database backed storage medium which defaults to using the default `Zend_Db` adapter with a table name of "subscription". It is recommended to set a custom storage solution where these

defaults are not apt either by passing in a new Model supporting the required interface or by passing a new instance of `Zend_Db_Table_Abstract` to the default Model's constructor to change the used table name.

11.4.2. Handling Subscriber Callbacks

Whenever a subscription or unsubscription request is made, the Hub must verify the request by forwarding a new verification request to the Callback URL set in the subscription/unsubscription parameters. To handle these Hub requests, which will include all future communications containing Topic (feed) updates, the Callback URL should trigger the execution of an instance of `Zend_Pubsubhubbub_Subscriber_Callback` to handle the request.

The Callback class should be configured to use the same storage medium as the Subscriber class. Using it is quite simple since most of its work is performed internally.

```
$storage = new Zend_Feed_Pubsubhubbub_Model_Subscription;
    $callback = new Zend_Feed_Pubsubhubbub_Subscriber_Callback;
    $callback->setStorage($storage);
    $callback->handle();
    $callback->sendResponse();

    /**
     * Check if the callback resulting in the receipt of a feed update.
     * Otherwise it was either a (un)sub verification request or invalid request.
     * Typically we need do nothing other than add feed update handling - the rest
     * is handled internally by the class.
     */
    if ($callback->hasFeedUpdate()) {
        $feedString = $callback->getFeedUpdate();
        /**
         *  Process the feed update asynchronously to avoid a Hub timeout.
         */
    }
```

It should be noted that `Zend_Feed_Pubsubhubbub_Subscriber_Callback` may independently parse any incoming query string and other parameters. This is necessary since PHP alters the structure and keys of a query string when it is parsed into the `$_GET` or `$_POST` superglobals. For example, all duplicate keys are ignored and periods are converted to underscores. Pubsubhubbub features both of these in the query strings it generates.

It is essential that developers recognise that Hubs are only concerned with sending requests and receiving a response which verifies its receipt. If a feed update is received, it should never be processed on the spot since this leaves the Hub waiting for a response. Rather, any processing should be offloaded to another process or deferred until after a response has been returned to the Hub. One symptom of a failure to promptly complete Hub requests is that a Hub may continue to attempt delivery of the update/verification request leading to duplicated update attempts being processed by the Subscriber. This appears problematic - but in reality a Hub may apply a timeout of just a few seconds, and if no response is received within that time it may disconnect (assuming a delivery failure) and retry later. Note that Hubs are expected to distribute vast volumes of updates so their resources are stretched - please do process

feeds asynchronously (e.g. in a separate process or a job queue or even a cron scheduled task) as much as possible.

11.4.3. Setting Up And Using A Callback URL Route

As noted earlier, the `Zend_Feed_Pubsubhubbub_Subscriber_Callback` class receives the combined key associated with any subscription from the Hub via one of two methods. The technically preferred method is to add this key to the Callback URL employed by the Hub in all future requests using a query string parameter with the key "xhub.subscription". However, for historical reasons, primarily that this was not supported in Pubsubhubbub 0.1 (it was recently added in 0.2 only), it is strongly recommended to use the most compatible means of adding this key to the Callback URL by appending it to the URL's path.

Thus the URL http://www.example.com/callback?xhub.subscription=key would become http://www.example.com/callback/key.

Since the query string method is the default in anticipation of a greater level of future support for the full 0.2 specification, this requires some additional work to implement.

The first step to to make the `Zend_Feed_Pubsubhubbub_Subscriber_Callback` class aware of the path contained subscription key. It's manually injected therefore since it also requires manually defining a route for this purpose. This is achieved simply by called the method `Zend_Feed_Pubsubhubbub_Subscriber_Callback::setSubscriptionKey()` with the parameter being the key value available from the Router. The example below demonstrates this using a Zend Framework controller.

```
class CallbackController extends Zend_Controller_Action
{

    public function indexAction()
    {
        $storage = new Zend_Feed_Pubsubhubbub_Model_Subscription;
        $callback = new Zend_Feed_Pubsubhubbub_Subscriber_Callback;
        $callback->setStorage($storage);
        /**
         * Inject subscription key parsing from URL path using
         * a parameter from Router.
         */
        $subscriptionKey = $this->_getParam('subkey');
        $callback->setSubscriptionKey($subscriptionKey);
        $callback->handle();
        $callback->sendResponse();

        /**
         * Check if the callback resulting in the receipt of a feed update.
         * Otherwise it was either a (un)sub verification request or invalid request.
         * Typically we need do nothing other than add feed update handling - the rest
         * is handled internally by the class.
         */
        if ($callback->hasFeedUpdate()) {
            $feedString = $callback->getFeedUpdate();
            /**
             *  Process the feed update asynchronously to avoid a Hub timeout.
             */
        }
    }
```

```
}
```

Actually adding the route which would map the path-appended key to a parameter for retrieval from a controller can be accomplished using a Route configuration such as the INI formatted example below for use with `Zend_Application` bootstrapping.

```
; Callback Route to enable appending a PuSH Subscription's lookup key
resources.router.routes.callback.route = "callback/:subkey"
resources.router.routes.callback.defaults.module = "default"
resources.router.routes.callback.defaults.controller = "callback"
resources.router.routes.callback.defaults.action = "index"
```

Zend_File

1. Zend_File_Transfer

`Zend_File_Transfer` provides extensive support for file uploads and downloads. It comes with built-in validators for files plus functionality to change files with filters. Protocol adapters allow `Zend_File_Transfer` to expose the same API for transport protocols like HTTP, FTP, WEBDAV and more.

Limitation

The current implementation of `Zend_File_Transfer` is limited to HTTP Post Uploads. Other adapters supporting downloads and other protocols will be added in future releases. Unimplemented methods will throw an exception. For now, you should use `Zend_File_Transfer_Adapter_Http` directly. As soon as there are multiple adapters available you can use a common interface.

Forms

When you are using `Zend_Form` you should use the APIs provided by `Zend_Form` and not `Zend_File_Transfer` directly. The file transfer support in `Zend_Form` is implemented with `Zend_File_Transfer`, so the information in this chapter may be useful for advanced users of `Zend_Form`.

The usage of `Zend_File_Transfer` is relatively simple. It consists of two parts. The HTTP form does the upload, while the `Zend_File_Transfer` handles the uploaded files. See the following example:

Example 340. Simple Form for Uploading Files

This example illustrates basic file uploading. The first part is the file form. In our example there is one file to upload.

```
<form enctype="multipart/form-data" action="/file/upload" method="POST">
    <input type="hidden" name="MAX_FILE_SIZE" value="100000" />
        Choose a file to upload: <input name="uploadedfile" type="file" />
    <br />
    <input type="submit" value="Upload File" />
</form>
```

For convenience, you can use Zend_Form_Element_File instead of building the HTML manually.

The next step is to create the receiver of the upload. In our example the receiver is located at `/file/upload`. So next we will create the `file` controller and the `upload` action.

```
$adapter = new Zend_File_Transfer_Adapter_Http();

$adapter->setDestination('C:\temp');

if (!$adapter->receive()) {
    $messages = $adapter->getMessages();
    echo implode("\n", $messages);
}
```

This code listing demonstrates the simplest usage of `Zend_File_Transfer`. A local destination is set with the `setDestination` method, then the `receive()` method is called. if there are any upload errors, an error will be returned.

Attention

This example is suitable only for demonstrating the basic API of `Zend_File_Transfer`. You should *never* use this code listing in a production environment, because severe security issues may be introduced. You should always use validators to increase security.

1.1. Supported Adapters for Zend_File_Transfer

`Zend_File_Transfer` is designed to support a variety of adapters and transfer directions. With `Zend_File_Transfer` you can upload, download and even forward (upload one adapter and download with another adapter at the same time) files.

1.2. Options for Zend_File_Transfer

`Zend_File_Transfer` and its adapters support different options. You can set all options either by passing them to the constructor or by calling `setOptions($options)`. `getOptions()` will return the options that are currently set. The following is a list of all supported options.

- *ignoreNoFile*: If this option is set to `TRUE`, all validators will ignore files that have not been uploaded by the form. The default value is `FALSE` which results in an error if no files were specified.

1.3. Checking Files

`Zend_File_Transfer` has several methods that check for various states of the specified file. These are useful if you must process files after they have been uploaded. These methods include:

- *isValid($files = null)*: This method will check if the given files are valid, based on the validators that are attached to the files. If no files are specified, all files will be checked. You can call `isValid()` before calling `receive()`; in this case, `receive()` will not call `isValid` internally again when receiving the file.
- *isUploaded($files = null)*: This method will check if the specified files have been uploaded by the user. This is useful when you have defined one or more optional files. When no files are specified, all files will be checked.
- *isReceived($files = null)*: This method will check if the given files have already been received. When no files are specified, all files will be checked.

Example 341. Checking Files

```
$upload = new Zend_File_Transfer();

// Returns all known internal file information
$files = $upload->getFileInfo();

foreach ($files as $file => $info) {
    // file uploaded ?
    if (!$upload->isUploaded($file)) {
        print "Why havn't you uploaded the file ?";
        continue;
    }

    // validators are ok ?
    if (!$upload->isValid($file)) {
        print "Sorry but $file is not what we wanted";
        continue;
    }
}

$upload->receive();
```

1.4. Additional File Informations

`Zend_File_Transfer` can return additional information on files. The following methods are available:

- *getFileName($file = null, $path = true)*: This method will return the real file name of a transferred file.
- *getFileInfo($file = null)*: This method will return all internal information for the given file.
- *getFileSize($file = null)*: This method will return the real filesize for the given file.
- *getHash($hash = 'crc32', $files = null)*: This method returns a hash of the content of a given transferred file.
- *getMimeType($files = null)*: This method returns the mimetype of a given transferred file.

`getFileName()` accepts the name of the element as first parameter. If no name is given, all known filenames will be returned in an array. If the file is a multifile, you will also get an array. If there is only a single file a string will be returned.

By default file names will be returned with the complete path. If you only need the file name without path, you can set the second parameter, `$path`, which will truncate the file path when set to FALSE.

Example 342. Getting the Filename

```
$upload = new Zend_File_Transfer();
$upload->receive();

// Returns the file names from all files
$names = $upload->getFileName();

// Returns the file names from the 'foo' form element
$names = $upload->getFileName('foo');
```

Note that the file name can change after you receive the file, because all filters will be applied once the file is received. So you should always call `getFileName()` after the files have been received.

`getFileSize()` returns per default the real filesize in SI notation which means you will get `2kB` instead of `2048`. If you need only the plain size set the `useByteString` option to FALSE.

Example 343. Getting the size of a file

```
$upload = new Zend_File_Transfer();
$upload->receive();

// Returns the sizes from all files as array if more than one file was uploaded
$size = $upload->getFileSize();

// Switches of the SI notation to return plain numbers
$upload->setOption(array('useByteString' => false));
$size = $upload->getFileSize();
```

Client given filesize

Note that the filesize which is given by the client is not seen as save input. Therefor the real size of the file will be detected and returned instead of the filesize sent by the client.

`getHash()` accepts the name of a hash algorithm as first parameter. For a list of known algorithms refer to PHP's hash_algos method. If you don't specify an algorithm, the `crc32` algorithm will be used by default.

Example 344. Getting the hash of a file

```
$upload = new Zend_File_Transfer();
$upload->receive();

// Returns the hashes from all files as array if more than one file was uploaded
$hash = $upload->getHash('md5');

// Returns the hash for the 'foo' form element
$names = $upload->getHash('crc32', 'foo');
```

Return value

Note that if the given file or form name contains more than one file, the returned value will be an array.

`getMimeType()` returns the mimetype of a file. If more than one file was uploaded it returns an array, otherwise a string.

Example 345. Getting the mimetype of a file

```
$upload = new Zend_File_Transfer();
$upload->receive();

$mime = $upload->getMimeType();

// Returns the mimetype for the 'foo' form element
$names = $upload->getMimeType('foo');
```

Client given mimetype

Note that the mimetype which is given by the client is not seen as save input. Therefor the real mimetype of the file will be detected and returned instead of the mimetype sent by the client.

Possible exception

Note that this method uses the fileinfo extension if it is available. If this extension can not be found, it uses the mimemagic extension. When no extension was found it raises an exception.

Original data within $_FILES

Due to security reasons also the original data within $_FILES will be overridden as soon as `Zend_File_Transfer` is initiated. When you want to omit this behaviour and have the original data simply set the detectInfos option to FALSE at initiation.

This option will have no effect after you initiated `Zend_File_Transfer`.

1.5. Progress for file uploads

`Zend_File_Transfer` can give you the actual state of a fileupload in progress. To use this feature you need either the APC extension which is provided with most default PHP installations,

or the uploadprogress extension. Both extensions are detected and used automatically. To be able to get the progress you need to meet some prerequisites.

First, you need to have either APC or uploadprogress to be enabled. Note that you can disable this feature of APC within your php.ini.

Second, you need to have the proper hidden fields added in the form which sends the files. When you use Zend_Form_Element_File this hidden fields are automatically added by Zend_Form.

When the above two points are provided then you are able to get the actual progress of the file upload by using the getProgress method. Actually there are 2 official ways to handle this.

1.5.1. Using a progressbar adapter

You can use the convinient *Zend_ProgressBar* to get the actual progress and can display it in a simple manner to your user.

To archive this, you have to add the wished *Zend_ProgressBar_Adapter* to getProgress() when you are calling it the first time. For details about the right adapter to use, look into the chapter Zend_ProgressBar Standard Adapters.

Example 346. Using the progressbar adapter to retrieve the actual state

```
$adapter = new Zend_ProgressBar_Adapter_Console();
$upload  = Zend_File_Transfer_Adapter_Http::getProgress($adapter);

$upload = null;
while (!$upload['done']) {
    $upload = Zend_File_Transfer_Adapter_Http:getProgress($upload);
}
```

The complete handling is done by getProgress() for you in the background.

1.5.2. Using getProgress() manually

You can also work manually with getProgress() without the usage of Zend_ProgressBar.

Call getProgress() without settings. It will return you an array with several keys. They differ according to the used PHP extension. But the following keys are given independently of the extension:

- *id*: The ID of this upload. This ID identifies the upload within the extension. It is filled automatically. You should never change or give this value yourself.
- *total*: The total filesize of the uploaded files in bytes as integer.
- *current*: The current uploaded filesize in bytes as integer.
- *rate*: The average upload speed in bytes per second as integer.
- *done*: Returns TRUE when the upload is finished and FALSE otherwise.
- *message*: The actual message. Either the progress as text in the form *10kB / 200kB*, or a helpful message in the case of a problem. Problems could be, that there is no upload in progress, that there was a failure while retrieving the data for the progress, or that the upload has been canceled.

- *progress*: This optional key takes a instance of `Zend_ProgressBar_Adapter` or `Zend_ProgressBar` and allows to get the actual upload state within a progressbar.

- *session*: This optional key takes the name of a session namespace which will be used within `Zend_ProgressBar`. When this key is not given it defaults to `Zend_File_Transfer_Adapter_Http_ProgressBar`.

All other returned keys are provided directly from the extensions and will not be checked.

The following example shows a possible manual usage:

Example 347. Manual usage of the file progress

```
$upload  = Zend_File_Transfer_Adapter_Http::getProgress();

while (!$upload['done']) {
    $upload = Zend_File_Transfer_Adapter_Http:getProgress($upload);
    print "\nActual progress:".$upload['message'];
    // do whatever you need
}
```

2. Validators for Zend_File_Transfer

`Zend_File_Transfer` is delivered with several file-related validators which can be used to increase security and prevent possible attacks. Note that these validators are only as effective as how effectively you apply them. All validators provided with `Zend_File_Transfer` can be found in the `Zend_Validator` component and are named `Zend_Validate_File_*`. The following validators are available:

- `Count`: This validator checks for the number of files. A minimum and maximum range can be specified. An error will be thrown if either limit is crossed.

- `Crc32`: This validator checks for the crc32 hash value of the content from a file. It is based on the `Hash` validator and provides a convenient and simple validator that only supports Crc32.

- `ExcludeExtension`: This validator checks the extension of files. It will throw an error when an given file has a defined extension. With this validator, you can exclude defined extensions from being validated.

- `ExcludeMimeType`: This validator validates the MIME type of files. It can also validate MIME types and will throw an error if the MIME type of specified file matches.

- `Exists`: This validator checks for the existence of files. It will throw an error when a specified file does not exist.

- `Extension`: This validator checks the extension of files. It will throw an error when a specified file has an undefined extension.

- `FilesSize`: This validator checks the size of validated files. It remembers internally the size of all checked files and throws an error when the sum of all specified files exceed the defined size. It also provides minimum and maximum values.

- `ImageSize`: This validator checks the size of image. It validates the width and height and enforces minimum and maximum dimensions.

- `IsCompressed`: This validator checks whether the file is compressed. It is based on the `MimeType` validator and validates for compression archives like zip or arc. You can also limit it to other archives.
- `IsImage`: This validator checks whether the file is an image. It is based on the `MimeType` validator and validates for image files like jpg or gif. You can also limit it to other image types.
- `Hash`: This validator checks the hash value of the content from a file. It supports multiple algorithms.
- `Md5`: This validator checks for the md5 hash value of the content from a file. It is based on the `Hash` validator and provides a convenient and simple validator that only supports Md5.
- `MimeType`: This validator validates the MIME type of files. It can also validate MIME types and will throw an error if the MIME type of a specified file does not match.
- `NotExists`: This validator checks for the existence of files. It will throw an error when an given file does exist.
- `Sha1`: This validator checks for the sha1 hash value of the content from a file. It is based on the `Hash` validator and provides a convenient and simple validator that only supports sha1.
- `Size`: This validator is able to check files for its file size. It provides a minimum and maximum size range and will throw an error when either of these thesholds are crossed.
- `Upload`: This validator is internal. It checks if an upload has resulted in an error. You must not set it, as it's automatically set by `Zend_File_Transfer` itself. So you do not use this validator directly. You should only know that it exists.
- `WordCount`: This validator is able to check the number of words within files. It provides a minimum and maximum count and will throw an error when either of these thresholds are crossed.

2.1. Using Validators with Zend_File_Transfer

Putting validators to work is quite simple. There are several methods for adding and manipulating validators:

- `isValid($files = null)`: Checks the specified files using all validators. `$files` may be either a real filename, the element's name or the name of the temporary file.
- `addValidator($validator, $breakChainOnFailure, $options = null, $files = null)`: Adds the specified validator to the validator stack (optionally only to the file(s) specified). `$validator` may be either an actual validator instance or a short name specifying the validator type (e.g., 'Count').
- `addValidators(array $validators, $files = null)`: Adds the specified validators to the stack of validators. Each entry may be either a validator type/options pair or an array with the key 'validator' specifying the validator. All other options will be considered validator options for instantiation.
- `setValidators(array $validators, $files = null)`: Overwrites any existing validators with the validators specified. The validators should follow the syntax for `addValidators()`.
- `hasValidator($name)`: Indicates whether a validator has been registered.

- `getValidator($name)`: Returns a previously registered validator.
- `getValidators($files = null)`: Returns registered validators. If `$files` is specified, returns validators for that particular file or set of files.
- `removeValidator($name)`: Removes a previously registered validator.
- `clearValidators()`: Clears all registered validators.

Example 348. Add Validators to a File Transfer Object

```
$upload = new Zend_File_Transfer();

// Set a file size with 20000 bytes
$upload->addValidator('Size', false, 20000);

// Set a file size with 20 bytes minimum and 20000 bytes maximum
$upload->addValidator('Size', false, array('min' => 20, 'max' => 20000));

// Set a file size with 20 bytes minimum and 20000 bytes maximum and
// a file count in one step
$upload->setValidators(array(
    'Size'  => array('min' => 20, 'max' => 20000),
    'Count' => array('min' => 1, 'max' => 3),
));
```

Example 349. Limit Validators to Single Files

`addValidator()`, `addValidators()`, and `setValidators()` each accept a final `$files` argument. This argument can be used to specify a particular file or array of files on which to set the given validator.

```
$upload = new Zend_File_Transfer();

// Set a file size with 20000 bytes and limits it only to 'file2'
$upload->addValidator('Size', false, 20000, 'file2');
```

Normally, you should use the `addValidators()` method, which can be called multiple times.

Example 350. Add Multiple Validators

Often it's simpler just to call `addValidator()` multiple times with one call for each validator. This also increases readability and makes your code more maintainable. All methods provide a fluent interface, so you can couple the calls as shown below:

```
$upload = new Zend_File_Transfer();

// Set a file size with 20000 bytes
$upload->addValidator('Size', false, 20000)
       ->addValidator('Count', false, 2)
       ->addValidator('Filessize', false, 25000);
```

Note that setting the same validator multiple times is allowed, but doing so can lead to issues when using different options for the same validator.

Last but not least, you can simply check the files using `isValid()`.

Example 351. Validate the Files

`isValid()` accepts the file name of the uploaded or downloaded file, the temporary file name and or the name of the form element. If no parameter or null is given all files will be validated

```
$upload = new Zend_File_Transfer();

// Set a file size with 20000 bytes
$upload->addValidator('Size', false, 20000)
       ->addValidator('Count', false, 2)
       ->addValidator('Filessize', false, 25000);

if ($upload->isValid()) {
    print "Validation failure";
}
```

Note that `isValid()` will be called automatically when you receive the files and have not called it previously.

When validation has failed it is a good idea to get information about the problems found. To get this information, you can use the methods `getMessages()` which returns all validation messages as array, `getErrors()` which returns all error codes, and `hasErrors()` which returns TRUE as soon as a validation error has been found.

2.2. Count Validator

The `Count` validator checks for the number of files which are provided. It supports the following option keys:

- `min`: Sets the minimum number of files to transfer.

  When using this option you must give the minimum number of files when calling this validator the first time; otherwise you will get an error in return.

  With this option you can define the minimum number of files you expect to receive.

- `max`: Sets the maximum number of files to transfer.

  With this option you can limit the number of files which are accepted but also detect a possible attack when more files are given than defined in your form.

If you initiate this validator with a string or integer, the value will be used as `max`. Or you can also use the methods `setMin()` and `setMax()` to set both options afterwards and `getMin()` and `getMax()` to retrieve the actual set values.

Example 352. Using the Count Validator

```
$upload = new Zend_File_Transfer();

// Limit the amount of files to maximum 2
$upload->addValidator('Count', false, 2);

// Limit the amount of files to maximum 5 and minimum 1 file
$upload->addValidator('Count', false, array('min' =>1, 'max' => 5));
```

Note that this validator stores the number of checked files internally. The file which exceeds the maximum will be returned as error.

2.3. Crc32 Validator

The `Crc32` validator checks the content of a transferred file by hashing it. This validator uses the hash extension from PHP with the crc32 algorithm. It supports the following options:

- `*`: Sets any key or use a numeric array. The values will be used as hash to validate against.

  You can set multiple hashes by using different keys. Each will be checked and the validation will fail only if all values fail.

Example 353. Using the Crc32 Validator

```
$upload = new Zend_File_Transfer();

// Checks whether the content of the uploaded file has the given hash
$upload->addValidator('Crc32', false, '3b3652f');

// Limits this validator to two different hashes
$upload->addValidator('Crc32', false, array('3b3652f', 'e612b69'));
```

2.4. ExcludeExtension Validator

The `ExcludeExtension` validator checks the file extension of the specified files. It supports the following options:

- `*`: Sets any key or use a numeric array. The values will be used to check whether the given file does not use this file extension.

- `case`: Sets a boolean indicating whether validation should be case-sensitive. The default is not case sensitive. Note that this key can be applied to for all available extensions.

This validator accepts multiple extensions, either as a comma-delimited string, or as an array. You may also use the methods `setExtension()`, `addExtension()`, and `getExtension()` to set and retrieve extensions.

In some cases it is useful to match in a case-sensitive fashion. So the constructor allows a second parameter called `$case` which, if set to `TRUE`, validates the extension by comparing it with the specified values in a case-sensitive fashion.

Example 354. Using the ExcludeExtension Validator

```
$upload = new Zend_File_Transfer();

// Do not allow files with extension php or exe
$upload->addValidator('ExcludeExtension', false, 'php,exe');

// Do not allow files with extension php or exe, but use array notation
$upload->addValidator('ExcludeExtension', false, array('php', 'exe'));

// Check in a case-sensitive fashion
$upload->addValidator('ExcludeExtension',
                      false,
                      array('php', 'exe', 'case' => true));
$upload->addValidator('ExcludeExtension',
                      false,
                      array('php', 'exe', 'case' => true));
```

Note that this validator only checks the file extension. It does not check the file's MIME type.

2.5. ExcludeMimeType Validator

The `ExcludeMimeType` validator checks the MIME type of transferred files. It supports the following options:

- `*`: Sets any key individually or use a numeric array. Sets the MIME type to validate against.

  With this option you can define the MIME type of files that are not to be accepted.

- `headerCheck`: If set to `TRUE` this option will check the HTTP Information for the file type when the *fileInfo* or *mimeMagic* extensions can not be found. The default value for this option is `FALSE`.

This validator accepts multiple MIME types, either as a comma-delimited string, or as an array. You may also use the methods `setMimeType()`, `addMimeType()`, and `getMimeType()` to set and retrieve the MIME types.

Example 355. Using the ExcludeMimeType Validator

```
$upload = new Zend_File_Transfer();

// Does not allow MIME type of gif images for all files
$upload->addValidator('ExcludeMimeType', false, 'image/gif');

// Does not allow MIME type of gif and jpg images for all given files
$upload->addValidator('ExcludeMimeType', false, array('image/gif',
                                                      'image/jpeg');

// Does not allow MIME type of the group images for all given files
$upload->addValidator('ExcludeMimeType', false, 'image');
```

The above example shows that it is also possible to disallow groups of MIME types. For example, to disallow all images, just use 'image' as the MIME type. This can be used for all groups of MIME types like 'image', 'audio', 'video', 'text', etc.

Note that disallowing groups of MIME types will disallow all members of this group even if this is not intentional. When you disallow 'image' you will disallow all types of images like 'image/jpeg' or 'image/vasa'. When you are not sure if you want to disallow all types, you should disallow only specific MIME types instead of complete groups.

2.6. Exists Validator

The `Exists` validator checks for the existence of specified files. It supports the following options:

- `*`: Sets any key or use a numeric array to check if the specific file exists in the given directory.

This validator accepts multiple directories, either as a comma-delimited string, or as an array. You may also use the methods `setDirectory()`, `addDirectory()`, and `getDirectory()` to set and retrieve directories.

Example 356. Using the Exists Validator

```
$upload = new Zend_File_Transfer();

// Add the temp directory to check for
$upload->addValidator('Exists', false, '\temp');

// Add two directories using the array notation
$upload->addValidator('Exists',
                      false,
                      array('\home\images', '\home\uploads'));
```

Note that this validator checks whether the specified file exists in all of the given directories. The validation will fail if the file does not exist in any of the given directories.

2.7. Extension Validator

The `Extension` validator checks the file extension of the specified files. It supports the following options:

- `*`: Sets any key or use a numeric array to check whether the specified file has this file extension.
- `case`: Sets whether validation should be done in a case-sensitive fashion. The default is no case sensitivity. Note the this key is used for all given extensions.

This validator accepts multiple extensions, either as a comma-delimited string, or as an array. You may also use the methods `setExtension()`, `addExtension()`, and `getExtension()` to set and retrieve extension values.

In some cases it is useful to test in a case-sensitive fashion. Therefore the constructor takes a second parameter `$case`, which, if set to `TRUE`, will validate the extension in a case-sensitive fashion.

Example 357. Using the Extension Validator

```
$upload = new Zend_File_Transfer();

// Limit the extensions to jpg and png files
$upload->addValidator('Extension', false, 'jpg,png');

// Limit the extensions to jpg and png files but use array notation
$upload->addValidator('Extension', false, array('jpg', 'png'));

// Check case sensitive
$upload->addValidator('Extension', false, array('mo', 'png', 'case' => true));
if (!$upload->isValid('C:\temp\myfile.MO')) {
    print 'Not valid because MO and mo do not match with case sensitivity;
}
```

Note that this validator only checks the file extension. It does not check the file's MIME type.

2.8. FilesSize Validator

The `FilesSize` validator checks for the aggregate size of all transferred files. It supports the following options:

- `min`: Sets the minimum aggregate file size. This option defines the minimum aggregate file size to be transferred.
- `max`: Sets the maximum aggregate file size.

  This option limits the aggregate file size of all transferred files, but not the file size of individual files.
- `bytestring`: Defines whether a failure is to return a user-friendly number or the plain file size.

  This option defines whether the user sees '10864' or '10MB'. The default value is `TRUE`, so '10MB' is returned if you did not specify otherwise.

You can initialize this validator with a string, which will then be used to set the `max` option. You can also use the methods `setMin()` and `setMax()` to set both options after construction, along with `getMin()` and `getMax()` to retrieve the values that have been set previously.

The size itself is also accepted in SI notation as handled by most operating systems. That is, instead of specifying *20000 bytes*, *20kB* may be given. All file sizes are converted using 1024 as the base value. The following Units are accepted: `kB`, `MB`, `GB`, `TB`, `PB` and `EB`. Note that 1kB is equal to 1024 bytes, 1MB is equal to 1024kB, and so on.

Example 358. Using the FilesSize Validator

```
$upload = new Zend_File_Transfer();

// Limit the size of all files to be uploaded to 40000 bytes
$upload->addValidator('FilesSize', false, 40000);

// Limit the size of all files to be uploaded to maximum 4MB and mimimum 10kB
$upload->addValidator('FilesSize',
                      false,
                      array('min' => '10kB', 'max' => '4MB'));

// As before, but returns the plain file size instead of a user-friendly string
$upload->addValidator('FilesSize',
                      false,
                      array('min' => '10kB',
                            'max' => '4MB',
                            'bytestring' => false));
```

Note that this validator internally stores the file size of checked files. The file which exceeds the size will be returned as an error.

2.9. ImageSize Validator

The `ImageSize` validator checks the size of image files. It supports the following options:

- `minheight`: Sets the minimum image height.
- `maxheight`: Sets the maximum image height.
- `minwidth`: Sets the minimum image width.
- `maxwidth`: Sets the maximum image width.

The methods `setImageMin()` and `setImageMax()` also set both minimum and maximum values, while the methods `getMin()` and `getMax()` return the currently set values.

For your convenience there are also the `setImageWidth()` and `setImageHeight()` methods, which set the minimum and maximum height and width of the image file. They, too, have corresponding `getImageWidth()` and `getImageHeight()` methods to retrieve the currently set values.

To bypass validation of a particular dimension, the relevent option simply should not be set.

Example 359. Using the ImageSize Validator

```
$upload = new Zend_File_Transfer();

// Limit the size of a image to a height of 100-200 and a width of
// 40-80 pixel
$upload->addValidator('ImageSize', false,
                      array('minwidth' => 40,
                            'maxwidth' => 80,
                            'minheight' => 100,
                            'maxheight' => 200)
                      );

// Reset the width for validation
$upload->setImageWidth(array('minwidth' => 20, 'maxwidth' => 200));
```

2.10. IsCompressed Validator

The `IsCompressed` validator checks if a transferred file is a compressed archive, such as zip or arc. This validator is based on the `MimeType` validator and supports the same methods and options. You may also limit this validator to particular compression types with the methods described there.

Example 360. Using the IsCompressed Validator

```
$upload = new Zend_File_Transfer();

// Checks is the uploaded file is a compressed archive
$upload->addValidator('IsCompressed', false);

// Limits this validator to zip files only
$upload->addValidator('IsCompressed', false, array('application/zip'));

// Limits this validator to zip files only using simpler notation
$upload->addValidator('IsCompressed', false, 'zip');
```

Note that there is no check if you set a MIME type that is not a archive. For example, it would be possible to define gif files to be accepted by this validator. Using the 'MimeType' validator for files which are not archived will result in more readable code.

2.11. IsImage Validator

The `IsImage` validator checks if a transferred file is a image file, such as gif or jpeg. This validator is based on the `MimeType` validator and supports the same methods and options. You can limit this validator to particular image types with the methods described there.

Example 361. Using the IsImage Validator

```
$upload = new Zend_File_Transfer();

// Checks whether the uploaded file is a image file
$upload->addValidator('IsImage', false);

// Limits this validator to gif files only
$upload->addValidator('IsImage', false, array('application/gif'));

// Limits this validator to jpeg files only using a simpler notation
$upload->addValidator('IsImage', false, 'jpeg');
```

Note that there is no check if you set a MIME type that is not an image. For example, it would be possible to define zip files to be accepted by this validator. Using the 'MimeType' validator for files which are not images will result in more readable code.

2.12. Hash Validator

The `Hash` validator checks the content of a transferred file by hashing it. This validator uses the hash extension from PHP. It supports the following options:

- `*`: Takes any key or use a numeric array. Sets the hash to validate against.

  You can set multiple hashes by passing them as an array. Each file is checked, and the validation will fail only if all files fail validation.

- `algorithm`: Sets the algorithm to use for hashing the content.

  You can set multiple algorithm by calling the `addHash()` method multiple times.

Example 362. Using the Hash Validator

```
$upload = new Zend_File_Transfer();

// Checks if the content of the uploaded file contains the given hash
$upload->addValidator('Hash', false, '3b3652f');

// Limits this validator to two different hashes
$upload->addValidator('Hash', false, array('3b3652f', 'e612b69'));

// Sets a different algorithm to check against
$upload->addValidator('Hash',
                      false,
                      array('315b3cd8273d44912a7',
                            'algorithm' => 'md5'));
```

This validator supports about 34 different hash algorithms. The most common include 'crc32', 'md5' and 'sha1'. A comprehesive list of supports hash algorithms can be found at the hash_algos method on the php.net site.

2.13. Md5 Validator

The `Md5` validator checks the content of a transferred file by hashing it. This validator uses the hash extension for PHP with the md5 algorithm. It supports the following options:

- *: Takes any key or use a numeric array.

  You can set multiple hashes by passing them as an array. Each file is checked, and the validation will fail only if all files fail validation.

  Example 363. Using the Md5 Validator

```
$upload = new Zend_File_Transfer();

// Checks if the content of the uploaded file has the given hash
$upload->addValidator('Md5', false, '3b3652f336522365223');

// Limits this validator to two different hashes
$upload->addValidator('Md5',
                      false,
                      array('3b3652f336522365223',
                            'eb3365f3365ddc65365'));
```

2.14. MimeType Validator

The `MimeType` validator checks the MIME type of transferred files. It supports the following options:

- *: Sets any key or use a numeric array. Sets the MIME type to validate against.

  Defines the MIME type of files to be accepted.

- `headerCheck`: If set to TRUE this option will check the HTTP Information for the file type when the *fileInfo* or *mimeMagic* extensions can not be found. The default value for this option is FALSE.

- `magicfile`: The magicfile to be used.

  With this option you can define which magicfile to use. When it's not set or empty, the MAGIC constant will be used instead. This option is available since Zend Framework 1.7.1.

This validator accepts multiple MIME type, either as a comma-delimited string, or as an array. You may also use the methods `setMimeType()`, `addMimeType()`, and `getMimeType()` to set and retrieve MIME type.

You can also set the magicfile which shall be used by fileinfo with the 'magicfile' option. Additionally there are convenient `setMagicFile()` and `getMagicFile()` methods which allow later setting and retrieving of the magicfile parameter. This methods are available since Zend Framework 1.7.1.

Example 364. Using the MimeType Validator

```
$upload = new Zend_File_Transfer();

// Limit the MIME type of all given files to gif images
$upload->addValidator('MimeType', false, 'image/gif');

// Limit the MIME type of all given files to gif and jpeg images
$upload->addValidator('MimeType', false, array('image/gif', 'image/jpeg');

// Limit the MIME type of all given files to the group images
$upload->addValidator('MimeType', false, 'image');

// Use a different magicfile
$upload->addValidator('MimeType',
                      false,
                      array('image',
                            'magicfile' => '/path/to/magicfile.mgx'));
```

The above example shows that it is also possible to limit the accepted MIME type to a group of MIME types. To allow all images just use 'image' as MIME type. This can be used for all groups of MIME types like 'image', 'audio', 'video', 'text, and so on.

Note that allowing groups of MIME types will accept all members of this group even if your application does not support them. When you allow 'image' you will also get 'image/xpixmap' or 'image/vasa' which could be problematic. When you are not sure if your application supports all types you should better allow only defined MIME types instead of the complete group.

This component will use the `fileinfo` extension if it is available. If it's not, it will degrade to the `mime_content_type` function. And if the function call fails it will use the MIME type which is given by HTTP.

You should be aware of possible security problems when you have whether `fileinfo` nor `mime_content_type` available. The MIME type given by HTTP is not secure and can be easily manipulated.

2.15. NotExists Validator

The `NotExists` validator checks for the existence of the provided files. It supports the following options:

- *: Set any key or use a numeric array. Checks whether the file exists in the given directory.

This validator accepts multiple directories either as a comma-delimited string, or as an array. You may also use the methods `setDirectory()`, `addDirectory()`, and `getDirectory()` to set and retrieve directories.

Example 365. Using the NotExists Validator

```
$upload = new Zend_File_Transfer();

// Add the temp directory to check
$upload->addValidator('NotExists', false, '\temp');

// Add two directories using the array notation
$upload->addValidator('NotExists', false,
                      array('\home\images',
                            '\home\uploads')
                     );
```

Note that this validator checks if the file does not exist in all of the provided directories. The validation will fail if the file does exist in any of the given directories.

2.16. Sha1 Validator

The `Sha1` validator checks the content of a transferred file by hashing it. This validator uses the hash extension for PHP with the sha1 algorithm. It supports the following options:

- `*`: Takes any key or use a numeric array.

  You can set multiple hashes by passing them as an array. Each file is checked, and the validation will fail only if all files fail validation.

Example 366. Using the sha1 Validator

```
$upload = new Zend_File_Transfer();

// Checks if the content of the uploaded file has the given hash
$upload->addValidator('sha1', false, '3b3652f336522365223');

// Limits this validator to two different hashes
$upload->addValidator('Sha1',
                      false, array('3b3652f336522365223',
                                   'eb3365f3365ddc65365'));
```

2.17. Size Validator

The `Size` validator checks for the size of a single file. It supports the following options:

- `min`: Sets the minimum file size.
- `max`: Sets the maximum file size.
- `bytestring`: Defines whether a failure is returned with a user-friendly number, or with the plain file size.

  With this option you can define if the user gets '10864' or '10MB'. Default value is `TRUE` which returns '10MB'.

You can initialize this validator with a string, which will then be used to set the `max` option. You can also use the methods `setMin()` and `setMax()` to set both options after construction, along with `getMin()` and `getMax()` to retrieve the values that have been set previously.

The size itself is also accepted in SI notation as handled by most operating systems. That is, instead of specifying *20000 bytes*, *20kB* may be given. All file sizes are converted using 1024 as the base value. The following Units are accepted: kB, MB, GB, TB, PB and EB. Note that 1kB is equal to 1024 bytes, 1MB is equal to 1024kB, and so on.

Example 367. Using the Size Validator

```
$upload = new Zend_File_Transfer();

// Limit the size of a file to 40000 bytes
$upload->addValidator('Size', false, 40000);

// Limit the size a given file to maximum 4MB and mimimum 10kB
// Also returns the plain number in case of an error
// instead of a user-friendly number
$upload->addValidator('Size',
                      false,
                      array('min' => '10kB',
                            'max' => '4MB',
                            'bytestring' => false));
```

2.18. WordCount Validator

The `WordCount` validator checks for the number of words within provided files. It supports the following option keys:

- `min`: Sets the minimum number of words to be found.
- `max`: Sets the maximum number of words to be found.

If you initiate this validator with a string or integer, the value will be used as `max`. Or you can also use the methods `setMin()` and `setMax()` to set both options afterwards and `getMin()` and `getMax()` to retrieve the actual set values.

Example 368. Using the WordCount Validator

```
$upload = new Zend_File_Transfer();

// Limit the amount of words within files to maximum 2000
$upload->addValidator('WordCount', false, 2000);

// Limit the amount of words within files to maximum 5000 and minimum 1000 words
$upload->addValidator('WordCount', false, array('min' => 1000, 'max' => 5000));
```

3. Filters for Zend_File_Transfer

`Zend_File_Transfer` is delivered with several file related filters which can be used to automate several tasks which are often done on files. Note that file filters are applied after validation. Also file filters behave slightly different that other filters. They will always return the file name and not the changed content (which would be a bad idea when working on 1GB files). All filters which are provided with `Zend_File_Transfer` can be found in the `Zend_Filter` component and are named `Zend_Filter_File_*`. The following filters are actually available:

- `Decrypt`: This filter can decrypt a encrypted file.
- `Encrypt`: This filter can encrypt a file.

- `LowerCase`: This filter can lowercase the content of a textfile.
- `Rename`: This filter can rename files, change the location and even force overwriting of existing files.
- `UpperCase`: This filter can uppercase the content of a textfile.

3.1. Using filters with Zend_File_Transfer

The usage of filters is quite simple. There are several methods for adding and manipulating filters.

- `addFilter($filter, $options = null, $files = null)`: Adds the given filter to the filter stack (optionally only to the file(s) specified). `$filter` may be either an actual filter instance, or a short name specifying the filter type (e.g., 'Rename').
- `addFilters(array $filters, $files = null)`: Adds the given filters to the stack of filters. Each entry may be either a filter type/options pair, or an array with the key 'filter' specifying the filter (all other options will be considered filter options for instantiation).
- `setFilters(array $filters, $files = null)`: Overwrites any existing filters with the filters specified. The filters should follow the syntax for `addFilters()`.
- `hasFilter($name)`: Indicates if a filter has been registered.
- `getFilter($name)`: Returns a previously registered filter.
- `getFilters($files = null)`: Returns registered filters; if `$files` is passed, returns filters for that particular file or set of files.
- `removeFilter($name)`: Removes a previously registered filter.
- `clearFilters()`: Clears all registered filters.

Example 369. Add filters to a file transfer

```
$upload = new Zend_File_Transfer();

// Set a new destination path
$upload->addFilter('Rename', 'C:\picture\uploads');

// Set a new destination path and overwrites existing files
$upload->addFilter('Rename',
                   array('target' => 'C:\picture\uploads',
                         'overwrite' => true));
```

Example 370. Limit filters to single files

`addFilter()`, `addFilters()`, and `setFilters()` each accept a final `$files` argument. This argument can be used to specify a particular file or array of files on which to set the given filter.

```
$upload = new Zend_File_Transfer();

// Set a new destination path and limits it only to 'file2'
$upload->addFilter('Rename', 'C:\picture\uploads', 'file2');
```

Generally you should simply use the `addFilters()` method, which can be called multiple times.

Example 371. Add multiple filters

Often it's simpler just to call `addFilter()` multiple times. One call for each filter. This also increases the readability and makes your code more maintainable. As all methods provide a fluent interface you can couple the calls as shown below:

```
$upload = new Zend_File_Transfer();

// Set a filesize with 20000 bytes
$upload->addFilter('Rename', 'C:\picture\newjpg', 'file1')
       ->addFilter('Rename', 'C:\picture\newgif', 'file2');
```

Note that even though setting the same filter multiple times is allowed, doing so can lead to issues when using different options for the same filter.

3.2. Decrypt filter

The `Decrypt` filter allows to decrypt a encrypted file.

This filter makes use of `Zend_Filter_Decrypt`. It supports the `Mcrypt` and `OpenSSL` extensions from PHP. Please read the related section for details about how to set the options for decryption and which options are supported.

This filter supports one additional option which can be used to save the decrypted file with another filename. Set the `filename` option to change the filename where the decrypted file will be stored. If you suppress this option, the decrypted file will overwrite the original encrypted file.

Example 372. Using the Decrypt filter with Mcrypt

```
$upload = new Zend_File_Transfer_Adapter_Http();

// Adds a filter to decrypt the uploaded encrypted file
// with mcrypt and the key mykey
$upload->addFilter('Decrypt',
    array('adapter' => 'mcrypt', 'key' => 'mykey'));
```

Example 373. Using the Decrypt filter with OpenSSL

```
$upload = new Zend_File_Transfer_Adapter_Http();

// Adds a filter to decrypt the uploaded encrypted file
// with openssl and the provided keys
$upload->addFilter('Decrypt',
    array('adapter' => 'openssl',
          'private' => '/path/to/privatekey.pem',
          'envelope' => '/path/to/envelopekey.pem'));
```

3.3. Encrypt filter

The `Encrypt` filter allows to encrypt a file.

This filter makes use of `Zend_Filter_Encrypt`. It supports the `Mcrypt` and `OpenSSL` extensions from PHP. Please read the related section for details about how to set the options for encryption and which options are supported.

This filter supports one additional option which can be used to save the encrypted file with another filename. Set the `filename` option to change the filename where the encrypted file will be stored. If you suppress this option, the encrypted file will overwrite the original file.

Example 374. Using the Encrypt filter with Mcrypt

```
$upload = new Zend_File_Transfer_Adapter_Http();

// Adds a filter to encrypt the uploaded file
// with mcrypt and the key mykey
$upload->addFilter('Encrypt',
    array('adapter' => 'mcrypt', 'key' => 'mykey'));
```

Example 375. Using the Encrypt filter with OpenSSL

```
$upload = new Zend_File_Transfer_Adapter_Http();

// Adds a filter to encrypt the uploaded file
// with openssl and the provided keys
$upload->addFilter('Encrypt',
    array('adapter' => 'openssl',
          'public' => '/path/to/publickey.pem'));
```

3.4. LowerCase filter

The `LowerCase` filter allows to change the content of a file to lowercase. You should use this filter only on textfiles.

At initiation you can give a string which will then be used as encoding. Or you can use the `setEncoding()` method to set it afterwards.

Example 376. Using the LowerCase filter

```
$upload = new Zend_File_Transfer_Adapter_Http();
$upload->addValidator('MimeType', 'text');

// Adds a filter to lowercase the uploaded textfile
$upload->addFilter('LowerCase');

// Adds a filter to lowercase the uploaded file but only for uploadfile1
$upload->addFilter('LowerCase', null, 'uploadfile1');

// Adds a filter to lowercase with encoding set to ISO-8859-1
$upload->addFilter('LowerCase', 'ISO-8859-1');
```

Note that due to the fact that the options for the LowerCase filter are optional, you must give a `NULL` as second parameter (the options) when you want to limit it to a single file element.

3.5. Rename filter

The `Rename` filter allows to change the destination of the upload, the filename and also to overwrite existing files. It supports the following options:

- `source`: The name and destination of the old file which shall be renamed.
- `target`: The new directory, or filename of the file.

- `overwrite`: Sets if the old file overwrites the new one if it already exists. The default value is FALSE.

Additionally you can also use the method `setFile()` to set files, which erases all previous set, `addFile()` to add a new file to existing ones, and `getFile()` to get all actually set files. To simplify things, this filter understands several notations and that methods and constructor understand the same notations.

Example 377. Using the Rename filter

```
$upload = new Zend_File_Transfer_Adapter_Http();

// Set a new destination path for all files
$upload->addFilter('Rename', 'C:\mypics\new');

// Set a new destination path only for uploadfile1
$upload->addFilter('Rename', 'C:\mypics\newgifs', 'uploadfile1');
```

You can use different notations. Below is a table where you will find a description and the intention for the supported notations. Note that when you use the Adapter or the Form Element you will not be able to use all described notations.

Table 62. Different notations of the rename filter and their meaning

notation	description
`addFile('C:\uploads')`	Specifies a new location for all files when the given string is a directory. Note that you will get an exception when the file already exists, see the overwriting parameter.
`addFile('C:\uploads\file.ext')`	Specifies a new location and filename for all files when the given string is not detected as directory. Note that you will get an exception when the file already exists, see the overwriting parameter.
`addFile(array('C:\uploads \file.ext', 'overwrite' => true))`	Specifies a new location and filename for all files when the given string is not detected as directory and overwrites an existing file with the same target name. Note, that you will get no notification that a file was overwritten.
`addFile(array('source' => 'C: \temp\uploads', 'target' => 'C: \uploads'))`	Specifies a new location for all files in the old location when the given strings are detected as directory. Note that you will get an exception when the file already exists, see the overwriting parameter.
addFile(array('source' => 'C:\temp\uploads', 'target' => 'C:\uploads', 'overwrite' => true))	Specifies a new location for all files in the old location when the given strings are detected as directory and overwrites and existing file with the same target name. Note, that you will get no notification that a file was overwritten.

3.6. UpperCase filter

The `UpperCase` filter allows to change the content of a file to uppercase. You should use this filter only on textfiles.

At initiation you can give a string which will then be used as encoding. Or you can use the `setEncoding()` method to set it afterwards.

Example 378. Using the UpperCase filter

```
$upload = new Zend_File_Transfer_Adapter_Http();
$upload->addValidator('MimeType', 'text');

// Adds a filter to uppercase the uploaded textfile
$upload->addFilter('UpperCase');

// Adds a filter to uppercase the uploaded file but only for uploadfile1
$upload->addFilter('UpperCase', null, 'uploadfile1');

// Adds a filter to uppercase with encoding set to ISO-8859-1
$upload->addFilter('UpperCase', 'ISO-8859-1');
```

Note that due to the fact that the options for the UpperCase filter are optional, you must give a `NULL` as second parameter (the options) when you want to limit it to a single file element.

Zend_Filter

1. Introduction

The `Zend_Filter` component provides a set of commonly needed data filters. It also provides a simple filter chaining mechanism by which multiple filters may be applied to a single datum in a user-defined order.

1.1. What is a filter?

In the physical world, a filter is typically used for removing unwanted portions of input, and the desired portion of the input passes through as filter output (e.g., coffee). In such scenarios, a filter is an operator that produces a subset of the input. This type of filtering is useful for web applications - removing illegal input, trimming unnecessary white space, etc.

This basic definition of a filter may be extended to include generalized transformations upon input. A common transformation applied in web applications is the escaping of HTML entities. For example, if a form field is automatically populated with untrusted input (e.g., from a web browser), this value should either be free of HTML entities or contain only escaped HTML entities, in order to prevent undesired behavior and security vulnerabilities. To meet this requirement, HTML entities that appear in the input must either be removed or escaped. Of course, which approach is more appropriate depends on the situation. A filter that removes the HTML entities operates within the scope of the first definition of filter - an operator that produces a subset of the input. A filter that escapes the HTML entities, however, transforms the input (e.g., "&" is transformed to "`&`"). Supporting such use cases for web developers is important, and "to filter," in the context of using `Zend_Filter`, means to perform some transformations upon input data.

1.2. Basic usage of filters

Having this filter definition established provides the foundation for `Zend_Filter_Interface`, which requires a single method named `filter()` to be implemented by a filter class.

Following is a basic example of using a filter upon two input data, the ampersand (&) and double quote (") characters:

```
$htmlEntities = new Zend_Filter_HtmlEntities();

echo $htmlEntities->filter('&'); // &
echo $htmlEntities->filter('"'); // >
```

1.3. Using the static staticFilter() method

If it is inconvenient to load a given filter class and create an instance of the filter, you can use the static method `Zend_Filter::filterStatic()` as an alternative invocation style. The first argument of this method is a data input value, that you would pass to the `filter()` method. The second argument is a string, which corresponds to the basename of the filter class, relative to the Zend_Filter namespace. The `staticFilter()` method automatically loads the class, creates an instance, and applies the `filter()` method to the data input.

```
echo Zend_Filter::filterStatic('&', 'HtmlEntities');
```

You can also pass an array of constructor arguments, if they are needed for the filter class.

```
echo Zend_Filter::filterStatic('"',
                               'HtmlEntities',
                               array('quotestyle' => ENT_QUOTES));
```

The static usage can be convenient for invoking a filter ad hoc, but if you have the need to run a filter for multiple inputs, it's more efficient to follow the first example above, creating an instance of the filter object and calling its `filter()` method.

Also, the `Zend_Filter_Input` class allows you to instantiate and run multiple filter and validator classes on demand to process sets of input data. See Section 5, “Zend_Filter_Input”.

1.3.1. Namespaces

When working with self defined filters you can give a forth parameter to `Zend_Filter::filterStatic()` which is the namespace where your filter can be found.

```
echo Zend_Filter::filterStatic(
    '"',
    'MyFilter',
    array($parameters),
    array('FirstNamespace', 'SecondNamespace')
);
```

`Zend_Filter` allows also to set namespaces as default. This means that you can set them once in your bootstrap and have not to give them again for each call of `Zend_Filter::filterStatic()`. The following code snippet is identical to the above one.

```
Zend_Filter::setDefaultNamespaces(array('FirstNamespace', 'SecondNamespace'));
echo Zend_Filter::filterStatic('"', 'MyFilter', array($parameters));
echo Zend_Filter::filterStatic('"', 'OtherFilter', array($parameters));
```

For your convinience there are following methods which allow the handling of namespaces:

- *`Zend_Filter::getDefaultNamespaces()`*: Returns all set default namespaces as array.
- *`Zend_Filter::setDefaultNamespaces()`*: Sets new default namespaces and overrides any previous set. It accepts either a string for a single namespace of an array for multiple namespaces.
- *`Zend_Filter::addDefaultNamespaces()`*: Adds additional namespaces to already set ones. It accepts either a string for a single namespace of an array for multiple namespaces.
- *`Zend_Filter::hasDefaultNamespaces()`*: Returns `TRUE` when one or more default namespaces are set, and `FALSE` when no default namespaces are set.

2. Standard Filter Classes

Zend Framework comes with a standard set of filters, which are ready for you to use.

2.1. Alnum

Returns the string `$value`, removing all but alphabetic and digit characters. This filter includes an option to also allow white space characters.

The alphabetic characters mean characters that makes up words in each language. However, the english alphabet is treated as the alphabetic characters in following languages: Chinese, Japanese, Korean. The language is specified by `Zend_Locale`.

2.2. Alpha

Returns the string `$value`, removing all but alphabetic characters. This filter includes an option to also allow white space characters.

2.3. BaseName

Given a string containing a path to a file, this filter will return the base name of the file

2.4. Boolean

This filter changes a given input to be a `BOOLEAN` value. This is often useful when working with databases or when processing form values.

2.4.1. Default behaviour for Zend_Filter_Boolean

By default, this filter works by casting the input to a `BOOLEAN` value; in other words, it operates in a similar fashion to calling **(boolean) $value**.

```
$filter = new Zend_Filter_Boolean();
$value  = '';
$result = $filter->filter($value);
// returns false
```

This means that without providing any configuration, `Zend_Filter_Boolean` accepts all input types and returns a `BOOLEAN` just as you would get by type casting to `BOOLEAN`.

2.4.2. Changing behaviour for Zend_Filter_Boolean

Sometimes casting with **(boolean)** will not suffice. `Zend_Filter_Boolean` allows you to configure specific types to convert, as well as which to omit.

The following types can be handled:

- *boolean*: Returns a boolean value as is.
- *integer*: Converts an integer *0* value to `FALSE`.
- *float*: Converts a float *0.0* value to `FALSE`.
- *string*: Converts an empty string *''* to `FALSE`.
- *zero*: Converts a string containing the single character zero (*'0'*) to `FALSE`.
- *empty_array*: Converts an empty *array* to `FALSE`.
- *null*: Converts a `NULL` value to `FALSE`.
- *php*: Converts values according to PHP when casting them to `BOOLEAN`.

- *false_string*: Converts a string containing the word "false" to a boolean FALSE.
- *yes*: Converts a localized string which contains the word "no" to FALSE.
- *all*: Converts all above types to BOOLEAN.

All other given values will return TRUE by default.

There are several ways to select which of the above types are filtered. You can give one or multiple types and add them, you can give an array, you can use constants, or you can give a textual string. See the following examples:

```
// converts 0 to false
$filter = new Zend_Filter_Boolean(Zend_Filter_Boolean::INTEGER);

// converts 0 and '0' to false
$filter = new Zend_Filter_Boolean(
    Zend_Filter_Boolean::INTEGER + Zend_Filter_Boolean::ZERO
);

// converts 0 and '0' to false
$filter = new Zend_Filter_Boolean(array(
    'type' => array(
        Zend_Filter_Boolean::INTEGER,
        Zend_Filter_Boolean::ZERO,
    ),
));

// converts 0 and '0' to false
$filter = new Zend_Filter_Boolean(array(
    'type' => array(
        'integer',
        'zero',
    ),
));
```

You can also give an instance of `Zend_Config` to set the desired types. To set types after instantiation, use the `setType()` method.

2.4.3. Localized booleans

As mentioned previously, `Zend_Filter_Boolean` can also recognise localized "yes" and "no" strings. This means that you can ask your customer in a form for "yes" or "no" within his native language and `Zend_Filter_Boolean` will convert the response to the appropriate boolean value.

To set the desired locale, you can either use the locale option, or the method `setLocale()`.

```
$filter = new Zend_Filter_Boolean(array(
    'type'   => Zend_Filter_Boolean::ALL,
    'locale' => 'de',
));

// returns false
echo $filter->filter('nein');

$filter->setLocale('en');
```

```
// returns true
$filter->filter('yes');
```

2.4.4. Disable casting

Sometimes it is necessary to recognise only TRUE or FALSE and return all other values without changes. Zend_Filter_Boolean allows you to do this by setting the casting option to FALSE.

In this case Zend_Filter_Boolean will work as described in the following table, which shows which values return TRUE or FALSE. All other given values are returned without change when casting is set to FALSE

Table 63. Usage without casting

Type	True	False
Zend_Filter_Boolean::BOOLEAN	TRUE	FALSE
Zend_Filter_Boolean::INTEGER	0	1
Zend_Filter_Boolean::FLOAT	0.0	1.0
Zend_Filter_Boolean::STRING	""	
Zend_Filter_Boolean::ZERO	"0"	"1"
Zend_Filter_Boolean::EMPTY_ARRAY	array()	
Zend_Filter_Boolean::NULL	NULL	
Zend_Filter_Boolean::FALSE_STRING	"false" (case independently)	"true" (case independently)
Zend_Filter_Boolean::YES	localized "yes" (case independently)	localized "no" (case independently)

The following example shows the behaviour when changing the casting option:

```
$filter = new Zend_Filter_Boolean(array(
    'type'    => Zend_Filter_Boolean::ALL,
    'casting' => false,
));

// returns false
echo $filter->filter(0);

// returns true
echo $filter->filter(1);

// returns the value
echo $filter->filter(2);
```

2.5. Callback

This filter allows you to use own methods in conjunction with Zend_Filter. You don't have to create a new filter when you already have a method which does the job.

Let's expect we want to create a filter which reverses a string.

```
$filter = new Zend_Filter_Callback('strrev');

print $filter->filter('Hello!');
// returns "!olleH"
```

As you can see it's really simple to use a callback to define a own filter. It is also possible to use a method, which is defined within a class, by giving an array as callback.

```
// Our classdefinition
class MyClass
{
    public function Reverse($param);
}

// The filter definition
$filter = new Zend_Filter_Callback(array('MyClass', 'Reverse'));
print $filter->filter('Hello!');
```

To get the actual set callback use `getCallback()` and to set another callback use `setCallback()`.

It is also possible to define default parameters, which are given to the called method as array when the filter is executed. This array will be concatenated with the value which will be filtered.

```
$filter = new Zend_Filter_Callback(
    array(
        'callback' => 'MyMethod',
        'options'  => array('key' => 'param1', 'key2' => 'param2')
    )
);
$filter->filter(array('value' => 'Hello'));
```

When you would call the above method definition manually it would look like this:

```
$value = MyMethod('Hello', 'param1', 'param2');
```

You should note that defining a callback method which can not be called will raise an exception.

2.6. Compress and Decompress

These two filters are capable of compressing and decompressing strings, files, and directories. They make use of adapters and support the following compression formats:

- *Bz2*
- *Gz*
- *Lzf*
- *Rar*
- *Tar*
- *Zip*

Each compression format has different capabilities as described below. All compression filters may be used in approximately the same ways, and differ primarily in the options available and the type of compression they offer (both algorithmically as well as string vs. file vs. directory)

2.6.1. Generic handling

To create a compression filter you need to select the compression format you want to use. The following description takes the *Bz2* adapter. Details for all other adapters are described after this section.

The two filters are basically identical, in that they utilize the same backends. `Zend_Filter_Compress` should be used when you wish to compress items, and `Zend_Filter_Decompress` should be used when you wish to decompress items.

For instance, if we want to compress a string, we have to initiate `Zend_Filter_Compress` and indicate the desired adapter.

```
$filter = new Zend_Filter_Compress('Bz2');
```

To use a different adapter, you simply specify it to the constructor.

You may also provide an array of options or `Zend_Config` object. If you do, provide minimally the key "adapter", and then either the key "options" or "adapterOptions" (which should be an array of options to provide to the adapter on instantiation).

```
$filter = new Zend_Filter_Compress(array(
    'adapter' => 'Bz2',
    'options' => array(
        'blocksize' => 8,
    ),
));
```

Default compression Adapter

When no compression adapter is given, then the *Gz* adapter will be used.

Almost the same usage is we want to decompress a string. We just have to use the decompression filter in this case.

```
$filter = new Zend_Filter_Decompress('Bz2');
```

To get the compressed string, we have to give the original string. The filtered value is the compressed version of the original string.

```
$filter     = new Zend_Filter_Compress('Bz2');
$compressed = $filter->filter('Uncompressed string');
// Returns the compressed string
```

Decompression works the same way.

```
$filter     = new Zend_Filter_Decompress('Bz2');
$compressed = $filter->filter('Compressed string');
// Returns the uncompressed string
```

Note on string compression

Not all adapters support string compression. Compression formats like *Rar* can only handle files and directories. For details, consult the section for the adapter you wish to use.

2.6.2. Creating an archive

Creating an archive file works almost the same as compressing a string. However, in this case we need an additional parameter which holds the name of the archive we want to create.

```
$filter     = new Zend_Filter_Compress(array(
    'adapter' => 'Bz2',
    'options' => array(
        'archive' => 'filename.bz2',
    ),
));
$compressed = $filter->filter('Uncompressed string');
// Returns true on success and creates the archive file
```

In the above example the uncompressed string is compressed, and is then written into the given archive file.

Existing archives will be overwritten

The content of any existing file will be overwritten when the given filename of the archive already exists.

When you want to compress a file, then you must give the name of the file with its path.

```
$filter     = new Zend_Filter_Compress(array(
    'adapter' => 'Bz2',
    'options' => array(
        'archive' => 'filename.bz2'
    ),
));
$compressed = $filter->filter('C:\temp\compressme.txt');
// Returns true on success and creates the archive file
```

You may also specify a directory instead of a filename. In this case the whole directory with all its files and subdirectories will be compressed into the archive.

```
$filter     = new Zend_Filter_Compress(array(
    'adapter' => 'Bz2',
    'options' => array(
        'archive' => 'filename.bz2'
    ),
));
$compressed = $filter->filter('C:\temp\somedir');
// Returns true on success and creates the archive file
```

Do not compress large or base directories

You should never compress large or base directories like a complete partition. Compressing a complete partition is a very time consuming task which can lead to massive problems on your server when there is not enough space or your script takes too much time.

2.6.3. Decompressing an archive

Decompressing an archive file works almost like compressing it. You must specify either the archive parameter, or give the filename of the archive when you decompress the file.

```
$filter     = new Zend_Filter_Decompress('Bz2');
$compressed = $filter->filter('filename.bz2');
// Returns true on success and decompresses the archive file
```

Some adapters support decompressing the archive into another subdirectory. In this case you can set the target parameter.

```
$filter     = new Zend_Filter_Decompress(array(
    'adapter' => 'Zip',
    'options' => array(
        'target' => 'C:\temp',
    )
));
$compressed = $filter->filter('filename.zip');
// Returns true on success and decompresses the archive file
// into the given target directory
```

Directories to extract to must exist

When you want to decompress an archive into a directory, then that directory must exist.

2.6.4. Bz2 Adapter

The Bz2 Adapter can compress and decompress:

- Strings
- Files
- Directories

This adapter makes use of PHP's Bz2 extension.

To customize compression, this adapter supports the following options:

- *Archive*: This parameter sets the archive file which should be used or created.
- *Blocksize*: This parameter sets the blocksize to use. It can be from '0' to '9'. The default value is '4'.

All options can be set at instantiation or by using a related method. For example, the related methods for 'Blocksize' are `getBlocksize()` and `setBlocksize()`. You can also use the `setOptions()` method which accepts all options as array.

2.6.5. Gz Adapter

The Gz Adapter can compress and decompress:

- Strings
- Files
- Directories

This adapter makes use of PHP's Zlib extension.

To customize the compression this adapter supports the following options:

- *Archive*: This parameter sets the archive file which should be used or created.
- *Level*: This compression level to use. It can be from '0' to '9'. The default value is '9'.
- *Mode*: There are two supported modes. 'compress' and 'deflate'. The default value is 'compress'.

All options can be set at initiation or by using a related method. For example, the related methods for 'Level' are `getLevel()` and `setLevel()`. You can also use the `setOptions()` method which accepts all options as array.

2.6.6. Lzf Adapter

The Lzf Adapter can compress and decompress:

- Strings

Lzf supports only strings

The Lzf adapter can not handle files and directories.

This adapter makes use of PHP's Lzf extension.

There are no options available to customize this adapter.

2.6.7. Rar Adapter

The Rar Adapter can compress and decompress:

- Files
- Directories

Rar does not support strings

The Rar Adapter can not handle strings.

This adapter makes use of PHP's Rar extension.

Rar compression not supported

Due to restrictions with the Rar compression format, there is no compression available for free. When you want to compress files into a new Rar archive, you must provide a callback to the adapter that can invoke a Rar compression program.

To customize the compression this adapter supports the following options:

- *Archive*: This parameter sets the archive file which should be used or created.
- *Callback*: A callback which provides compression support to this adapter.
- *Password*: The password which has to be used for decompression.

- *Target*: The target where the decompressed files will be written to.

All options can be set at instantiation or by using a related method. For example, the related methods for 'Target' are `getTarget()` and `setTarget()`. You can also use the `setOptions()` method which accepts all options as array.

2.6.8. Tar Adapter

The Tar Adapter can compress and decompress:

- Files
- Directories

Tar does not support strings

The Tar Adapter can not handle strings.

This adapter makes use of PEAR's `Archive_Tar` component.

To customize the compression this adapter supports the following options:

- *Archive*: This parameter sets the archive file which should be used or created.
- *Mode*: A mode to use for compression. Supported are either 'NULL' which means no compression at all, 'Gz' which makes use of PHP's Zlib extension and 'Bz2' which makes use of PHP's Bz2 extension. The default value is 'NULL'.
- *Target*: The target where the decompressed files will be written to.

All options can be set at instantiation or by using a related method. For example, the related methods for 'Target' are `getTarget()` and `setTarget()`. You can also use the `setOptions()` method which accepts all options as array.

Directory usage

When compressing directories with Tar then the complete file path is used. This means that created Tar files will not only have the subdirectory but the complete path for the compressed file.

2.6.9. Zip Adapter

The Zip Adapter can compress and decompress:

- Strings
- Files
- Directories

Zip does not support string decompression

The Zip Adapter can not handle decompression to a string; decompression will always be written to a file.

This adapter makes use of PHP's `Zip` extension.

To customize the compression this adapter supports the following options:

- *Archive*: This parameter sets the archive file which should be used or created.
- *Target*: The target where the decompressed files will be written to.

All options can be set at instantiation or by using a related method. For example, the related methods for 'Target' are `getTarget()` and `setTarget()`. You can also use the `setOptions()` method which accepts all options as array.

2.7. Decrypt

This filter will decrypt any given string with the provided setting. Therefor it makes use of Adapters. Actually there are adapters for the `Mcrypt` and `OpenSSL` extensions from php.

For details about how to encrypt content look at the `Encrypt` filter. As the basics are covered within the `Encrypt` filter, we will describe here only the needed additional methods and changes for decryption.

2.7.1. Decryption with Mcrypt

For decrypting content which was previously encrypted with `Mcrypt` you need to have the options with which the encryption has been called.

There is one eminent difference for you. When you did not provide a vector at encryption you need to get it after you encrypted the content by using the `getVector()` method on the encryption filter. Without the correct vector you will not be able to decrypt the content.

As soon as you have provided all options decryption is as simple as encryption.

```
// Use the default blowfish settings
$filter = new Zend_Filter_Decrypt('myencryptionkey');

// Set the vector with which the content was encrypted
$filter->setVector('myvector');

$decrypted = $filter->filter('encoded_text_normally_unreadable');
print $decrypted;
```

Note that you will get an exception if the mcrypt extension is not available in your environment.

You should also note that all settings which be checked when you create the instance or when you call setEncryption(). If mcrypt detects problem with your settings an exception will be thrown.

2.7.2. Decryption with OpenSSL

Decryption with `OpenSSL` is as simple as encryption. But you need to have all data from the person who encrypted the content.

For decryption with OpenSSL you need:

- *private*: Your private key which will be used for decrypting the content. The private key can be either a filename with path of the key file, or just the content of the key file itself.
- *envelope*: The encrypted envelope key from the user who encrypted the content. You can either provide the path and filename of the key file, or just the content of the key file itself.

```
// Use openssl and provide a private key
$filter = new Zend_Filter_Decrypt(array(
    'adapter' => 'openssl',
    'private' => '/path/to/mykey/private.pem'
));

// of course you can also give the envelope keys at initiation
$filter->setEnvelopeKey(array(
    '/key/from/encoder/first.pem',
    '/key/from/encoder/second.pem'
));
```

Note that the OpenSSL adapter will not work when you do not provide valid keys.

Optionally it could be necessary to provide the passphrase for decrypting the keys themself by using the `setPassphrase()` method.

```
// Use openssl and provide a private key
$filter = new Zend_Filter_Decrypt(array(
    'adapter' => 'openssl',
    'private' => '/path/to/mykey/private.pem'
));

// of course you can also give the envelope keys at initiation
$filter->setEnvelopeKey(array(
    '/key/from/encoder/first.pem',
    '/key/from/encoder/second.pem'
));
$filter->setPassphrase('mypassphrase');
```

At last, decode the content. Our complete example for decrypting the previously encrypted content looks like this.

```
// Use openssl and provide a private key
$filter = new Zend_Filter_Decrypt(array(
    'adapter' => 'openssl',
    'private' => '/path/to/mykey/private.pem'
));

// of course you can also give the envelope keys at initiation
$filter->setEnvelopeKey(array(
    '/key/from/encoder/first.pem',
    '/key/from/encoder/second.pem'
));
$filter->setPassphrase('mypassphrase');

$decrypted = $filter->filter('encoded_text_normally_unreadable');
```

```
print $decrypted;
```

2.8. Digits

Returns the string `$value`, removing all but digit characters.

2.9. Dir

Returns directory name component of path.

2.10. Encrypt

This filter will encrypt any given string with the provided setting. Therefor it makes use of Adapters. Actually there are adapters for the `Mcrypt` and `OpenSSL` extensions from php.

As these two encryption methodologies work completely different, also the usage of the adapters differ. You have to select the adapter you want to use when initiating the filter.

```
// Use the Mcrypt adapter
$filter1 = new Zend_Filter_Encrypt(array('adapter' => 'mcrypt'));

// Use the OpenSSL adapter
$filter2 = new Zend_Filter_Encrypt(array('adapter' => 'openssl'));
```

To set another adapter you can also use `setAdapter()`, and the `getAdapter()` method to receive the actual set adapter.

```
// Use the Mcrypt adapter
$filter = new Zend_Filter_Encrypt();
$filter->setAdapter('openssl');
```

When you do not supply the `adapter` option or do not use setAdapter, then the `Mcrypt` adapter will be used per default.

2.10.1. Encryption with Mcrypt

When you have installed the `Mcrypt` extension you can use the `Mcrypt` adapter. This adapter supports the following options at initiation:

- *key*: The encryption key with which the input will be encrypted. You need the same key for decryption.
- *algorithm*: The algorithm which has to be used. It should be one of the algorithm ciphers which can be found under PHP's mcrypt ciphers. If not set it defaults to `blowfish`.
- *algorithm_directory*: The directory where the algorithm can be found. If not set it defaults to the path set within the mcrypt extension.
- *mode*: The encryption mode which has to be used. It should be one of the modes which can be found under PHP's mcrypt modes. If not set it defaults to `cbc`.
- *mode_directory*: The directory where the mode can be found. If not set it defaults to the path set within the `mcrypt` extension.

- *vector*: The initialization vector which shall be used. If not set it will be a random vector.
- *salt*: If the key should be used as salt value. The key used for encryption will then itself also be encrypted. Default is FALSE.

If you give a string instead of an array, this string will be used as key.

You can get/set the encryption values also afterwards with the `getEncryption()` and `setEncryption()` methods.

Note that you will get an exception if the mcrypt extension is not available in your environment.

You should also note that all settings which be checked when you create the instance or when you call setEncryption(). If mcrypt detects problem with your settings an exception will be thrown.

You can get/set the encryption vector by calling `getVector()` and `setVector()`. A given string will be truncated or padded to the needed vector size of the used algorithm.

Note that when you are not using an own vector, you must get the vector and store it. Otherwise you will not be able to decode the encoded string.

```
// Use the default blowfish settings
$filter = new Zend_Filter_Encrypt('myencryptionkey');

// Set a own vector, otherwise you must call getVector()
// and store this vector for later decryption
$filter->setVector('myvector');
// $filter->getVector();

$encrypted = $filter->filter('text_to_be_encoded');
print $encrypted;

// For decryption look at the Decrypt filter
```

2.10.2. Encryption with OpenSSL

When you have installed the `OpenSSL` extension you can use the `OpenSSL` adapter. This adapter supports the following options at initiation:

- *public*: The public key of the user whom you want to provide the encrpted content. You can give multiple public keys by using an array. You can eigther provide the path and filename of the key file, or just the content of the key file itself.
- *private*: Your private key which will be used for encrypting the content. Also the private key can be either a filename with path of the key file, or just the content of the key file itself.

You can get/set the public keys also afterwards with the `getPublicKey()` and `setPublicKey()` methods. The private key can also be get and set with the related `getPrivateKey()` and `setPrivateKey()` methods.

```
// Use openssl and provide a private key
```

```
$filter = new Zend_Filter_Encrypt(array(
    'adapter' => 'openssl',
    'private' => '/path/to/mykey/private.pem'
));

// of course you can also give the public keys at initiation
$filter->setPublicKey(array(
    '/public/key/path/first.pem',
    '/public/key/path/second.pem'
));
```

Note that the OpenSSL adapter will not work when you do not provide valid keys.

When you want to encode also the keys, then you have to provide a passphrase with the `setPassphrase()` method. When you want to decode content which was encoded with a passphrase you will not only need the public key, but also the passphrase to decode the encrypted key.

```
// Use openssl and provide a private key
$filter = new Zend_Filter_Encrypt(array(
    'adapter' => 'openssl',
    'private' => '/path/to/mykey/private.pem'
));

// of course you can also give the public keys at initiation
$filter->setPublicKey(array(
    '/public/key/path/first.pem',
    '/public/key/path/second.pem'
));
$filter->setPassphrase('mypassphrase');
```

At last, when you use OpenSSL you need to give the receiver the encrypted content, the passphrase when have provided one, and the envelope keys for decryption.

This means for you, that you have to get the envelope keys after the encryption with the `getEnvelopeKey()` method.

So our complete example for encrypting content with OpenSSL look like this.

```
// Use openssl and provide a private key
$filter = new Zend_Filter_Encrypt(array(
    'adapter' => 'openssl',
    'private' => '/path/to/mykey/private.pem'
));

// of course you can also give the public keys at initiation
$filter->setPublicKey(array(
    '/public/key/path/first.pem',
    '/public/key/path/second.pem'
));
$filter->setPassphrase('mypassphrase');

$encrypted = $filter->filter('text_to_be_encoded');
$envelope  = $filter->getEnvelopeKey();
print $encrypted;
```

```
// For decryption look at the Decrypt filter
```

2.11. HtmlEntities

Returns the string `$value`, converting characters to their corresponding HTML entity equivalents where they exist.

2.12. Int

Returns (int) `$value`

2.13. LocalizedToNormalized

This filter will change any given localized input to it's normalized representation. It uses in Background `Zend_Locale` to do this transformation for you.

This allows your user to enter informations in his own language notation, and you can then store the normalized value into your database for example.

Please note that normalization is not equal to translation. This filter can not translate strings from one language into another like you could expect with months or names of days.

The following input types can be normalized:

- *integer*: Integer numbers, which are localized, will be normalized to the english notation.
- *float*: Float numbers, which are localized, will be normalized to the english notation.
- *numbers*: Other numbers, like real, will be normalized to the english notation.
- *time*: Time values, will be normalized to a named array.
- *date*: Date values, will be normalized to a named array.

Any other input will be returned as it, without changing it.

You should note that normalized output is always given as string. Otherwise your environment would transfer the normalized output automatically to the notation used by the locale your environment is set to.

2.13.1. Normalization for numbers

Any given number like integer, float or real value, can be normalized. Note, that numbers in scientific notation, can actually not be handled by this filter.

So how does this normalization work in detail for numbers:

```
// Initiate the filter
$filter = new Zend_Filter_LocalizedToNormalized();
$filter->filter('123.456,78');
// returns the value '123456.78'
```

Let's expect you have set the locale 'de' as application wide locale. `Zend_Filter_LocalizedToNormalized` will take the set locale and use it to detect which sort of input you gave. In our example it was a value with precision. Now the filter will return you the normalized representation for this value as string.

You can also control how your normalized number has to look like. Therefor you can give all options which are also used by `Zend_Locale_Format`. The most common are:

- *date_format*
- *locale*
- *precision*

For details about how these options are used take a look into this Zend_Locale chapter.

Below is a example with defined precision so you can see how options work:

```
// Numeric Filter
$filter = new Zend_Filter_LocalizedToNormalized(array('precision' => 2));

$filter->filter('123.456');
// returns the value '123456.00'

$filter->filter('123.456,78901');
// returns the value '123456.79'
```

2.13.2. Normalization for date and time

Input for date and time values can also be normalized. All given date and time values will be returned as array, where each date part is given within a own key.

```
// Initiate the filter
$filter = new Zend_Filter_LocalizedToNormalized();
$filter->filter('12.April.2009');
// returns array('day' => '12', 'month' => '04', 'year' => '2009')
```

Let's expect you have set the locale 'de' again. Now the input is automatically detected as date, and you will get a named array in return.

Of course you can also control how your date input looks like with the *date_format* and the *locale* option.

```
// Date Filter
$filter = new Zend_Filter_LocalizedToNormalized(
    array('date_format' => 'ss:mm:HH')
);

$filter->filter('11:22:33');
// returns array('hour' => '33', 'minute' => '22', 'second' => '11')
```

2.14. NormalizedToLocalized

This filter is the reverse of the filter `Zend_Filter_LocalizedToNormalized` and will change any given normalized input to it's localized representation. It uses in Background `Zend_Locale` to do this transformation for you.

This allows you to give your user any stored normalised value in a localized manner, your user is more common to.

Please note that localization is not equal to translation. This filter can not translate strings from one language into another like you could expect with months or names of days.

The following input types can be localized:

- *integer*: Integer numbers, which are normalized, will be localized to the set notation.
- *float*: Float numbers, which are normalized, will be localized to the set notation.
- *numbers*: Other numbers, like real, will be localized to the set notation.
- *time*: Time values, will be localized to a string.
- *date*: Date values, will be normalized to a string.

Any other input will be returned as it, without changing it.

2.14.1. Localization for numbers

Any given number like integer, float or real value, can be localized. Note, that numbers in scientific notation, can actually not be handled by this filter.

So how does localization work in detail for numbers:

```
// Initiate the filter
$filter = new Zend_Filter_NormalizedToLocalized();
$filter->filter(123456.78);
// returns the value '123.456,78'
```

Let's expect you have set the locale 'de' as application wide locale. `Zend_Filter_NormalizedToLocalized` will take the set locale and use it to detect which sort of output you want to have. In our example it was a value with precision. Now the filter will return you the localized representation for this value as string.

You can also control how your localized number has to look like. Therefor you can give all options which are also used by `Zend_Locale_Format`. The most common are:

- *date_format*
- *locale*
- *precision*

For details about how these options are used take a look into this Zend_Locale chapter.

Below is a example with defined precision so you can see how options work:

```
// Numeric Filter
$filter = new Zend_Filter_NormalizedToLocalized(array('precision' => 2));

$filter->filter(123456);
// returns the value '123.456,00'
```

```
$filter->filter(123456.78901);
// returns the value '123.456,79'
```

2.14.2. Localization for date and time

Normalized for date and time values can also be localized. All given date and time values will be returned as string, with the format defined by the set locale.

```
// Initiate the filter
$filter = new Zend_Filter_NormalizedToLocalized();
$filter->filter(array('day' => '12', 'month' => '04', 'year' => '2009');
// returns '12.04.2009'
```

Let's expect you have set the locale 'de' again. Now the input is automatically detected as date, and will be returned in the format defined by the locale 'de'.

Of course you can also control how your date input looks like with the *date_format*, and the *locale* option.

```
// Date Filter
$filter = new Zend_Filter_LocalizedToNormalized(
    array('date_format' => 'ss:mm:HH')
);

$filter->filter(array('hour' => '33', 'minute' => '22', 'second' => '11'));
// returns '11:22:33'
```

2.15. Null

This filter will change the given input to be `NULL` if it meets specific criteria. This is often necessary when you work with databases and want to have a `NULL` value instead of a boolean or any other type.

2.15.1. Default behaviour for Zend_Filter_Null

Per default this filter works like PHP's `empty()` method; in other words, if `empty()` returns a boolean `TRUE`, then a `NULL` value will be returned.

```
$filter = new Zend_Filter_Null();
$value  = '';
$result = $filter->filter($value);
// returns null instead of the empty string
```

This means that without providing any configuration, `Zend_Filter_Null` will accept all input types and return `NULL` in the same cases as `empty()`.

Any other value will be returned as is, without any changes.

2.15.2. Changing behaviour for Zend_Filter_Null

Sometimes it's not enough to filter based on `empty()`. Therefor `Zend_Filter_Null` allows you to configure which type will be converted and which not.

The following types can be handled:

- *boolean*: Converts a boolean *FALSE* value to `NULL`.

- *integer*: Converts an integer *0* value to `NULL`.
- *empty_array*: Converts an empty *array* to `NULL`.
- *string*: Converts an empty string *''* to `NULL`.
- *zero*: Converts a string containing the single character zero (*'0'*) to `NULL`.
- *all*: Converts all above types to `NULL`. (This is the default behavior.)

There are several ways to select which of the above types are filtered. You can give one or multiple types and add them, you can give an array, you can use constants, or you can give a textual string. See the following examples:

```
// converts false to null
$filter = new Zend_Filter_Null(Zend_Filter_Null::BOOLEAN);

// converts false and 0 to null
$filter = new Zend_Filter_Null(
    Zend_Filter_Null::BOOLEAN + Zend_Filter_Null::INTEGER
);

// converts false and 0 to null
$filter = new Zend_Filter_Null( array(
    Zend_Filter_Null::BOOLEAN,
    Zend_Filter_Null::INTEGER
));

// converts false and 0 to null
$filter = new Zend_Filter_Null(array(
    'boolean',
    'integer',
));
```

You can also give an instance of `Zend_Config` to set the wished types. To set types afterwards use `setType()`.

2.16. PregReplace

`Zend_Filter_PregReplace` performs a search using regular expressions and replaces all found elements.

The option match has to be given to set the pattern which will be searched for. It can be a string for a single pattern, or an array of strings for multiple pattern.

To set the pattern which will be used as replacement the option replace has to be used. It can be a string for a single pattern, or an array of strings for multiple pattern.

```
$filter = new Zend_Filter_PregReplace(array('match' => 'bob',
                                            'replace' => 'john'));
$input  = 'Hy bob!";

$filter->filter($input);
// returns 'Hy john!'
```

You can use `getMatchPattern()` and `setMatchPattern()` to set the matching pattern afterwards. To set the replacement pattern you can use `getReplacement()` and `setReplacement()`.

```
$filter = new Zend_Filter_PregReplace();
$filter->setMatchPattern(array('bob', 'Hy'))
       ->setReplacement(array('john', 'Bye'));
$input  = 'Hy bob!";

$filter->filter($input);
// returns 'Bye john!'
```

For a more complex usage take a look into PHP's PCRE Pattern Chapter.

2.17. RealPath

This filter will resolve given links and pathnames and returns canonicalized absolute pathnames. References to `'/./'`, `'/../'` and extra `'/'` characters in the input path will be stripped. The resulting path will not have any symbolic link, `'/./'` or `'/../'` character.

`Zend_Filter_RealPath` will return FALSE on failure, e.g. if the file does not exist. On BSD systems `Zend_Filter_RealPath` doesn't fail if only the last path component doesn't exist, while other systems will return FALSE.

```
$filter = new Zend_Filter_RealPath();
$path   = '/www/var/path/../../mypath';
$filtered = $filter->filter($path);

// returns '/www/mypath'
```

Sometimes it is useful to get also paths when they don't exist, f.e. when you want to get the real path for a path which you want to create. You can then either give a FALSE at initiation, or use `setExists()` to set it.

```
$filter = new Zend_Filter_RealPath(false);
$path   = '/www/var/path/../../non/existing/path';
$filtered = $filter->filter($path);

// returns '/www/non/existing/path'
// even when file_exists or realpath would return false
```

2.18. StringToLower

This filter converts any input to be lowercased.

```
$filter = new Zend_Filter_StringToLower();

print $filter->filter('SAMPLE');
// returns "sample"
```

Per default it will only handle characters from the actual locale of your server. Characters from other charsets would be ignored. Still, it's possible to also lowercase them when the mbstring extension is available in your environment. Simply set the wished encoding when initiating the `StringToLower` filter. Or use the `setEncoding()` method to change the encoding afterwards.

```
// using UTF-8
$filter = new Zend_Filter_StringToLower('UTF-8');

// or give an array which can be useful when using a configuration
$filter = new Zend_Filter_StringToLower(array('encoding' => 'UTF-8'));
```

```
// or do this afterwards
$filter->setEncoding('ISO-8859-1');
```

Setting wrong encodings

Be aware that you will get an exception when you want to set an encoding and the mbstring extension is not available in your environment.

Also when you are trying to set an encoding which is not supported by your mbstring extension you will get an exception.

2.19. StringToUpper

This filter converts any input to be uppercased.

```
$filter = new Zend_Filter_StringToUpper();

print $filter->filter('Sample');
// returns "SAMPLE"
```

Like the `StringToLower` filter, this filter handles only characters from the actual locale of your server. Using different character sets works the same as with `StringToLower`.

```
$filter = new Zend_Filter_StringToUpper(array('encoding' => 'UTF-8'));

// or do this afterwards
$filter->setEncoding('ISO-8859-1');
```

2.20. StringTrim

Returns the string `$value` with characters stripped from the beginning and end.

2.21. StripNewlines

Returns the string `$value` without any newline control characters.

2.22. StripTags

This filter returns the input string, with all HTML and PHP tags stripped from it, except those that have been explicitly allowed. In addition to the ability to specify which tags are allowed, developers can specify which attributes are allowed across all allowed tags and for specific tags only.

3. Filter Chains

Often multiple filters should be applied to some value in a particular order. For example, a login form accepts a username that should be only lowercase, alphabetic characters. `Zend_Filter` provides a simple method by which filters may be chained together. The following code illustrates how to chain together two filters for the submitted username:

```
// Create a filter chain and add filters to the chain
$filterChain = new Zend_Filter();
$filterChain->addFilter(new Zend_Filter_Alpha())
```

```
            ->addFilter(new Zend_Filter_StringToLower());

// Filter the username
$username = $filterChain->filter($_POST['username']);
```

Filters are run in the order they were added to `Zend_Filter`. In the above example, the username is first removed of any non-alphabetic characters, and then any uppercase characters are converted to lowercase.

Any object that implements `Zend_Filter_Interface` may be used in a filter chain.

3.1. Changing filter chain order

Since 1.10, the `Zend_Filter` chain also supports altering the chain by prepending or appending filters. For example, the next piece of code does exactly the same as the other username filter chain example:

```
// Create a filter chain and add filters to the chain
$filterChain = new Zend_Filter();

// this filter will be appended to the filter chain
$filterChain->appendFilter(new Zend_Filter_StringToLower());

// this filter will be prepended at the beginning of the filter chain.
$filterChain->prependFilter(new Zend_Filter_Alpha());

// Filter the username
$username = $filterChain->filter($_POST['username']);
```

4. Writing Filters

`Zend_Filter` supplies a set of commonly needed filters, but developers will often need to write custom filters for their particular use cases. The task of writing a custom filter is facilitated by implementing `Zend_Filter_Interface`.

`Zend_Filter_Interface` defines a single method, `filter()`, that may be implemented by user classes. An object that implements this interface may be added to a filter chain with `Zend_Filter::addFilter()`.

The following example demonstrates how to write a custom filter:

```
class MyFilter implements Zend_Filter_Interface
{
    public function filter($value)
    {
        // perform some transformation upon $value to arrive on $valueFiltered

        return $valueFiltered;
    }
}
```

To add an instance of the filter defined above to a filter chain:

```
$filterChain = new Zend_Filter();
$filterChain->addFilter(new MyFilter());
```

5. Zend_Filter_Input

`Zend_Filter_Input` provides a declarative interface to associate multiple filters and validators, apply them to collections of data, and to retrieve input values after they have been processed by the filters and validators. Values are returned in escaped format by default for safe HTML output.

Consider the metaphor that this class is a cage for external data. Data enter the application from external sources, such as HTTP request parameters, HTTP headers, a web service, or even read from a database or another file. Data are first put into the cage, and subsequently the application can access data only by telling the cage what the data should be and how they plan to use it. The cage inspects the data for validity. It might apply escaping to the data values for the appropriate context. The cage releases data only if it can fulfill these responsibilities. With a simple and convenient interface, it encourages good programming habits and makes developers think about how data are used.

- *Filters* transform input values, by removing or changing characters within the value. The goal is to "normalize" input values until they match an expected format. For example, if a string of numeric digits is needed, and the input value is "abc123", then it might be a reasonable transformation to change the value to the string "123".

- *Validators* check input values against criteria and report whether they passed the test or not. The value is not changed, but the check may fail. For example, if a string must look like an email address, and the input value is "abc123", then the value is not considered valid.

- *Escapers* transform a value by removing magic behavior of certain characters. In some output contexts, special characters have meaning. For example, the characters '<' and '>' delimit HTML tags, and if a string containing those characters is output in an HTML context, the content between them might affect the output or functionality of the HTML presentation. Escaping the characters removes the special meaning, so they are output as literal characters.

To use `Zend_Filter_Input`, perform the following steps:

1. Declare filter and validator rules

2. Create the filter and validator processor

3. Provide input data

4. Retrieve validated fields and other reports

The following sections describe the steps for using this class.

5.1. Declaring Filter and Validator Rules

Before creating an instance of `Zend_Filter_Input`, declare an array of filter rules and an array of validator rules. This associative array maps a rule name to a filter or validator or a chain of filters or validators.

The following example filter rule set that declares the field 'month' is filtered by `Zend_Filter_Digits`, and the field 'account' is filtered by `Zend_Filter_StringTrim`. Then a validation rule set declares that the field 'account' is valid only if it contains only alphabetical characters.

```
$filters = array(
```

```
    'month'   => 'Digits',
    'account' => 'StringTrim'
);

$validators = array(
    'account' => 'Alpha'
);
```

Each key in the `$filters` array above is the name of a rule for applying a filter to a specific data field. By default, the name of the rule is also the name of the input data field to which to apply the rule.

You can declare a rule in several formats:

- A single string scalar, which is mapped to a class name.

```
$validators = array(
    'month'   => 'Digits',
);
```

- An object instance of one of the classes that implement Zend_Filter_Interface or Zend_Validate_Interface.

```
$digits = new Zend_Validate_Digits();

$validators = array(
    'month'   => $digits
);
```

- An array, to declare a chain of filters or validators. The elements of this array can be strings mapping to class names or filter/validator objects, as in the cases described above. In addition, you can use a third choice: an array containing a string mapping to the class name followed by arguments to pass to its constructor.

```
$validators = array(
    'month'   => array(
        'Digits',               // string
        new Zend_Validate_Int(), // object instance
        array('Between', 1, 12)  // string with constructor arguments
    )
);
```

If you declare a filter or validator with constructor arguments in an array, then you must make an array for the rule, even if the rule has only one filter or validator.

You can use a special "wildcard" rule key `'*'` in either the filters array or the validators array. This means that the filters or validators declared in this rule will be applied to all input data fields. Note that the order of entries in the filters array or validators array is significant; the rules are applied in the same order in which you declare them.

```
$filters = array(
    '*'     => 'StringTrim',
    'month' => 'Digits'
);
```

5.2. Creating the Filter and Validator Processor

After declaring the filters and validators arrays, use them as arguments in the constructor of `Zend_Filter_Input`. This returns an object that knows all your filtering and validating rules, and you can use this object to process one or more sets of input data.

```
$input = new Zend_Filter_Input($filters, $validators);
```

You can specify input data as the third constructor argument. The data structure is an associative array. The keys are field names, and the values are data values. The standard `$_GET` and `$_POST` superglobal variables in PHP are examples of this format. You can use either of these variables as input data for `Zend_Filter_Input`.

```
$data = $_GET;

$input = new Zend_Filter_Input($filters, $validators, $data);
```

Alternatively, use the `setData()` method, passing an associative array of key/value pairs the same format as described above.

```
$input = new Zend_Filter_Input($filters, $validators);
$input->setData($newData);
```

The `setData()` method redefines data in an existing `Zend_Filter_Input` object without changing the filtering and validation rules. Using this method, you can run the same rules against different sets of input data.

5.3. Retrieving Validated Fields and other Reports

After you have declared filters and validators and created the input processor, you can retrieve reports of missing, unknown, and invalid fields. You also can get the values of fields after filters have been applied.

5.3.1. Querying if the input is valid

If all input data pass the validation rules, the `isValid()` method returns `TRUE`. If any field is invalid or any required field is missing, `isValid()` returns `FALSE`.

```
if ($input->isValid()) {
  echo "OK\n";
}
```

This method accepts an optional string argument, naming an individual field. If the specified field passed validation and is ready for fetching, `isValid('fieldName')` returns `TRUE`.

```
if ($input->isValid('month')) {
  echo "Field 'month' is OK\n";
}
```

5.3.2. Getting Invalid, Missing, or Unknown Fields

- *Invalid* fields are those that don't pass one or more of their validation checks.
- *Missing* fields are those that are not present in the input data, but were declared with the metacommand `'presence'=>'required'` (see the later section on metacommands).

- *Unknown* fields are those that are not declared in any rule in the array of validators, but appear in the input data.

```
if ($input->hasInvalid() || $input->hasMissing()) {
  $messages = $input->getMessages();
}

// getMessages() simply returns the merge of getInvalid() and
// getMissing()

if ($input->hasInvalid()) {
  $invalidFields = $input->getInvalid();
}

if ($input->hasMissing()) {
  $missingFields = $input->getMissing();
}

if ($input->hasUnknown()) {
  $unknownFields = $input->getUnknown();
}
```

The results of the `getMessages()` method is an associative array, mapping a rule name to an array of error messages related to that rule. Note that the index of this array is the rule name used in the rule declaration, which may be different from the names of fields checked by the rule.

The `getMessages()` method returns the merge of the arrays returned by the `getInvalid()` and `getMissing()`. These methods return subsets of the messages, related to validation failures, or fields that were declared as required but missing from the input.

The `getErrors()` method returns an associative array, mapping a rule name to an array of error identifiers. Error identifiers are fixed strings, to identify the reason for a validation failure, while messages can be customized. See Section 1.2, “Basic usage of validators” for more information.

You can specify the message returned by `getMissing()` using the 'missingMessage' option, as an argument to the `Zend_Filter_Input` constructor or using the `setOptions()` method.

```
$options = array(
    'missingMessage' => "Field '%field%' is required"
);

$input = new Zend_Filter_Input($filters, $validators, $data, $options);

// alternative method:

$input = new Zend_Filter_Input($filters, $validators, $data);
$input->setOptions($options);
```

And you can also add a translator which gives you the ability to provide multiple languages for the messages which are returned by `Zend_Filter_Input`.

```
$translate = new Zend_Translate_Adapter_Array(array(
    Zend_Filter_Input::MISSING_MESSAGE => "Where is the field?"
);

$input = new Zend_Filter_Input($filters, $validators, $data);
$input->setTranslator($translate);
```

When you are using an application wide translator, then it will also be used by `Zend_Filter_Input`. In this case you will not have to set the translator manually.

The results of the `getUnknown()` method is an associative array, mapping field names to field values. Field names are used as the array keys in this case, instead of rule names, because no rule mentions the fields considered to be unknown fields.

5.3.3. Getting Valid Fields

All fields that are neither invalid, missing, nor unknown are considered valid. You can get values for valid fields using a magic accessor. There are also non-magic accessor methods `getEscaped()` and `getUnescaped()`.

```
$m = $input->month;                // escaped output from magic accessor
$m = $input->getEscaped('month');  // escaped output
$m = $input->getUnescaped('month'); // not escaped
```

By default, when retrieving a value, it is filtered with the `Zend_Filter_HtmlEntities`. This is the default because it is considered the most common usage to output the value of a field in HTML. The HtmlEntities filter helps prevent unintentional output of code, which can result in security problems.

As shown above, you can retrieve the unescaped value using the `getUnescaped()` method, but you must write code to use the value safely, and avoid security issues such as vulnerability to cross-site scripting attacks.

Escaping unvalidated fields

As mentioned before `getEscaped()` returns only validated fields. Fields which do not have an associated validator can not be received this way. Still, there is a possible way. You can add a empty validator for all fields.

```
$validators = array('*' => array());

$input = new Zend_Filter_Input($filters, $validators, $data, $options);
```

But be warned that using this notation introduces a security leak which could be used for cross-site scripting attacks. Therefor you should always set individual validators for each field.

You can specify a different filter for escaping values, by specifying it in the constructor options array:

```
$options = array('escapeFilter' => 'StringTrim');
$input = new Zend_Filter_Input($filters, $validators, $data, $options);
```

Alternatively, you can use the `setDefaultEscapeFilter()` method:

```
$input = new Zend_Filter_Input($filters, $validators, $data);
$input->setDefaultEscapeFilter(new Zend_Filter_StringTrim());
```

In either usage, you can specify the escape filter as a string base name of the filter class, or as an object instance of a filter class. The escape filter can be an instance of a filter chain, an object of the class `Zend_Filter`.

Filters to escape output should be run in this way, to make sure they run after validation. Other filters you declare in the array of filter rules are applied to input data before data are validated. If escaping filters were run before validation, the process of validation would be more complex, and it would be harder to provide both escaped and unescaped versions of the data. So it is recommended to declare filters to escape output using `setDefaultEscapeFilter()`, not in the `$filters` array.

There is only one method `getEscaped()`, and therefore you can specify only one filter for escaping (although this filter can be a filter chain). If you need a single instance of `Zend_Filter_Input` to return escaped output using more than one filtering method, you should extend `Zend_Filter_Input` and implement new methods in your subclass to get values in different ways.

5.4. Using Metacommands to Control Filter or Validator Rules

In addition to declaring the mapping from fields to filters or validators, you can specify some "metacommands" in the array declarations, to control some optional behavior of `Zend_Filter_Input`. Metacommands appear as string-indexed entries in a given filter or validator array value.

5.4.1. The FIELDS metacommand

If the rule name for a filter or validator is different than the field to which it should apply, you can specify the field name with the 'fields' metacommand.

You can specify this metacommand using the class constant `Zend_Filter_Input::FIELDS` instead of the string.

```
$filters = array(
    'month' => array(
        'Digits',        // filter name at integer index [0]
        'fields' => 'mo' // field name at string index ['fields']
    )
);
```

In the example above, the filter rule applies the 'digits' filter to the input field named 'mo'. The string 'month' simply becomes a mnemonic key for this filtering rule; it is not used as the field name if the field is specified with the 'fields' metacommand, but it is used as the rule name.

The default value of the 'fields' metacommand is the index of the current rule. In the example above, if the 'fields' metacommand is not specified, the rule would apply to the input field named 'month'.

Another use of the 'fields' metacommand is to specify fields for filters or validators that require multiple fields as input. If the 'fields' metacommand is an array, the argument to the corresponding filter or validator is an array of the values of those fields. For example, it is common for users to specify a password string in two fields, and they must type the same string in both fields. Suppose you implement a validator class that takes an array argument, and returns TRUE if all the values in the array are equal to each other.

```
$validators = array(
    'password' => array(
        'StringEquals',
        'fields' => array('password1', 'password2')
    )
);
```

```
// Invokes hypothetical class Zend_Validate_StringEquals,
// passing an array argument containing the values of the two input
// data fields named 'password1' and 'password2'.
```

If the validation of this rule fails, the rule key (`'password'`) is used in the return value of `getInvalid()`, not any of the fields named in the 'fields' metacommand.

5.4.2. The PRESENCE metacommand

Each entry in the validator array may have a metacommand called 'presence'. If the value of this metacommand is 'required' then the field must exist in the input data, or else it is reported as a missing field.

You can specify this metacommand using the class constant `Zend_Filter_Input::PRESENCE` instead of the string.

```
$validators = array(
    'month' => array(
        'digits',
        'presence' => 'required'
    )
);
```

The default value of this metacommand is 'optional'.

5.4.3. The DEFAULT_VALUE metacommand

If a field is not present in the input data, and you specify a value for the 'default' metacommand for that rule, the field takes the value of the metacommand.

You can specify this metacommand using the class constant `Zend_Filter_Input::DEFAULT_VALUE` instead of the string.

This default value is assigned to the field before any of the validators are invoked. The default value is applied to the field only for the current rule; if the same field is referenced in a subsequent rule, the field has no value when evaluating that rule. Thus different rules can declare different default values for a given field.

```
$validators = array(
    'month' => array(
        'digits',
        'default' => '1'
    )
);

// no value for 'month' field
$data = array();

$input = new Zend_Filter_Input(null, $validators, $data);
echo $input->month; // echoes 1
```

If your rule uses the `FIELDS` metacommand to define an array of multiple fields, you can define an array for the `DEFAULT_VALUE` metacommand and the defaults of corresponding keys are used for any missing fields. If `FIELDS` defines multiple fields but `DEFAULT_VALUE` is a scalar, then that default value is used as the value for any missing fields in the array.

There is no default value for this metacommand.

5.4.4. The ALLOW_EMPTY metacommand

By default, if a field exists in the input data, then validators are applied to it, even if the value of the field is an empty string (`''`). This is likely to result in a failure to validate. For example, if the validator checks for digit characters, and there are none because a zero-length string has no characters, then the validator reports the data as invalid.

If in your case an empty string should be considered valid, you can set the metacommand 'allowEmpty' to `TRUE`. Then the input data passes validation if it is present in the input data, but has the value of an empty string.

You can specify this metacommand using the class constant `Zend_Filter_Input::ALLOW_EMPTY` instead of the string.

```
$validators = array(
    'address2' => array(
        'Alnum',
        'allowEmpty' => true
    )
);
```

The default value of this metacommand is `FALSE`.

In the uncommon case that you declare a validation rule with no validators, but the 'allowEmpty' metacommand is `FALSE` (that is, the field is considered invalid if it is empty), `Zend_Filter_Input` returns a default error message that you can retrieve with `getMessages()`. You can specify this message using the 'notEmptyMessage' option, as an argument to the `Zend_Filter_Input` constructor or using the `setOptions()` method.

```
$options = array(
    'notEmptyMessage' => "A non-empty value is required for field '%field%'"
);

$input = new Zend_Filter_Input($filters, $validators, $data, $options);

// alternative method:

$input = new Zend_Filter_Input($filters, $validators, $data);
$input->setOptions($options);
```

5.4.5. The BREAK_CHAIN metacommand

By default if a rule has more than one validator, all validators are applied to the input, and the resulting messages contain all error messages caused by the input.

Alternatively, if the value of the 'breakChainOnFailure' metacommand is `TRUE`, the validator chain terminates after the first validator fails. The input data is not checked against subsequent validators in the chain, so it might cause more violations even if you correct the one reported.

You can specify this metacommand using the class constant `Zend_Filter_Input::BREAK_CHAIN` instead of the string.

```
$validators = array(
    'month' => array(
        'Digits',
        new Zend_Validate_Between(1,12),
        new Zend_Validate_GreaterThan(0),
```

```
        'breakChainOnFailure' => true
    )
);
$input = new Zend_Filter_Input(null, $validators);
```

The default value of this metacommand is FALSE.

The validator chain class, `Zend_Validate`, is more flexible with respect to breaking chain execution than `Zend_Filter_Input`. With the former class, you can set the option to break the chain on failure independently for each validator in the chain. With the latter class, the defined value of the 'breakChainOnFailure' metacommand for a rule applies uniformly for all validators in the rule. If you require the more flexible usage, you should create the validator chain yourself, and use it as an object in the validator rule definition:

```
// Create validator chain with non-uniform breakChainOnFailure
// attributes
$chain = new Zend_Validate();
$chain->addValidator(new Zend_Validate_Digits(), true);
$chain->addValidator(new Zend_Validate_Between(1,12), false);
$chain->addValidator(new Zend_Validate_GreaterThan(0), true);

// Declare validator rule using the chain defined above
$validators = array(
    'month' => $chain
);
$input = new Zend_Filter_Input(null, $validators);
```

5.4.6. The MESSAGES metacommand

You can specify error messages for each validator in a rule using the metacommand 'messages'. The value of this metacommand varies based on whether you have multiple validators in the rule, or if you want to set the message for a specific error condition in a given validator.

You can specify this metacommand using the class constant `Zend_Filter_Input::MESSAGES` instead of the string.

Below is a simple example of setting the default error message for a single validator.

```
$validators = array(
    'month' => array(
        'digits',
        'messages' => 'A month must consist only of digits'
    )
);
```

If you have multiple validators for which you want to set the error message, you should use an array for the value of the 'messages' metacommand.

Each element of this array is applied to the validator at the same index position. You can specify a message for the validator at position *n* by using the value *n* as the array index. Thus you can allow some validators to use their default message, while setting the message for a subsequent validator in the chain.

```
$validators = array(
    'month' => array(
        'digits',
        new Zend_Validate_Between(1, 12),
```

```
        'messages' => array(
            // use default message for validator [0]
            // set new message for validator [1]
            1 => 'A month value must be between 1 and 12'
        )
    )
);
```

If one of your validators has multiple error messages, they are identified by a message key. There are different keys in each validator class, serving as identifiers for error messages that the respective validator class might generate. Each validate class defines constants for its message keys. You can use these keys in the 'messages' metacommand by passing an associative array instead of a string.

```
$validators = array(
    'month' => array(
        'digits', new Zend_Validate_Between(1, 12),
        'messages' => array(
            'A month must consist only of digits',
            array(
                Zend_Validate_Between::NOT_BETWEEN =>
                    'Month value %value% must be between ' .
                    '%min% and %max%',
                Zend_Validate_Between::NOT_BETWEEN_STRICT =>
                    'Month value %value% must be strictly between ' .
                    '%min% and %max%'
            )
        )
    )
);
```

You should refer to documentation for each validator class to know if it has multiple error messages, the keys of these messages, and the tokens you can use in the message templates.

If you have only one validator in validation rule or all used validators has the same messages set, then they can be referenced without additional array construction:

```
$validators = array(
    'month' => array(
        new Zend_Validate_Between(1, 12),
        'messages' => array(
                        Zend_Validate_Between::NOT_BETWEEN =>
                            'Month value %value% must be between ' .
                            '%min% and %max%',
                        Zend_Validate_Between::NOT_BETWEEN_STRICT =>
                            'Month value %value% must be strictly between ' .
                            '%min% and %max%'
        )
    )
);
```

5.4.7. Using options to set metacommands for all rules

The default value for 'allowEmpty', 'breakChainOnFailure', and 'presence' metacommands can be set for all rules using the `$options` argument to the constructor of `Zend_Filter_Input`. This allows you to set the default value for all rules, without requiring you to set the metacommand for every rule.

```
// The default is set so all fields allow an empty string.
$options = array('allowEmpty' => true);

// You can override this in a rule definition,
// if a field should not accept an empty string.
$validators = array(
    'month' => array(
        'Digits',
        'allowEmpty' => false
    )
);

$input = new Zend_Filter_Input($filters, $validators, $data, $options);
```

The 'fields', 'messages', and 'default' metacommands cannot be set using this technique.

5.5. Adding Filter Class Namespaces

By default, when you declare a filter or validator as a string, `Zend_Filter_Input` searches for the corresponding classes under the `Zend_Filter` or `Zend_Validate` namespaces. For example, a filter named by the string 'digits' is found in the class `Zend_Filter_Digits`.

If you write your own filter or validator classes, or use filters or validators provided by a third-party, the classes may exist in different namespaces than `Zend_Filter` or `Zend_Validate`. You can tell `Zend_Filter_Input` to search more namespaces. You can specify namespaces in the constructor options:

```
$options = array('filterNamespace' => 'My_Namespace_Filter',
                 'validatorNamespace' => 'My_Namespace_Validate');
$input = new Zend_Filter_Input($filters, $validators, $data, $options);
```

Alternatively, you can use the `addValidatorPrefixPath($prefix, $path)` or `addFilterPrefixPath($prefix, $path)` methods, which directly proxy to the plugin loader that is used by `Zend_Filter_Input`:

```
$input->addValidatorPrefixPath('Other_Namespace', 'Other/Namespace');
$input->addFilterPrefixPath('Foo_Namespace', 'Foo/Namespace');

// Now the search order for validators is:
// 1. My_Namespace_Validate
// 2. Other_Namespace
// 3. Zend_Validate

// The search order for filters is:
// 1. My_Namespace_Filter
// 2. Foo_Namespace
// 3. Zend_Filter
```

You cannot remove `Zend_Filter` and `Zend_Validate` as namespaces, you only can add namespaces. User-defined namespaces are searched first, Zend namespaces are searched last.

As of version 1.5 the function `addNamespace($namespace)` was deprecated and exchanged with the plugin loader and the `addFilterPrefixPath` and `addValidatorPrefixPath` were added. Also the constant `Zend_Filter_Input::INPUT_NAMESPACE` is now

deprecated. The constants `Zend_Filter_Input::VALIDATOR_NAMESPACE` and `Zend_Filter_Input::FILTER_NAMESPACE` are available in releases after 1.7.0.

As of version 1.0.4, `Zend_Filter_Input::NAMESPACE`, having value `namespace`, was changed to `Zend_Filter_Input::INPUT_NAMESPACE`, having value `inputNamespace`, in order to comply with the PHP 5.3 reservation of the keyword `namespace`.

6. Zend_Filter_Inflector

`Zend_Filter_Inflector` is a general purpose tool for rules-based inflection of strings to a given target.

As an example, you may find you need to transform MixedCase or camelCasedWords into a path; for readability, OS policies, or other reasons, you also need to lower case this, and you want to separate the words using a dash ('-'). An inflector can do this for you.

`Zend_Filter_Inflector` implements `Zend_Filter_Interface`; you perform inflection by calling `filter()` on the object instance.

Example 379. Transforming MixedCase and camelCaseText to another format

```
$inflector = new Zend_Filter_Inflector('pages/:page.:suffix');
$inflector->setRules(array(
    ':page'  => array('Word_CamelCaseToDash', 'StringToLower'),
    'suffix' => 'html'
));

$string   = 'camelCasedWords';
$filtered = $inflector->filter(array('page' => $string));
// pages/camel-cased-words.html

$string   = 'this_is_not_camel_cased';
$filtered = $inflector->filter(array('page' => $string));
// pages/this_is_not_camel_cased.html
```

6.1. Operation

An inflector requires a *target* and one or more *rules*. A target is basically a string that defines placeholders for variables you wish to substitute. These are specified by prefixing with a ':': `:script`.

When calling `filter()`, you then pass in an array of key/value pairs corresponding to the variables in the target.

Each variable in the target can have zero or more rules associated with them. Rules may be either *static* or refer to a `Zend_Filter` class. Static rules will replace with the text provided. Otherwise, a class matching the rule provided will be used to inflect the text. Classes are typically specified using a short name indicating the filter name stripped of any common prefix.

As an example, you can use any `Zend_Filter` concrete implementations; however, instead of referring to them as 'Zend_Filter_Alpha' or 'Zend_Filter_StringToLower', you'd specify only 'Alpha' or 'StringToLower'.

6.2. Setting Paths To Alternate Filters

`Zend_Filter_Inflector` uses `Zend_Loader_PluginLoader` to manage loading filters to use with inflection. By default, any filter prefixed with `Zend_Filter` will be available. To access filters with that prefix but which occur deeper in the hierarchy, such as the various Word filters, simply strip off the `Zend_Filter` prefix:

```
// use Zend_Filter_Word_CamelCaseToDash as a rule
$inflector->addRules(array('script' => 'Word_CamelCaseToDash'));
```

To set alternate paths, `Zend_Filter_Inflector` has a utility method that proxies to the plugin loader, `addFilterPrefixPath()`:

```
$inflector->addFilterPrefixPath('My_Filter', 'My/Filter/');
```

Alternatively, you can retrieve the plugin loader from the inflector, and interact with it directly:

```
$loader = $inflector->getPluginLoader();
$loader->addPrefixPath('My_Filter', 'My/Filter/');
```

For more options on modifying the paths to filters, please see the PluginLoader documentation.

6.3. Setting the Inflector Target

The inflector target is a string with some placeholders for variables. Placeholders take the form of an identifier, a colon (':') by default, followed by a variable name: ':script', ':path', etc. The `filter()` method looks for the identifier followed by the variable name being replaced.

You can change the identifier using the `setTargetReplacementIdentifier()` method, or passing it as the third argument to the constructor:

```
// Via constructor:
$inflector = new Zend_Filter_Inflector('#foo/#bar.#sfx', null, '#');

// Via accessor:
$inflector->setTargetReplacementIdentifier('#');
```

Typically, you will set the target via the constructor. However, you may want to re-set the target later (for instance, to modify the default inflector in core components, such as the `ViewRenderer` or `Zend_Layout`). `setTarget()` can be used for this purpose:

```
$inflector = $layout->getInflector();
$inflector->setTarget('layouts/:script.phtml');
```

Additionally, you may wish to have a class member for your class that you can use to keep the inflector target updated -- without needing to directly update the target each time (thus saving on method calls). `setTargetReference()` allows you to do this:

```
class Foo
{
    /**
     * @var string Inflector target
     */
    protected $_target = 'foo/:bar/:baz.:suffix';
```

```
    /**
     * Constructor
     * @return void
     */
    public function __construct()
    {
        $this->_inflector = new Zend_Filter_Inflector();
        $this->_inflector->setTargetReference($this->_target);
    }

    /**
     * Set target; updates target in inflector
     *
     * @param  string $target
     * @return Foo
     */
    public function setTarget($target)
    {
        $this->_target = $target;
        return $this;
    }
}
```

6.4. Inflection Rules

As mentioned in the introduction, there are two types of rules: static and filter-based.

It is important to note that regardless of the method in which you add rules to the inflector, either one-by-one, or all-at-once; the order is very important. More specific names, or names that might contain other rule names, must be added before least specific names. For example, assuming the two rule names 'moduleDir' and 'module', the 'moduleDir' rule should appear before module since 'module' is contained within 'moduleDir'. If 'module' were added before 'moduleDir', 'module' will match part of 'moduleDir' and process it leaving 'Dir' inside of the target uninflected.

6.4.1. Static Rules

Static rules do simple string substitution; use them when you have a segment in the target that will typically be static, but which you want to allow the developer to modify. Use the `setStaticRule()` method to set or modify the rule:

```
$inflector = new Zend_Filter_Inflector(':script.:suffix');
$inflector->setStaticRule('suffix', 'phtml');

// change it later:
$inflector->setStaticRule('suffix', 'php');
```

Much like the target itself, you can also bind a static rule to a reference, allowing you to update a single variable instead of require a method call; this is often useful when your class uses an inflector internally, and you don't want your users to need to fetch the inflector in order to update it. The `setStaticRuleReference()` method is used to accomplish this:

```
class Foo
{
    /**
```

```
     * @var string Suffix
     */
    protected $_suffix = 'phtml';

    /**
     * Constructor
     * @return void
     */
    public function __construct()
    {
        $this->_inflector = new Zend_Filter_Inflector(':script.:suffix');
        $this->_inflector->setStaticRuleReference('suffix', $this->_suffix);
    }

    /**
     * Set suffix; updates suffix static rule in inflector
     *
     * @param  string $suffix
     * @return Foo
     */
    public function setSuffix($suffix)
    {
        $this->_suffix = $suffix;
        return $this;
    }
}
```

6.4.2. Filter Inflector Rules

`Zend_Filter` filters may be used as inflector rules as well. Just like static rules, these are bound to a target variable; unlike static rules, you may define multiple filters to use when inflecting. These filters are processed in order, so be careful to register them in an order that makes sense for the data you receive.

Rules may be added using `setFilterRule()` (which overwrites any previous rules for that variable) or `addFilterRule()` (which appends the new rule to any existing rule for that variable). Filters are specified in one of the following ways:

- *String*. The string may be a filter class name, or a class name segment minus any prefix set in the inflector's plugin loader (by default, minus the 'Zend_Filter' prefix).
- *Filter object*. Any object instance implementing `Zend_Filter_Interface` may be passed as a filter.
- *Array*. An array of one or more strings or filter objects as defined above.

```
$inflector = new Zend_Filter_Inflector(':script.:suffix');

// Set rule to use Zend_Filter_Word_CamelCaseToDash filter
$inflector->setFilterRule('script', 'Word_CamelCaseToDash');

// Add rule to lowercase string
$inflector->addFilterRule('script', new Zend_Filter_StringToLower());

// Set rules en-masse
$inflector->setFilterRule('script', array(
    'Word_CamelCaseToDash',
    new Zend_Filter_StringToLower()
```

```
));
```

6.4.3. Setting Many Rules At Once

Typically, it's easier to set many rules at once than to configure a single variable and its inflection rules at a time. `Zend_Filter_Inflector`'s `addRules()` and `setRules()` method allow this.

Each method takes an array of variable/rule pairs, where the rule may be whatever the type of rule accepts (string, filter object, or array). Variable names accept a special notation to allow setting static rules and filter rules, according to the following notation:

- *':' prefix*: filter rules.
- *No prefix*: static rule.

Example 380. Setting Multiple Rules at Once

```
// Could also use setRules() with this notation:
$inflector->addRules(array(
    // filter rules:
    ':controller' => array('CamelCaseToUnderscore','StringToLower'),
    ':action'     => array('CamelCaseToUnderscore','StringToLower'),

    // Static rule:
    'suffix'      => 'phtml'
));
```

6.5. Utility Methods

`Zend_Filter_Inflector` has a number of utility methods for retrieving and setting the plugin loader, manipulating and retrieving rules, and controlling if and when exceptions are thrown.

- `setPluginLoader()` can be used when you have configured your own plugin loader and wish to use it with `Zend_Filter_Inflector`; `getPluginLoader()` retrieves the currently set one.
- `setThrowTargetExceptionsOn()` can be used to control whether or not `filter()` throws an exception when a given replacement identifier passed to it is not found in the target. By default, no exceptions are thrown. `isThrowTargetExceptionsOn()` will tell you what the current value is.
- `getRules($spec = null)` can be used to retrieve all registered rules for all variables, or just the rules for a single variable.
- `getRule($spec, $index)` fetches a single rule for a given variable; this can be useful for fetching a specific filter rule for a variable that has a filter chain. `$index` must be passed.
- `clearRules()` will clear all currently registered rules.

6.6. Using Zend_Config with Zend_Filter_Inflector

You can use `Zend_Config` to set rules, filter prefix paths, and other object state in your inflectors, either by passing a `Zend_Config` object to the constructor or `setConfig()`. The following settings may be specified:

- `target` specifies the inflection target.

- `filterPrefixPath` specifies one or more filter prefix/path pairs for use with the inflector.
- `throwTargetExceptionsOn` should be a boolean indicating whether or not to throw exceptions when a replacement identifier is still present after inflection.
- `targetReplacementIdentifier` specifies the character to use when identifying replacement variables in the target string.
- `rules` specifies an array of inflection rules; it should consist of keys that specify either values or arrays of values, consistent with `addRules()`.

Example 381. Using Zend_Config with Zend_Filter_Inflector

```
// With the constructor:
$config    = new Zend_Config($options);
$inflector = new Zend_Filter_Inflector($config);

// Or with setConfig():
$inflector = new Zend_Filter_Inflector();
$inflector->setConfig($config);
```

Zend_Form

1. Zend_Form

`Zend_Form` simplifies form creation and handling in your web application. It performs the following tasks:

- Element input filtering and validation
- Element ordering
- Element and Form rendering, including escaping
- Element and form grouping
- Element and form-level configuration

`Zend_Form` makes use of several Zend Framework components to accomplish its goals, including `Zend_Config`, `Zend_Validate`, `Zend_Filter`, `Zend_Loader_PluginLoader`, and optionally `Zend_View`.

2. Zend_Form Quick Start

This quick start guide covers the basics of creating, validating, and rendering forms with `Zend_Form`.

2.1. Create a form object

Creating a form object is very simple: simply instantiate `Zend_Form`:

```
$form = new Zend_Form;
```

For advanced use cases, you may want to create a `Zend_Form` subclass, but for simple forms, you can create a form programmatically using a `Zend_Form` object.

If you wish to specify the form action and method (always good ideas), you can do so with the `setAction()` and `setMethod()` accessors:

```
$form->setAction('/resource/process')
     ->setMethod('post');
```

The above code sets the form action to the partial URL "/resource/process" and the form method to HTTP POST. This will be reflected during final rendering.

You can set additional HTML attributes for the `<form>` tag by using the setAttrib() or setAttribs() methods. For instance, if you wish to set the id, set the "id" attribute:

```
$form->setAttrib('id', 'login');
```

2.2. Add elements to the form

A form is nothing without its elements. `Zend_Form` ships with some default elements that render XHTML via `Zend_View` helpers. These are as follows:

- button
- checkbox (or many checkboxes at once with multiCheckbox)
- hidden
- image
- password
- radio
- reset
- select (both regular and multi-select types)
- submit
- text
- textarea

You have two options for adding elements to a form: you can instantiate concrete elements and pass in these objects, or you can pass in simply the element type and have `Zend_Form` instantiate an object of the correct type for you.

Some examples:

```
// Instantiating an element and passing to the form object:
$form->addElement(new Zend_Form_Element_Text('username'));

// Passing a form element type to the form object:
$form->addElement('text', 'username');
```

By default, these do not have any validators or filters. This means you will need to configure your elements with at least validators, and potentially filters. You can either do this (a) before you pass the element to the form, (b) via configuration options passed in when creating an element via `Zend_Form`, or (c) by pulling the element from the form object and configuring it after the fact.

Let's first look at creating validators for a concrete element instance. You can either pass in `Zend_Validate_*` objects, or the name of a validator to utilize:

```
$username = new Zend_Form_Element_Text('username');

// Passing a Zend_Validate_* object:
$username->addValidator(new Zend_Validate_Alnum());

// Passing a validator name:
$username->addValidator('alnum');
```

When using this second option, you can pass constructor arguments in an array as the third parameter if the validator can accept tem:

```
// Pass a pattern
$username->addValidator('regex', false, array('/^[a-z]/i'));
```

(The second parameter is used to indicate whether or not failure of this validator should prevent later validators from running; by default, this is FALSE.)

You may also wish to specify an element as required. This can be done using an accessor or passing an option when creating the element. In the former case:

```
// Make this element required:
$username->setRequired(true);
```

When an element is required, a 'NotEmpty' validator is added to the top of the validator chain, ensuring that the element has a value when required.

Filters are registered in basically the same way as validators. For illustration purposes, let's add a filter to lowercase the final value:

```
$username->addFilter('StringtoLower');
```

The final element setup might look like this:

```
$username->addValidator('alnum')
         ->addValidator('regex', false, array('/^[a-z]/'))
         ->setRequired(true)
         ->addFilter('StringToLower');

// or, more compactly:
$username->addValidators(array('alnum',
        array('regex', false, '/^[a-z]/i')
    ))
    ->setRequired(true)
    ->addFilters(array('StringToLower'));
```

Simple as this is, repeating it this for every element in a form can be a bit tedious. Let's try option (b) from above. When we create a new element using `Zend_Form::addElement()` as a factory, we can optionally pass in configuration options. These can include validators and filters. To do all of the above implicitly, try the following:

```
$form->addElement('text', 'username', array(
    'validators' => array(
        'alnum',
        array('regex', false, '/^[a-z]/i')
    ),
    'required' => true,
    'filters'  => array('StringToLower'),
));
```

If you find you are setting up elements using the same options in many locations, you may want to consider creating your own `Zend_Form_Element` subclass and utilizing that class instead; this will save you typing in the long-run.

2.3. Render a form

Rendering a form is simple. Most elements use a `Zend_View` helper to render themselves, and thus need a view object in order to render. Other than that, you have two options: use the form's render() method, or simply echo it.

```
// Explicitly calling render(), and passing an optional view object:
echo $form->render($view);

// Assuming a view object has been previously set via setView():
```

```
echo $form;
```

By default, `Zend_Form` and `Zend_Form_Element` will attempt to use the view object initialized in the `ViewRenderer`, which means you won't need to set the view manually when using the Zend Framework MVC. To render a form in a view, you simply have to do the following:

```
<?php echo $this->form ?>
```

Under the hood, `Zend_Form` uses "decorators" to perform rendering. These decorators can replace content, append content, or prepend content, and can fully introspect the element passed to them. As a result, you can combine multiple decorators to achieve custom effects. By default, `Zend_Form_Element` actually combines four decorators to achieve its output; setup looks something like this:

```
$element->addDecorators(array(
    'ViewHelper',
    'Errors',
    array('HtmlTag', array('tag' => 'dd')),
    array('Label', array('tag' => 'dt')),
));
```

(Where <HELPERNAME> is the name of a view helper to use, and varies based on the element.)

The above creates output like the following:

```
<dt><label for="username" class="required">Username</dt>
<dd>
    <input type="text" name="username" value="123-abc" />
    <ul class="errors">
        <li>'123-abc' has not only alphabetic and digit characters</li>
        <li>'123-abc' does not match against pattern '/^[a-z]/i'</li>
    </ul>
</dd>
```

(Albeit not with the same formatting.)

You can change the decorators used by an element if you wish to have different output; see the section on decorators for more information.

The form itself simply loops through the elements, and dresses them in an HTML `<form>`. The action and method you provided when setting up the form are provided to the `<form>` tag, as are any attributes you set via `setAttribs()` and family.

Elements are looped either in the order in which they were registered, or, if your element contains an order attribute, that order will be used. You can set an element's order using:

```
$element->setOrder(10);
```

Or, when creating an element, by passing it as an option:

```
$form->addElement('text', 'username', array('order' => 10));
```

2.4. Check if a form is valid

After a form is submitted, you will need to check and see if it passes validations. Each element is checked against the data provided; if a key matching the element name is not present, and the item is marked as required, validations are run with a `NULL` value.

Where does the data come from? You can use `$_POST` or `$_GET`, or any other data source you might have at hand (web service requests, for instance):

```
if ($form->isValid($_POST)) {
    // success!
} else {
    // failure!
}
```

With AJAX requests, you can sometimes get away with validating a single element, or groups of elements. `isValidPartial()` will validate a partial form. Unlike `isValid()`, however, if a particular key is not present, it will not run validations for that particular element:

```
if ($form->isValidPartial($_POST)) {
    // elements present all passed validations
} else {
    // one or more elements tested failed validations
}
```

An additional method, `processAjax()`, can be used for validating partial forms. Unlike `isValidPartial()`, it returns a JSON-formatted string containing error messages on failure.

Assuming your validations have passed, you can now fetch the filtered values:

```
$values = $form->getValues();
```

If you need the unfiltered values at any point, use:

```
$unfiltered = $form->getUnfilteredValues();
```

If you on the other hand need all the valid and filtered values of a partially valid form, you can call:

```
$values = $form->getValidValues($_POST);
```

2.5. Get error status

Did your form have failed validations on submission? In most cases, you can simply render the form again, and errors will be displayed when using the default decorators:

```
if (!$form->isValid($_POST)) {
    echo $form;

    // or assign to the view object and render a view...
    $this->view->form = $form;
    return $this->render('form');
}
```

If you want to inspect the errors, you have two methods. `getErrors()` returns an associative array of element names / codes (where codes is an array of error codes). `getMessages()` returns an associative array of element names / messages (where messages is an associative array of error code / error message pairs). If a given element does not have any errors, it will not be included in the array.

2.6. Putting it together

Let's build a simple login form. It will need elements representing:

- username
- password
- submit

For our purposes, let's assume that a valid username should be alphanumeric characters only, start with a letter, have a minimum length of 6, and maximum length of 20; they will be normalized to lowercase. Passwords must be a minimum of 6 characters. We'll simply toss the submit value when done, so it can remain unvalidated.

We'll use the power of `Zend_Form`'s configuration options to build the form:

```
$form = new Zend_Form();
$form->setAction('/user/login')
     ->setMethod('post');

// Create and configure username element:
$username = $form->createElement('text', 'username');
$username->addValidator('alnum')
         ->addValidator('regex', false, array('/^[a-z]+/'))
         ->addValidator('stringLength', false, array(6, 20))
         ->setRequired(true)
         ->addFilter('StringToLower');

// Create and configure password element:
$password = $form->createElement('password', 'password');
$password->addValidator('StringLength', false, array(6))
         ->setRequired(true);

// Add elements to form:
$form->addElement($username)
     ->addElement($password)
     // use addElement() as a factory to create 'Login' button:
     ->addElement('submit', 'login', array('label' => 'Login'));
```

Next, we'll create a controller for handling this:

```
class UserController extends Zend_Controller_Action
{
    public function getForm()
    {
        // create form as above
        return $form;
    }

    public function indexAction()
    {
        // render user/form.phtml
        $this->view->form = $this->getForm();
        $this->render('form');
    }

    public function loginAction()
    {
        if (!$this->getRequest()->isPost()) {
            return $this->_forward('index');
        }
        $form = $this->getForm();
```

```
        if (!$form->isValid($_POST)) {
            // Failed validation; redisplay form
            $this->view->form = $form;
            return $this->render('form');
        }

        $values = $form->getValues();
        // now try and authenticate....
    }
}
```

And a view script for displaying the form:

```
<h2>Please login:</h2>
<?php echo $this->form ?>
```

As you'll note from the controller code, there's more work to do: while the submission may be valid, you may still need to do some authentication using `Zend_Auth` or another authorization mechanism.

2.7. Using a Zend_Config Object

All `Zend_Form` classes are configurable using `Zend_Config`; you can either pass a `Zend_Config` object to the constructor or pass it in with `setConfig()`. Let's look at how we might create the above form using an INI file. First, let's follow the recommendations, and place our configurations into sections reflecting the release location, and focus on the 'development' section. Next, we'll setup a section for the given controller ('user'), and a key for the form ('login'):

```
[development]
; general form metainformation
user.login.action = "/user/login"
user.login.method = "post"

; username element
user.login.elements.username.type = "text"
user.login.elements.username.options.validators.alnum.validator = "alnum"
user.login.elements.username.options.validators.regex.validator = "regex"
user.login.elements.username.options.validators.regex.options.pattern = "/^[a-z]/i"
user.login.elements.username.options.validators.strlen.validator = "StringLength"
user.login.elements.username.options.validators.strlen.options.min = "6"
user.login.elements.username.options.validators.strlen.options.max = "20"
user.login.elements.username.options.required = true
user.login.elements.username.options.filters.lower.filter = "StringToLower"

; password element
user.login.elements.password.type = "password"
user.login.elements.password.options.validators.strlen.validator = "StringLength"
user.login.elements.password.options.validators.strlen.options.min = "6"
user.login.elements.password.options.required = true

; submit element
user.login.elements.submit.type = "submit"
```

You would then pass this to the form constructor:

```
$config = new Zend_Config_Ini($configFile, 'development');
$form   = new Zend_Form($config->user->login);
```

and the entire form will be defined.

2.8. Conclusion

Hopefully with this little tutorial, you should now be well on your way to unlocking the power and flexibility of `Zend_Form`. Read on for more in-depth information!

3. Creating Form Elements Using Zend_Form_Element

A form is made of elements that typically correspond to HTML form input. `Zend_Form_Element` encapsulates single form elements, with the following areas of responsibility:

- validation (is submitted data valid?)
  - capturing of validation error codes and messages
- filtering (how is the element escaped or normalized prior to validation and/or for output?)
- rendering (how is the element displayed?)
- metadata and attributes (what information further qualifies the element?)

The base class, `Zend_Form_Element`, has reasonable defaults for many cases, but it is best to extend the class for commonly used special purpose elements. Additionally, Zend Framework ships with a number of standard XHTML elements; you can read about them in the Standard Elements chapter.

3.1. Plugin Loaders

`Zend_Form_Element` makes use of Zend_Loader_PluginLoader to allow developers to specify locations of alternate validators, filters, and decorators. Each has its own plugin loader associated with it, and general accessors are used to retrieve and modify each.

The following loader types are used with the various plugin loader methods: 'validate', 'filter', and 'decorator'. The type names are case insensitive.

The methods used to interact with plugin loaders are as follows:

- `setPluginLoader($loader, $type)`: `$loader` is the plugin loader object itself, while `$type` is one of the types specified above. This sets the plugin loader for the given type to the newly specified loader object.
- `getPluginLoader($type)`: retrieves the plugin loader associated with `$type`.
- `addPrefixPath($prefix, $path, $type = null)`: adds a prefix/path association to the loader specified by `$type`. If `$type` is `NULL`, it will attempt to add the path to all loaders, by appending the prefix with each of "_Validate", "_Filter", and "_Decorator"; and appending the path with "Validate/", "Filter/", and "Decorator/". If you have all your extra form element classes under a common hierarchy, this is a convenience method for setting the base prefix for them.
- `addPrefixPaths(array $spec)`: allows you to add many paths at once to one or more plugin loaders. It expects each array item to be an array with the keys 'path', 'prefix', and 'type'.

Custom validators, filters, and decorators are an easy way to share functionality between forms and to encapsulate custom functionality.

Example 382. Custom Label

One common use case for plugins is to provide replacements for standard classes. For instance, if you want to provide a different implementation of the 'Label' decorator -- for instance, to always append a colon -- you could create your own 'Label' decorator with your own class prefix, and then add it to your prefix path.

Let's start with a custom Label decorator. We'll give it the class prefix "My_Decorator", and the class itself will be in the file "My/Decorator/Label.php".

```
class My_Decorator_Label extends Zend_Form_Decorator_Abstract
{
    protected $_placement = 'PREPEND';

    public function render($content)
    {
        if (null === ($element = $this->getElement())) {
            return $content;
        }
        if (!method_exists($element, 'getLabel')) {
            return $content;
        }

        $label = $element->getLabel() . ':';

        if (null === ($view = $element->getView())) {
            return $this->renderLabel($content, $label);
        }

        $label = $view->formLabel($element->getName(), $label);

        return $this->renderLabel($content, $label);
    }

    public function renderLabel($content, $label)
    {
        $placement = $this->getPlacement();
        $separator = $this->getSeparator();

        switch ($placement) {
            case 'APPEND':
                return $content . $separator . $label;
            case 'PREPEND':
            default:
                return $label . $separator . $content;
        }
    }
}
```

Now we can tell the element to use this plugin path when looking for decorators:

```
$element->addPrefixPath('My_Decorator', 'My/Decorator/', 'decorator');
```

Alternately, we can do that at the form level to ensure all decorators use this path:

```
$form->addElementPrefixPath('My_Decorator', 'My/Decorator/', 'decorator');
```

After it added as in the example above, the 'My/Decorator/' path will be searched first to see if the decorator exists there when you add a decorator. As a result, 'My_Decorator_Label' will now be used when the 'Label' decorator is requested.

3.2. Filters

It's often useful and/or necessary to perform some normalization on input prior to validation. For example, you may want to strip out all HTML, but run your validations on what remains to ensure the submission is valid. Or you may want to trim empty space surrounding input so that a StringLength validator will use the correct length of the input without counting leading or trailing whitespace characters. These operations may be performed using `Zend_Filter`. `Zend_Form_Element` has support for filter chains, allowing you to specify multiple, sequential filters. Filtering happens both during validation and when you retrieve the element value via `getValue()`:

```
$filtered = $element->getValue();
```

Filters may be added to the chain in two ways:

- passing in a concrete filter instance
- providing a filter name – either a short name or fully qualified class name

Let's see some examples:

```
// Concrete filter instance:
$element->addFilter(new Zend_Filter_Alnum());

// Fully qualified class name:
$element->addFilter('Zend_Filter_Alnum');

// Short filter name:
$element->addFilter('Alnum');
$element->addFilter('alnum');
```

Short names are typically the filter name minus the prefix. In the default case, this will mean minus the 'Zend_Filter_' prefix. The first letter can be upper-cased or lower-cased.

Using Custom Filter Classes

If you have your own set of filter classes, you can tell `Zend_Form_Element` about these using `addPrefixPath()`. For instance, if you have filters under the 'My_Filter' prefix, you can tell `Zend_Form_Element` about this as follows:

```
$element->addPrefixPath('My_Filter', 'My/Filter/', 'filter');
```

(Recall that the third argument indicates which plugin loader on which to perform the action.)

If at any time you need the unfiltered value, use the `getUnfilteredValue()` method:

```
$unfiltered = $element->getUnfilteredValue();
```

For more information on filters, see the Zend_Filter documentation.

Methods associated with filters include:

- `addFilter($nameOfFilter, array $options = null)`
- `addFilters(array $filters)`

- `setFilters(array $filters)` (overwrites all filters)
- `getFilter($name)` (retrieve a filter object by name)
- `getFilters()` (retrieve all filters)
- `removeFilter($name)` (remove filter by name)
- `clearFilters()` (remove all filters)

3.3. Validators

If you subscribe to the security mantra of "filter input, escape output," you'll should use validator to filter input submitted with your form. In `Zend_Form`, each element includes its own validator chain, consisting of `Zend_Validate_*` validators.

Validators may be added to the chain in two ways:

- passing in a concrete validator instance
- providing a validator name – either a short name or fully qualified class name

Let's see some examples:

```
// Concrete validator instance:
$element->addValidator(new Zend_Validate_Alnum());

// Fully qualified class name:
$element->addValidator('Zend_Validate_Alnum');

// Short validator name:
$element->addValidator('Alnum');
$element->addValidator('alnum');
```

Short names are typically the validator name minus the prefix. In the default case, this will mean minus the 'Zend_Validate_' prefix. As is the case with filters, the first letter can be upper-cased or lower-cased.

Using Custom Validator Classes

If you have your own set of validator classes, you can tell `Zend_Form_Element` about these using `addPrefixPath()`. For instance, if you have validators under the 'My_Validator' prefix, you can tell `Zend_Form_Element` about this as follows:

```
$element->addPrefixPath('My_Validator', 'My/Validator/', 'validate');
```

(Recall that the third argument indicates which plugin loader on which to perform the action.)

If failing a particular validation should prevent later validators from firing, pass boolean `TRUE` as the second parameter:

```
$element->addValidator('alnum', true);
```

If you are using a string name to add a validator, and the validator class accepts arguments to the constructor, you may pass these to the third parameter of `addValidator()` as an array:

```
$element->addValidator('StringLength', false, array(6, 20));
```

Arguments passed in this way should be in the order in which they are defined in the constructor. The above example will instantiate the `Zend_Validate_StringLenth` class with its `$min` and `$max` parameters:

```
$validator = new Zend_Validate_StringLength(6, 20);
```

Providing Custom Validator Error Messages

Some developers may wish to provide custom error messages for a validator. The `$options` argument of the `Zend_Form_Element::addValidator()` method allows you to do so by providing the key 'messages' and mapping it to an array of key/value pairs for setting the message templates. You will need to know the error codes of the various validation error types for the particular validator.

A better option is to use a `Zend_Translate_Adapter` with your form. Error codes are automatically passed to the adapter by the default Errors decorator; you can then specify your own error message strings by setting up translations for the various error codes of your validators.

You can also set many validators at once, using `addValidators()`. The basic usage is to pass an array of arrays, with each array containing 1 to 3 values, matching the constructor of `addValidator()`:

```
$element->addValidators(array(
    array('NotEmpty', true),
    array('alnum'),
    array('stringLength', false, array(6, 20)),
));
```

If you want to be more verbose or explicit, you can use the array keys 'validator', 'breakChainOnFailure', and 'options':

```
$element->addValidators(array(
    array(
        'validator'           => 'NotEmpty',
        'breakChainOnFailure' => true),
    array('validator' => 'alnum'),
    array(
        'validator' => 'stringLength',
        'options'   => array(6, 20)),
));
```

This usage is good for illustrating how you could then configure validators in a config file:

```
element.validators.notempty.validator = "NotEmpty"
element.validators.notempty.breakChainOnFailure = true
element.validators.alnum.validator = "Alnum"
element.validators.strlen.validator = "StringLength"
element.validators.strlen.options.min = 6
element.validators.strlen.options.max = 20
```

Notice that every item has a key, whether or not it needs one; this is a limitation of using configuration files -- but it also helps make explicit what the arguments are for. Just remember that any validator options must be specified in order.

To validate an element, pass the value to `isValid()`:

```
if ($element->isValid($value)) {
    // valid
} else {
    // invalid
}
```

Validation Operates On Filtered Values

`Zend_Form_Element::isValid()` filters values through the provided filter chain prior to validation. See the Filters section for more information.

Validation Context

`Zend_Form_Element::isValid()` supports an additional argument, `$context`. `Zend_Form::isValid()` passes the entire array of data being processed to `$context` when validating a form, and `Zend_Form_Element::isValid()`, in turn, passes it to each validator. This means you can write validators that are aware of data passed to other form elements. As an example, consider a standard registration form that has fields for both password and a password confirmation; one validation would be that the two fields match. Such a validator might look like the following:

```
class My_Validate_PasswordConfirmation extends Zend_Validate_Abstract
{
    const NOT_MATCH = 'notMatch';

    protected $_messageTemplates = array(
        self::NOT_MATCH => 'Password confirmation does not match'
    );

    public function isValid($value, $context = null)
    {
        $value = (string) $value;
        $this->_setValue($value);

        if (is_array($context)) {
            if (isset($context['password_confirm'])
                && ($value == $context['password_confirm']))
            {
                return true;
            }
        } elseif (is_string($context) && ($value == $context)) {
            return true;
        }

        $this->_error(self::NOT_MATCH);
        return false;
    }
}
```

Validators are processed in order. Each validator is processed, unless a validator created with a `TRUE` `breakChainOnFailure` value fails its validation. Be sure to specify your validators in a reasonable order.

After a failed validation, you can retrieve the error codes and messages from the validator chain:

```
$errors   = $element->getErrors();
$messages = $element->getMessages();
```

(Note: error messages returned are an associative array of error code / error message pairs.)

In addition to validators, you can specify that an element is required, using `setRequired(true)`. By default, this flag is FALSE, meaning that your validator chain will be skipped if no value is passed to `isValid()`. You can modify this behavior in a number of ways:

- By default, when an element is required, a flag, 'allowEmpty', is also TRUE. This means that if a value evaluating to empty is passed to `isValid()`, the validators will be skipped. You can toggle this flag using the accessor `setAllowEmpty($flag)`; when the flag is FALSE and a value is passed, the validators will still run.

- By default, if an element is required but does not contain a 'NotEmpty' validator, `isValid()` will add one to the top of the stack, with the `breakChainOnFailure` flag set. This behavior lends required flag semantic meaning: if no value is passed, we immediately invalidate the submission and notify the user, and prevent other validators from running on what we already know is invalid data.

  If you do not want this behavior, you can turn it off by passing a FALSE value to `setAutoInsertNotEmptyValidator($flag)`; this will prevent `isValid()` from placing the 'NotEmpty' validator in the validator chain.

For more information on validators, see the Zend_Validate documentation.

Using Zend_Form_Elements as general-purpose validators

`Zend_Form_Element` implements `Zend_Validate_Interface`, meaning an element may also be used as a validator in other, non-form related validation chains.

When is an element detected as empty?

As mentioned the 'NotEmpty' validator is used to detect if an element is empty or not. But `Zend_Validate_NotEmpty` does, per default, not work like PHP's method `empty()`.

This means when an element contains an integer *0* or an string *'0'* then the element will be seen as not empty. If you want to have a different behaviour you must create your own instance of `Zend_Validate_NotEmpty`. There you can define the behaviour of this validator. See Zend_Validate_NotEmpty for details.

Methods associated with validation include:

- `setRequired($flag)` and `isRequired()` allow you to set and retrieve the status of the 'required' flag. When set to boolean TRUE, this flag requires that the element be in the data processed by `Zend_Form`.

- `setAllowEmpty($flag)` and `getAllowEmpty()` allow you to modify the behaviour of optional elements (i.e., elements where the required flag is FALSE). When the 'allow empty' flag is TRUE, empty values will not be passed to the validator chain.

- `setAutoInsertNotEmptyValidator($flag)` allows you to specify whether or not a 'NotEmpty' validator will be prepended to the validator chain when the element is required. By default, this flag is `TRUE`.
- `addValidator($nameOrValidator, $breakChainOnFailure = false, array $options = null)`
- `addValidators(array $validators)`
- `setValidators(array $validators)` (overwrites all validators)
- `getValidator($name)` (retrieve a validator object by name)
- `getValidators()` (retrieve all validators)
- `removeValidator($name)` (remove validator by name)
- `clearValidators()` (remove all validators)

3.3.1. Custom Error Messages

At times, you may want to specify one or more specific error messages to use instead of the error messages generated by the validators attached to your element. Additionally, at times you may want to mark the element invalid yourself. As of 1.6.0, this functionality is possible via the following methods.

- `addErrorMessage($message)`: add an error message to display on form validation errors. You may call this more than once, and new messages are appended to the stack.
- `addErrorMessages(array $messages)`: add multiple error messages to display on form validation errors.
- `setErrorMessages(array $messages)`: add multiple error messages to display on form validation errors, overwriting all previously set error messages.
- `getErrorMessages()`: retrieve the list of custom error messages that have been defined.
- `clearErrorMessages()`: remove all custom error messages that have been defined.
- `markAsError()`: mark the element as having failed validation.
- `hasErrors()`: determine whether the element has either failed validation or been marked as invalid.
- `addError($message)`: add a message to the custom error messages stack and flag the element as invalid.
- `addErrors(array $messages)`: add several messages to the custom error messages stack and flag the element as invalid.
- `setErrors(array $messages)`: overwrite the custom error messages stack with the provided messages and flag the element as invalid.

All errors set in this fashion may be translated. Additionally, you may insert the placeholder "%value%" to represent the element value; this current element value will be substituted when the error messages are retrieved.

3.4. Decorators

One particular pain point for many web developers is the creation of the XHTML forms themselves. For each element, the developer needs to create markup for the element itself (typically a label) and special markup for displaying validation error messages. The more elements on the page, the less trivial this task becomes.

`Zend_Form_Element` tries to solve this issue through the use of "decorators". Decorators are simply classes that have access to the element and a method for rendering content. For more information on how decorators work, please see the section on Zend_Form_Decorator.

The default decorators used by `Zend_Form_Element` are:

- *ViewHelper*: specifies a view helper to use to render the element. The 'helper' element attribute can be used to specify which view helper to use. By default, `Zend_Form_Element` specifies the 'formText' view helper, but individual subclasses specify different helpers.
- *Errors*: appends error messages to the element using `Zend_View_Helper_FormErrors`. If none are present, nothing is appended.
- *Description*: appends the element description. If none is present, nothing is appended. By default, the description is rendered in a <p> tag with a class of 'description'.
- *HtmlTag*: wraps the element and errors in an HTML <dd> tag.
- *Label*: prepends a label to the element using `Zend_View_Helper_FormLabel`, and wraps it in a <dt> tag. If no label is provided, just the definition term tag is rendered.

Default Decorators Do Not Need to Be Loaded

By default, the default decorators are loaded during object initialization. You can disable this by passing the 'disableLoadDefaultDecorators' option to the constructor:

```
$element = new Zend_Form_Element('foo',
                                 array('disableLoadDefaultDecorators' =>
                                      true)
                                );
```

This option may be mixed with any other options you pass, both as array options or in a `Zend_Config` object.

Since the order in which decorators are registered matters- the first decorator registered is executed first- you will need to make sure you register your decorators in an appropriate order, or ensure that you set the placement options in a sane fashion. To give an example, here is the code that registers the default decorators:

```
$this->addDecorators(array(
    array('ViewHelper'),
    array('Errors'),
    array('Description', array('tag' => 'p', 'class' => 'description')),
    array('HtmlTag', array('tag' => 'dd')),
    array('Label', array('tag' => 'dt')),
));
```

The initial content is created by the 'ViewHelper' decorator, which creates the form element itself. Next, the 'Errors' decorator fetches error messages from the element, and, if any are present,

passes them to the 'FormErrors' view helper to render. If a description is present, the 'Description' decorator will append a paragraph of class 'description' containing the descriptive text to the aggregated content. The next decorator, 'HtmlTag', wraps the element, errors, and description in an HTML <dd> tag. Finally, the last decorator, 'label', retrieves the element's label and passes it to the 'FormLabel' view helper, wrapping it in an HTML <dt> tag; the value is prepended to the content by default. The resulting output looks basically like this:

```
<dt><label for="foo" class="optional">Foo</label></dt>
<dd>
    <input type="text" name="foo" id="foo" value="123" />
    <ul class="errors">
        <li>"123" is not an alphanumeric value</li>
    </ul>
    <p class="description">
        This is some descriptive text regarding the element.
    </p>
</dd>
```

For more information on decorators, read the Zend_Form_Decorator section.

Using Multiple Decorators of the Same Type

Internally, `Zend_Form_Element` uses a decorator's class as the lookup mechanism when retrieving decorators. As a result, you cannot register multiple decorators of the same type; subsequent decorators will simply overwrite those that existed before.

To get around this, you can use *aliases*. Instead of passing a decorator or decorator name as the first argument to `addDecorator()`, pass an array with a single element, with the alias pointing to the decorator object or name:

```
// Alias to 'FooBar':
$element->addDecorator(array('FooBar' => 'HtmlTag'),
                       array('tag' => 'div'));

// And retrieve later:
$decorator = $element->getDecorator('FooBar');
```

In the `addDecorators()` and `setDecorators()` methods, you will need to pass the 'decorator' option in the array representing the decorator:

```
// Add two 'HtmlTag' decorators, aliasing one to 'FooBar':
$element->addDecorators(
    array('HtmlTag', array('tag' => 'div')),
    array(
        'decorator' => array('FooBar' => 'HtmlTag'),
        'options' => array('tag' => 'dd')
    ),
);

// And retrieve later:
$htmlTag = $element->getDecorator('HtmlTag');
$fooBar  = $element->getDecorator('FooBar');
```

Methods associated with decorators include:

- `addDecorator($nameOrDecorator, array $options = null)`
- `addDecorators(array $decorators)`
- `setDecorators(array $decorators)` (overwrites all decorators)
- `getDecorator($name)` (retrieve a decorator object by name)
- `getDecorators()` (retrieve all decorators)
- `removeDecorator($name)` (remove decorator by name)
- `clearDecorators()` (remove all decorators)

`Zend_Form_Element` also uses overloading to allow rendering specific decorators. `__call()` will intercept methods that lead with the text 'render' and use the remainder of the method name to lookup a decorator; if found, it will then render that *single* decorator. Any arguments passed to the method call will be used as content to pass to the decorator's `render()` method. As an example:

```
// Render only the ViewHelper decorator:
echo $element->renderViewHelper();

// Render only the HtmlTag decorator, passing in content:
echo $element->renderHtmlTag("This is the html tag content");
```

If the decorator does not exist, an exception is raised.

3.5. Metadata and Attributes

`Zend_Form_Element` handles a variety of attributes and element metadata. Basic attributes include:

- *name*: the element name. Uses the `setName()` and `getName()` accessors.
- *label*: the element label. Uses the `setLabel()` and `getLabel()` accessors.
- *order*: the index at which an element should appear in the form. Uses the `setOrder()` and `getOrder()` accessors.
- *value*: the current element value. Uses the `setValue()` and `getValue()` accessors.
- *description*: a description of the element; often used to provide tooltip or javascript contextual hinting describing the purpose of the element. Uses the `setDescription()` and `getDescription()` accessors.
- *required*: flag indicating whether or not the element is required when performing form validation. Uses the `setRequired()` and `getRequired()` accessors. This flag is `FALSE` by default.
- *allowEmpty*: flag indicating whether or not a non-required (optional) element should attempt to validate empty values. If it is set to `TRUE` and the required flag is `FALSE`, empty values are not passed to the validator chain and are presumed `TRUE`. Uses the `setAllowEmpty()` and `getAllowEmpty()` accessors. This flag is `TRUE` by default.
- *autoInsertNotEmptyValidator*: flag indicating whether or not to insert a 'NotEmpty' validator when the element is required. By default, this flag is `TRUE`. Set the flag with `setAutoInsertNotEmptyValidator($flag)` and determine the value with `autoInsertNotEmptyValidator()`.

Form elements may require additional metadata. For XHTML form elements, for instance, you may want to specify attributes such as the class or id. To facilitate this are a set of accessors:

- *setAttrib($name, $value)*: add an attribute
- *setAttribs(array $attribs)*: like addAttribs(), but overwrites
- *getAttrib($name)*: retrieve a single attribute value
- *getAttribs()*: retrieve all attributes as key/value pairs

Most of the time, however, you can simply access them as object properties, as `Zend_Form_Element` utilizes overloading to facilitate access to them:

```
// Equivalent to $element->setAttrib('class', 'text'):
$element->class = 'text;
```

By default, all attributes are passed to the view helper used by the element during rendering, and rendered as HTML attributes of the element tag.

3.6. Standard Elements

`Zend_Form` ships with a number of standard elements; please read the Standard Elements chapter for full details.

3.7. Zend_Form_Element Methods

`Zend_Form_Element` has many, many methods. What follows is a quick summary of their signatures, grouped by type:

- Configuration:
  - `setOptions(array $options)`
  - `setConfig(Zend_Config $config)`
- I18n:
  - `setTranslator(Zend_Translate_Adapter $translator = null)`
  - `getTranslator()`
  - `setDisableTranslator($flag)`
  - `translatorIsDisabled()`
- Properties:
  - `setName($name)`
  - `getName()`
  - `setValue($value)`
  - `getValue()`
  - `getUnfilteredValue()`

    - `setLabel($label)`
    - `getLabel()`
    - `setDescription($description)`
    - `getDescription()`
    - `setOrder($order)`
    - `getOrder()`
    - `setRequired($flag)`
    - `getRequired()`
    - `setAllowEmpty($flag)`
    - `getAllowEmpty()`
    - `setAutoInsertNotEmptyValidator($flag)`
    - `autoInsertNotEmptyValidator()`
    - `setIgnore($flag)`
    - `getIgnore()`
    - `getType()`
    - `setAttrib($name, $value)`
    - `setAttribs(array $attribs)`
    - `getAttrib($name)`
    - `getAttribs()`
- Plugin loaders and paths:
    - `setPluginLoader(Zend_Loader_PluginLoader_Interface $loader, $type)`
    - `getPluginLoader($type)`
    - `addPrefixPath($prefix, $path, $type = null)`
    - `addPrefixPaths(array $spec)`
- Validation:
    - `addValidator($validator, $breakChainOnFailure = false, $options = array())`
    - `addValidators(array $validators)`
    - `setValidators(array $validators)`
    - `getValidator($name)`

    - `getValidators()`
    - `removeValidator($name)`
    - `clearValidators()`
    - `isValid($value, $context = null)`
    - `getErrors()`
    - `getMessages()`
- Filters:
    - `addFilter($filter, $options = array())`
    - `addFilters(array $filters)`
    - `setFilters(array $filters)`
    - `getFilter($name)`
    - `getFilters()`
    - `removeFilter($name)`
    - `clearFilters()`
- Rendering:
    - `setView(Zend_View_Interface $view = null)`
    - `getView()`
    - `addDecorator($decorator, $options = null)`
    - `addDecorators(array $decorators)`
    - `setDecorators(array $decorators)`
    - `getDecorator($name)`
    - `getDecorators()`
    - `removeDecorator($name)`
    - `clearDecorators()`
    - `render(Zend_View_Interface $view = null)`

3.8. Configuration

`Zend_Form_Element`'s constructor accepts either an array of options or a `Zend_Config` object containing options, and it can also be configured using either `setOptions()` or `setConfig()`. Generally speaking, keys are named as follows:

- If 'set' + key refers to a `Zend_Form_Element` method, then the value provided will be passed to that method.

- Otherwise, the value will be used to set an attribute.

Exceptions to the rule include the following:

- `prefixPath` will be passed to `addPrefixPaths()`
- The following setters cannot be set in this way:
  - `setAttrib` (though `setAttribs` *will* work)
  - `setConfig`
  - `setOptions`
  - `setPluginLoader`
  - `setTranslator`
  - `setView`

As an example, here is a config file that passes configuration for every type of configurable data:

```
[element]
name = "foo"
value = "foobar"
label = "Foo:"
order = 10
required = true
allowEmpty = false
autoInsertNotEmptyValidator = true
description = "Foo elements are for examples"
ignore = false
attribs.id = "foo"
attribs.class = "element"
; sets 'onclick' attribute
onclick = "autoComplete(this, '/form/autocomplete/element')"
prefixPaths.decorator.prefix = "My_Decorator"
prefixPaths.decorator.path = "My/Decorator/"
disableTranslator = 0
validators.required.validator = "NotEmpty"
validators.required.breakChainOnFailure = true
validators.alpha.validator = "alpha"
validators.regex.validator = "regex"
validators.regex.options.pattern = "/^[A-F].*/$"
filters.ucase.filter = "StringToUpper"
decorators.element.decorator = "ViewHelper"
decorators.element.options.helper = "FormText"
decorators.label.decorator = "Label"
```

3.9. Custom Elements

You can create your own custom elements by simply extending the `Zend_Form_Element` class. Common reasons to do so include:

- Elements that share common validators and/or filters
- Elements that have custom decorator functionality

There are two methods typically used to extend an element: `init()`, which can be used to add custom initialization logic to your element, and `loadDefaultDecorators()`, which can be used to set a list of default decorators used by your element.

As an example, let's say that all text elements in a form you are creating need to be filtered with `StringTrim`, validated with a common regular expression, and that you want to use a custom decorator you've created for displaying them, 'My_Decorator_TextItem'. In addition, you have a number of standard attributes, including 'size', 'maxLength', and 'class' you wish to specify. You could define an element to accomplish this as follows:

```
class My_Element_Text extends Zend_Form_Element
{
    public function init()
    {
        $this->addPrefixPath('My_Decorator', 'My/Decorator/', 'decorator')
             ->addFilters('StringTrim')
             ->addValidator('Regex', false, array('/^[a-z0-9]{6,}$/i'))
             ->addDecorator('TextItem')
             ->setAttrib('size', 30)
             ->setAttrib('maxLength', 45)
             ->setAttrib('class', 'text');
    }
}
```

You could then inform your form object about the prefix path for such elements, and start creating elements:

```
$form->addPrefixPath('My_Element', 'My/Element/', 'element')
     ->addElement('text', 'foo');
```

The 'foo' element will now be of type `My_Element_Text`, and exhibit the behaviour you've outlined.

Another method you may want to override when extending `Zend_Form_Element` is the `loadDefaultDecorators()` method. This method conditionally loads a set of default decorators for your element; you may wish to substitute your own decorators in your extending class:

```
class My_Element_Text extends Zend_Form_Element
{
    public function loadDefaultDecorators()
    {
        $this->addDecorator('ViewHelper')
             ->addDecorator('DisplayError')
             ->addDecorator('Label')
             ->addDecorator('HtmlTag',
                            array('tag' => 'div', 'class' => 'element'));
    }
}
```

There are many ways to customize elements. Read the API documentation of `Zend_Form_Element` to learn about all of the available methods.

4. Creating Forms Using Zend_Form

The `Zend_Form` class is used to aggregate form elements, display groups, and subforms. It can then perform the following actions on those items:

- Validation, including retrieving error codes and messages
- Value aggregation, including populating items and retrieving both filtered and unfiltered values from all items
- Iteration over all items, in the order in which they are entered or based on the order hints retrieved from each item
- Rendering of the entire form, either via a single decorator that performs custom rendering or by iterating over each item in the form

While forms created with `Zend_Form` may be complex, probably the best use case is for simple forms; its best use is for Rapid Application Development (RAD) and prototyping.

At its most basic, you simply instantiate a form object:

```
// Generic form object:
$form = new Zend_Form();

// Custom form object:
$form = new My_Form()
```

You can optionally pass in a instance of `Zend_Config` or an array, which will be used to set object state and potentially create new elements:

```
// Passing in configuration options:
$form = new Zend_Form($config);
```

`Zend_Form` is iterable, and will iterate over elements, display groups, and subforms, using the order they were registered and any order index each may have. This is useful in cases where you wish to render the elements manually in the appropriate order.

`Zend_Form`'s magic lies in its ability to serve as a factory for elements and display groups, as well as the ability to render itself through decorators.

4.1. Plugin Loaders

`Zend_Form` makes use of `Zend_Loader_PluginLoader` to allow developers to specify the locations of alternate elements and decorators. Each has its own plugin loader associated with it, and general accessors are used to retrieve and modify each.

The following loader types are used with the various plugin loader methods: 'element' and 'decorator'. The type names are case insensitive.

The methods used to interact with plugin loaders are as follows:

- `setPluginLoader($loader, $type)`: $loader is the plugin loader object itself, while type is one of the types specified above. This sets the plugin loader for the given type to the newly specified loader object.
- `getPluginLoader($type)`: retrieves the plugin loader associated with $type.
- `addPrefixPath($prefix, $path, $type = null)`: adds a prefix/path association to the loader specified by $type. If $type is `NULL`, it will attempt to add the path to all loaders, by appending the prefix with each of "_Element" and "_Decorator"; and appending the path with

"Element/" and "Decorator/". If you have all your extra form element classes under a common hierarchy, this is a convenience method for setting the base prefix for them.

- `addPrefixPaths(array $spec)`: allows you to add many paths at once to one or more plugin loaders. It expects each array item to be an array with the keys 'path', 'prefix', and 'type'.

Additionally, you can specify prefix paths for all elements and display groups created through a `Zend_Form` instance using the following methods:

- `addElementPrefixPath($prefix, $path, $type = null)`: Just like `addPrefixPath()`, you must specify a class prefix and a path. `$type`, when specified, must be one of the plugin loader types specified by `Zend_Form_Element`; see the element plugins section for more information on valid `$type` values. If no `$type` is specified, the method will assume it is a general prefix for all types.

- `addDisplayGroupPrefixPath($prefix, $path)`: Just like `addPrefixPath()`, you must specify a class prefix and a path; however, since display groups only support decorators as plugins, no `$type` is necessary.

Custom elements and decorators are an easy way to share functionality between forms and encapsulate custom functionality. See the Custom Label example in the elements documentation for an example of how custom elements can be used as replacements for standard classes.

4.2. Elements

`Zend_Form` provides several accessors for adding and removing form elements from a form. These can take element object instances or serve as factories for instantiating the element objects themselves.

The most basic method for adding an element is `addElement()`. This method can take either an object of type `Zend_Form_Element` (or of a class extending `Zend_Form_Element`), or arguments for building a new element -- including the element type, name, and any configuration options.

Some examples:

```
// Using an element instance:
$element = new Zend_Form_Element_Text('foo');
$form->addElement($element);

// Using a factory
//
// Creates an element of type Zend_Form_Element_Text with the
// name of 'foo':
$form->addElement('text', 'foo');

// Pass label option to the element:
$form->addElement('text', 'foo', array('label' => 'Foo:'));
```

addElement() Implements Fluent Interface

`addElement()` implements a fluent interface; that is to say, it returns the `Zend_Form` object, and not the element. This is done to allow you to chain together multiple addElement() methods or other form methods that implement the fluent interface (all setters in `Zend_Form` implement the pattern).

If you wish to return the element instead, use `createElement()`, which is outlined below. Be aware, however, that `createElement()` does not attach the element to the form.

Internally, `addElement()` actually uses `createElement()` to create the element before attaching it to the form.

Once an element has been added to the form, you can retrieve it by name. This can be done either by using the `getElement()` method or by using overloading to access the element as an object property:

```
// getElement():
$foo = $form->getElement('foo');

// As object property:
$foo = $form->foo;
```

Occasionally, you may want to create an element without attaching it to the form (for instance, if you wish to make use of the various plugin paths registered with the form, but wish to later attach the object to a sub form). The `createElement()` method allows you to do so:

```
// $username becomes a Zend_Form_Element_Text object:
$username = $form->createElement('text', 'username');
```

4.2.1. Populating and Retrieving Values

After validating a form, you will typically need to retrieve the values so you can perform other operations, such as updating a database or notifying a web service. You can retrieve all values for all elements using `getValues()`; `getValue($name)` allows you to retrieve a single element's value by element name:

```
// Get all values:
$values = $form->getValues();

// Get only 'foo' element's value:
$value = $form->getValue('foo');
```

Sometimes you'll want to populate the form with specified values prior to rendering. This can be done with either the `setDefaults()` or `populate()` methods:

```
$form->setDefaults($data);
$form->populate($data);
```

On the flip side, you may want to clear a form after populating or validating it; this can be done using the `reset()` method:

```
$form->reset();
```

4.2.2. Global Operations

Occasionally you will want certain operations to affect all elements. Common scenarios include needing to set plugin prefix paths for all elements, setting decorators for all elements, and setting filters for all elements. As examples:

Example 383. Setting Prefix Paths for All Elements

You can set prefix paths for all elements by type, or using a global prefix. Some examples:

```
// Set global prefix path:
// Creates paths for prefixes My_Foo_Filter, My_Foo_Validate,
// and My_Foo_Decorator
$form->addElementPrefixPath('My_Foo', 'My/Foo/');

// Just filter paths:
$form->addElementPrefixPath('My_Foo_Filter',
                            'My/Foo/Filter',
                            'filter');

// Just validator paths:
$form->addElementPrefixPath('My_Foo_Validate',
                            'My/Foo/Validate',
                            'validate');

// Just decorator paths:
$form->addElementPrefixPath('My_Foo_Decorator',
                            'My/Foo/Decorator',
                            'decorator');
```

Example 384. Setting Decorators for All Elements

You can set decorators for all elements. `setElementDecorators()` accepts an array of decorators, just like `setDecorators()`, and will overwrite any previously set decorators in each element. In this example, we set the decorators to simply a ViewHelper and a Label:

```
$form->setElementDecorators(array(
    'ViewHelper',
    'Label'
));
```

Example 385. Setting Decorators for Some Elements

You can also set decorators for a subset of elements, either by inclusion or exclusion. The second argument to `setElementDecorators()` may be an array of element names; by default, specifying such an array will set the specified decorators on those elements only. You may also pass a third argument, a flag indicating whether this list of elements is for inclusion or exclusion purposes. If the flag is FALSE, it will decorate all elements *except* those in the passed list. As with standard usage of the method, any decorators passed will overwrite any previously set decorators in each element.

In the following snippet, we indicate that we want only the ViewHelper and Label decorators for the 'foo' and 'bar' elements:

```
$form->setElementDecorators(
    array(
        'ViewHelper',
        'Label'
    ),
    array(
        'foo',
        'bar'
    )
);
```

On the flip side, with this snippet, we'll now indicate that we want to use only the ViewHelper and Label decorators for every element *except* the 'foo' and 'bar' elements:

```
$form->setElementDecorators(
    array(
        'ViewHelper',
        'Label'
    ),
    array(
        'foo',
        'bar'
    ),
    false
);
```

Some Decorators are Inappropriate for Some Elements

While `setElementDecorators()` may seem like a good solution, there are some cases where it may actually end up with unexpected results. For example, the various button elements (Submit, Button, Reset) currently use the label as the value of the button, and only use ViewHelper and DtDdWrapper decorators -- preventing an additional labels, errors, and hints from being rendered. The example above would duplicate some content (the label) for button elements.

You can use the inclusion/exclusion array to overcome this issue as noted in the previous example.

So, use this method wisely, and realize that you may need to exclude some elements or manually change some elements' decorators to prevent unwanted output.

Example 386. Setting Filters for All Elements

In some cases, you may want to apply the same filter to all elements; a common case is to `trim()` all values:

```
$form->setElementFilters(array('StringTrim'));
```

4.2.3. Methods For Interacting With Elements

The following methods may be used to interact with elements:

- `createElement($element, $name = null, $options = null)`
- `addElement($element, $name = null, $options = null)`
- `addElements(array $elements)`
- `setElements(array $elements)`
- `getElement($name)`
- `getElements()`
- `removeElement($name)`
- `clearElements()`
- `setDefaults(array $defaults)`
- `setDefault($name, $value)`
- `getValue($name)`
- `getValues()`
- `getUnfilteredValue($name)`
- `getUnfilteredValues()`
- `setElementFilters(array $filters)`
- `setElementDecorators(array $decorators)`
- `addElementPrefixPath($prefix, $path, $type = null)`
- `addElementPrefixPaths(array $spec)`

4.3. Display Groups

Display groups are a way to create virtual groupings of elements for display purposes. All elements remain accessible by name in the form, but when iterating over the form or rendering, any elements in a display group are rendered together. The most common use case for this is for grouping elements in fieldsets.

The base class for display groups is `Zend_Form_DisplayGroup`. While it can be instantiated directly, it is usually best to use `Zend_Form`'s `addDisplayGroup()` method to do so. This method takes an array of elements as its first argument, and a name for the display group as

its second argument. You may optionally pass in an array of options or a `Zend_Config` object as the third argument.

Assuming that the elements 'username' and 'password' are already set in the form, the following code would group these elements in a 'login' display group:

```
$form->addDisplayGroup(array('username', 'password'), 'login');
```

You can access display groups using the `getDisplayGroup()` method, or via overloading using the display group's name:

```
// Using getDisplayGroup():
$login = $form->getDisplayGroup('login');

// Using overloading:
$login = $form->login;
```

Default Decorators Do Not Need to Be Loaded

By default, the default decorators are loaded during object initialization. You can disable this by passing the 'disableLoadDefaultDecorators' option when creating a display group:

```
$form->addDisplayGroup(
    array('foo', 'bar'),
    'foobar',
    array('disableLoadDefaultDecorators' => true)
);
```

This option may be mixed with any other options you pass, both as array options or in a `Zend_Config` object.

4.3.1. Global Operations

Just as with elements, there are some operations which might affect all display groups; these include setting decorators and setting the plugin path in which to look for decorators.

Example 387. Setting Decorator Prefix Path for All Display Groups

By default, display groups inherit whichever decorator paths the form uses; however, if they should look in alternate locations, you can use the `addDisplayGroupPrefixPath()` method.

```
$form->addDisplayGroupPrefixPath('My_Foo_Decorator', 'My/Foo/Decorator');
```

Example 388. Setting Decorators for All Display Groups

You can set decorators for all display groups. `setDisplayGroupDecorators()` accepts an array of decorators, just like `setDecorators()`, and will overwrite any previously set decorators in each display group. In this example, we set the decorators to simply a fieldset (the FormElements decorator is necessary to ensure that the elements are iterated):

```
$form->setDisplayGroupDecorators(array(
    'FormElements',
    'Fieldset'
));
```

4.3.2. Using Custom Display Group Classes

By default, `Zend_Form` uses the `Zend_Form_DisplayGroup` class for display groups. You may find you need to extend this class in order to provided custom functionality. `addDisplayGroup()` does not allow passing in a concrete instance, but does allow specifying the class to use as one of its options, using the 'displayGroupClass' key:

```
// Use the 'My_DisplayGroup' class
$form->addDisplayGroup(
    array('username', 'password'),
    'user',
    array('displayGroupClass' => 'My_DisplayGroup')
);
```

If the class has not yet been loaded, `Zend_Form` will attempt to do so using `Zend_Loader`.

You can also specify a default display group class to use with the form such that all display groups created with the form object will use that class:

```
// Use the 'My_DisplayGroup' class for all display groups:
$form->setDefaultDisplayGroupClass('My_DisplayGroup');
```

This setting may be specified in configurations as 'defaultDisplayGroupClass', and will be loaded early to ensure all display groups use that class.

4.3.3. Methods for Interacting With Display Groups

The following methods may be used to interact with display groups:

- `addDisplayGroup(array $elements, $name, $options = null)`
- `addDisplayGroups(array $groups)`
- `setDisplayGroups(array $groups)`
- `getDisplayGroup($name)`
- `getDisplayGroups()`
- `removeDisplayGroup($name)`
- `clearDisplayGroups()`
- `setDisplayGroupDecorators(array $decorators)`
- `addDisplayGroupPrefixPath($prefix, $path)`
- `setDefaultDisplayGroupClass($class)`
- `getDefaultDisplayGroupClass($class)`

4.3.4. Zend_Form_DisplayGroup Methods

`Zend_Form_DisplayGroup` has the following methods, grouped by type:

- Configuration:
  - `setOptions(array $options)`
  - `setConfig(Zend_Config $config)`

- Metadata:
  - `setAttrib($key, $value)`
  - `addAttribs(array $attribs)`
  - `setAttribs(array $attribs)`
  - `getAttrib($key)`
  - `getAttribs()`
  - `removeAttrib($key)`
  - `clearAttribs()`
  - `setName($name)`
  - `getName()`
  - `setDescription($value)`
  - `getDescription()`
  - `setLegend($legend)`
  - `getLegend()`
  - `setOrder($order)`
  - `getOrder()`
- Elements:
  - `createElement($type, $name, array $options = array())`
  - `addElement($typeOrElement, $name, array $options = array())`
  - `addElements(array $elements)`
  - `setElements(array $elements)`
  - `getElement($name)`
  - `getElements()`
  - `removeElement($name)`
  - `clearElements()`
- Plugin loaders:
  - `setPluginLoader(Zend_Loader_PluginLoader $loader)`
  - `getPluginLoader()`
  - `addPrefixPath($prefix, $path)`
  - `addPrefixPaths(array $spec)`

- Decorators:
  - `addDecorator($decorator, $options = null)`
  - `addDecorators(array $decorators)`
  - `setDecorators(array $decorators)`
  - `getDecorator($name)`
  - `getDecorators()`
  - `removeDecorator($name)`
  - `clearDecorators()`
- Rendering:
  - `setView(Zend_View_Interface $view = null)`
  - `getView()`
  - `render(Zend_View_Interface $view = null)`
- I18n:
  - `setTranslator(Zend_Translate_Adapter $translator = null)`
  - `getTranslator()`
  - `setDisableTranslator($flag)`
  - `translatorIsDisabled()`

4.4. Sub Forms

Sub forms serve several purposes:

- Creating logical element groups. Since sub forms are simply forms, you can validate subforms as individual entities.
- Creating multi-page forms. Since sub forms are simply forms, you can display a separate sub form per page, building up multi-page forms where each form has its own validation logic. Only once all sub forms validate would the form be considered complete.
- Display groupings. Like display groups, sub forms, when rendered as part of a larger form, can be used to group elements. Be aware, however, that the master form object will have no awareness of the elements in sub forms.

A sub form may be a `Zend_Form` object, or, more typically, a `Zend_Form_SubForm` object. The latter contains decorators suitable for inclusion in a larger form (i.e., it does not render additional HTML form tags, but does group elements). To attach a sub form, simply add it to the form and give it a name:

```
$form->addSubForm($subForm, 'subform');
```

You can retrieve a sub form using either `getSubForm($name)` or overloading using the sub form name:

```
// Using getSubForm():
$subForm = $form->getSubForm('subform');

// Using overloading:
$subForm = $form->subform;
```

Sub forms are included in form iteration, although the elements they contain are not.

4.4.1. Global Operations

Like elements and display groups, there are some operations that might need to affect all sub forms. Unlike display groups and elements, however, sub forms inherit most functionality from the master form object, and the only real operation that may need to be performed globally is setting decorators for sub forms. For this purpose, there is the `setSubFormDecorators()` method. In the next example, we'll set the decorator for all subforms to be simply a fieldset (the FormElements decorator is needed to ensure its elements are iterated):

```
$form->setSubFormDecorators(array(
    'FormElements',
    'Fieldset'
));
```

4.4.2. Methods for Interacting With Sub Forms

The following methods may be used to interact with sub forms:

- `addSubForm(Zend_Form $form, $name, $order = null)`
- `addSubForms(array $subForms)`
- `setSubForms(array $subForms)`
- `getSubForm($name)`
- `getSubForms()`
- `removeSubForm($name)`
- `clearSubForms()`
- `setSubFormDecorators(array $decorators)`

4.5. Metadata and Attributes

While a form's usefulness primarily derives from the elements it contains, it can also contain other metadata, such as a name (often used as a unique ID in the HTML markup); the form action and method; the number of elements, groups, and sub forms it contains; and arbitrary metadata (usually used to set HTML attributes for the form tag itself).

You can set and retrieve a form's name using the name accessors:

```
// Set the name:
$form->setName('registration');

// Retrieve the name:
$name = $form->getName();
```

To set the action (url to which the form submits) and method (method by which it should submit, e.g., 'POST' or 'GET'), use the action and method accessors:

```
// Set the action and method:
$form->setAction('/user/login')
     ->setMethod('post');
```

You may also specify the form encoding type specifically using the enctype accessors. `Zend_Form` defines two constants, `Zend_Form::ENCTYPE_URLENCODED` and `Zend_Form::ENCTYPE_MULTIPART`, corresponding to the values 'application/x-www-form-urlencoded' and 'multipart/form-data', respectively; however, you can set this to any arbitrary encoding type.

```
// Set the action, method, and enctype:
$form->setAction('/user/login')
     ->setMethod('post')
     ->setEnctype(Zend_Form::ENCTYPE_MULTIPART);
```

The method, action, and enctype are only used internally for rendering, and not for any sort of validation.

`Zend_Form` implements the `Countable` interface, allowing you to pass it as an argument to count:

```
$numItems = count($form);
```

Setting arbitrary metadata is done through the attribs accessors. Since overloading in `Zend_Form` is used to access elements, display groups, and sub forms, this is the only method for accessing metadata.

```
// Setting attributes:
$form->setAttrib('class', 'zend-form')
     ->addAttribs(array(
         'id'       => 'registration',
         'onSubmit' => 'validate(this)',
     ));

// Retrieving attributes:
$class = $form->getAttrib('class');
$attribs = $form->getAttribs();

// Remove an attribute:
$form->removeAttrib('onSubmit');

// Clear all attributes:
$form->clearAttribs();
```

4.6. Decorators

Creating the markup for a form is often a time-consuming task, particularly if you plan on re-using the same markup to show things such as validation errors, submitted values, etc. `Zend_Form`'s answer to this issue is *decorators*.

Decorators for `Zend_Form` objects can be used to render a form. The FormElements decorator will iterate through all items in a form -- elements, display groups, and sub forms -- and render

them, returning the result. Additional decorators may then be used to wrap this content, or append or prepend it.

The default decorators for `Zend_Form` are FormElements, HtmlTag (wraps in a definition list), and Form; the equivalent code for creating them is as follows:

```
$form->setDecorators(array(
    'FormElements',
    array('HtmlTag', array('tag' => 'dl')),
    'Form'
));
```

This creates output like the following:

```
<form action="/form/action" method="post">
<dl>
...
</dl>
</form>
```

Any attributes set on the form object will be used as HTML attributes of the `<form>` tag.

Default Decorators Do Not Need to Be Loaded

By default, the default decorators are loaded during object initialization. You can disable this by passing the 'disableLoadDefaultDecorators' option to the constructor:

```
$form = new Zend_Form(array('disableLoadDefaultDecorators' => true));
```

This option may be mixed with any other options you pass, both as array options or in a `Zend_Config` object.

Using Multiple Decorators of the Same Type

Internally, `Zend_Form` uses a decorator's class as the lookup mechanism when retrieving decorators. As a result, you cannot register multiple decorators of the same type; subsequent decorators will simply overwrite those that existed before.

To get around this, you can use aliases. Instead of passing a decorator or decorator name as the first argument to `addDecorator()`, pass an array with a single element, with the alias pointing to the decorator object or name:

```
// Alias to 'FooBar':
$form->addDecorator(array('FooBar' => 'HtmlTag'), array('tag' => 'div'));

// And retrieve later:
$form = $element->getDecorator('FooBar');
```

In the `addDecorators()` and `setDecorators()` methods, you will need to pass the 'decorator' option in the array representing the decorator:

```
// Add two 'HtmlTag' decorators, aliasing one to 'FooBar':
$form->addDecorators(
    array('HtmlTag', array('tag' => 'div')),
    array(
```

```
        'decorator' => array('FooBar' => 'HtmlTag'),
        'options' => array('tag' => 'dd')
    ),
);

// And retrieve later:
$htmlTag = $form->getDecorator('HtmlTag');
$fooBar  = $form->getDecorator('FooBar');
```

You may create your own decorators for generating the form. One common use case is if you know the exact HTML you wish to use; your decorator could create the exact HTML and simply return it, potentially using the decorators from individual elements or display groups.

The following methods may be used to interact with decorators:

- `addDecorator($decorator, $options = null)`
- `addDecorators(array $decorators)`
- `setDecorators(array $decorators)`
- `getDecorator($name)`
- `getDecorators()`
- `removeDecorator($name)`
- `clearDecorators()`

`Zend_Form` also uses overloading to allow rendering specific decorators. `__call()` will intercept methods that lead with the text 'render' and use the remainder of the method name to lookup a decorator; if found, it will then render that *single* decorator. Any arguments passed to the method call will be used as content to pass to the decorator's `render()` method. As an example:

```
// Render only the FormElements decorator:
echo $form->renderFormElements();

// Render only the fieldset decorator, passing in content:
echo $form->renderFieldset("<p>This is fieldset content</p>");
```

If the decorator does not exist, an exception is raised.

4.7. Validation

A primary use case for forms is validating submitted data. `Zend_Form` allows you to validate an entire form, a partial form, or responses for XmlHttpRequests (AJAX). If the submitted data is not valid, it has methods for retrieving the various error codes and messages for elements and sub forms.

To validate a full form, use the `isValid()` method:

```
if (!$form->isValid($_POST)) {
    // failed validation
}
```

`isValid()` will validate every required element, and any unrequired element contained in the submitted data.

Sometimes you may need to validate only a subset of the data; for this, use `isValidPartial($data)`:

```
if (!$form->isValidPartial($data)) {
    // failed validation
}
```

`isValidPartial()` only attempts to validate those items in the data for which there are matching elements; if an element is not represented in the data, it is skipped.

When validating elements or groups of elements for an AJAX request, you will typically be validating a subset of the form, and want the response back in JSON. `processAjax()` does precisely that:

```
$json = $form->processAjax($data);
```

You can then simply send the JSON response to the client. If the form is valid, this will be a boolean TRUE response. If not, it will be a javascript object containing key/message pairs, where each 'message' is an array of validation error messages.

For forms that fail validation, you can retrieve both error codes and error messages, using `getErrors()` and `getMessages()`, respectively:

```
$codes = $form->getErrors();
$messages = $form->getMessages();
```

Since the messages returned by `getMessages()` are an array of error code/message pairs, `getErrors()` is typically not needed.

You can retrieve codes and error messages for individual elements by simply passing the element name to each:

```
$codes = $form->getErrors('username');
$messages = $form->getMessages('username');
```

Note: When validating elements, `Zend_Form` sends a second argument to each element's `isValid()` method: the array of data being validated. This can then be used by individual validators to allow them to utilize other submitted values when determining the validity of the data. An example would be a registration form that requires both a password and password confirmation; the password element could use the password confirmation as part of its validation.

4.7.1. Custom Error Messages

At times, you may want to specify one or more specific error messages to use instead of the error messages generated by the validators attached to your elements. Additionally, at times you may want to mark the form invalid yourself. This functionality is possible via the following methods.

- `addErrorMessage($message)`: add an error message to display on form validation errors. You may call this more than once, and new messages are appended to the stack.
- `addErrorMessages(array $messages)`: add multiple error messages to display on form validation errors.

- `setErrorMessages(array $messages)`: add multiple error messages to display on form validation errors, overwriting all previously set error messages.
- `getErrorMessages()`: retrieve the list of custom error messages that have been defined.
- `clearErrorMessages()`: remove all custom error messages that have been defined.
- `markAsError()`: mark the form as having failed validation.
- `addError($message)`: add a message to the custom error messages stack and flag the form as invalid.
- `addErrors(array $messages)`: add several messages to the custom error messages stack and flag the form as invalid.
- `setErrors(array $messages)`: overwrite the custom error messages stack with the provided messages and flag the form as invalid.

All errors set in this fashion may be translated.

4.7.2. Retrieving Valid Values Only

There are scenarios when you want to allow your user to work on a valid form in several steps. Meanwhile you allow the user to save the form with any set of values inbetween. Then if all the data is specified you can transfer the model from the building or prototying stage to a valid stage.

You can retrieve all the valid values that match the submitted data by calling:

```
$validValues = $form->getValidValues($_POST);
```

4.8. Methods

The following is a full list of methods available to `Zend_Form`, grouped by type:

- Configuration and Options:
  - `setOptions(array $options)`
  - `setConfig(Zend_Config $config)`
- Plugin Loaders and paths:
  - `setPluginLoader(Zend_Loader_PluginLoader_Interface $loader, $type = null)`
  - `getPluginLoader($type = null)`
  - `addPrefixPath($prefix, $path, $type = null)`
  - `addPrefixPaths(array $spec)`
  - `addElementPrefixPath($prefix, $path, $type = null)`
  - `addElementPrefixPaths(array $spec)`
  - `addDisplayGroupPrefixPath($prefix, $path)`

- Metadata:
  - `setAttrib($key, $value)`
  - `addAttribs(array $attribs)`
  - `setAttribs(array $attribs)`
  - `getAttrib($key)`
  - `getAttribs()`
  - `removeAttrib($key)`
  - `clearAttribs()`
  - `setAction($action)`
  - `getAction()`
  - `setMethod($method)`
  - `getMethod()`
  - `setName($name)`
  - `getName()`
- Elements:
  - `addElement($element, $name = null, $options = null)`
  - `addElements(array $elements)`
  - `setElements(array $elements)`
  - `getElement($name)`
  - `getElements()`
  - `removeElement($name)`
  - `clearElements()`
  - `setDefaults(array $defaults)`
  - `setDefault($name, $value)`
  - `getValue($name)`
  - `getValues()`
  - `getUnfilteredValue($name)`
  - `getUnfilteredValues()`
  - `setElementFilters(array $filters)`
  - `setElementDecorators(array $decorators)`

- Sub forms:
  - `addSubForm(Zend_Form $form, $name, $order = null)`
  - `addSubForms(array $subForms)`
  - `setSubForms(array $subForms)`
  - `getSubForm($name)`
  - `getSubForms()`
  - `removeSubForm($name)`
  - `clearSubForms()`
  - `setSubFormDecorators(array $decorators)`
- Display groups:
  - `addDisplayGroup(array $elements, $name, $options = null)`
  - `addDisplayGroups(array $groups)`
  - `setDisplayGroups(array $groups)`
  - `getDisplayGroup($name)`
  - `getDisplayGroups()`
  - `removeDisplayGroup($name)`
  - `clearDisplayGroups()`
  - `setDisplayGroupDecorators(array $decorators)`
- Validation
  - `populate(array $values)`
  - `isValid(array $data)`
  - `isValidPartial(array $data)`
  - `processAjax(array $data)`
  - `persistData()`
  - `getErrors($name = null)`
  - `getMessages($name = null)`
- Rendering:
  - `setView(Zend_View_Interface $view = null)`
  - `getView()`
  - `addDecorator($decorator, $options = null)`

    - addDecorators(array $decorators)
    - setDecorators(array $decorators)
    - getDecorator($name)
    - getDecorators()
    - removeDecorator($name)
    - clearDecorators()
    - render(Zend_View_Interface $view = null)
- I18n:
    - setTranslator(Zend_Translate_Adapter $translator = null)
    - getTranslator()
    - setDisableTranslator($flag)
    - translatorIsDisabled()

4.9. Configuration

Zend_Form is fully configurable via setOptions() and setConfig() (or by passing options or a Zend_Config object to the constructor). Using these methods, you can specify form elements, display groups, decorators, and metadata.

As a general rule, if 'set' + the option key refers to a Zend_Form method, then the value provided will be passed to that method. If the accessor does not exist, the key is assumed to reference an attribute, and will be passed to setAttrib().

Exceptions to the rule include the following:

- prefixPaths will be passed to addPrefixPaths()
- elementPrefixPaths will be passed to addElementPrefixPaths()
- displayGroupPrefixPaths will be passed to addDisplayGroupPrefixPaths()
- the following setters cannot be set in this way:
    - setAttrib (though setAttribs *will* work)
    - setConfig
    - setDefault
    - setOptions
    - setPluginLoader
    - setSubForms
    - setTranslator

- `setView`

As an example, here is a config file that passes configuration for every type of configurable data:

```
[element]
name = "registration"
action = "/user/register"
method = "post"
attribs.class = "zend_form"
attribs.onclick = "validate(this)"

disableTranslator = 0

prefixPath.element.prefix = "My_Element"
prefixPath.element.path = "My/Element/"
elementPrefixPath.validate.prefix = "My_Validate"
elementPrefixPath.validate.path = "My/Validate/"
displayGroupPrefixPath.prefix = "My_Group"
displayGroupPrefixPath.path = "My/Group/"

elements.username.type = "text"
elements.username.options.label = "Username"
elements.username.options.validators.alpha.validator = "Alpha"
elements.username.options.filters.lcase = "StringToLower"
; more elements, of course...

elementFilters.trim = "StringTrim"
;elementDecorators.trim = "StringTrim"

displayGroups.login.elements.username = "username"
displayGroups.login.elements.password = "password"
displayGroupDecorators.elements.decorator = "FormElements"
displayGroupDecorators.fieldset.decorator = "Fieldset"

decorators.elements.decorator = "FormElements"
decorators.fieldset.decorator = "FieldSet"
decorators.fieldset.decorator.options.class = "zend_form"
decorators.form.decorator = "Form"
```

The above could easily be abstracted to an XML or PHP array-based configuration file.

4.10. Custom forms

An alternative to using configuration-based forms is to subclass `Zend_Form`. This has several benefits:

- You can unit test your form easily to ensure validations and rendering perform as expected.
- Fine-grained control over individual elements.
- Re-use of form objects, and greater portability (no need to track config files).
- Implementing custom functionality.

The most typical use case would be to use the `init()` method to setup specific form elements and configuration:

```
class My_Form_Login extends Zend_Form
{
```

```
    public function init()
    {
        $username = new Zend_Form_Element_Text('username');
        $username->class = 'formtext';
        $username->setLabel('Username:')
                 ->setDecorators(array(
                     array('ViewHelper',
                           array('helper' => 'formText')),
                     array('Label',
                           array('class' => 'label'))
                 ));

        $password = new Zend_Form_Element_Password('password');
        $password->class = 'formtext';
        $password->setLabel('Username:')
                 ->setDecorators(array(
                     array('ViewHelper',
                           array('helper' => 'formPassword')),
                     array('Label',
                           array('class' => 'label'))
                 ));

        $submit = new Zend_Form_Element_Submit('login');
        $submit->class = 'formsubmit';
        $submit->setValue('Login')
               ->setDecorators(array(
                   array('ViewHelper',
                   array('helper' => 'formSubmit'))
               ));

        $this->addElements(array(
            $username,
            $password,
            $submit
        ));

        $this->setDecorators(array(
            'FormElements',
            'Fieldset',
            'Form'
        ));
    }
}
```

This form can then be instantiated with simply:

```
$form = new My_Form_Login();
```

and all functionality is already setup and ready; no config files needed. (Note that this example is greatly simplified, as it contains no validators or filters for the elements.)

Another common reason for extension is to define a set of default decorators. You can do this by overriding the `loadDefaultDecorators()` method:

```
class My_Form_Login extends Zend_Form
{
    public function loadDefaultDecorators()
    {
        $this->setDecorators(array(
```

```
            'FormElements',
            'Fieldset',
            'Form'
        ));
    }
}
```

5. Creating Custom Form Markup Using Zend_Form_Decorator

Rendering a form object is completely optional -- you do not need to use `Zend_Form`'s render() methods at all. However, if you do, decorators are used to render the various form objects.

An arbitrary number of decorators may be attached to each item (elements, display groups, sub forms, or the form object itself); however, only one decorator of a given type may be attached to each item. Decorators are called in the order they are registered. Depending on the decorator, it may replace the content passed to it, or append or prepend the content.

Object state is set via configuration options passed to the constructor or the decorator's `setOptions()` method. When creating decorators via an item's `addDecorator()` or related methods, options may be passed as an argument to the method. These can be used to specify placement, a separator to use between passed in content and newly generated content, and whatever options the decorator supports.

Before each decorator's `render()` method is called, the current item is set in the decorator using `setElement()`, giving the decorator awareness of the item being rendered. This allows you to create decorators that only render specific portions of the item -- such as the label, the value, error messages, etc. By stringing together several decorators that render specific element segments, you can build complex markup representing the entire item.

5.1. Operation

To configure a decorator, pass an array of options or a `Zend_Config` object to its constructor, an array to `setOptions()`, or a `Zend_Config` object to `setConfig()`.

Standard options include:

- `placement`: Placement can be either 'append' or 'prepend' (case insensitive), and indicates whether content passed to `render()` will be appended or prepended, respectively. In the case that a decorator replaces the content, this setting is ignored. The default setting is to append.
- `separator`: The separator is used between the content passed to `render()` and new content generated by the decorator, or between items rendered by the decorator (e.g. FormElements uses the separator between each item rendered). In the case that a decorator replaces the content, this setting may be ignored. The default value is `PHP_EOL`.

The decorator interface specifies methods for interacting with options. These include:

- `setOption($key, $value)`: set a single option.
- `getOption($key)`: retrieve a single option value.
- `getOptions()`: retrieve all options.
- `removeOption($key)`: remove a single option.
- `clearOptions()`: remove all options.

Decorators are meant to interact with the various `Zend_Form` class types: `Zend_Form`, `Zend_Form_Element`, `Zend_Form_DisplayGroup`, and all classes deriving from them. The method `setElement()` allows you to set the object the decorator is currently working with, and `getElement()` is used to retrieve it.

Each decorator's `render()` method accepts a string, `$content`. When the first decorator is called, this string is typically empty, while on subsequent calls it will be populated. Based on the type of decorator and the options passed in, the decorator will either replace this string, prepend the string, or append the string; an optional separator will be used in the latter two situations.

5.2. Standard Decorators

`Zend_Form` ships with many standard decorators; see the chapter on Standard Decorators for details.

5.3. Custom Decorators

If you find your rendering needs are complex or need heavy customization, you should consider creating a custom decorator.

Decorators need only implement `Zend_Form_Decorator_Interface`. The interface specifies the following:

```
interface Zend_Form_Decorator_Interface
{
    public function __construct($options = null);
    public function setElement($element);
    public function getElement();
    public function setOptions(array $options);
    public function setConfig(Zend_Config $config);
    public function setOption($key, $value);
    public function getOption($key);
    public function getOptions();
    public function removeOption($key);
    public function clearOptions();
    public function render($content);
}
```

To make this simpler, you can simply extend `Zend_Form_Decorator_Abstract`, which implements all methods except `render()`.

As an example, let's say you want to reduce the number of decorators you use, and build a "composite" decorator to take care of rendering the label, element, any error messages, and description in an HTML `div`. You might build such a 'Composite' decorator as follows:

```
class My_Decorator_Composite extends Zend_Form_Decorator_Abstract
{
    public function buildLabel()
    {
        $element = $this->getElement();
        $label = $element->getLabel();
        if ($translator = $element->getTranslator()) {
            $label = $translator->translate($label);
        }
        if ($element->isRequired()) {
            $label .= '*';
        }
```

```
        $label .= ':';
        return $element->getView()
                       ->formLabel($element->getName(), $label);
    }

    public function buildInput()
    {
        $element = $this->getElement();
        $helper  = $element->helper;
        return $element->getView()->$helper(
            $element->getName(),
            $element->getValue(),
            $element->getAttribs(),
            $element->options
        );
    }

    public function buildErrors()
    {
        $element  = $this->getElement();
        $messages = $element->getMessages();
        if (empty($messages)) {
            return '';
        }
        return '<div class="errors">' .
               $element->getView()->formErrors($messages) . '</div>';
    }

    public function buildDescription()
    {
        $element = $this->getElement();
        $desc    = $element->getDescription();
        if (empty($desc)) {
            return '';
        }
        return '<div class="description">' . $desc . '</div>';
    }

    public function render($content)
    {
        $element = $this->getElement();
        if (!$element instanceof Zend_Form_Element) {
            return $content;
        }
        if (null === $element->getView()) {
            return $content;
        }

        $separator = $this->getSeparator();
        $placement = $this->getPlacement();
        $label     = $this->buildLabel();
        $input     = $this->buildInput();
        $errors    = $this->buildErrors();
        $desc      = $this->buildDescription();

        $output = '<div class="form element">'
                . $label
                . $input
                . $errors
                . $desc
```

```
                . '</div>'

        switch ($placement) {
            case (self::PREPEND):
                return $output . $separator . $content;
            case (self::APPEND):
            default:
                return $content . $separator . $output;
        }
    }
}
```

You can then place this in the decorator path:

```
// for an element:
$element->addPrefixPath('My_Decorator',
                        'My/Decorator/',
                        'decorator');

// for all elements:
$form->addElementPrefixPath('My_Decorator',
                            'My/Decorator/',
                            'decorator');
```

You can then specify this decorator as 'Composite' and attach it to an element:

```
// Overwrite existing decorators with this single one:
$element->setDecorators(array('Composite'));
```

While this example showed how to create a decorator that renders complex output from several element properties, you can also create decorators that handle a single aspect of an element; the 'Decorator' and 'Label' decorators are excellent examples of this practice. Doing so allows you to mix and match decorators to achieve complex output -- and also override single aspects of decoration to customize for your needs.

For example, if you wanted to simply display that an error occurred when validating an element, but not display each of the individual validation error messages, you might create your own 'Errors' decorator:

```
class My_Decorator_Errors
{
    public function render($content = '')
    {
        $output = '<div class="errors">The value you provided was invalid;
            please try again</div>';

        $placement = $this->getPlacement();
        $separator = $this->getSeparator();

        switch ($placement) {
            case 'PREPEND':
                return $output . $separator . $content;
            case 'APPEND':
            default:
                return $content . $separator . $output;
        }
    }
}
```

In this particular example, because the decorator's final segment, 'Errors', matches the same as `Zend_Form_Decorator_Errors`, it will be rendered *in place of* that decorator -- meaning you would not need to change any decorators to modify the output. By naming your decorators after existing standard decorators, you can modify decoration without needing to modify your elements' decorators.

5.4. Rendering Individual Decorators

Since decorators can target distinct metadata of the element or form they decorate, it's often useful to render one individual decorator at a time. This behavior is possible via method overloading in each major form class type (forms, sub form, display group, element).

To do so, simply call `render[DecoratorName]()`, where "[DecoratorName]" is the "short name" of your decorator; optionally, you can pass in content you want decorated. For example:

```
// render just the element label decorator:
echo $element->renderLabel();

// render just the display group fieldset, with some content:
echo $group->renderFieldset('fieldset content');

// render just the form HTML tag, with some content:
echo $form->renderHtmlTag('wrap this content');
```

If the decorator does not exist, an exception is raised.

This can be useful particularly when rendering a form with the ViewScript decorator; each element can use its attached decorators to generate content, but with fine-grained control.

6. Standard Form Elements Shipped With Zend Framework

Zend Framework ships with concrete element classes covering most HTML form elements. Most simply specify a particular view helper for use when decorating the element, but several offer additional functionality. The following is a list of all such classes, as well as descriptions of the functionality they offer.

6.1. Zend_Form_Element_Button

Used for creating HTML button elements, `Zend_Form_Element_Button` extends Zend_Form_Element_Submit, specifying some custom functionality. It specifies the 'formButton' view helper for decoration.

Like the submit element, it uses the element's label as the element value for display purposes; in other words, to set the text of the button, set the value of the element. The label will be translated if a translation adapter is present.

Because the label is used as part of the element, the button element uses only the ViewHelper and DtDdWrapper decorators.

After populating or validating a form, you can check if the given button was clicked using the `isChecked()` method.

6.2. Zend_Form_Element_Captcha

CAPTCHAs are used to prevent automated submission of forms by bots and other automated processes.

The Captcha form element allows you to specify which Zend_Captcha adapter you wish to utilize as a form CAPTCHA. It then sets this adapter as a validator to the object, and uses a Captcha decorator for rendering (which proxies to the CAPTCHA adapter).

Adapters may be any adapters in `Zend_Captcha`, as well as any custom adapters you may have defined elsewhere. To allow this, you may pass an additional plugin loader type key, 'CAPTCHA' or 'captcha', when specifying a plugin loader prefix path:

```
$element->addPrefixPath('My_Captcha', 'My/Captcha/', 'captcha');
```

Captcha's may then be registered using the `setCaptcha()` method, which can take either a concrete CAPTCHA instance, or the short name of a CAPTCHA adapter:

```
// Concrete instance:
$element->setCaptcha(new Zend_Captcha_Figlet());

// Using shortnames:
$element->setCaptcha('Dumb');
```

If you wish to load your element via configuration, specify either the key 'captcha' with an array containing the key 'captcha', or both the keys 'captcha' and 'captchaOptions':

```
// Using single captcha key:
$element = new Zend_Form_Element_Captcha('foo', array(
    'label' => "Please verify you're a human",
    'captcha' => array(
        'captcha' => 'Figlet',
        'wordLen' => 6,
        'timeout' => 300,
    ),
));

// Using both captcha and captchaOptions:
$element = new Zend_Form_Element_Captcha('foo', array(
    'label' => "Please verify you're a human",
    'captcha' => 'Figlet',
    'captchaOptions' => array(
        'captcha' => 'Figlet',
        'wordLen' => 6,
        'timeout' => 300,
    ),
));
```

The decorator used is determined by querying the captcha adapter. By default, the Captcha decorator is used, but an adapter may specify a different one via its `getDecorator()` method.

As noted, the captcha adapter itself acts as a validator for the element. Additionally, the NotEmpty validator is not used, and the element is marked as required. In most cases, you should need to do nothing else to have a captcha present in your form.

6.3. Zend_Form_Element_Checkbox

HTML checkboxes allow you return a specific value, but basically operate as booleans. When checked, the checkbox's value is submitted. When the checkbox is not checked, nothing is submitted. Internally, `Zend_Form_Element_Checkbox` enforces this state.

By default, the checked value is '1', and the unchecked value '0'. You can specify the values to use using the `setCheckedValue()` and `setUncheckedValue()` accessors, respectively.

Internally, any time you set the value, if the provided value matches the checked value, then it is set, but any other value causes the unchecked value to be set.

Additionally, setting the value sets the `checked` property of the checkbox. You can query this using `isChecked()` or simply accessing the property. Using the `setChecked($flag)` method will both set the state of the flag as well as set the appropriate checked or unchecked value in the element. Please use this method when setting the checked state of a checkbox element to ensure the value is set properly.

`Zend_Form_Element_Checkbox` uses the 'formCheckbox' view helper. The checked value is always used to populate it.

6.4. Zend_Form_Element_File

The File form element provides a mechanism for supplying file upload fields to your form. It utilizes Zend_File_Transfer internally to provide this functionality, and the `FormFile` view helper as also the `File` decorator to display the form element.

By default, it uses the `Http` transfer adapter, which introspects the `$_FILES` array and allows you to attach validators and filters. Validators and filters attached to the form element are in turn attached to the transfer adapter.

Example 389. File form element usage

The above explanation of using the File form element may seem arcane, but actual usage is relatively trivial:

```
$element = new Zend_Form_Element_File('foo');
$element->setLabel('Upload an image:')
        ->setDestination('/var/www/upload');
// ensure only 1 file
$element->addValidator('Count', false, 1);
// limit to 100K
$element->addValidator('Size', false, 102400);
// only JPEG, PNG, and GIFs
$element->addValidator('Extension', false, 'jpg,png,gif');
$form->addElement($element, 'foo');
```

You also need to ensure that the correct encoding type is provided to the form; you should use 'multipart/form-data'. You can do this by setting the 'enctype' attribute on the form:

```
$form->setAttrib('enctype', 'multipart/form-data');
```

After the form is validated successfully, you must receive the file to store it in the final destination using `receive()`. Additionally you can determinate the final location using `getFileName()`:

```
if (!$form->isValid()) {
    print "Uh oh... validation error";
}

if (!$form->foo->receive()) {
    print "Error receiving the file";
}

$location = $form->foo->getFileName();
```

Default Upload Location

By default, files are uploaded to the system temp directory.

File values

Within HTTP a file element has no value. For this reason and because of security concerns `getValue()` returns only the uploaded filename and not the complete path. If you need the file path, call `getFileName()`, which returns both the path and the name of the file.

Per default the file will automatically be received when you call `getValues()` on the form. The reason behind this behaviour is, that the file itself is the value of the file element.

```
$form->getValues();
```

Therefor another call of `receive()` after calling `getValues()` will not have an effect. Also creating a instance of `Zend_File_Transfer` will not have an effect as there no file anymore to receive.

Still, sometimes you may want to call `getValues()` without receiving the file. You can archive this by calling `setValueDisabled(true)`. To get the actual value of this flag you can call `isValueDisabled()`.

Example 390. Explicit file retrievement

First call `setValueDisabled(true)`.

```
$element = new Zend_Form_Element_File('foo');
$element->setLabel('Upload an image:')
        ->setDestination('/var/www/upload')
        ->setValueDisabled(true);
```

Now the file will not be received when you call `getValues()`. So you must call `receive()` on the file element, or an instance of `Zend_File_Transfer` yourself.

```
$values = $form->getValues();

if ($form->isValid($form->getPost())) {
    if (!$form->foo->receive()) {
        print "Upload error";
    }
}
```

There are several states of the uploaded file which can be checked with the following methods:

- `isUploaded()`: Checks if the file element has been uploaded or not.
- `isReceived()`: Checks if the file element has already been received.
- `isFiltered()`: Checks if the filters have already been applied to the file element or not.

Example 391. Checking if an optional file has been uploaded

```
$element = new Zend_Form_Element_File('foo');
$element->setLabel('Upload an image:')
        ->setDestination('/var/www/upload')
        ->setRequired(false);
$element->addValidator('Size', false, 102400);
$form->addElement($element, 'foo');

// The foo file element is optional but when it's given go into here
if ($form->foo->isUploaded()) {
    // foo file given... do something
}
```

`Zend_Form_Element_File` also supports multiple files. By calling the `setMultiFile($count)` method you can set the number of file elements you want to create. This keeps you from setting the same settings multiple times.

Example 392. Setting multiple files

Creating a multifile element is the same as setting a single element. Just call `setMultiFile()` after the element is created:

```
$element = new Zend_Form_Element_File('foo');
$element->setLabel('Upload an image:')
        ->setDestination('/var/www/upload');
// ensure minimum 1, maximum 3 files
$element->addValidator('Count', false, array('min' => 1, 'max' => 3));
// limit to 100K
$element->addValidator('Size', false, 102400);
// only JPEG, PNG, and GIFs
$element->addValidator('Extension', false, 'jpg,png,gif');
// defines 3 identical file elements
$element->setMultiFile(3);
$form->addElement($element, 'foo');
```

You now have 3 identical file upload elements with the same settings. To get the set multifile number simply call `getMultiFile()`.

File elements in Subforms

When you use file elements in subforms you must set unique names. For example, if you name a file element in subform1 "file", you must give any file element in subform2 a different name.

If there are 2 file elements with the same name, the second element is not be displayed or submitted.

Additionally, file elements are not rendered within the sub-form. So when you add a file element into a subform, then the element will be rendered within the main form.

To limit the size of the file uploaded, you can specify the maximum file size by setting the `MAX_FILE_SIZE` option on the form. When you set this value by using the `setMaxFileSize($size)` method, it will be rendered with the file element.

```
$element = new Zend_Form_Element_File('foo');
$element->setLabel('Upload an image:')
```

```
        ->setDestination('/var/www/upload')
        ->addValidator('Size', false, 102400) // limit to 100K
        ->setMaxFileSize(102400); // limits the filesize on the client side
$form->addElement($element, 'foo');
```

MaxFileSize with Multiple File Elements

When you use multiple file elements in your form you should set the `MAX_FILE_SIZE` only once. Setting it again will overwrite the previous value.

Note, that this is also the case when you use multiple forms.

6.5. Zend_Form_Element_Hidden

Hidden elements inject data that should be submitted, but that should not manipulated by the user . `Zend_Form_Element_Hidden` accomplishes this with the 'formHidden' view helper.

6.6. Zend_Form_Element_Hash

This element provides protection from CSRF attacks on forms, ensuring the data is submitted by the user session that generated the form and not by a rogue script. Protection is achieved by adding a hash element to a form and verifying it when the form is submitted.

The name of the hash element should be unique. We recommend using the `salt` option for the element- two hashes with same names and different salts would not collide:

```
$form->addElement('hash', 'no_csrf_foo', array('salt' => 'unique'));
```

You can set the salt later using the `setSalt($salt)` method.

Internally, the element stores a unique identifier using `Zend_Session_Namespace`, and checks for it at submission (checking that the TTL has not expired). The 'Identical' validator is then used to ensure the submitted hash matches the stored hash.

The 'formHidden' view helper is used to render the element in the form.

6.7. Zend_Form_Element_Image

Images can be used as form elements, and you can use these images as graphical elements on form buttons.

Images need an image source. `Zend_Form_Element_Image` allows you to specify this by using the `setImage()` accessor (or 'image' configuration key). You can also optionally specify a value to use when submitting the image using the `setImageValue()` accessor (or 'imageValue' configuration key). When the value set for the element matches the `imageValue`, then the accessor `isChecked()` will return `TRUE`.

Image elements use the Image Decorator for rendering, in addition to the standard Errors, HtmlTag, and Label decorators. You can optionally specify a tag to the `Image` decorator that will then wrap the image element.

6.8. Zend_Form_Element_MultiCheckbox

Often you have a set of related checkboxes, and you wish to group the results. This is much like a Multiselect, but instead of them being in a dropdown list, you need to show checkbox/value pairs.

`Zend_Form_Element_MultiCheckbox` makes this a snap. Like all other elements extending the base Multi element, you can specify a list of options, and easily validate against that same list. The 'formMultiCheckbox' view helper ensures that these are returned as an array in the form submission.

By default, this element registers an `InArray` validator which validates against the array keys of registered options. You can disable this behavior by either calling `setRegisterInArrayValidator(false)`, or by passing a `FALSE` value to the `registerInArrayValidator` configuration key.

You may manipulate the various checkbox options using the following methods:

- `addMultiOption($option, $value)`
- `addMultiOptions(array $options)`
- `setMultiOptions(array $options)` (overwrites existing options)
- `getMultiOption($option)`
- `getMultiOptions()`
- `removeMultiOption($option)`
- `clearMultiOptions()`

To mark checked items, you need to pass an array of values to `setValue()`. The following will check the values "bar" and "bat":

```
$element = new Zend_Form_Element_MultiCheckbox('foo', array(
    'multiOptions' => array(
        'foo' => 'Foo Option',
        'bar' => 'Bar Option',
        'baz' => 'Baz Option',
        'bat' => 'Bat Option',
    );
));

$element->setValue(array('bar', 'bat'));
```

Note that even when setting a single value, you must pass an array.

6.9. Zend_Form_Element_Multiselect

XHTML `select` elements allow a 'multiple' attribute, indicating multiple options may be selected for submission, instead of the usual one. `Zend_Form_Element_Multiselect` extends Zend_Form_Element_Select, and sets the `multiple` attribute to 'multiple'. Like other classes that inherit from the base `Zend_Form_Element_Multi` class, you can manipulate the options for the select using:

- `addMultiOption($option, $value)`
- `addMultiOptions(array $options)`
- `setMultiOptions(array $options)` (overwrites existing options)
- `getMultiOption($option)`

- `getMultiOptions()`
- `removeMultiOption($option)`
- `clearMultiOptions()`

If a translation adapter is registered with the form and/or element, option values will be translated for display purposes.

By default, this element registers an `InArray` validator which validates against the array keys of registered options. You can disable this behavior by either calling `setRegisterInArrayValidator(false)`, or by passing a `FALSE` value to the `registerInArrayValidator` configuration key.

6.10. Zend_Form_Element_Password

Password elements are basically normal text elements -- except that you typically do not want the submitted password displayed in error messages or the element itself when the form is re-displayed.

`Zend_Form_Element_Password` achieves this by calling `setObscureValue(true)` on each validator (ensuring that the password is obscured in validation error messages), and using the 'formPassword' view helper (which does not display the value passed to it).

6.11. Zend_Form_Element_Radio

Radio elements allow you to specify several options, of which you need a single value returned. `Zend_Form_Element_Radio` extends the base `Zend_Form_Element_Multi` class, allowing you to specify a number of options, and then uses the `formRadio` view helper to display these.

By default, this element registers an `InArray` validator which validates against the array keys of registered options. You can disable this behavior by either calling `setRegisterInArrayValidator(false)`, or by passing a `FALSE` value to the `registerInArrayValidator` configuration key.

Like all elements extending the Multi element base class, the following methods may be used to manipulate the radio options displayed:

- `addMultiOption($option, $value)`
- `addMultiOptions(array $options)`
- `setMultiOptions(array $options)` (overwrites existing options)
- `getMultiOption($option)`
- `getMultiOptions()`
- `removeMultiOption($option)`
- `clearMultiOptions()`

6.12. Zend_Form_Element_Reset

Reset buttons are typically used to clear a form, and are not part of submitted data. However, as they serve a purpose in the display, they are included in the standard elements.

`Zend_Form_Element_Reset` extends Zend_Form_Element_Submit. As such, the label is used for the button display, and will be translated if a translation adapter is present. It utilizes only the 'ViewHelper' and 'DtDdWrapper' decorators, as there should never be error messages for such elements, nor will a label be necessary.

6.13. Zend_Form_Element_Select

Select boxes are a common way of limiting to specific choices for a given form datum. `Zend_Form_Element_Select` allows you to generate these quickly and easily.

By default, this element registers an `InArray` validator which validates against the array keys of registered options. You can disable this behavior by either calling `setRegisterInArrayValidator(false)`, or by passing a `FALSE` value to the `registerInArrayValidator` configuration key.

As it extends the base Multi element, the following methods may be used to manipulate the select options:

- `addMultiOption($option, $value)`
- `addMultiOptions(array $options)`
- `setMultiOptions(array $options)` (overwrites existing options)
- `getMultiOption($option)`
- `getMultiOptions()`
- `removeMultiOption($option)`
- `clearMultiOptions()`

`Zend_Form_Element_Select` uses the 'formSelect' view helper for decoration.

6.14. Zend_Form_Element_Submit

Submit buttons are used to submit a form. You may use multiple submit buttons; you can use the button used to submit the form to decide what action to take with the data submitted. `Zend_Form_Element_Submit` makes this decisioning easy, by adding a `isChecked()` method; as only one button element will be submitted by the form, after populating or validating the form, you can call this method on each submit button to determine which one was used.

`Zend_Form_Element_Submit` uses the label as the "value" of the submit button, translating it if a translation adapter is present. `isChecked()` checks the submitted value against the label in order to determine if the button was used.

The ViewHelper and DtDdWrapper decorators to render the element. No label decorator is used, as the button label is used when rendering the element; also, typically, you will not associate errors with a submit element.

6.15. Zend_Form_Element_Text

By far the most prevalent type of form element is the text element, allowing for limited text entry; it's an ideal element for most data entry. `Zend_Form_Element_Text` simply uses the 'formText' view helper to display the element.

6.16. Zend_Form_Element_Textarea

Textareas are used when large quantities of text are expected, and place no limits on the amount of text submitted (other than maximum size limits as dictated by your server or PHP). `Zend_Form_Element_Textarea` uses the 'textArea' view helper to display such elements, placing the value as the content of the element.

7. Standard Form Decorators Shipped With Zend Framework

`Zend_Form` ships with several standard decorators. For more information on general decorator usage, see the Decorators section.

7.1. Zend_Form_Decorator_Callback

The Callback decorator can execute an arbitrary callback to render content. Callbacks should be specified via the 'callback' option passed in the decorator configuration, and can be any valid PHP callback type. Callbacks should accept three arguments, `$content` (the original content passed to the decorator), `$element` (the item being decorated), and an array of `$options`. As an example callback:

```
class Util
{
    public static function label($content, $element, array $options)
    {
        return '<span class="label">' . $element->getLabel() . "</span>";
    }
}
```

This callback would be specified as `array('Util', 'label')`, and would generate some (bad) HTML markup for the label. The Callback decorator would then either replace, append, or prepend the original content with the return value of this.

The Callback decorator allows specifying a `NULL` value for the placement option, which will replace the original content with the callback return value; 'prepend' and 'append' are still valid as well.

7.2. Zend_Form_Decorator_Captcha

The Captcha decorator is for use with the CAPTCHA form element. It utilizes the CAPTCHA adapter's `render()` method to generate the output.

A variant on the Captcha decorator, 'Captcha_Word', is also commonly used, and creates two elements, an id and input. The id indicates the session identifier to compare against, and the input is for the user verification of the CAPTCHA. These are validated as a single element.

7.3. Zend_Form_Decorator_Description

The Description decorator can be used to display a description set on a `Zend_Form`, `Zend_Form_Element`, or `Zend_Form_DisplayGroup` item; it pulls the description using the object's `getDescription()` method. Common use cases are for providing UI hints for your elements.

By default, if no description is present, no output is generated. If the description is present, then it is wrapped in an HTML p tag by default, though you may specify a tag by passing a `tag` option

when creating the decorator, or calling `setTag()`. You may additionally specify a class for the tag using the `class` option or by calling `setClass()`; by default, the class 'hint' is used.

The description is escaped using the view object's escaping mechanisms by default. You can disable this by passing a `FALSE` value to the decorator's 'escape' option or `setEscape()` method.

7.4. Zend_Form_Decorator_DtDdWrapper

The default decorators utilize definition lists (`<dl>`) to render form elements. Since form items can appear in any order, display groups and sub forms can be interspersed with other form items. To keep these particular item types within the definition list, the DtDdWrapper creates a new, empty definition term (`<dt>`) and wraps its content in a new definition datum (`<dd>`). The output looks something like this:

```
<dt></dt>
<dd><fieldset id="subform">
    <legend>User Information</legend>
    ...
</fieldset></dd>
```

This decorator replaces the content provided to it by wrapping it within the `<dd>` element.

7.5. Zend_Form_Decorator_Errors

Element errors get their own decorator with the Errors decorator. This decorator proxies to the FormErrors view helper, which renders error messages in an unordered list (`<ul>`) as list items. The `<ul>` element receives a class of "errors".

The Errors decorator can either prepend or append the content provided to it.

7.6. Zend_Form_Decorator_Fieldset

Display groups and sub forms render their content within fieldsets by default. The Fieldset decorator checks for either a 'legend' option or a `getLegend()` method in the registered element, and uses that as a legend if non-empty. Any content passed in is wrapped in the HTML fieldset, replacing the original content. Any attributes set in the decorated item are passed to the fieldset as HTML attributes.

7.7. Zend_Form_Decorator_File

File Elements have special notation when you use multiple file elements or subforms. The File decorator is used by `Zend_Form_Element_File` and allows to set multiple file elements with only a single methodcall. It is used automatically and fixes the elements name.

7.8. Zend_Form_Decorator_Form

`Zend_Form` objects typically need to render an HTML form tag. The Form decorator proxies to the Form view helper. It wraps any provided content in an HTML form element, using the `Zend_Form` object's action and method, and any attributes as HTML attributes.

7.9. Zend_Form_Decorator_FormElements

Forms, display groups, and sub forms are collections of elements. In order to render these elements, they utilize the FormElements decorator, which iterates through all items, calling

`render()` on each and joining them with the registered separator. It can either append or prepend content passed to it.

7.10. Zend_Form_Decorator_FormErrors

Some developers and designers prefer to group all error messages at the top of the form. The FormErrors decorator allows you to do this.

By default, the generated list of errors has the following markup:

```
<ul class="form-errors>
    <li><b>[element label or name]</b><ul>
            <li>[error message]</li>
            <li>[error message]</li>
        </ul>
    </li>
    <li><ul>
        <li><b>[subform element label or name</b><ul>
                <li>[error message]</li>
                <li>[error message]</li>
            </ul>
        </li>
    </ul></li>
</ul>
```

You can pass in a variety of options to configure the generated output:

- `ignoreSubForms`: whether or not to disable recursion into subforms. Default value: FALSE (i.e., allow recursion).
- `markupElementLabelEnd`: Markup to append to element labels. Default value: ''
- `markupElementLabelStart`: Markup to prepend to element labels. Default value: ''
- `markupListEnd`: Markup to append error message lists with. Default value: '
'.
- `markupListItemEnd`: Markup to append individual error messages with. Default value: '
'
- `markupListItemStart`: Markup to prepend individual error messages with. Default value: '<li>'
- `markupListStart`: Markup to append error message lists with. Default value: '<ul class="form-errors">'

The FormErrors decorator can either prepend or append the content provided to it.

7.11. Zend_Form_Decorator_HtmlTag

The HtmlTag decorator allows you to utilize HTML tags to decorate content; the tag utilized is passed in the 'tag' option, and any other options are used as HTML attributes to that tag. The tag by default is assumed to be block level, and replaces the content by wrapping it in the given tag. However, you can specify a placement to append or prepend a tag as well.

7.12. Zend_Form_Decorator_Image

The Image decorator allows you to create an HTML image input (`<input type="image" ... />`), and optionally render it within another HTML tag.

By default, the decorator uses the element's src property, which can be set with the `setImage()` method, as the image source. Additionally, the element's label will be used as the alt tag, and the `imageValue` (manipulated with the Image element's `setImageValue()` and `getImageValue()` accessors) will be used for the value.

To specify an HTML tag with which to wrap the element, either pass a 'tag' option to the decorator, or explicitly call `setTag()`.

7.13. Zend_Form_Decorator_Label

Form elements typically have labels, and the Label decorator is used to render these labels. It proxies to the FormLabel view helper, and pulls the element label using the `getLabel()` method of the element. If no label is present, none is rendered. By default, labels are translated when a translation adapter exists and a translation for the label exists.

You may optionally specify a 'tag' option; if provided, it wraps the label in that block-level tag. If the 'tag' option is present, and no label present, the tag is rendered with no content. You can specify the class to use with the tag with the 'class' option or by calling `setClass()`.

Additionally, you can specify prefixes and suffixes to use when displaying the element, based on whether or not the label is for an optional or required element. Common use cases would be to append a ':' to the label, or a '*' indicating an item is required. You can do so with the following options and methods:

- `optionalPrefix`: set the text to prefix the label with when the element is optional. Use the `setOptionalPrefix()` and `getOptionalPrefix()` accessors to manipulate it.
- `optionalSuffix`: set the text to append the label with when the element is optional. Use the `setOptionalSuffix()` and `getOptionalSuffix()` accessors to manipulate it.
- `requiredPrefix`: set the text to prefix the label with when the element is required. Use the `setRequiredPrefix()` and `getRequiredPrefix()` accessors to manipulate it.
- `requiredSuffix`: set the text to append the label with when the element is required. Use the `setRequiredSuffix()` and `getRequiredSuffix()` accessors to manipulate it.

By default, the Label decorator prepends to the provided content; specify a 'placement' option of 'append' to place it after the content.

7.14. Zend_Form_Decorator_PrepareElements

Forms, display groups, and sub forms are collections of elements. When using the ViewScript decorator with your form or sub form, it's useful to be able to recursively set the view object, translator, and all fully qualifid names (as determined by sub form array notation). The 'PrepareElements' decorator can do this for you. Typically, you will set it as the first decorator in the list.

```
$form->setDecorators(array(
    'PrepareElements',
    array('ViewScript', array('viewScript' => 'form.phtml')),
));
```

7.15. Zend_Form_Decorator_ViewHelper

Most elements utilize `Zend_View` helpers for rendering, and this is done with the ViewHelper decorator. With it, you may specify a 'helper' tag to explicitly set the view helper to utilize; if

none is provided, it uses the last segment of the element's class name to determine the helper, prepending it with the string 'form': e.g., 'Zend_Form_Element_Text' would look for a view helper of 'formText'.

Any attributes of the provided element are passed to the view helper as element attributes.

By default, this decorator appends content; use the 'placement' option to specify alternate placement.

7.16. Zend_Form_Decorator_ViewScript

Sometimes you may wish to use a view script for creating your elements; this way you can have fine-grained control over your elements, turn the view script over to a designer, or simply create a way to easily override setting based on which module you're using (each module could optionally override element view scripts to suit their own needs). The ViewScript decorator solves this problem.

The ViewScript decorator requires a 'viewScript' option, either provided to the decorator, or as an attribute of the element. It then renders that view script as a partial script, meaning each call to it has its own variable scope; no variables from the view will be injected other than the element itself. Several variables are then populated:

- `element`: the element being decorated
- `content`: the content passed to the decorator
- `decorator`: the decorator object itself
- Additionally, all options passed to the decorator via `setOptions()` that are not used internally (such as placement, separator, etc.) are passed to the view script as view variables.

As an example, you might have the following element:

```
// Setting the decorator for the element to a single, ViewScript,
// decorator, specifying the viewScript as an option, and some extra
// options:
$element->setDecorators(array(array('ViewScript', array(
    'viewScript' => '_element.phtml',
    'class'      => 'form element'
))));

// OR specifying the viewScript as an element attribute:
$element->viewScript = '_element.phtml';
$element->setDecorators(array(array('ViewScript',
                                    array('class' => 'form element'))));
```

You could then have a view script something like this:

```
<div class="<?php echo $this->class ?>">
    <?php echo $this->formLabel($this->element->getName(),
                          $this->element->getLabel()) ?>
    <?php echo $this->{$this->element->helper}(
        $this->element->getName(),
        $this->element->getValue(),
        $this->element->getAttribs()
    ) ?>
    <?php echo $this->formErrors($this->element->getMessages()) ?>
    <div class="hint"><?php echo $this->element->getDescription() ?></div>
```

```
</div>
```

Replacing content with a view script

You may find it useful for the view script to replace the content provided to the decorator -- for instance, if you want to wrap it. You can do so by specifying a boolean FALSE value for the decorator's 'placement' option:

```
// At decorator creation:
$element->addDecorator('ViewScript', array('placement' => false));

// Applying to an existing decorator instance:
$decorator->setOption('placement', false);

// Applying to a decorator already attached to an element:
$element->getDecorator('ViewScript')->setOption('placement', false);

// Within a view script used by a decorator:
$this->decorator->setOption('placement', false);
```

Using the ViewScript decorator is recommended for when you want to have very fine-grained control over how your elements are rendered.

8. Internationalization of Zend_Form

Increasingly, developers need to tailor their content for multiple languages and regions. `Zend_Form` aims to make such a task trivial, and leverages functionality in both Zend_Translate and Zend_Validate to do so.

By default, no internationalisation (I18n) is performed. To turn on I18n features in `Zend_Form`, you will need to instantiate a `Zend_Translate` object with an appropriate adapter, and attach it to `Zend_Form` and/or `Zend_Validate`. See the Zend_Translate documentation for more information on creating the translate object and translation files

Translation Can Be Turned Off Per Item

You can disable translation for any form, element, display group, or sub form by calling its `setDisableTranslator($flag)` method or passing a disableTranslator option to the object. This can be useful when you want to selectively disable translation for individual elements or sets of elements.

8.1. Initializing I18n in Forms

In order to initialize I18n in forms, you will need either a `Zend_Translate` object or a `Zend_Translate_Adapter` object, as detailed in the `Zend_Translate` documentation. Once you have a translation object, you have several options:

- *Easiest:* add it to the registry. All I18n aware components of Zend Framework will autodiscover a translate object that is in the registry under the 'Zend_Translate' key and use it to perform translation and/or localization:

```
// use the 'Zend_Translate' key; $translate is a Zend_Translate object:
Zend_Registry::set('Zend_Translate', $translate);
```

This will be picked up by `Zend_Form`, `Zend_Validate`, and `Zend_View_Helper_Translate`.

- If all you are worried about is translating validation error messages, you can register the translation object with `Zend_Validate_Abstract`:

```
// Tell all validation classes to use a specific translate adapter:
Zend_Validate_Abstract::setDefaultTranslator($translate);
```

- Alternatively, you can attach to the `Zend_Form` object as a global translator. This has the side effect of also translating validation error messages:

```
// Tell all form classes to use a specific translate adapter, as well
// as use this adapter to translate validation error messages:
Zend_Form::setDefaultTranslator($translate);
```

- Finally, you can attach a translator to a specific form instance or to specific elements using their `setTranslator()` methods:

```
// Tell *this* form instance to use a specific translate adapter; it
// will also be used to translate validation error messages for all
// elements:
$form->setTranslator($translate);

// Tell *this* element to use a specific translate adapter; it will
// also be used to translate validation error messages for this
// particular element:
$element->setTranslator($translate);
```

8.2. Standard I18n Targets

Now that you've attached a translation object to, what exactly can you translate by default?

- *Validation error messages.* Validation error messages may be translated. To do so, use the various error code constants from the `Zend_Validate` validation classes as the message IDs. For more information on these codes, see the Zend_Validate documentation.

  Alternately, as of 1.6.0, you may provide translation strings using the actual error messages as message identifiers. This is the preferred use case for 1.6.0 and up, as we will be deprecating translation of message keys in future releases.

- *Labels.* Element labels will be translated, if a translation exists.

- *Fieldset Legends.* Display groups and sub forms render in fieldsets by default. The Fieldset decorator attempts to translate the legend before rendering the fieldset.

- *Form and Element Descriptions.* All form types (element, form, display group, sub form) allow specifying an optional item description. The Description decorator can be used to render this, and by default will take the value and attempt to translate it.

- *Multi-option Values.* for the various items inheriting from `Zend_Form_Element_Multi` (including the MultiCheckbox, Multiselect, and Radio elements), the option values (not keys) will be translated if a translation is available; this means that the option labels presented to the user will be translated.

- *Submit and Button Labels.* The various Submit and Button elements (Button, Submit, and Reset) will translate the label displayed to the user.

9. Advanced Zend_Form Usage

`Zend_Form` has a wealth of functionality, much of it aimed at experienced developers. This chapter aims to document some of this functionality with examples and use cases.

9.1. Array Notation

Many experienced web developers like to group related form elements using array notation in the element names. For example, if you have two addresses you wish to capture, a shipping and a billing address, you may have identical elements; by grouping them in an array, you can ensure they are captured separately. Take the following form, for example:

```
<form>
    <fieldset>
        <legend>Shipping Address</legend>
        <dl>
            <dt><label for="recipient">Ship to:</label></dt>
            <dd><input name="recipient" type="text" value="" /></dd>

            <dt><label for="address">Address:</label></dt>
            <dd><input name="address" type="text" value="" /></dd>

            <dt><label for="municipality">City:</label></dt>
            <dd><input name="municipality" type="text" value="" /></dd>

            <dt><label for="province">State:</label></dt>
            <dd><input name="province" type="text" value="" /></dd>

            <dt><label for="postal">Postal Code:</label></dt>
            <dd><input name="postal" type="text" value="" /></dd>
        </dl>
    </fieldset>

    <fieldset>
        <legend>Billing Address</legend>
        <dl>
            <dt><label for="payer">Bill To:</label></dt>
            <dd><input name="payer" type="text" value="" /></dd>

            <dt><label for="address">Address:</label></dt>
            <dd><input name="address" type="text" value="" /></dd>

            <dt><label for="municipality">City:</label></dt>
            <dd><input name="municipality" type="text" value="" /></dd>

            <dt><label for="province">State:</label></dt>
            <dd><input name="province" type="text" value="" /></dd>

            <dt><label for="postal">Postal Code:</label></dt>
            <dd><input name="postal" type="text" value="" /></dd>
        </dl>
    </fieldset>

    <dl>
        <dt><label for="terms">I agree to the Terms of Service</label></dt>
        <dd><input name="terms" type="checkbox" value="" /></dd>

        <dt></dt>
```

```
        <dd><input name="save" type="submit" value="Save" /></dd>
    </dl>
</form>
```

In this example, the billing and shipping address contain some identical fields, which means one would overwrite the other. We can solve this solution using array notation:

```
<form>
    <fieldset>
        <legend>Shipping Address</legend>
        <dl>
            <dt><label for="shipping-recipient">Ship to:</label></dt>
            <dd><input name="shipping[recipient]" id="shipping-recipient"
                type="text" value="" /></dd>

            <dt><label for="shipping-address">Address:</label></dt>
            <dd><input name="shipping[address]" id="shipping-address"
                type="text" value="" /></dd>

            <dt><label for="shipping-municipality">City:</label></dt>
            <dd><input name="shipping[municipality]" id="shipping-municipality"
                type="text" value="" /></dd>

            <dt><label for="shipping-province">State:</label></dt>
            <dd><input name="shipping[province]" id="shipping-province"
                type="text" value="" /></dd>

            <dt><label for="shipping-postal">Postal Code:</label></dt>
            <dd><input name="shipping[postal]" id="shipping-postal"
                type="text" value="" /></dd>
        </dl>
    </fieldset>

    <fieldset>
        <legend>Billing Address</legend>
        <dl>
            <dt><label for="billing-payer">Bill To:</label></dt>
            <dd><input name="billing[payer]" id="billing-payer"
                type="text" value="" /></dd>

            <dt><label for="billing-address">Address:</label></dt>
            <dd><input name="billing[address]" id="billing-address"
                type="text" value="" /></dd>

            <dt><label for="billing-municipality">City:</label></dt>
            <dd><input name="billing[municipality]" id="billing-municipality"
                type="text" value="" /></dd>

            <dt><label for="billing-province">State:</label></dt>
            <dd><input name="billing[province]" id="billing-province"
                type="text" value="" /></dd>

            <dt><label for="billing-postal">Postal Code:</label></dt>
            <dd><input name="billing[postal]" id="billing-postal"
                type="text" value="" /></dd>
        </dl>
    </fieldset>

    <dl>
```

```
        <dt><label for="terms">I agree to the Terms of Service</label></dt>
        <dd><input name="terms" type="checkbox" value="" /></dd>

        <dt></dt>
        <dd><input name="save" type="submit" value="Save" /></dd>
    </dl>
</form>
```

In the above sample, we now get separate addresses. In the submitted form, we'll now have three elements, the 'save' element for the submit, and then two arrays, 'shipping' and 'billing', each with keys for their various elements.

`Zend_Form` attempts to automate this process with its sub forms. By default, sub forms render using the array notation as shown in the previous HTML form listing, complete with ids. The array name is based on the sub form name, with the keys based on the elements contained in the sub form. Sub forms may be nested arbitrarily deep, and this will create nested arrays to reflect the structure. Additionally, the various validation routines in `Zend_Form` honor the array structure, ensuring that your form validates correctly, no matter how arbitrarily deep you nest your sub forms. You need do nothing to benefit from this; this behaviour is enabled by default.

Additionally, there are facilities that allow you to turn on array notation conditionally, as well as specify the specific array to which an element or collection belongs:

- `Zend_Form::setIsArray($flag)`: By setting the flag `TRUE`, you can indicate that an entire form should be treated as an array. By default, the form's name will be used as the name of the array, unless `setElementsBelongTo()` has been called. If the form has no specified name, or if `setElementsBelongTo()` has not been set, this flag will be ignored (as there is no array name to which the elements may belong).

  You may determine if a form is being treated as an array using the `isArray()` accessor.

- `Zend_Form::setElementsBelongTo($array)`: Using this method, you can specify the name of an array to which all elements of the form belong. You can determine the name using the `getElementsBelongTo()` accessor.

Additionally, on the element level, you can specify individual elements may belong to particular arrays using `Zend_Form_Element::setBelongsTo()` method. To discover what this value is -- whether set explicitly or implicitly via the form -- you may use the `getBelongsTo()` accessor.

9.2. Multi-Page Forms

Currently, Multi-Page forms are not officially supported in `Zend_Form`; however, most support for implementing them is available and can be utilized with a little extra tooling.

The key to creating a multi-page form is to utilize sub forms, but to display only one such sub form per page. This allows you to submit a single sub form at a time and validate it, but not process the form until all sub forms are complete.

```
            )),

            new Zend_Form_Element_Text('familyName', array(
                'required'   => true,
                'label'      => 'Family (Last) Name:',
                'filters'    => array('StringTrim'),
                'validators' => array(
                    array('Regex',
                          false,
                          array('/^[a-z][a-z0-9., \'-]{2,}$/i'))
                )
            )),

            new Zend_Form_Element_Text('location', array(
                'required'   => true,
                'label'      => 'Your Location:',
                'filters'    => array('StringTrim'),
                'validators' => array(
                    array('StringLength', false, array(2))
                )
            )),
        ));

        // Create mailing lists sub form
```

```
class My_Form_Registration extends Zend_Form
{
    // ...

    /**
     * Prepare a sub form for display
     *
     * @param  string|Zend_Form_SubForm $spec
```

```
class RegistrationController extends Zend_Controller_Action
{
    // ...

    protected $_namespace = 'RegistrationController';
    protected $_session;

    /**
     * Get the session namespace we're using
     *
     * @return Zend_Session_Namespace
     */
    public function getSessionNamespace()
    {
        if (null === $this->_session) {
            $this->_session =
                new Zend_Session_Namespace($this->_namespace);
        }

        return $this->_session;
    }
```

```
<?php // registration/index.phtml ?>
<h2>Registration</h2>
<?php echo $this->form ?>
<?php // registration/verification.phtml ?>
<h2>Thank you for registering!</h2>
<p>
    Here is the information you provided:
</p>

<?
// Have to do this construct due to how items are stored in session
// namespaces
foreach ($this->info as $info):
    foreach ($info as $form => $data): ?>
<h4><?php echo ucfirst($form) ?>:</h4>
<dl>
    <?php foreach ($data as $key => $value): ?>
    <dt><?php echo ucfirst($key) ?></dt>
    <?php if (is_array($value)):
        foreach ($value as $label => $val): ?>
```

Upcoming releases of Zend Framework will include components to make multi page forms simpler by abstracting the session and ordering logic. In the meantime, the above example should serve as a reasonable guideline on how to accomplish this task for your site.

Zend_Gdata

1. Introduction

Google Data APIs provide programmatic interface to some of Google's online services. The Google data Protocol is based upon the Atom Publishing Protocol and allows client applications to retrieve data matching queries, post data, update data and delete data using standard HTTP and the Atom syndication formation. The `Zend_Gdata` component is a PHP 5 interface for accessing Google Data from PHP. The `Zend_Gdata` component also supports accessing other services implementing the Atom Publishing Protocol.

See http://code.google.com/apis/gdata/ for more information about Google Data API.

The services that are accessible by `Zend_Gdata` include the following:

- Google Calendar is a popular online calendar application.
- Google Spreadsheets provides an online collaborative spreadsheets tool which can be used as a simple data store for your applications.
- Google Documents List provides an online list of all spreadsheets, word processing documents, and presentations stored in a Google account.
- Google Provisioning provides the ability to create, retrieve, update, and delete user accounts, nicknames, and email lists on a Google Apps hosted domain.
- Google Base provides the ability to retrieve, post, update, and delete items in Google Base.
- YouTube provides the ability to search and retrieve videos, comments, favorites, subscriptions, user profiles and more.
- Picasa Web Albums provides an online photo sharing application.
- Google Blogger is a popular Internet provider of "push-button publishing" and syndication.
- Google CodeSearch allows you to search public source code from many projects.
- Google Notebook allows you to view public Notebook content.

Unsupported services

`Zend_Gdata` does not provide an interface to any other Google service, such as Search, Gmail, Translation, or Maps. Only services that support the Google Data API are supported.

1.1. Structure of Zend_Gdata

`Zend_Gata` is composed of several types of classes:

- Service classes - inheriting from `Zend_Gdata_App`. These also include other classes such as `Zend_Gdata`, `Zend_Gdata_Spreadsheets`, etc. These classes enable interacting with

APP or GData services and provide the ability to retrieve feeds, retrieve entries, post entries, update entries and delete entries.

- Query classes - inheriting from `Zend_Gdata_Query`. These also include other classes for specific services, such as `Zend_Gdata_Spreadsheets_ListQuery` and `Zend_Gdata_Spreadsheets_CellQuery`. Query classes provide methods used to construct a query for data to be retrieved from GData services. Methods include getters and setters like `setUpdatedMin()`, `setStartIndex()`, and `getPublishedMin()`. The query classes also have a method to generate a URL representing the constructed query -- `getQueryUrl`. Alternatively, the query string component of the URL can be retrieved used the `getQueryString()` method.

- Feed classes - inheriting from `Zend_Gdata_App_Feed`. These also include other classes such as `Zend_Gdata_Feed`, `Zend_Gdata_Spreadsheets_SpreadsheetFeed`, and `Zend_Gdata_Spreadsheets_ListFeed`. These classes represent feeds of entries retrieved from services. They are primarily used to retrieve data returned from services.

- Entry classes - inheriting from `Zend_Gdata_App_Entry`. These also include other classes such as `Zend_Gdata_Entry`, and `Zend_Gdata_Spreadsheets_ListEntry`. These classes represent entries retrieved from services or used for constructing data to send to services. In addition to being able to set the properties of an entry (such as the spreadsheet cell value), you can use an entry object to send update or delete requests to a service. For example, you can call `$entry->save()` to save changes made to an entry back to service from which the entry initiated, or `$entry->delete()` to delete an entry from the server.

- Other Data model classes - inheriting from `Zend_Gdata_App_Extension`. These include classes such as `Zend_Gdata_App_Extension_Title` (representing the atom:title XML element), `Zend_Gdata_Extension_When` (representing the gd:when XML element used by the GData Event "Kind"), and `Zend_Gdata_Extension_Cell` (representing the gs:cell XML element used by Google Spreadsheets). These classes are used purely to store the data retrieved back from services and for constructing data to be sent to services. These include getters and setters such as `setText()` to set the child text node of an element, `getText()` to retrieve the text node of an element, `getStartTime()` to retrieve the start time attribute of a When element, and other similiar methods. The data model classes also include methods such as `getDOM()` to retrieve a DOM representation of the element and all children and `transferFromDOM()` to construct a data model representation of a DOM tree.

1.2. Interacting with Google Services

Google data services are based upon the Atom Publishing Protocol (APP) and the Atom syndication format. To interact with APP or Google services using the `Zend_Gdata` component, you need to use the service classes such as `Zend_Gdata_App`, `Zend_Gdata`, `Zend_Gdata_Spreadsheets`, etc. These service classes provide methods to retrieve data from services as feeds, insert new entries into feeds, update entries, and delete entries.

Note: A full example of working with `Zend_Gdata` is available in the `demos/Zend/Gdata` directory. This example is runnable from the command-line, but the methods contained within are easily portable to a web application.

1.3. Obtaining instances of Zend_Gdata classes

The Zend Framework naming standards require that all classes be named based upon the directory structure in which they are located. For instance, extensions related to Spreadsheets are stored in: `Zend/Gdata/Spreadsheets/Extension/...` and, as a result of this, are

named `Zend_Gdata_Spreadsheets_Extension_...`. This causes a lot of typing if you're trying to construct a new instance of a spreadsheet cell element!

We've implemented a magic factory method in all service classes (such as `Zend_Gdata_App`, `Zend_Gdata`, `Zend_Gdata_Spreadsheets`) that should make constructing new instances of data model, query and other classes much easier. This magic factory is implemented by using the magic `__call` method to intercept all attempts to call `$service->newXXX(arg1, arg2, ...)`. Based off the value of XXX, a search is performed in all registered 'packages' for the desired class. Here's some examples:

```
$ss = new Zend_Gdata_Spreadsheets();

// creates a Zend_Gdata_App_Spreadsheets_CellEntry
$entry = $ss->newCellEntry();

// creates a Zend_Gdata_App_Spreadsheets_Extension_Cell
$cell = $ss->newCell();
$cell->setText('My cell value');
$cell->setRow('1');
$cell->setColumn('3');
$entry->cell = $cell;

// ... $entry can then be used to send an update to a Google Spreadsheet
```

Each service class in the inheritance tree is responsible for registering the appropriate 'packages' (directories) which are to be searched when calling the magic factory method.

1.4. Google Data Client Authentication

Most Google Data services require client applications to authenticate against the Google server before accessing private data, or saving or deleting data. There are two implementations of authentication for Google Data: AuthSub and ClientLogin. `Zend_Gdata` offers class interfaces for both of these methods.

Most other types of queries against Google Data services do not require authentication.

1.5. Dependencies

`Zend_Gdata` makes use of Zend_Http_Client to send requests to google.com and fetch results. The response to most Google Data requests is returned as a subclass of the `Zend_Gdata_App_Feed` or `Zend_Gdata_App_Entry` classes.

`Zend_Gdata` assumes your PHP application is running on a host that has a direct connection to the Internet. The `Zend_Gdata` client operates by contacting Google Data servers.

1.6. Creating a new Gdata client

Create a new object of class `Zend_Gdata_App`, `Zend_Gdata`, or one of the subclasses available that offer helper methods for service-specific behavior.

The single optional parameter to the `Zend_Gdata_App` constructor is an instance of Zend_Http_Client. If you don't pass this parameter, `Zend_Gdata` creates a default `Zend_Http_Client` object, which will not have associated credentials to access private feeds. Specifying the `Zend_Http_Client` object also allows you to pass configuration options to that client object.

```
$client = new Zend_Http_Client();
$client->setConfig( ...options... );

$gdata = new Zend_Gdata($client);
```

Beginning with Zend Framework 1.7, support has been added for protocol versioning. This allows the client and server to support new features while maintaining backwards compatibility. While most services will manage this for you, if you create a `Zend_Gdata` instance directly (as opposed to one of its subclasses), you may need to specify the desired protocol version to access certain server functionality.

```
$client = new Zend_Http_Client();
$client->setConfig( ...options... );

$gdata = new Zend_Gdata($client);
$gdata->setMajorProtocolVersion(2);
$gdata->setMinorProtocolVersion(null);
```

Also see the sections on authentication for methods to create an authenticated `Zend_Http_Client` object.

1.7. Common Query Parameters

You can specify parameters to customize queries with `Zend_Gdata`. Query parameters are specified using subclasses of `Zend_Gdata_Query`. The `Zend_Gdata_Query` class includes methods to set all query parameters used throughout GData services. Individual services, such as Spreadsheets, also provide query classes to defined parameters which are custom to the particular service and feeds. Spreadsheets includes a CellQuery class to query the Cell Feed and a ListQuery class to query the List Feed, as different query parameters are applicable to each of those feed types. The GData-wide parameters are described below.

- The q parameter specifies a full-text query. The value of the parameter is a string.

  Set this parameter with the `setQuery()` function.

- The `alt` parameter specifies the feed type. The value of the parameter can be `atom`, `rss`, `json`, or `json-in-script`. If you don't specify this parameter, the default feed type is `atom`. NOTE: Only the output of the atom feed format can be processed using `Zend_Gdata`. The `Zend_Http_Client` could be used to retrieve feeds in other formats, using query URLs generated by the `Zend_Gdata_Query` class and its subclasses.

  Set this parameter with the `setAlt()` function.

- The `maxResults` parameter limits the number of entries in the feed. The value of the parameter is an integer. The number of entries returned in the feed will not exceed this value.

  Set this parameter with the `setMaxResults()` function.

- The `startIndex` parameter specifies the ordinal number of the first entry returned in the feed. Entries before this number are skipped.

  Set this parameter with the `setStartIndex()` function.

- The `updatedMin` and `updatedMax` parameters specify bounds on the entry date. If you specify a value for `updatedMin`, no entries that were updated earlier than the date you specify

are included in the feed. Likewise no entries updated after the date specified by `updatedMax` are included.

You can use numeric timestamps, or a variety of date/time string representations as the value for these parameters.

Set this parameter with the `setUpdatedMin()` and `setUpdatedMax()` functions.

There is a `get` function for each `set` function.

```
$query = new Zend_Gdata_Query();
$query->setMaxResults(10);
echo $query->getMaxResults();   // returns 10
```

The `Zend_Gdata` class also implements "magic" getter and setter methods, so you can use the name of the parameter as a virtual member of the class.

```
$query = new Zend_Gdata_Query();
$query->maxResults = 10;
echo $query->maxResults;        // returns 10
```

You can clear all parameters with the `resetParameters()` function. This is useful to do if you reuse a `Zend_Gdata` object for multiple queries.

```
$query = new Zend_Gdata_Query();
$query->maxResults = 10;
// ...get feed...

$query->resetParameters();      // clears all parameters
// ...get a different feed...
```

1.8. Fetching a Feed

Use the `getFeed()` function to retrieve a feed from a specified URI. This function returns an instance of class specified as the second argument to getFeed, which defaults to `Zend_Gdata_Feed`.

```
$gdata = new Zend_Gdata();
$query = new Zend_Gdata_Query(
        'http://www.blogger.com/feeds/blogID/posts/default');
$query->setMaxResults(10);
$feed = $gdata->getFeed($query);
```

See later sections for special functions in each helper class for Google Data services. These functions help you to get feeds from the URI that is appropriate for the respective service.

1.9. Working with Multi-page Feeds

When retrieving a feed that contains a large number of entries, the feed may be broken up into many smaller "pages" of feeds. When this occurs, each page will contain a link to the next page in the series. This link can be accessed by calling `getLink('next')`. The following example shows how to retrieve the next page of a feed:

```
function getNextPage($feed) {
    $nextURL = $feed->getLink('next');
```

```
    if ($nextURL !== null) {
        return $gdata->getFeed($nextURL);
    } else {
        return null;
    }
}
```

If you would prefer not to work with pages in your application, pass the first page of the feed into `Zend_Gdata_App::retrieveAllEntriesForFeed()`, which will consolidate all entries from each page into a single feed. This example shows how to use this function:

```
$gdata = new Zend_Gdata();
$query = new Zend_Gdata_Query(
        'http://www.blogger.com/feeds/blogID/posts/default');
$feed = $gdata->retrieveAllEntriesForFeed($gdata->getFeed($query));
```

Keep in mind when calling this function that it may take a long time to complete on large feeds. You may need to increase PHP's execution time limit by calling `set_time_limit()`.

1.10. Working with Data in Feeds and Entries

After retrieving a feed, you can read the data from the feed or the entries contained in the feed using either the accessors defined in each of the data model classes or the magic accessors. Here's an example:

```
$client = Zend_Gdata_ClientLogin::getHttpClient($user, $pass, $service);
$gdata = new Zend_Gdata($client);
$query = new Zend_Gdata_Query(
        'http://www.blogger.com/feeds/blogID/posts/default');
$query->setMaxResults(10);
$feed = $gdata->getFeed($query);
foreach ($feed as $entry) {
    // using the magic accessor
    echo 'Title: ' . $entry->title->text;
    // using the defined accessors
    echo 'Content: ' . $entry->getContent()->getText();
}
```

1.11. Updating Entries

After retrieving an entry, you can update that entry and save changes back to the server. Here's an example:

```
$client = Zend_Gdata_ClientLogin::getHttpClient($user, $pass, $service);
$gdata = new Zend_Gdata($client);
$query = new Zend_Gdata_Query(
        'http://www.blogger.com/feeds/blogID/posts/default');
$query->setMaxResults(10);
$feed = $gdata->getFeed($query);
foreach ($feed as $entry) {
    // update the title to append 'NEW'
    echo 'Old Title: ' . $entry->title->text;
    $entry->title->text = $entry->title->text . ' NEW';

    // update the entry on the server
    $newEntry = $entry->save();
```

```
    echo 'New Title: ' . $newEntry->title->text;
}
```

1.12. Posting Entries to Google Servers

The Zend_Gdata object has a function insertEntry() with which you can upload data to save new entries to Google Data services.

You can use the data model classes for each service to construct the appropriate entry to post to Google's services. The insertEntry() function will accept a child of Zend_Gdata_App_Entry as data to post to the service. The method returns a child of Zend_Gdata_App_Entry which represents the state of the entry as it was returned from the server.

Alternatively, you could construct the XML structure for an entry as a string and pass the string to the insertEntry() function.

```
$gdata = new Zend_Gdata($authenticatedHttpClient);

$entry = $gdata->newEntry();
$entry->title = $gdata->newTitle('Playing football at the park');
$content =
    $gdata->newContent('We will visit the park and play football');
$content->setType('text');
$entry->content = $content;

$entryResult = $gdata->insertEntry($entry,
        'http://www.blogger.com/feeds/blogID/posts/default');

echo 'The <id> of the resulting entry is: ' . $entryResult->id->text;
```

To post entries, you must be using an authenticated Zend_Http_Client that you created using the Zend_Gdata_AuthSub or Zend_Gdata_ClientLogin classes.

1.13. Deleting Entries on Google Servers

Option 1: The Zend_Gdata object has a function delete() with which you can delete entries from Google Data services. Pass the edit URL value from a feed entry to the delete() method.

Option 2: Alternatively, you can call $entry->delete() on an entry retrieved from a Google service.

```
$gdata = new Zend_Gdata($authenticatedHttpClient);
// a Google Data feed
$feedUri = ...;
$feed = $gdata->getFeed($feedUri);
foreach ($feed as $feedEntry) {
    // Option 1 - delete the entry directly
    $feedEntry->delete();
    // Option 2 - delete the entry by passing the edit URL to
    // $gdata->delete()
    // $gdata->delete($feedEntry->getEditLink()->href);
}
```

To delete entries, you must be using an authenticated Zend_Http_Client that you created using the Zend_Gdata_AuthSub or Zend_Gdata_ClientLogin classes.

2. Authenticating with AuthSub

The AuthSub mechanism enables you to write web applications that acquire authenticated access Google Data services, without having to write code that handles user credentials.

See http://code.google.com/apis/accounts/AuthForWebApps.html for more information about Google Data AuthSub authentication.

The Google documentation says the ClientLogin mechanism is appropriate for "installed applications" whereas the AuthSub mechanism is for "web applications." The difference is that AuthSub requires interaction from the user, and a browser interface that can react to redirection requests. The ClientLogin solution uses PHP code to supply the account credentials; the user is not required to enter her credentials interactively.

The account credentials supplied via the AuthSub mechanism are entered by the user of the web application. Therefore they must be account credentials that are known to that user.

Registered applications

`Zend_Gdata` currently does not support use of secure tokens, because the AuthSub authentication does not support passing a digital certificate to acquire a secure token.

2.1. Creating an AuthSub authenticated Http Client

Your PHP application should provide a hyperlink to the Google URL that performs authentication. The static function `Zend_Gdata_AuthSub::getAuthSubTokenUri()` provides the correct URL. The arguments to this function include the URL to your PHP application so that Google can redirect the user's browser back to your application after the user's credentials have been verified.

After Google's authentication server redirects the user's browser back to the current application, a GET request parameter is set, called *token*. The value of this parameter is a single-use token that can be used for authenticated access. This token can be converted into a multi-use token and stored in your session.

Then use the token value in a call to `Zend_Gdata_AuthSub::getHttpClient()`. This function returns an instance of `Zend_Http_Client`, with appropriate headers set so that subsequent requests your application submits using that Http Client are also authenticated.

Below is an example of PHP code for a web application to acquire authentication to use the Google Calendar service and create a `Zend_Gdata` client object using that authenticated Http Client.

```
$my_calendar = 'http://www.google.com/calendar/feeds/default/private/full';

if (!isset($_SESSION['cal_token'])) {
    if (isset($_GET['token'])) {
        // You can convert the single-use token to a session token.
        $session_token =
            Zend_Gdata_AuthSub::getAuthSubSessionToken($_GET['token']);
        // Store the session token in our session.
        $_SESSION['cal_token'] = $session_token;
    } else {
```

```
        // Display link to generate single-use token
        $googleUri = Zend_Gdata_AuthSub::getAuthSubTokenUri(
            'http://'. $_SERVER['SERVER_NAME'] . $_SERVER['REQUEST_URI'],
            $my_calendar, 0, 1);
        echo "Click <a href='$googleUri'>here</a> " .
             "to authorize this application.";
        exit();
    }
}

// Create an authenticated HTTP Client to talk to Google.
$client = Zend_Gdata_AuthSub::getHttpClient($_SESSION['cal_token']);

// Create a Gdata object using the authenticated Http Client
$cal = new Zend_Gdata_Calendar($client);
```

2.2. Revoking AuthSub authentication

To terminate the authenticated status of a given token, use the `Zend_Gdata_AuthSub::AuthSubRevokeToken()` static function. Otherwise, the token is still valid for some time.

```
// Carefully construct this value to avoid application security problems.
$php_self = htmlentities(substr($_SERVER['PHP_SELF'],
                         0,
                         strcspn($_SERVER['PHP_SELF'], "\n\r")),
                         ENT_QUOTES);

if (isset($_GET['logout'])) {
    Zend_Gdata_AuthSub::AuthSubRevokeToken($_SESSION['cal_token']);
    unset($_SESSION['cal_token']);
    header('Location: ' . $php_self);
    exit();
}
```

Security notes

The treatment of the `$php_self` variable in the example above is a general security guideline, it is not specific to `Zend_Gdata`. You should always filter content you output to http headers.

Regarding revoking authentication tokens, it is recommended to do this when the user is finished with her Google Data session. The possibility that someone can intercept the token and use it for malicious purposes is very small, but nevertheless it is a good practice to terminate authenticated access to any service.

3. Using the Book Search Data API

The Google Book Search Data API allows client applications to view and update Book Search content in the form of Google Data API feeds.

Your client application can use the Book Search Data API to issue full-text searches for books and to retrieve standard book information, ratings, and reviews. You can also access individual users' library collections and public reviews. Finally, your application can submit authenticated

requests to enable users to create and modify library collections, ratings, labels, reviews, and other account-specific entities.

For more information on the Book Search Data API, please refer to the official PHP Developer's Guide on code.google.com.

3.1. Authenticating to the Book Search service

You can access both public and private feeds using the Book Search Data API. Public feeds don't require any authentication, but they are read-only. If you want to modify user libraries, submit reviews or ratings, or add labels, then your client needs to authenticate before requesting private feeds. It can authenticate using either of two approaches: AuthSub proxy authentication or ClientLogin username/password authentication. Please refer to the Authentication section in the PHP Developer's Guide for more detail.

3.2. Searching for books

The Book Search Data API provides a number of feeds that list collections of books.

The most common action is to retrieve a list of books that match a search query. To do so you create a `VolumeQuery` object and pass it to the `Books::getVolumeFeed` method.

For example, to perform a keyword query, with a filter on viewability to restrict the results to partial or full view books, use the `setMinViewability` and `setQuery` methods of the `VolumeQuery` object. The following code snippet prints the title and viewability of all volumes whose metadata or text matches the query term "domino":

```
$books = new Zend_Gdata_Books();
$query = $books->newVolumeQuery();

$query->setQuery('domino');
$query->setMinViewability('partial_view');

$feed = $books->getVolumeFeed($query);

foreach ($feed as $entry) {
    echo $entry->getVolumeId();
    echo $entry->getTitle();
    echo $entry->getViewability();
}
```

The `Query` class, and subclasses like `VolumeQuery`, are responsible for constructing feed URLs. The VolumeQuery shown above constructs a URL equivalent to the following:

```
http://www.google.com/books/feeds/volumes?q=keyword&min-viewability=partial
```

Note: Since Book Search results are public, you can issue a Book Search query without authentication.

Here are some of the most common `VolumeQuery` methods for setting search parameters:

`setQuery`: Specifies a search query term. Book Search searches all book metadata and full text for books matching the term. Book metadata includes titles, keywords, descriptions, author names, and subjects. Note that any spaces, quotes or other punctuation in the parameter value must be URL-escaped. (Use a plus (+) for a space.) To search for an exact phrase, enclose the phrase in quotation marks. For example, to search for books matching the phrase "spy plane", set

the q parameter to `%22spy+plane%22`. You can also use any of the advanced search operators supported by Book Search. For example, `jane+austen+-inauthor:austen` returns matches that mention (but are not authored by) Jane Austen.

`setStartIndex:` Specifies the index of the first matching result that should be included in the result set. This parameter uses a one-based index, meaning the first result is 1, the second result is 2 and so forth. This parameter works in conjunction with the max-results parameter to determine which results to return. For example, to request the third set of 10 results—results 21-30—set the `start-index` parameter to `21` and the max-results parameter to `10`. Note: This isn't a general cursoring mechanism. If you first send a query with `?start-index=1&max-results=10` and then send another query with `?start-index=11&max-results=10`, the service cannot guarantee that the results are equivalent to `?start-index=1&max-results=20`, because insertions and deletions could have taken place in between the two queries.

`setMaxResults:` Specifies the maximum number of results that should be included in the result set. This parameter works in conjunction with the start-index parameter to determine which results to return. The default value of this parameter is `10` and the maximum value is `20`.

`setMinViewability:` Allows you to filter the results according to the books' viewability status. This parameter accepts one of three values: `'none'` (the default, returning all matching books regardless of viewability), `'partial_view'` (returning only books that the user can preview or view in their entirety), or `'full_view'` (returning only books that the user can view in their entirety).

3.2.1. Partner Co-Branded Search

Google Book Search provides Co-Branded Search, which lets content partners provide full-text search of their books from their own websites.

If you are a partner who wants to do Co-Branded Search using the Book Search Data API, you may do so by modifying the feed URL above to point to your Co-Branded Search implementation. If, for example, a Co-Branded Search is available at the following URL:

```
http://www.google.com/books/p/PARTNER_COBRAND_ID?q=ball
```

then you can obtain the same results using the Book Search Data API at the following URL:

```
http://www.google.com/books/feeds/p/PARTNER_COBRAND_ID/volumes?q=ball+-soccer
```

To specify an alternate URL when querying a volume feed, you can provide an extra parameter to `newVolumeQuery`

```
$query =
    $books->newVolumeQuery('http://www.google.com/books/p/PARTNER_COBRAND_ID');
```

For additional information or support, visit our Partner help center.

3.3. Using community features

3.3.1. Adding a rating

A user can add a rating to a book. Book Search uses a 1-5 rating system in which 1 is the lowest rating. Users cannot update or delete ratings.

To add a rating, add a `Rating` object to a `VolumeEntry` and post it to the annotation feed. In the example below, we start from an empty `VolumeEntry` object.

```
$entry = new Zend_Gdata_Books_VolumeEntry();
$entry->setId(new Zend_Gdata_App_Extension_Id(VOLUME_ID));
$entry->setRating(new Zend_Gdata_Extension_Rating(3, 1, 5, 1));
$books->insertVolume($entry, Zend_Gdata_Books::MY_ANNOTATION_FEED_URI);
```

3.3.2. Reviews

In addition to ratings, authenticated users can submit reviews or edit their reviews. For information on how to request previously submitted reviews, see Retrieving annotations.

3.3.2.1. Adding a review

To add a review, add a `Review` object to a `VolumeEntry` and post it to the annotation feed. In the example below, we start from an existing `VolumeEntry` object.

```
$annotationUrl = $entry->getAnnotationLink()->href;
$review        = new Zend_Gdata_Books_Extension_Review();

$review->setText("This book is amazing!");
$entry->setReview($review);
$books->insertVolume($entry, $annotationUrl);
```

3.3.2.2. Editing a review

To update an existing review, first you retrieve the review you want to update, then you modify it, and then you submit it to the annotation feed.

```
$entryUrl = $entry->getId()->getText();
$review   = new Zend_Gdata_Books_Extension_Review();

$review->setText("This book is actually not that good!");
$entry->setReview($review);
$books->updateVolume($entry, $entryUrl);
```

3.3.3. Labels

You can use the Book Search Data API to label volumes with keywords. A user can submit, retrieve and modify labels. See Retrieving annotations for how to read previously submitted labels.

3.3.3.1. Submitting a set of labels

To submit labels, add a `Category` object with the scheme `LABELS_SCHEME` to a `VolumeEntry` and post it to the annotation feed.

```
$annotationUrl = $entry->getAnnotationLink()->href;
$category      = new Zend_Gdata_App_Extension_Category(
    'rated',
    'http://schemas.google.com/books/2008/labels');
$entry->setCategory(array($category));
$books->insertVolume($entry, Zend_Gdata_Books::MY_ANNOTATION_FEED_URI);
```

3.3.4. Retrieving annotations: reviews, ratings, and labels

You can use the Book Search Data API to retrieve annotations submitted by a given user. Annotations include reviews, ratings, and labels. To retrieve any user's annotations, you can send an unauthenticated request that includes the user's user ID. To retrieve the authenticated user's annotations, use the value `me` as the user ID.

```
$feed = $books->getVolumeFeed(
            'http://www.google.com/books/feeds/users/USER_ID/volumes');
<i>(or)</i>
$feed = $books->getUserAnnotationFeed();

// print title(s) and rating value
foreach ($feed as $entry) {
    foreach ($feed->getTitles() as $title) {
        echo $title;
    }
    if ($entry->getRating()) {
        echo 'Rating: ' . $entry->getRating()->getAverage();
    }
}
```

For a list of the supported query parameters, see the query parameters section.

3.3.5. Deleting Annotations

If you retrieved an annotation entry containing ratings, reviews, and/or labels, you can remove all annotations by calling `deleteVolume` on that entry.

```
$books->deleteVolume($entry);
```

3.4. Book collections and My Library

Google Book Search provides a number of user-specific book collections, each of which has its own feed.

The most important collection is the user's My Library, which represents the books the user would like to remember, organize, and share with others. This is the collection the user sees when accessing his or her My Library page.

3.4.1. Retrieving books in a user's library

The following sections describe how to retrieve a list of books from a user's library, with or without query parameters.

You can query a Book Search public feed without authentication.

3.4.1.1. Retrieving all books in a user's library

To retrieve the user's books, send a query to the My Library feed. To get the library of the authenticated user, use `me` in place of `USER_ID`.

```
$feed = $books->getUserLibraryFeed();
```

Note: The feed may not contain all of the user's books, because there's a default limit on the number of results returned. For more information, see the `max-results` query parameter in Searching for books.

3.4.1.2. Searching for books in a user's library

Just as you can search across all books, you can do a full-text search over just the books in a user's library. To do this, just set the appropriate paramters on the `VolumeQuery` object.

For example, the following query returns all the books in your library that contain the word "bear":

```
$query = $books->newVolumeQuery(
    'http://www.google.com/books/feeds/users' .
    '/USER_ID/collections/library/volumes');
$query->setQuery('bear');
$feed = $books->getVolumeFeed($query);
```

For a list of the supported query parameters, see the query parameters section. In addition, you can search for books that have been labeled by the user:

```
$query = $books->newVolumeQuery(
    'http://www.google.com/books/feeds/users/' .
    'USER_ID/collections/library/volumes');
$query->setCategory(
$query->setCategory('favorites');
$feed = $books->getVolumeFeed($query);
```

3.4.2. Updating books in a user's library

You can use the Book Search Data API to add a book to, or remove a book from, a user's library. Ratings, reviews, and labels are valid across all the collections of a user, and are thus edited using the annotation feed (see Using community features).

3.4.2.1. Adding a book to a library

After authenticating, you can add books to the current user's library.

You can either create an entry from scratch if you know the volume ID, or insert an entry read from any feed.

The following example creates a new entry and adds it to the library:

```
$entry = new Zend_Gdata_Books_VolumeEntry();
$entry->setId(new Zend_Gdata_App_Extension_Id(VOLUME_ID));
$books->insertVolume(
    $entry,
    Zend_Gdata_Books::MY_LIBRARY_FEED_URI
);
```

The following example adds an existing `VolumeEntry` object to the library:

```
$books->insertVolume(
    $entry,
    Zend_Gdata_Books::MY_LIBRARY_FEED_URI
);
```

3.4.2.2. Removing a book from a library

To remove a book from a user's library, call `deleteVolume` on the `VolumeEntry` object.

```
$books->deleteVolume($entry);
```

4. Authenticating with ClientLogin

The ClientLogin mechanism enables you to write PHP application that acquire authenticated access to Google Services, specifying a user's credentials in the Http Client.

See http://code.google.com/apis/accounts/AuthForInstalledApps.html for more information about Google Data ClientLogin authentication.

The Google documentation says the ClientLogin mechanism is appropriate for "installed applications" whereas the AuthSub mechanism is for "web applications." The difference is that AuthSub requires interaction from the user, and a browser interface that can react to redirection requests. The ClientLogin solution uses PHP code to supply the account credentials; the user is not required to enter her credentials interactively.

The account credentials supplied via the ClientLogin mechanism must be valid credentials for Google services, but they are not required to be those of the user who is using the PHP application.

4.1. Creating a ClientLogin authenticated Http Client

The process of creating an authenticated Http client using the ClientLogin mechanism is to call the static function `Zend_Gdata_ClientLogin::getHttpClient()` and pass the Google account credentials in plain text. The return value of this function is an object of class `Zend_Http_Client`.

The optional third parameter is the name of the Google Data service. For instance, this can be 'cl' for Google Calendar. The default is "xapi", which is recognized by Google Data servers as a generic service name.

The optional fourth parameter is an instance of `Zend_Http_Client`. This allows you to set options in the client, such as proxy server settings. If you pass `NULL` for this parameter, a generic `Zend_Http_Client` object is created.

The optional fifth parameter is a short string that Google Data servers use to identify the client application for logging purposes. By default this is string "Zend-ZendFramework";

The optional sixth parameter is a string ID for a CAPTCHA™ challenge that has been issued by the server. It is only necessary when logging in after receiving a CAPTCHA™ challenge from a previous login attempt.

The optional seventh parameter is a user's response to a CAPTCHA™ challenge that has been issued by the server. It is only necessary when logging in after receiving a CAPTCHA™ challenge from a previous login attempt.

Below is an example of PHP code for a web application to acquire authentication to use the Google Calendar service and create a `Zend_Gdata` client object using that authenticated `Zend_Http_Client`.

```
// Enter your Google account credentials
$email = 'johndoe@gmail.com';
$passwd = 'xxxxxxxx';
try {
```

```
    $client = Zend_Gdata_ClientLogin::getHttpClient($email, $passwd, 'cl');
} catch (Zend_Gdata_App_CaptchaRequiredException $cre) {
    echo 'URL of CAPTCHA image: ' . $cre->getCaptchaUrl() . "\n";
    echo 'Token ID: ' . $cre->getCaptchaToken() . "\n";
} catch (Zend_Gdata_App_AuthException $ae) {
    echo 'Problem authenticating: ' . $ae->exception() . "\n";
}

$cal = new Zend_Gdata_Calendar($client);
```

4.2. Terminating a ClientLogin authenticated Http Client

There is no method to revoke ClientLogin authentication as there is in the AuthSub token-based solution. The credentials used in the ClientLogin authentication are the login and password to a Google account, and therefore these can be used repeatedly in the future.

5. Using Google Calendar

You can use the `Zend_Gdata_Calendar` class to view, create, update, and delete events in the online Google Calendar service.

See http://code.google.com/apis/calendar/overview.html for more information about the Google Calendar API.

5.1. Connecting To The Calendar Service

The Google Calendar API, like all GData APIs, is based off of the Atom Publishing Protocol (APP), an XML based format for managing web-based resources. Traffic between a client and the Google Calendar servers occurs over HTTP and allows for both authenticated and unauthenticated connections.

Before any transactions can occur, this connection needs to be made. Creating a connection to the calendar servers involves two steps: creating an HTTP client and binding a `Zend_Gdata_Calendar` service instance to that client.

5.1.1. Authentication

The Google Calendar API allows access to both public and private calendar feeds. Public feeds do not require authentication, but are read-only and offer reduced functionality. Private feeds offers the most complete functionality but requires an authenticated connection to the calendar servers. There are three authentication schemes that are supported by Google Calendar:

- *ClientAuth* provides direct username/password authentication to the calendar servers. Since this scheme requires that users provide your application with their password, this authentication is only recommended when other authentication schemes are insufficient.

- *AuthSub* allows authentication to the calendar servers via a Google proxy server. This provides the same level of convenience as ClientAuth but without the security risk, making this an ideal choice for web-based applications.

- *MagicCookie* allows authentication based on a semi-random URL available from within the Google Calendar interface. This is the simplest authentication scheme to implement, but requires that users manually retrieve their secure URL before they can authenticate, doesn't provide access to calendar lists, and is limited to read-only access.

The Zend_Gdata library provides support for all three authentication schemes. The rest of this chapter will assume that you are familiar the authentication schemes available and how to create an appropriate authenticated connection. For more information, please see section the Authentication section of this manual or the Authentication Overview in the Google Data API Developer's Guide.

5.1.2. Creating A Service Instance

In order to interact with Google Calendar, this library provides the Zend_Gdata_Calendar service class. This class provides a common interface to the Google Data and Atom Publishing Protocol models and assists in marshaling requests to and from the calendar servers.

Once deciding on an authentication scheme, the next step is to create an instance of Zend_Gdata_Calendar. The class constructor takes an instance of Zend_Http_Client as a single argument. This provides an interface for AuthSub and ClientAuth authentication, as both of these require creation of a special authenticated HTTP client. If no arguments are provided, an unauthenticated instance of Zend_Http_Client will be automatically created.

The example below shows how to create a Calendar service class using ClientAuth authentication:

```
// Parameters for ClientAuth authentication
$service = Zend_Gdata_Calendar::AUTH_SERVICE_NAME;
$user = "sample.user@gmail.com";
$pass = "pa$$w0rd";

// Create an authenticated HTTP client
$client = Zend_Gdata_ClientLogin::getHttpClient($user, $pass, $service);

// Create an instance of the Calendar service
$service = new Zend_Gdata_Calendar($client);
```

A Calendar service using AuthSub can be created in a similar, though slightly more lengthy fashion:

```
/*
 * Retrieve the current URL so that the AuthSub server knows where to
 * redirect the user after authentication is complete.
 */
function getCurrentUrl()
{
    global $_SERVER;

    // Filter php_self to avoid a security vulnerability.
    $php_request_uri =
        htmlentities(substr($_SERVER['REQUEST_URI'],
                            0,
                            strcspn($_SERVER['REQUEST_URI'], "\n\r")),
                            ENT_QUOTES);

    if (isset($_SERVER['HTTPS']) &&
        strtolower($_SERVER['HTTPS']) == 'on') {
        $protocol = 'https://';
    } else {
        $protocol = 'http://';
    }
    $host = $_SERVER['HTTP_HOST'];
    if ($_SERVER['HTTP_PORT'] != '' &&
```

```
        (($protocol == 'http://' && $_SERVER['HTTP_PORT'] != '80') ||
        ($protocol == 'https://' && $_SERVER['HTTP_PORT'] != '443'))) {
        $port = ':' . $_SERVER['HTTP_PORT'];
    } else {
        $port = '';
    }
    return $protocol . $host . $port . $php_request_uri;
}

/**
 * Obtain an AuthSub authenticated HTTP client, redirecting the user
 * to the AuthSub server to login if necessary.
 */
function getAuthSubHttpClient()
{
    global $_SESSION, $_GET;

    // if there is no AuthSub session or one-time token waiting for us,
    // redirect the user to the AuthSub server to get one.
    if (!isset($_SESSION['sessionToken']) && !isset($_GET['token'])) {
        // Parameters to give to AuthSub server
        $next = getCurrentUrl();
        $scope = "http://www.google.com/calendar/feeds/";
        $secure = false;
        $session = true;

        // Redirect the user to the AuthSub server to sign in

        $authSubUrl = Zend_Gdata_AuthSub::getAuthSubTokenUri($next,
                                                             $scope,
                                                             $secure,
                                                             $session);
         header("HTTP/1.0 307 Temporary redirect");

         header("Location: " . $authSubUrl);

         exit();
    }

    // Convert an AuthSub one-time token into a session token if needed
    if (!isset($_SESSION['sessionToken']) && isset($_GET['token'])) {
        $_SESSION['sessionToken'] =
            Zend_Gdata_AuthSub::getAuthSubSessionToken($_GET['token']);
    }

    // At this point we are authenticated via AuthSub and can obtain an
    // authenticated HTTP client instance

    // Create an authenticated HTTP client
    $client = Zend_Gdata_AuthSub::getHttpClient($_SESSION['sessionToken']);
    return $client;
}

// -> Script execution begins here <-

// Make sure that the user has a valid session, so we can record the
// AuthSub session token once it is available.
session_start();

// Create an instance of the Calendar service, redirecting the user
```

```
// to the AuthSub server if necessary.
$service = new Zend_Gdata_Calendar(getAuthSubHttpClient());
```

Finally, an unauthenticated server can be created for use with either public feeds or MagicCookie authentication:

```
// Create an instance of the Calendar service using an unauthenticated
// HTTP client

$service = new Zend_Gdata_Calendar();
```

Note that MagicCookie authentication is not supplied with the HTTP connection, but is instead specified along with the desired visibility when submitting queries. See the section on retrieving events below for an example.

5.2. Retrieving A Calendar List

The calendar service supports retrieving a list of calendars for the authenticated user. This is the same list of calendars which are displayed in the Google Calendar UI, except those marked as "`hidden`" are also available.

The calendar list is always private and must be accessed over an authenticated connection. It is not possible to retrieve another user's calendar list and it cannot be accessed using MagicCookie authentication. Attempting to access a calendar list without holding appropriate credentials will fail and result in a 401 (Authentication Required) status code.

```
$service = Zend_Gdata_Calendar::AUTH_SERVICE_NAME;
$client = Zend_Gdata_ClientLogin::getHttpClient($user, $pass, $service);
$service = new Zend_Gdata_Calendar($client);

try {
    $listFeed= $service->getCalendarListFeed();
} catch (Zend_Gdata_App_Exception $e) {
    echo "Error: " . $e->getMessage();
}
```

Calling `getCalendarListFeed()` creates a new instance of `Zend_Gdata_Calendar_ListFeed` containing each available calendar as an instance of `Zend_Gdata_Calendar_ListEntry`. After retrieving the feed, you can use the iterator and accessors contained within the feed to inspect the enclosed calendars.

```
echo "<h1>Calendar List Feed</h1>";
echo "<ul>";
foreach ($listFeed as $calendar) {
    echo "<li>" . $calendar->title .
         " (Event Feed: " . $calendar->id . ")</li>";
}
echo "</ul>";
```

5.3. Retrieving Events

Like the list of calendars, events are also retrieved using the `Zend_Gdata_Calendar` service class. The event list returned is of type `Zend_Gdata_Calendar_EventFeed` and contains each event as an instance of `Zend_Gdata_Calendar_EventEntry`. As before, the iterator and accessors contained within the event feed instance allow inspection of individual events.

5.3.1. Queries

When retrieving events using the Calendar API, specially constructed query URLs are used to describe what events should be returned. The `Zend_Gdata_Calendar_EventQuery` class simplifies this task by automatically constructing a query URL based on provided parameters. A full list of these parameters is available at the Queries section of the Google Data APIs Protocol Reference. However, there are three parameters that are worth special attention:

- *User* is used to specify the user whose calendar is being searched for, and is specified as an email address. If no user is provided, "default" will be used instead to indicate the currently authenticated user (if authenticated).
- *Visibility* specifies whether a users public or private calendar should be searched. If using an unauthenticated session and no MagicCookie is available, only the public feed will be available.
- *Projection* specifies how much data should be returned by the server and in what format. In most cases you will want to use the "full" projection. Also available is the "basic" projection, which places most meta-data into each event's content field as human readable text, and the "composite" projection which includes complete text for any comments alongside each event. The "composite" view is often much larger than the "full" view.

5.3.2. Retrieving Events In Order Of Start Time

The example below illustrates the use of the `Zend_Gdata_Query` class and specifies the private visibility feed, which requires that an authenticated connection is available to the calendar servers. If a MagicCookie is being used for authentication, the visibility should be instead set to "`private-magicCookieValue`", where magicCookieValue is the random string obtained when viewing the private XML address in the Google Calendar UI. Events are requested chronologically by start time and only events occurring in the future are returned.

```
$query = $service->newEventQuery();
$query->setUser('default');
// Set to $query->setVisibility('private-magicCookieValue') if using
// MagicCookie auth
$query->setVisibility('private');
$query->setProjection('full');
$query->setOrderby('starttime');
$query->setFutureevents('true');

// Retrieve the event list from the calendar server
try {
    $eventFeed = $service->getCalendarEventFeed($query);
} catch (Zend_Gdata_App_Exception $e) {
    echo "Error: " . $e->getMessage();
}

// Iterate through the list of events, outputting them as an HTML list
echo "<ul>";
foreach ($eventFeed as $event) {
    echo "<li>" . $event->title . " (Event ID: " . $event->id . ")</li>";
}
echo "</ul>";
```

Additional properties such as ID, author, when, event status, visibility, web content, and content, among others are available within `Zend_Gdata_Calendar_EventEntry`. Refer to the Zend Framework API Documentation and the Calendar Protocol Reference for a complete list.

5.3.3. Retrieving Events In A Specified Date Range

To print out all events within a certain range, for example from December 1, 2006 through December 15, 2007, add the following two lines to the previous sample. Take care to remove "`$query->setFutureevents('true')`", since `futureevents` will override `startMin` and `startMax`.

```
$query->setStartMin('2006-12-01');
$query->setStartMax('2006-12-16');
```

Note that `startMin` is inclusive whereas `startMax` is exclusive. As a result, only events through 2006-12-15 23:59:59 will be returned.

5.3.4. Retrieving Events By Fulltext Query

To print out all events which contain a specific word, for example "dogfood", use the `setQuery()` method when creating the query.

```
$query->setQuery("dogfood");
```

5.3.5. Retrieving Individual Events

Individual events can be retrieved by specifying their event ID as part of the query. Instead of calling `getCalendarEventFeed()`, `getCalendarEventEntry()` should be called instead.

```
$query = $service->newEventQuery();
$query->setUser('default');
$query->setVisibility('private');
$query->setProjection('full');
$query->setEvent($eventId);

try {
    $event = $service->getCalendarEventEntry($query);
} catch (Zend_Gdata_App_Exception $e) {
    echo "Error: " . $e->getMessage();
}
```

In a similar fashion, if the event URL is known, it can be passed directly into `getCalendarEntry()` to retrieve a specific event. In this case, no query object is required since the event URL contains all the necessary information to retrieve the event.

```
$eventURL = "http://www.google.com/calendar/feeds/default/private"
          . "/full/g829on5sq4ag12se91d10uumko";

try {
    $event = $service->getCalendarEventEntry($eventURL);
} catch (Zend_Gdata_App_Exception $e) {
    echo "Error: " . $e->getMessage();
}
```

5.4. Creating Events

5.4.1. Creating Single-Occurrence Events

Events are added to a calendar by creating an instance of `Zend_Gdata_EventEntry` and populating it with the appropriate data. The calendar service instance (`Zend_Gdata_Calendar`)

is then used to used to transparently covert the event into XML and POST it to the calendar server. Creating events requires either an AuthSub or ClientAuth authenticated connection to the calendar server.

At a minimum, the following attributes should be set:

- *Title* provides the headline that will appear above the event within the Google Calendar UI.
- *When* indicates the duration of the event and, optionally, any reminders that are associated with it. See the next section for more information on this attribute.

Other useful attributes that may optionally set include:

- *Author* provides information about the user who created the event.
- *Content* provides additional information about the event which appears when the event details are requested from within Google Calendar.
- *EventStatus* indicates whether the event is confirmed, tentative, or canceled.
- *Hidden* removes the event from the Google Calendar UI.
- *Transparency* indicates whether the event should be consume time on the user's free/busy list.
- *WebContent* allows links to external content to be provided within an event.
- *Where* indicates the location of the event.
- *Visibility* allows the event to be hidden from the public event lists.

For a complete list of event attributes, refer to the Zend Framework API Documentation and the Calendar Protocol Reference. Attributes that can contain multiple values, such as where, are implemented as arrays and need to be created accordingly. Be aware that all of these attributes require objects as parameters. Trying instead to populate them using strings or primitives will result in errors during conversion to XML.

Once the event has been populated, it can be uploaded to the calendar server by passing it as an argument to the calendar service's `insertEvent()` function.

```
// Create a new entry using the calendar service's magic factory method
$event= $service->newEventEntry();

// Populate the event with the desired information
// Note that each attribute is crated as an instance of a matching class
$event->title = $service->newTitle("My Event");
$event->where = array($service->newWhere("Mountain View, California"));
$event->content =
    $service->newContent(" This is my awesome event. RSVP required.");

// Set the date using RFC 3339 format.
$startDate = "2008-01-20";
$startTime = "14:00";
$endDate = "2008-01-20";
$endTime = "16:00";
$tzOffset = "-08";

$when = $service->newWhen();
$when->startTime = "{$startDate}T{$startTime}:00.000{$tzOffset}:00";
```

```
$when->endTime = "{$endDate}T{$endTime}:00.000{$tzOffset}:00";
$event->when = array($when);

// Upload the event to the calendar server
// A copy of the event as it is recorded on the server is returned
$newEvent = $service->insertEvent($event);
```

5.4.2. Event Schedules and Reminders

An event's starting time and duration are determined by the value of its `when` property, which contains the properties `startTime`, `endTime`, and `valueString`. `StartTime` and `EndTime` control the duration of the event, while the `valueString` property is currently unused.

All-day events can be scheduled by specifying only the date omitting the time when setting `startTime` and `endTime`. Likewise, zero-duration events can be specified by omitting the `endTime`. In all cases, date/time values should be provided in RFC3339 format.

```
// Schedule the event to occur on December 05, 2007 at 2 PM PST (UTC-8)
// with a duration of one hour.
$when = $service->newWhen();
$when->startTime = "2007-12-05T14:00:00-08:00";
$when->endTime="2007-12-05T15:00:00:00-08:00";

// Apply the when property to an event
$event->when = array($when);
```

The `when` attribute also controls when reminders are sent to a user. Reminders are stored in an array and each event may have up to find reminders associated with it.

For a `reminder` to be valid, it needs to have two attributes set: `method` and a time. `Method` can accept one of the following strings: "alert", "email", or "sms". The time should be entered as an integer and can be set with either the property `minutes`, `hours`, `days`, or `absoluteTime`. However, a valid request may only have one of these attributes set. If a mixed time is desired, convert to the most precise unit available. For example, 1 hour and 30 minutes should be entered as 90 minutes.

```
// Create a new reminder object. It should be set to send an email
// to the user 10 minutes beforehand.
$reminder = $service->newReminder();
$reminder->method = "email";
$reminder->minutes = "10";

// Apply the reminder to an existing event's when property
$when = $event->when[0];
$when->reminders = array($reminder);
```

5.4.3. Creating Recurring Events

Recurring events are created the same way as single-occurrence events, except a recurrence attribute should be provided instead of a where attribute. The recurrence attribute should hold a string describing the event's recurrence pattern using properties defined in the iCalendar standard (RFC 2445).

Exceptions to the recurrence pattern will usually be specified by a distinct `recurrenceException` attribute. However, the iCalendar standard provides a secondary format for defining recurrences, and the possibility that either may be used must be accounted for.

Due to the complexity of parsing recurrence patterns, further information on this them is outside the scope of this document. However, more information can be found in the Common Elements section of the Google Data APIs Developer Guide, as well as in RFC 2445.

```
// Create a new entry using the calendar service's magic factory method
$event= $service->newEventEntry();

// Populate the event with the desired information
// Note that each attribute is crated as an instance of a matching class
$event->title = $service->newTitle("My Recurring Event");
$event->where = array($service->newWhere("Palo Alto, California"));
$event->content =
    $service->newContent(' This is my other awesome event, ' .
                         ' occurring all-day every Tuesday from .
                         '2007-05-01 until 207-09-04. No RSVP required.');

// Set the duration and frequency by specifying a recurrence pattern.

$recurrence = "DTSTART;VALUE=DATE:20070501\r\n" .
        "DTEND;VALUE=DATE:20070502\r\n" .
        "RRULE:FREQ=WEEKLY;BYDAY=Tu;UNTIL=20070904\r\n";

$event->recurrence = $service->newRecurrence($recurrence);

// Upload the event to the calendar server
// A copy of the event as it is recorded on the server is returned
$newEvent = $service->insertEvent($event);
```

5.4.4. Using QuickAdd

QuickAdd is a feature which allows events to be created using free-form text entry. For example, the string "Dinner at Joe's Diner on Thursday" would create an event with the title "Dinner", location "Joe's Diner", and date "Thursday". To take advantage of QuickAdd, create a new `QuickAdd` property set to `TRUE` and store the freeform text as a `content` property.

```
// Create a new entry using the calendar service's magic factory method
$event= $service->newEventEntry();

// Populate the event with the desired information
$event->content= $service->newContent("Dinner at Joe's Diner on Thursday");
$event->quickAdd = $service->newQuickAdd("true");

// Upload the event to the calendar server
// A copy of the event as it is recorded on the server is returned
$newEvent = $service->insertEvent($event);
```

5.5. Modifying Events

Once an instance of an event has been obtained, the event's attributes can be locally modified in the same way as when creating an event. Once all modifications are complete, calling the event's `save()` method will upload the changes to the calendar server and return a copy of the event as it was created on the server.

In the event another user has modified the event since the local copy was retrieved, `save()` will fail and the server will return a 409 (Conflict) status code. To resolve this a fresh copy of the event must be retrieved from the server before attempting to resubmit any modifications.

```
// Get the first event in the user's event list
$event = $eventFeed[0];

// Change the title to a new value
$event->title = $service->newTitle("Woof!");

// Upload the changes to the server
try {
    $event->save();
} catch (Zend_Gdata_App_Exception $e) {
    echo "Error: " . $e->getMessage();
}
```

5.6. Deleting Events

Calendar events can be deleted either by calling the calendar service's `delete()` method and providing the edit URL of an event or by calling an existing event's own `delete()` method.

In either case, the deleted event will still show up on a user's private event feed if an `updateMin` query parameter is provided. Deleted events can be distinguished from regular events because they will have their `eventStatus` property set to "http://schemas.google.com/g/2005#event.canceled".

```
// Option 1: Events can be deleted directly
$event->delete();
```

```
// Option 2: Events can be deleted supplying the edit URL of the event
// to the calendar service, if known
$service->delete($event->getEditLink()->href);
```

5.7. Accessing Event Comments

When using the full event view, comments are not directly stored within an entry. Instead, each event contains a URL to its associated comment feed which must be manually requested.

Working with comments is fundamentally similar to working with events, with the only significant difference being that a different feed and event class should be used and that the additional meta-data for events such as where and when does not exist for comments. Specifically, the comment's author is stored in the `author` property, and the comment text is stored in the `content` property.

```
// Extract the comment URL from the first event in a user's feed list
$event = $eventFeed[0];
$commentUrl = $event->comments->feedLink->url;

// Retrieve the comment list for the event
try {
$commentFeed = $service->getFeed($commentUrl);
} catch (Zend_Gdata_App_Exception $e) {
    echo "Error: " . $e->getMessage();
}

// Output each comment as an HTML list
echo "<ul>";
foreach ($commentFeed as $comment) {
    echo "<li><em>Comment By: " . $comment->author->name "</em><br/>" .
         $comment->content . "</li>";
```

```
}
echo "</ul>";
```

6. Using Google Documents List Data API

The Google Documents List Data API allows client applications to upload documents to Google Documents and list them in the form of Google Data API ("GData") feeds. Your client application can request a list of a user's documents, and query the content in an existing document.

See http://code.google.com/apis/documents/overview.html for more information about the Google Documents List API.

6.1. Get a List of Documents

You can get a list of the Google Documents for a particular user by using the `getDocumentListFeed()` method of the docs service. The service will return a `Zend_Gdata_Docs_DocumentListFeed` object containing a list of documents associated with the authenticated user.

```
$service = Zend_Gdata_Docs::AUTH_SERVICE_NAME;
$client = Zend_Gdata_ClientLogin::getHttpClient($user, $pass, $service);
$docs = new Zend_Gdata_Docs($client);
$feed = $docs->getDocumentListFeed();
```

The resulting `Zend_Gdata_Docs_DocumentListFeed` object represents the response from the server. This feed contains a list of `Zend_Gdata_Docs_DocumentListEntry` objects (**$feed->entries**), each of which represents a single Google Document.

6.2. Upload a Document

You can create a new Google Document by uploading a word processing document, spreadsheet, or presentation. This example is from the interactive Docs.php sample which comes with the library. It demonstrates uploading a file and printing information about the result from the server.

```
/**
 * Upload the specified document
 *
 * @param Zend_Gdata_Docs $docs The service object to use for communicating
 *     with the Google Documents server.
 * @param boolean $html True if output should be formatted for display in a
 *     web browser.
 * @param string $originalFileName The name of the file to be uploaded. The
 *     MIME type of the file is determined from the extension on this file
 *     name. For example, test.csv is uploaded as a comma separated volume
 *     and converted into a spreadsheet.
 * @param string $temporaryFileLocation (optional) The file in which the
 *     data for the document is stored. This is used when the file has been
 *     uploaded from the client's machine to the server and is stored in
 *     a temporary file which does not have an extension. If this parameter
 *     is null, the file is read from the originalFileName.
 */
function uploadDocument($docs, $html, $originalFileName,
                        $temporaryFileLocation) {
  $fileToUpload = $originalFileName;
```

```
  if ($temporaryFileLocation) {
    $fileToUpload = $temporaryFileLocation;
  }

  // Upload the file and convert it into a Google Document. The original
  // file name is used as the title of the document and the MIME type
  // is determined based on the extension on the original file name.
  $newDocumentEntry = $docs->uploadFile($fileToUpload, $originalFileName,
      null, Zend_Gdata_Docs::DOCUMENTS_LIST_FEED_URI);

  echo "New Document Title: ";

  if ($html) {
      // Find the URL of the HTML view of this document.
      $alternateLink = '';
      foreach ($newDocumentEntry->link as $link) {
          if ($link->getRel() === 'alternate') {
              $alternateLink = $link->getHref();
          }
      }
      // Make the title link to the document on docs.google.com.
      echo "<a href=\"$alternateLink\">\n";
  }
  echo $newDocumentEntry->title."\n";
  if ($html) {echo "</a>\n";}
}
```

6.3. Searching the documents feed

You can search the Document List using some of the standard Google Data API query parameters. Categories are used to restrict the type of document (word processor document, spreadsheet) returned. The full-text query string is used to search the content of all the documents. More detailed information on parameters specific to the Documents List can be found in the Documents List Data API Reference Guide.

6.3.1. Get a List of Word Processing Documents

You can also request a feed containing all of your documents of a specific type. For example, to see a list of your work processing documents, you would perform a category query as follows.

```
$feed = $docs->getDocumentListFeed(
    'http://docs.google.com/feeds/documents/private/full/-/document');
```

6.3.2. Get a List of Spreadsheets

To request a list of your Google Spreadsheets, use the following category query:

```
$feed = $docs->getDocumentListFeed(
    'http://docs.google.com/feeds/documents/private/full/-/spreadsheet');
```

6.3.3. Performing a text query

You can search the content of documents by using a `Zend_Gdata_Docs_Query` in your request. A Query object can be used to construct the query URI, with the search term being passed in as a parameter. Here is an example method which queries the documents list for documents which contain the search string:

```
$docsQuery = new Zend_Gdata_Docs_Query();
$docsQuery->setQuery($query);
$feed = $client->getDocumentListFeed($docsQuery);
```

7. Using Google Health

The Google Health Data API is designed to enable developers to do two things:

- Read a user's Google Health profile or query for medical records that match particular criteria and then use the results to provide personalized functionality based on the data.
- Add new medical records to a user's profile by including CCR data when sending a notice to a user's profile. Note: The CCR data is stored as an XML blob within the <atom> entry. The library does not provide direct accessors to the object model but it does have helpers for extracting specific fields.

There are three main feeds, each of which requires authentication. Unlike other Google Data APIs, each Google Health feed has a limited set of HTTP operations you can perform on it, depending on which authentication method you are using (ClientLogin or AuthSub/OAuth). For a list of permitted operations, see http://code.google.com/apis/health/reference.html#Authentication.

- *Profile Feed* use the profile feed to query a user's health profile for specific information.
- *Register Feed* use the register feed to reconcile new CCR data into a profile.
- *Profile List Feed* the profile list feed should be used to determine which of the user's Health profiles to interact with. This feed is only available when using ClientLogin.

See http://code.google.com/apis/health for more information about the Google Health API.

7.1. Connect To The Health Service

The Google Health API, like all Google Data APIs, is based off of the Atom Publishing Protocol (APP), an XML based format for managing web-based resources. Traffic between a client and the Google Health servers occurs over HTTP and allows for authenticated connections.

Before any transactions can occur, a connection needs to be made. Creating a connection to the Health servers involves two steps: creating an HTTP client and binding a `Zend_Gdata_Health` service instance to that client.

7.1.1. Authentication

The Google Health API allows programmatic access to a user's Health profile. There are three authentication schemes that are supported by Google Health:

- *ClientLogin* provides direct username/password authentication to the Health servers. Since this method requires that users provide your application with their password, this authentication scheme is only recommended for installed/desktop applications.
- *AuthSub* allows a user to authorize the sharing of their private data. This provides the same level of convenience as ClientLogin but without the security risk, making it an ideal choice for web-based applications. For Google Health, AuthSub must be used in registered and secure mode--meaning that all requests to the API must be digitally signed.
- *OAuth* is an alternative to AuthSub. Although this authentication scheme is not discussed in this document, more information can be found in the Health Data API Developer's Guide.

See Authentication Overview in the Google Data API documentation for more information on each authentication method.

7.1.2. Create A Health Service Instance

In order to interact with Google Health, the client library provides the `Zend_Gdata_Health` service class. This class provides a common interface to the Google Data and Atom Publishing Protocol models and assists in marshaling requests to and from the Health API.

Once you've decided on an authentication scheme, the next step is to create an instance of `Zend_Gdata_Health` . This class should be passed an instance of `Zend_Gdata_HttpClient`. This provides an interface for AuthSub/OAuth and ClientLogin to create a special authenticated HTTP client.

To test against the H9 Developer's (/h9) instead of Google Health (/health), the `Zend_Gdata_Health` constructor takes an optional third argument for you to specify the H9 service name 'weaver'.

The example below shows how to create a Health service class using ClientLogin authentication:

```
// Parameters for ClientLogin authentication
$healthServiceName = Zend_Gdata_Health::HEALTH_SERVICE_NAME;
//$h9ServiceName = Zend_Gdata_Health::H9_SANDBOX_SERVICE_NAME;
$user = "user@gmail.com";
$pass = "pa$$w0rd";

// Create an authenticated HTTP client
$client = Zend_Gdata_ClientLogin::getHttpClient($user,
                                                $pass,
                                                $healthServiceName);

// Create an instance of the Health service
$service = new Zend_Gdata_Health($client);
```

A Health service using AuthSub can be created in a similar, though slightly more lengthy fashion. AuthSub is the recommend interface to communicate with Google Health because each token is directly linked to a specific profile in the user's account. Unlike other Google Data APIs, it is required that all requests from your application be digitally signed.

```
/*
 * Retrieve the current URL so that the AuthSub server knows where to
 * redirect the user after authentication is complete.
 */
function getCurrentUrl() {
    $phpRequestUri = htmlentities(substr($_SERVER['REQUEST_URI'],
                                         0,
                                         strcspn($_SERVER['REQUEST_URI'],
                                                 "\n\r")),
                                  ENT_QUOTES);

    if (isset($_SERVER['HTTPS']) && strtolower($_SERVER['HTTPS']) == 'on') {
        $protocol = 'https://';
    } else {
        $protocol = 'http://';
    }
    $host = $_SERVER['HTTP_HOST'];
    if ($_SERVER['SERVER_PORT'] != '' &&
       (($protocol == 'http://' && $_SERVER['SERVER_PORT'] != '80') ||
```

```
       ($protocol == 'https://' && $_SERVER['SERVER_PORT'] != '443'))) {
        $port = ':' . $_SERVER['SERVER_PORT'];
    } else {
        $port = '';
    }
    return $protocol . $host . $port . $phpRequestUri;
}

/*
 * Redirect a user to AuthSub if they do not have a valid session token.
 * If they're coming back from AuthSub with a single-use token, instantiate
 * a new HTTP client and exchange the token for a long-lived session token
 * instead.
 */
function setupClient($singleUseToken = null) {
    $client = null;

    // Fetch a new AuthSub token?
    if (!$singleUseToken) {
        $next = getCurrentUrl();
        $scope = 'https://www.google.com/health/feeds';
        $authSubHandler = 'https://www.google.com/health/authsub';
        $secure = 1;
        $session = 1;
        $authSubURL =  Zend_Gdata_AuthSub::getAuthSubTokenUri($next,
                                                              $scope,
                                                              $secure,
                                                              $session,
                                                              $authSubHandler);

         // 1 - allows posting notices && allows reading profile data
        $permission = 1;
        $authSubURL .= '&permission=' . $permission;

        echo '<a href="' . $authSubURL . '">Your Google Health Account</a>';
    } else {
        $client = new Zend_Gdata_HttpClient();

        // This sets your private key to be used to sign subsequent requests
        $client->setAuthSubPrivateKeyFile('/path/to/your/rsa_private_key.pem',
                                          null,
                                          true);

        $sessionToken =
            Zend_Gdata_AuthSub::getAuthSubSessionToken(trim($singleUseToken),
                                                       $client);

        // Set the long-lived session token for subsequent requests
        $client->setAuthSubToken($sessionToken);
    }
    return $client;
}

// -> Script execution begins here <-

session_start();

$client = setupClient(@$_GET['token']);

// Create an instance of the Health service
```

```
$userH9Sandbox = false;
$healthService = new Zend_Gdata_Health($client,
                                       'googleInc-MyTestAppName-v1.0',
                                       $userH9Sandbox);
```

NOTE: the remainder of this document will assume you are using AuthSub for authentication.

7.2. Profile Feed

To query the user's profile feed, make sure your initial AuthSub token was requested with the `permission=1` parameter set. The process of extracting data from the profile requires two steps, sending a query and iterating through the resulting feed.

7.2.1. Send a Structured Query

You can send structured queries to retrieve specific records from a user's profile.

When retrieving the profile using the Health API, specifically constructed query URLs are used to describe what (CCR) data should be returned. The `Zend_Gdata_Health_Query` class helps simplify this task by automatically constructing a query URL based on the parameters you set.

7.2.1.1. Query The Feed

To execute a query against the profile feed, invoke a new instance of an `Zend_Gdata_Health_Query` and call the service's `getHealthProfileFeed()` method:

```
$healthService = new Zend_Gdata_Health($client);

// example query for the top 10 medications with 2 items each
$query = new Zend_Gdata_Health_Query();
$query->setDigest("true");
$query->setGrouped("true");
$query->setMaxResultsGroup(10);
$query->setMaxResultsInGroup(2);
$query->setCategory("medication");

$profileFeed = $healthService->getHealthProfileFeed($query);
```

Using `setDigest("true")` returns all of user's CCR data in a single Atom `<entry>`.

The `setCategory()` helper can be passed an additional parameter to return more specific CCR information. For example, to return just the medication Lipitor, use `setCategory("medication", "Lipitor")`. The same methodology can be applied to other categories such as conditions, allergies, lab results, etc.

A full list of supported query parameters is available in the query parameters section of the Health API Reference Guide.

7.2.2. Iterate Through The Profile Entries

Each Google Health entry contains CCR data, however, using the `digest=true` query parameter will consolidate all of the CCR elements (that match your query) into a single Atom `<entry>`.

To retrieve the full CCR information from an entry, make a call to the `Zend_Gdata_Health_ProfileEntry` class's `getCcr()` method. That returns a `Zend_Gdata_Health_Extension_CCR`:

```
$entries = $profileFeed->getEntries();
foreach ($entries as $entry) {
    $medications = $entry->getCcr()->getMedications();
    //$conditions = $entry->getCcr()->getConditions();
    //$immunizations = $entry->getCcr()->getImmunizations();

    // print the CCR xml (this will just be the entry's medications)
    foreach ($medications as $med) {
        $xmlStr = $med->ownerDocument->saveXML($med);
        echo "<pre>" . $xmlStr . "</pre>";
    }
}
```

Here, the `getCcr()` method is used in conjunction with a magic helper to drill down and extract just the medication data from the entry's CCR. The formentioned magic helper takes the form `getCATEGORYNAME()`, where `CATEGORYNAME` is a supported Google Health category. See the Google Health reference Guide for the possible categories.

To be more efficient, you can also use category queries to only return the necessary CCR from the Google Health servers. Then, iterate through those results:

```
$query = new Zend_Gdata_Health_Query();
$query->setDigest("true");
$query->setCategory("condition");
$profileFeed = $healthService->getHealthProfileFeed($query);

// Since the query contained digest=true, only one Atom entry is returned
$entry = $profileFeed->entry[0];
$conditions = $entry->getCcr()->getConditions();

// print the CCR xml (this will just be the profile's conditions)
foreach ($conditions as $cond) {
    $xmlStr = $cond->ownerDocument->saveXML($cond);
    echo "<pre>" . $xmlStr . "</pre>";
}
```

7.3. Profile List Feed

NOTE: This feed is only available when using ClientLogin

Since ClientLogin requires a profile ID with each of its feeds, applications will likely want to query this feed first in order to select the appropriate profile. The profile list feed returns Atom entries corresponding each profile in the user's Google Health account. The profile ID is returned in the Atom `<content>` and the profile name in the `<title>` element.

7.3.1. Query The Feed

To execute a query against the profile list feed, call the service's `getHealthProfileListFeed()` method:

```
$client = Zend_Gdata_ClientLogin::getHttpClient('user@gmail.com',
                                                'pa$$word',
                                                'health');
$healthService = new Zend_Gdata_Health($client);
$feed = $healthService->getHealthProfileListFeed();

// print each profile's name and id
```

```
$entries = $feed->getEntries();
foreach ($entries as $entry) {
    echo '<p>Profile name: ' . $entry->getProfileName() . '<br>';
    echo 'profile ID: ' . $entry->getProfileID() . '</p>';
}
```

Once you've determined which profile to use, call `setProfileID()` with the profileID as an argument. This will restrict subsequent API requests to be against that particular profile:

```
// use the first profile
$profileID = $feed->entry[0]->getProfileID();
$healthService->setProfileID($profileID);

$profileFeed = $healthService->getHealthProfileFeed();

$profileID = $healthService->getProfileID();
echo '<p><b>Queried profileID</b>: ' . $profileID . '</p>';
```

7.4. Sending Notices to the Register Feed

Individual posts to the register feed are known as notices. Notice are sent from third-party applications to inform the user of a new event. With AuthSub/OAuth, notices are the single means by which your application can add new CCR information into a user's profile. Notices can contain plain text (including certain XHTML elements), a CCR document, or both. As an example, notices might be sent to remind users to pick up a prescription, or they might contain lab results in the CCR format.

7.4.1. Sending a notice

Notices can be sent by using the `sendHealthNotice()` method for the Health service:

```
$healthService = new Zend_Gdata_Health($client);

$subject = "Title of your notice goes here";
$body = "Notice body can contain <b>html</b> entities";
$ccr = '<ContinuityOfCareRecord xmlns="urn:astm-org:CCR">
  <Body>
   <Problems>
    <Problem>
      <DateTime>
        <Type><Text>Start date</Text></Type>
        <ExactDateTime>2007-04-04T07:00:00Z</ExactDateTime>
      </DateTime>
      <Description>
        <Text>Aortic valve disorders</Text>
        <Code>
          <Value>410.10</Value>
          <CodingSystem>ICD9</CodingSystem>
          <Version>2004</Version>
        </Code>
      </Description>
      <Status><Text>Active</Text></Status>
    </Problem>
  </Problems>
  </Body>
</ContinuityOfCareRecord>';
```

```
$responseEntry = $healthService->sendHealthNotice($subject,
                                                  $body,
                                                  "html",
                                                  $ccr);
```

8. Using Google Spreadsheets

The Google Spreadsheets data API allows client applications to view and update Spreadsheets content in the form of Google data API feeds. Your client application can request a list of a user's spreadsheets, edit or delete content in an existing Spreadsheets worksheet, and query the content in an existing Spreadsheets worksheet.

See http://code.google.com/apis/spreadsheets/overview.html for more information about the Google Spreadsheets API.

8.1. Create a Spreadsheet

The Spreadsheets data API does not currently provide a way to programmatically create or delete a spreadsheet.

8.2. Get a List of Spreadsheets

You can get a list of spreadsheets for a particular user by using the `getSpreadsheetFeed` method of the Spreadsheets service. The service will return a `Zend_Gdata_Spreadsheets_SpreadsheetFeed` object containing a list of spreadsheets associated with the authenticated user.

```
$service = Zend_Gdata_Spreadsheets::AUTH_SERVICE_NAME;
$client = Zend_Gdata_ClientLogin::getHttpClient($user, $pass, $service);
$spreadsheetService = new Zend_Gdata_Spreadsheets($client);
$feed = $spreadsheetService->getSpreadsheetFeed();
```

8.3. Get a List of Worksheets

A given spreadsheet may contain multiple worksheets. For each spreadsheet, there's a worksheets metafeed listing all the worksheets in that spreadsheet.

Given the spreadsheet key from the <id> of a `Zend_Gdata_Spreadsheets_SpreadsheetEntry` object you've already retrieved, you can fetch a feed containing a list of worksheets associated with that spreadsheet.

```
$query = new Zend_Gdata_Spreadsheets_DocumentQuery();
$query->setSpreadsheetKey($spreadsheetKey);
$feed = $spreadsheetService->getWorksheetFeed($query);
```

The resulting `Zend_Gdata_Spreadsheets_WorksheetFeed` object feed represents the response from the server. Among other things, this feed contains a list of `Zend_Gdata_Spreadsheets_WorksheetEntry` objects (`$feed->entries`), each of which represents a single worksheet.

8.4. Interacting With List-based Feeds

A given worksheet generally contains multiple rows, each containing multiple cells. You can request data from the worksheet either as a list-based feed, in which each entry represents a

row, or as a cell-based feed, in which each entry represents a single cell. For information on cell-based feeds, see Interacting with cell-based feeds.

The following sections describe how to get a list-based feed, add a row to a worksheet, and send queries with various query parameters.

The list feed makes some assumptions about how the data is laid out in the spreadsheet.

In particular, the list feed treats the first row of the worksheet as a header row; Spreadsheets dynamically creates XML elements named after the contents of header-row cells. Users who want to provide Gdata feeds should not put any data other than column headers in the first row of a worksheet.

The list feed contains all rows after the first row up to the first blank row. The first blank row terminates the data set. If expected data isn't appearing in a feed, check the worksheet manually to see whether there's an unexpected blank row in the middle of the data. In particular, if the second row of the spreadsheet is blank, then the list feed will contain no data.

A row in a list feed is as many columns wide as the worksheet itself.

8.4.1. Get a List-based Feed

To retrieve a worksheet's list feed, use the `getListFeed` method of the Spreadsheets service.

```
$query = new Zend_Gdata_Spreadsheets_ListQuery();
$query->setSpreadsheetKey($spreadsheetKey);
$query->setWorksheetId($worksheetId);
$listFeed = $spreadsheetService->getListFeed($query);
```

The resulting `Zend_Gdata_Spreadsheets_ListFeed` object `$listfeed` represents a response from the server. Among other things, this feed contains an array of `Zend_Gdata_Spreadsheets_ListEntry` objects (`$listFeed->entries`), each of which represents a single row in a worksheet.

Each `Zend_Gdata_Spreadsheets_ListEntry` contains an array, `custom`, which contains the data for that row. You can extract and display this array:

```
$rowData = $listFeed->entries[1]->getCustom();
foreach($rowData as $customEntry) {
  echo $customEntry->getColumnName() . " = " . $customEntry->getText();
}
```

An alternate version of this array, `customByName`, allows direct access to an entry's cells by name. This is convenient when trying to access a specific header:

```
$customEntry = $listFeed->entries[1]->getCustomByName('my_heading');
echo $customEntry->getColumnName() . " = " . $customEntry->getText();
```

8.4.2. Reverse-sort Rows

By default, rows in the feed appear in the same order as the corresponding rows in the GUI; that is, they're in order by row number. To get rows in reverse order, set the reverse properties of the `Zend_Gdata_Spreadsheets_ListQuery` object to `TRUE`:

```
$query = new Zend_Gdata_Spreadsheets_ListQuery();
$query->setSpreadsheetKey($spreadsheetKey);
```

```
$query->setWorksheetId($worksheetId);
$query->setReverse('true');
$listFeed = $spreadsheetService->getListFeed($query);
```

Note that if you want to order (or reverse sort) by a particular column, rather than by position in the worksheet, you can set the `orderby` value of the `Zend_Gdata_Spreadsheets_ListQuery` object to `column:<the header of that column>`.

8.4.3. Send a Structured Query

You can set a `Zend_Gdata_Spreadsheets_ListQuery`'s `sq` value to produce a feed with entries that meet the specified criteria. For example, suppose you have a worksheet containing personnel data, in which each row represents information about a single person. You wish to retrieve all rows in which the person's name is "John" and the person's age is over 25. To do so, you would set `sq` as follows:

```
$query = new Zend_Gdata_Spreadsheets_ListQuery();
$query->setSpreadsheetKey($spreadsheetKey);
$query->setWorksheetId($worksheetId);
$query->setSpreadsheetQuery('name=John and age>25');
$listFeed = $spreadsheetService->getListFeed($query);
```

8.4.4. Add a Row

Rows can be added to a spreadsheet by using the `insertRow` method of the Spreadsheet service.

```
$insertedListEntry = $spreadsheetService->insertRow($rowData,
                                                    $spreadsheetKey,
                                                    $worksheetId);
```

The `$rowData` parameter contains an array of column keys to data values. The method returns a `Zend_Gdata_Spreadsheets_SpreadsheetsEntry` object which represents the inserted row.

Spreadsheets inserts the new row immediately after the last row that appears in the list-based feed, which is to say immediately before the first entirely blank row.

8.4.5. Edit a Row

Once a `Zend_Gdata_Spreadsheets_ListEntry` object is fetched, its rows can be updated by using the `updateRow` method of the Spreadsheet service.

```
$updatedListEntry = $spreadsheetService->updateRow($oldListEntry,
                                                   $newRowData);
```

The `$oldListEntry` parameter contains the list entry to be updated. `$newRowData` contains an array of column keys to data values, to be used as the new row data. The method returns a `Zend_Gdata_Spreadsheets_SpreadsheetsEntry` object which represents the updated row.

8.4.6. Delete a Row

To delete a row, simply invoke `deleteRow` on the `Zend_Gdata_Spreadsheets` object with the existing entry to be deleted:

```
$spreadsheetService->deleteRow($listEntry);
```

Alternatively, you can call the `delete` method of the entry itself:

```
$listEntry->delete();
```

8.5. Interacting With Cell-based Feeds

In a cell-based feed, each entry represents a single cell.

Note that we don't recommend interacting with both a cell-based feed and a list-based feed for the same worksheet at the same time.

8.5.1. Get a Cell-based Feed

To retrieve a worksheet's cell feed, use the `getCellFeed` method of the Spreadsheets service.

```
$query = new Zend_Gdata_Spreadsheets_CellQuery();
$query->setSpreadsheetKey($spreadsheetKey);
$query->setWorksheetId($worksheetId);
$cellFeed = $spreadsheetService->getCellFeed($query);
```

The resulting `Zend_Gdata_Spreadsheets_CellFeed` object `$cellFeed` represents a response from the server. Among other things, this feed contains an array of `Zend_Gdata_Spreadsheets_CellEntry` objects (`$cellFeed>entries`), each of which represents a single cell in a worksheet. You can display this information:

```
foreach($cellFeed as $cellEntry) {
  $row = $cellEntry->cell->getRow();
  $col = $cellEntry->cell->getColumn();
  $val = $cellEntry->cell->getText();
  echo "$row, $col = $val\n";
}
```

8.5.2. Send a Cell Range Query

Suppose you wanted to retrieve the cells in the first column of a worksheet. You can request a cell feed containing only this column as follows:

```
$query = new Zend_Gdata_Spreadsheets_CellQuery();
$query->setMinCol(1);
$query->setMaxCol(1);
$query->setMinRow(2);
$feed = $spreadsheetService->getCellsFeed($query);
```

This requests all the data in column 1, starting with row 2.

8.5.3. Change Contents of a Cell

To modify the contents of a cell, call `updateCell` with the row, column, and new value of the cell.

```
$updatedCell = $spreadsheetService->updateCell($row,
                                               $col,
                                               $inputValue,
```

```
                                                $spreadsheetKey,
                                                $worksheetId);
```

The new data is placed in the specified cell in the worksheet. If the specified cell contains data already, it will be overwritten. Note: Use `updateCell` to change the data in a cell, even if the cell is empty.

9. Using Google Apps Provisioning

Google Apps is a service which allows domain administrators to offer their users managed access to Google services such as Mail, Calendar, and Docs & Spreadsheets. The Provisioning API offers a programmatic interface to configure this service. Specifically, this API allows administrators the ability to create, retrieve, update, and delete user accounts, nicknames, and email lists.

This library implements version 2.0 of the Provisioning API. Access to your account via the Provisioning API must be manually enabled for each domain using the Google Apps control panel. Only certain account types are able to enable this feature.

For more information on the Google Apps Provisioning API, including instructions for enabling API access, refer to the Provisioning API V2.0 Reference.

Authentication

The Provisioning API does not support authentication via AuthSub and anonymous access is not permitted. All HTTP connections must be authenticated using ClientAuth authentication.

9.1. Setting the current domain

In order to use the Provisioning API, the domain being administered needs to be specified in all request URIs. In order to ease development, this information is stored within both the Gapps service and query classes to use when constructing requests.

9.1.1. Setting the domain for the service class

To set the domain for requests made by the service class, either call `setDomain()` or specify the domain when instantiating the service class. For example:

```
$domain = "example.com";
$gdata = new Zend_Gdata_Gapps($client, $domain);
```

9.1.2. Setting the domain for query classes

Setting the domain for requests made by query classes is similar to setting it for the service class- either call `setDomain()` or specify the domain when creating the query. For example:

```
$domain = "example.com";
$query = new Zend_Gdata_Gapps_UserQuery($domain, $arg);
```

When using a service class factory method to create a query, the service class will automatically set the query's domain to match its own domain. As a result, it is not necessary to specify the domain as part of the constructor arguments.

```
$domain = "example.com";
$gdata = new Zend_Gdata_Gapps($client, $domain);
$query = $gdata->newUserQuery($arg);
```

9.2. Interacting with users

Each user account on a Google Apps hosted domain is represented as an instance of `Zend_Gdata_Gapps_UserEntry`. This class provides access to all account properties including name, username, password, access rights, and current quota.

9.2.1. Creating a user account

User accounts can be created by calling the `createUser()` convenience method:

```
$gdata->createUser('foo', 'Random', 'User', '••••••••');
```

Users can also be created by instantiating UserEntry, providing a username, given name, family name, and password, then calling `insertUser()` on a service object to upload the entry to the server.

```
$user = $gdata->newUserEntry();
$user->login = $gdata->newLogin();
$user->login->username = 'foo';
$user->login->password = '••••••••';
$user->name = $gdata->newName();
$user->name->givenName = 'Random';
$user->name->familyName = 'User';
$user = $gdata->insertUser($user);
```

The user's password should normally be provided as cleartext. Optionally, the password can be provided as an SHA-1 digest if `login->passwordHashFunction` is set to 'SHA-1'.

9.2.2. Retrieving a user account

Individual user accounts can be retrieved by calling the `retrieveUser()` convenience method. If the user is not found, `NULL` will be returned.

```
$user = $gdata->retrieveUser('foo');

echo 'Username: ' . $user->login->userName . "\n";
echo 'Given Name: ' . $user->name->givenName . "\n";
echo 'Family Name: ' . $user->name->familyName . "\n";
echo 'Suspended: ' . ($user->login->suspended ? 'Yes' : 'No') . "\n";
echo 'Admin: ' . ($user->login->admin ? 'Yes' : 'No') . "\n"
echo 'Must Change Password: ' .
     ($user->login->changePasswordAtNextLogin ? 'Yes' : 'No') . "\n";
echo 'Has Agreed To Terms: ' .
     ($user->login->agreedToTerms ? 'Yes' : 'No') . "\n";
```

Users can also be retrieved by creating an instance of `Zend_Gdata_Gapps_UserQuery`, setting its username property to equal the username of the user that is to be retrieved, and calling `getUserEntry()` on a service object with that query.

```
$query = $gdata->newUserQuery('foo');
$user = $gdata->getUserEntry($query);
```

```
echo 'Username: ' . $user->login->userName . "\n";
echo 'Given Name: ' . $user->login->givenName . "\n";
echo 'Family Name: ' . $user->login->familyName . "\n";
echo 'Suspended: ' . ($user->login->suspended ? 'Yes' : 'No') . "\n";
echo 'Admin: ' . ($user->login->admin ? 'Yes' : 'No') . "\n"
echo 'Must Change Password: ' .
     ($user->login->changePasswordAtNextLogin ? 'Yes' : 'No') . "\n";
echo 'Has Agreed To Terms: ' .
     ($user->login->agreedToTerms ? 'Yes' : 'No') . "\n";
```

If the specified user cannot be located a ServiceException will be thrown with an error code of `Zend_Gdata_Gapps_Error::ENTITY_DOES_NOT_EXIST`. ServiceExceptions will be covered in Section 9.6, “Handling errors”.

9.2.3. Retrieving all users in a domain

To retrieve all users in a domain, call the `retrieveAllUsers()` convenience method.

```
$feed = $gdata->retrieveAllUsers();

foreach ($feed as $user) {
    echo "  * " . $user->login->username . ' (' . $user->name->givenName .
        ' ' . $user->name->familyName . ")\n";
}
```

This will create a `Zend_Gdata_Gapps_UserFeed` object which holds each user on the domain.

Alternatively, call `getUserFeed()` with no options. Keep in mind that on larger domains this feed may be paged by the server. For more information on paging, see Section 1.9, “Working with Multi-page Feeds”.

```
$feed = $gdata->getUserFeed();

foreach ($feed as $user) {
    echo "  * " . $user->login->username . ' (' . $user->name->givenName .
        ' ' . $user->name->familyName . ")\n";
}
```

9.2.4. Updating a user account

The easiest way to update a user account is to retrieve the user as described in the previous sections, make any desired changes, then call `save()` on that user. Any changes made will be propagated to the server.

```
$user = $gdata->retrieveUser('foo');
$user->name->givenName = 'Foo';
$user->name->familyName = 'Bar';
$user = $user->save();
```

9.2.4.1. Resetting a user's password

A user's password can be reset to a new value by updating the `login->password` property.

```
$user = $gdata->retrieveUser('foo');
$user->login->password = '••••••••';
$user = $user->save();
```

Note that it is not possible to recover a password in this manner as stored passwords are not made available via the Provisioning API for security reasons.

9.2.4.2. Forcing a user to change their password

A user can be forced to change their password at their next login by setting the `login->changePasswordAtNextLogin` property to `TRUE`.

```
$user = $gdata->retrieveUser('foo');
$user->login->changePasswordAtNextLogin = true;
$user = $user->save();
```

Similarly, this can be undone by setting the `login->changePasswordAtNextLogin` property to `FALSE`.

9.2.4.3. Suspending a user account

Users can be restricted from logging in without deleting their user account by instead *suspending* their user account. Accounts can be suspended or restored by using the `suspendUser()` and `restoreUser()` convenience methods:

```
$gdata->suspendUser('foo');
$gdata->restoreUser('foo');
```

Alternatively, you can set the UserEntry's `login->suspended` property to `TRUE`.

```
$user = $gdata->retrieveUser('foo');
$user->login->suspended = true;
$user = $user->save();
```

To restore the user's access, set the `login->suspended` property to `FALSE`.

9.2.4.4. Granting administrative rights

Users can be granted the ability to administer your domain by setting their `login->admin` property to `TRUE`.

```
$user = $gdata->retrieveUser('foo');
$user->login->admin = true;
$user = $user->save();
```

And as expected, setting a user's `login->admin` property to `FALSE` revokes their administrative rights.

9.2.5. Deleting user accounts

Deleting a user account to which you already hold a UserEntry is a simple as calling `delete()` on that entry.

```
$user = $gdata->retrieveUser('foo');
$user->delete();
```

If you do not have access to a UserEntry object for an account, use the `deleteUser()` convenience method.

```
$gdata->deleteUser('foo');
```

9.3. Interacting with nicknames

Nicknames serve as email aliases for existing users. Each nickname contains precisely two key properties: its name and its owner. Any email addressed to a nickname is forwarded to the user who owns that nickname.

Nicknames are represented as an instances of `Zend_Gdata_Gapps_NicknameEntry`.

9.3.1. Creating a nickname

Nicknames can be created by calling the `createNickname()` convenience method:

```
$gdata->createNickname('foo', 'bar');
```

Nicknames can also be created by instantiating NicknameEntry, providing the nickname with a name and an owner, then calling `insertNickname()` on a service object to upload the entry to the server.

```
$nickname = $gdata->newNicknameEntry();
$nickname->login = $gdata->newLogin('foo');
$nickname->nickname = $gdata->newNickname('bar');
$nickname = $gdata->insertNickname($nickname);
```

9.3.2. Retrieving a nickname

Nicknames can be retrieved by calling the `retrieveNickname()` convenience method. This will return `NULL` if a user is not found.

```
$nickname = $gdata->retrieveNickname('bar');

echo 'Nickname: ' . $nickname->nickname->name . "\n";
echo 'Owner: ' . $nickname->login->username . "\n";
```

Individual nicknames can also be retrieved by creating an instance of `Zend_Gdata_Gapps_NicknameQuery`, setting its nickname property to equal the nickname that is to be retrieved, and calling `getNicknameEntry()` on a service object with that query.

```
$query = $gdata->newNicknameQuery('bar');
$nickname = $gdata->getNicknameEntry($query);

echo 'Nickname: ' . $nickname->nickname->name . "\n";
echo 'Owner: ' . $nickname->login->username . "\n";
```

As with users, if no corresponding nickname is found a ServiceException will be thrown with an error code of `Zend_Gdata_Gapps_Error::ENTITY_DOES_NOT_EXIST`. Again, these will be discussed in Section 9.6, “Handling errors”.

9.3.3. Retrieving all nicknames for a user

To retrieve all nicknames associated with a given user, call the convenience method `retrieveNicknames()`.

```
$feed = $gdata->retrieveNicknames('foo');
```

```
foreach ($feed as $nickname) {
    echo '  * ' . $nickname->nickname->name . "\n";
}
```

This will create a `Zend_Gdata_Gapps_NicknameFeed` object which holds each nickname associated with the specified user.

Alternatively, create a new `Zend_Gdata_Gapps_NicknameQuery`, set its username property to the desired user, and submit the query by calling `getNicknameFeed()` on a service object.

```
$query = $gdata->newNicknameQuery();
$query->setUsername('foo');
$feed = $gdata->getNicknameFeed($query);

foreach ($feed as $nickname) {
    echo '  * ' . $nickname->nickname->name . "\n";
}
```

9.3.4. Retrieving all nicknames in a domain

To retrieve all nicknames in a feed, simply call the convenience method `retrieveAllNicknames()`

```
$feed = $gdata->retrieveAllNicknames();

foreach ($feed as $nickname) {
    echo '  * ' . $nickname->nickname->name . ' => ' .
        $nickname->login->username . "\n";
}
```

This will create a `Zend_Gdata_Gapps_NicknameFeed` object which holds each nickname on the domain.

Alternatively, call `getNicknameFeed()` on a service object with no arguments.

```
$feed = $gdata->getNicknameFeed();

foreach ($feed as $nickname) {
    echo '  * ' . $nickname->nickname->name . ' => ' .
        $nickname->login->username . "\n";
}
```

9.3.5. Deleting a nickname

Deleting a nickname to which you already hold a NicknameEntry for is a simple as calling `delete()` on that entry.

```
$nickname = $gdata->retrieveNickname('bar');
$nickname->delete();
```

For nicknames which you do not hold a NicknameEntry for, use the `deleteNickname()` convenience method.

```
$gdata->deleteNickname('bar');
```

9.4. Interacting with email lists

Email lists allow several users to retrieve email addressed to a single email address. Users do not need to be a member of this domain in order to subscribe to an email list provided their complete email address (including domain) is used.

Each email list on a domain is represented as an instance of `Zend_Gdata_Gapps_EmailListEntry`.

9.4.1. Creating an email list

Email lists can be created by calling the `createEmailList()` convenience method:

```
$gdata->createEmailList('friends');
```

Email lists can also be created by instantiating EmailListEntry, providing a name for the list, then calling `insertEmailList()` on a service object to upload the entry to the server.

```
$list = $gdata->newEmailListEntry();
$list->emailList = $gdata->newEmailList('friends');
$list = $gdata->insertEmailList($list);
```

9.4.2. Retrieving all email lists to which a recipient is subscribed

To retrieve all email lists to which a particular recipient is subscribed, call the `retrieveEmailLists()` convenience method:

```
$feed = $gdata->retrieveEmailLists('baz@somewhere.com');

foreach ($feed as $list) {
    echo '  * ' . $list->emailList->name . "\n";
}
```

This will create a `Zend_Gdata_Gapps_EmailListFeed` object which holds each email list associated with the specified recipient.

Alternatively, create a new `Zend_Gdata_Gapps_EmailListQuery`, set its recipient property to the desired email address, and submit the query by calling `getEmailListFeed()` on a service object.

```
$query = $gdata->newEmailListQuery();
$query->setRecipient('baz@somewhere.com');
$feed = $gdata->getEmailListFeed($query);

foreach ($feed as $list) {
    echo '  * ' . $list->emailList->name . "\n";
}
```

9.4.3. Retrieving all email lists in a domain

To retrieve all email lists in a domain, call the convenience method `retrieveAllEmailLists()`.

```
$feed = $gdata->retrieveAllEmailLists();
```

```
foreach ($feed as $list) {
    echo '  * ' . $list->emailList->name . "\n";
}
```

This will create a `Zend_Gdata_Gapps_EmailListFeed` object which holds each email list on the domain.

Alternatively, call `getEmailListFeed()` on a service object with no arguments.

```
$feed = $gdata->getEmailListFeed();

foreach ($feed as $list) {
    echo '  * ' . $list->emailList->name . "\n";
}
```

9.4.4. Deleting an email list

To delete an email list, call the deleteEmailList() convenience method:

```
$gdata->deleteEmailList('friends');
```

9.5. Interacting with email list recipients

Each recipient subscribed to an email list is represented by an instance of `Zend_Gdata_Gapps_EmailListRecipient`. Through this class, individual recipients can be added and removed from email lists.

9.5.1. Adding a recipient to an email list

To add a recipient to an email list, simply call the `addRecipientToEmailList()` convenience method:

```
$gdata->addRecipientToEmailList('bar@somewhere.com', 'friends');
```

9.5.2. Retrieving the list of subscribers to an email list

The convenience method `retrieveAllRecipients()` can be used to retrieve the list of subscribers to an email list:

```
$feed = $gdata->retrieveAllRecipients('friends');

foreach ($feed as $recipient) {
    echo '  * ' . $recipient->who->email . "\n";
}
```

Alternatively, construct a new EmailListRecipientQuery, set its emailListName property to match the desired email list, and call `getEmailListRecipientFeed()` on a service object.

```
$query = $gdata->newEmailListRecipientQuery();
$query->setEmailListName('friends');
$feed = $gdata->getEmailListRecipientFeed($query);
```

```
foreach ($feed as $recipient) {
    echo '  * ' . $recipient->who->email . "\n";
}
```

This will create a `Zend_Gdata_Gapps_EmailListRecipientFeed` object which holds each recipient for the selected email list.

9.5.3. Removing a recipient from an email list

To remove a recipient from an email list, call the `removeRecipientFromEmailList()` convenience method:

```
$gdata->removeRecipientFromEmailList('baz@somewhere.com', 'friends');
```

9.6. Handling errors

In addition to the standard suite of exceptions thrown by `Zend_Gdata`, requests using the Provisioning API may also throw a `Zend_Gdata_Gapps_ServiceException`. These exceptions indicate that a API specific error occurred which prevents the request from completing.

Each ServiceException instance may hold one or more Error objects. Each of these objects contains an error code, reason, and (optionally) the input which triggered the exception. A complete list of known error codes is provided in the Zend Framework API documentation under `Zend_Gdata_Gapps_Error`. Additionally, the authoritative error list is available online at Google Apps Provisioning API V2.0 Reference: Appendix D.

While the complete list of errors received is available within ServiceException as an array by calling `getErrors()`, often it is convenient to know if one specific error occurred. For these cases the presence of an error can be determined by calling `hasError()`.

The following example demonstrates how to detect if a requested resource doesn't exist and handle the fault gracefully:

```
function retrieveUser ($username) {
    $query = $gdata->newUserQuery($username);
    try {
        $user = $gdata->getUserEntry($query);
    } catch (Zend_Gdata_Gapps_ServiceException $e) {
        // Set the user to null if not found
        if ($e->hasError(Zend_Gdata_Gapps_Error::ENTITY_DOES_NOT_EXIST)) {
            $user = null;
        } else {
            throw $e;
        }
    }
    return $user;
}
```

10. Using Google Base

The Google Base data API is designed to enable developers to do two things:

- Query Google Base data to create applications and mashups.

- Input and manage Google Base items programmatically.

There are two item feeds: snippets feed and customer items feeds. The snippets feed contains all Google Base data and is available to anyone to query against without a need for authentication. The customer items feed is a customer-specific subset of data and only a customer/owner can access this feed to insert, update, or delete their own data. Queries are constructed the same way against both types of feeds.

See http://code.google.com/apis/base for more information about the Google Base API.

10.1. Connect To The Base Service

The Google Base API, like all GData APIs, is based off of the Atom Publishing Protocol (APP), an XML based format for managing web-based resources. Traffic between a client and the Google Base servers occurs over HTTP and allows for both authenticated and unauthenticated connections.

Before any transactions can occur, this connection needs to be made. Creating a connection to the base servers involves two steps: creating an HTTP client and binding a `Zend_Gdata_Gbase` service instance to that client.

10.1.1. Authentication

The Google Base API allows access to both public and private base feeds. Public feeds do not require authentication, but are read-only and offer reduced functionality. Private feeds offers the most complete functionality but requires an authenticated connection to the base servers. There are three authentication schemes that are supported by Google Base:

- *ClientAuth* provides direct username/password authentication to the base servers. Since this scheme requires that users provide your application with their password, this authentication is only recommended when other authentication schemes are insufficient.

- *AuthSub* allows authentication to the base servers via a Google proxy server. This provides the same level of convenience as ClientAuth but without the security risk, making this an ideal choice for web-based applications.

The `Zend_Gdata` library provides support for all three authentication schemes. The rest of this chapter will assume that you are familiar the authentication schemes available and how to create an appropriate authenticated connection. For more information, please see section Section 1.4, “Google Data Client Authentication”. or the Authentication Overview in the Google Data API Developer's Guide.

10.1.2. Create A Service Instance

In order to interact with Google Base, this library provides the `Zend_Gdata_Gbase` service class. This class provides a common interface to the Google Data and Atom Publishing Protocol models and assists in marshaling requests to and from the base servers.

Once deciding on an authentication scheme, the next step is to create an instance of `Zend_Gdata_Gbase` . This class takes in an instance of `Zend_Http_Client` as a single argument. This provides an interface for AuthSub and ClientAuth authentication, as both of these creation of a special authenticated HTTP client. If no arguments are provided, an unauthenticated instance of `Zend_Http_Client` will be automatically created.

The example below shows how to create a Base service class using ClientAuth authentication:

```php
// Parameters for ClientAuth authentication
$service = Zend_Gdata_Gbase::AUTH_SERVICE_NAME;
$user = "sample.user@gmail.com";
$pass = "pa$$w0rd";

// Create an authenticated HTTP client
$client = Zend_Gdata_ClientLogin::getHttpClient($user, $pass, $service);

// Create an instance of the Base service
$service = new Zend_Gdata_Gbase($client);
```

A Base service using AuthSub can be created in a similar, though slightly more lengthy fashion:

```php
/*
 * Retrieve the current URL so that the AuthSub server knows where to
 * redirect the user after authentication is complete.
 */
function getCurrentUrl()
{
    global $_SERVER;

    // Filter php_self to avoid a security vulnerability.
    $php_request_uri =
        htmlentities(substr($_SERVER['REQUEST_URI'],
                            0,
                            strcspn($_SERVER['REQUEST_URI'], "\n\r")),
                     ENT_QUOTES);

    if (isset($_SERVER['HTTPS']) &&
        strtolower($_SERVER['HTTPS']) == 'on') {
        $protocol = 'https://';
    } else {
        $protocol = 'http://';
    }
    $host = $_SERVER['HTTP_HOST'];
    if ($_SERVER['HTTP_PORT'] != '' &&
        (($protocol == 'http://' && $_SERVER['HTTP_PORT'] != '80') ||
        ($protocol == 'https://' && $_SERVER['HTTP_PORT'] != '443'))) {
        $port = ':' . $_SERVER['HTTP_PORT'];
    } else {
        $port = '';
    }
    return $protocol . $host . $port . $php_request_uri;
}

/**
 * Obtain an AuthSub authenticated HTTP client, redirecting the user
 * to the AuthSub server to login if necessary.
 */
function getAuthSubHttpClient()
{
    global $_SESSION, $_GET;

    // If there is no AuthSub session or one-time token waiting for us,
    // redirect the user to the AuthSub server to get one.
    if (!isset($_SESSION['sessionToken']) && !isset($_GET['token'])) {
        // Parameters to give to AuthSub server
        $next = getCurrentUrl();
        $scope = "http://www.google.com/base/feeds/items/";
```

```
        $secure = false;
        $session = true;

        // Redirect the user to the AuthSub server to sign in

        $authSubUrl = Zend_Gdata_AuthSub::getAuthSubTokenUri($next,
                                                             $scope,
                                                             $secure,
                                                             $session);
         header("HTTP/1.0 307 Temporary redirect");

         header("Location: " . $authSubUrl);

         exit();
    }

    // Convert an AuthSub one-time token into a session token if needed
    if (!isset($_SESSION['sessionToken']) && isset($_GET['token'])) {
        $_SESSION['sessionToken'] =
            Zend_Gdata_AuthSub::getAuthSubSessionToken($_GET['token']);
    }

    // At this point we are authenticated via AuthSub and can obtain an
    // authenticated HTTP client instance

    // Create an authenticated HTTP client
    $client = Zend_Gdata_AuthSub::getHttpClient($_SESSION['sessionToken']);
    return $client;
}

// -> Script execution begins here <-

// Make sure http://code.google.com/apis/gdata/reference.html#Queriesthat
// the user has a valid session, so we can record the
// AuthSub session token once it is available.
session_start();

// Create an instance of the Base service, redirecting the user
// to the AuthSub server if necessary.
$service = new Zend_Gdata_Gbase(getAuthSubHttpClient());
```

Finally, an unauthenticated server can be created for use with snippets feeds:

```
// Create an instance of the Base service using an unauthenticated HTTP client
$service = new Zend_Gdata_Gbase();
```

10.2. Retrieve Items

You can query customer items feed or snippets feed to retrieve items. It involves two steps, sending a query and iterating through the returned feed.

10.2.1. Send a Structured Query

You can send a structured query to retrieve items from your own customer items feed or from the public snippets feed.

When retrieving items using the Base API, specially constructed query URLs are used to describe what events should be returned. The `Zend_Gdata_Gbase_ItemQuery` and

Zend_Gdata_Gbase_SnippetQuery classes simplify this task by automatically constructing a query URL based on provided parameters.

10.2.1.1. Query Customer Items Feed

To execute a query against the customer items feed, invoke newItemQuery() and getGbaseItemFeed() methods:

```
$service = new Zend_Gdata_Gbase($client);
$query = $service->newItemQuery();
$query->setBq('[title:Programming]');
$query->setOrderBy('modification_time');
$query->setSortOrder('descending');
$query->setMaxResults('5');
$feed = $service->getGbaseItemFeed($query);
```

A full list of these parameters is available at the Query parameters section of the Customer Items Feed documentation.

10.2.1.2. Query Snippets Feed

To execute a query against the public snippets feed, invoke newSnippetQuery() and getGbaseSnippetFeed() methods:

```
$service = new Zend_Gdata_Gbase();
$query = $service->newSnippetQuery();
$query->setBq('[title:Programming]');
$query->setOrderBy('modification_time');
$query->setSortOrder('descending');
$query->setMaxResults('5');
$feed = $service->getGbaseSnippetFeed($query);
```

A full list of these parameters is available at the Query parameters section of the Snippets Feed documentation.

10.2.2. Iterate through the Items

Google Base items can contain item-specific attributes such as <g:main_ingredient> and <g:weight>.

To iterate through all attributes of a given item, invoke getGbaseAttributes() and iterate through the results:

```
foreach ($feed->entries as $entry) {
  // Get all attributes and print out the name and text value of each
  // attribute
  $baseAttributes = $entry->getGbaseAttributes();
  foreach ($baseAttributes as $attr) {
    echo "Attribute " . $attr->name . " : " . $attr->text . "<br>";
  }
}
```

Or, you can look for specific attribute name and iterate through the results that match:

```
foreach ($feed->entries as $entry) {
  // Print all main ingredients <g:main_ingredient>
  $baseAttributes = $entry->getGbaseAttribute("main_ingredient");
  foreach ($baseAttributes as $attr) {
    echo "Main ingredient: " . $attr->text . "<br>";
```

```
  }
}
```

10.3. Insert, Update, and Delete Customer Items

A customer/owner can access his own Customer Items feed to insert, update, or delete their items. These operations do not apply to the public snippets feed.

You can test a feed operation before it is actually executed by setting the dry-run flag (`$dryRun`) to TRUE. Once you are sure that you want to submit the data, set it to FALSE to execute the operation.

10.3.1. Insert an Item

Items can be added by using the `insertGbaseItem()` method for the Base service:

```
$service = new Zend_Gdata_Gbase($client);
$newEntry = $service->newItemEntry();

// Add title
$title = "PHP Developer Handbook";
$newEntry->title = $service->newTitle(trim($title));

// Add some content
$content = "Essential handbook for PHP developers.";
$newEntry->content = $service->newContent($content);
$newEntry->content->type = 'text';

// Define product type
$itemType = "Products";
$newEntry->itemType = $itemType;

// Add item specific attributes
$newEntry->addGbaseAttribute("product_type", "book", "text");
$newEntry->addGbaseAttribute("price", "12.99 USD", "floatUnit");
$newEntry->addGbaseAttribute("quantity", "10", "int");
$newEntry->addGbaseAttribute("weight", "2.2 lbs", "numberUnit");
$newEntry->addGbaseAttribute("condition", "New", "text");
$newEntry->addGbaseAttribute("author", "John Doe", "text");
$newEntry->addGbaseAttribute("edition", "First Edition", "text");
$newEntry->addGbaseAttribute("pages", "253", "number");
$newEntry->addGbaseAttribute("publisher", "My Press", "text");
$newEntry->addGbaseAttribute("year", "2007", "number");
$newEntry->addGbaseAttribute("payment_accepted", "Google Checkout", "text");

$dryRun = true;
$createdEntry = $service->insertGbaseItem($newEntry, $dryRun);
```

10.3.2. Modify an Item

You can update each attribute element of an item as you iterate through them:

```
// Update the title
$newTitle = "PHP Developer Handbook Second Edition";
$entry->title = $service->newTitle($newTitle);

// Find <g:price> attribute and update the price
$baseAttributes = $entry->getGbaseAttribute("price");
```

```
if (is_object($baseAttributes[0])) {
  $newPrice = "16.99 USD";
  $baseAttributes[0]->text = $newPrice;
}

// Find <g:pages> attribute and update the number of pages
$baseAttributes = $entry->getGbaseAttribute("pages");
if (is_object($baseAttributes[0])) {
  $newPages = "278";
  $baseAttributes[0]->text = $newPages;

  // Update the attribute type from "number" to "int"
  if ($baseAttributes[0]->type == "number") {
    $newType = "int";
    $baseAttributes[0]->type = $newType;
  }
}

// Remove <g:label> attributes
$baseAttributes = $entry->getGbaseAttribute("label");
foreach ($baseAttributes as $note) {
  $entry->removeGbaseAttribute($note);
}

// Add new attributes
$entry->addGbaseAttribute("note", "PHP 5", "text");
$entry->addGbaseAttribute("note", "Web Programming", "text");

// Save the changes by invoking save() on the entry object itself
$dryRun = true;
$entry->save($dryRun);

// Or, save the changes by calling updateGbaseItem() on the service object
// $dryRun = true;
// $service->updateGbaseItem($entry, $dryRun);
```

After making the changes, either invoke `save($dryRun)` method on the `Zend_Gdata_Gbase_ItemEntry` object or call `updateGbaseItem($entry, $dryRun)` method on the `Zend_Gdata_Gbase` object to save the changes.

10.3.3. Delete an Item

You can remove an item by calling `deleteGbaseItem()` method:

```
$dryRun = false;
$service->deleteGbaseItem($entry, $dryRun);
```

Alternatively, you can invoke `delete()` on the `Zend_Gdata_Gbase_ItemEntry` object:

```
$dryRun = false;
$entry->delete($dryRun);
```

11. Using Picasa Web Albums

Picasa Web Albums is a service which allows users to maintain albums of their own pictures, and browse the albums and pictures of others. The API offers a programmatic interface to this service, allowing users to add to, update, and remove from their albums, as well as providing the ability to tag and comment on photos.

Access to public albums and photos is not restricted by account, however, a user must be logged in for non-read-only access.

For more information on the API, including instructions for enabling API access, refer to the Picasa Web Albums Data API Overview.

Authentication

The API provides authentication via AuthSub (recommended) and ClientAuth. HTTP connections must be authenticated for write support, but non-authenticated connections have read-only access.

11.1. Connecting To The Service

The Picasa Web Albums API, like all GData APIs, is based off of the Atom Publishing Protocol (APP), an XML based format for managing web-based resources. Traffic between a client and the servers occurs over HTTP and allows for both authenticated and unauthenticated connections.

Before any transactions can occur, this connection needs to be made. Creating a connection to the Picasa servers involves two steps: creating an HTTP client and binding a `Zend_Gdata_Photos` service instance to that client.

11.1.1. Authentication

The Google Picasa API allows access to both public and private photo feeds. Public feeds do not require authentication, but are read-only and offer reduced functionality. Private feeds offers the most complete functionality but requires an authenticated connection to the Picasa servers. There are three authentication schemes that are supported by Google Picasa :

- *ClientAuth* provides direct username/password authentication to the Picasa servers. Since this scheme requires that users provide your application with their password, this authentication is only recommended when other authentication schemes are insufficient.
- *AuthSub* allows authentication to the Picasa servers via a Google proxy server. This provides the same level of convenience as ClientAuth but without the security risk, making this an ideal choice for web-based applications.

The `Zend_Gdata` library provides support for both authentication schemes. The rest of this chapter will assume that you are familiar the authentication schemes available and how to create an appropriate authenticated connection. For more information, please see section the Authentication section of this manual or the Authentication Overview in the Google Data API Developer's Guide.

11.1.2. Creating A Service Instance

In order to interact with the servers, this library provides the `Zend_Gdata_Photos` service class. This class provides a common interface to the Google Data and Atom Publishing Protocol models and assists in marshaling requests to and from the servers.

Once deciding on an authentication scheme, the next step is to create an instance of `Zend_Gdata_Photos`. The class constructor takes an instance of `Zend_Http_Client` as a single argument. This provides an interface for AuthSub and ClientAuth authentication, as both of these require creation of a special authenticated HTTP client. If no arguments are provided, an unauthenticated instance of `Zend_Http_Client` will be automatically created.

The example below shows how to create a service class using ClientAuth authentication:

```
// Parameters for ClientAuth authentication
$service = Zend_Gdata_Photos::AUTH_SERVICE_NAME;
$user = "sample.user@gmail.com";
$pass = "pa$$w0rd";

// Create an authenticated HTTP client
$client = Zend_Gdata_ClientLogin::getHttpClient($user, $pass, $service);

// Create an instance of the service
$service = new Zend_Gdata_Photos($client);
```

A service instance using AuthSub can be created in a similar, though slightly more lengthy fashion:

```
session_start();

/**
 * Returns the full URL of the current page, based upon env variables
 *
 * Env variables used:
 * $_SERVER['HTTPS'] = (on|off|)
 * $_SERVER['HTTP_HOST'] = value of the Host: header
 * $_SERVER['SERVER_PORT'] = port number (only used if not http/80,https/443)
 * $_SERVER['REQUEST_URI'] = the URI after the method of the HTTP request
 *
 * @return string Current URL
 */
function getCurrentUrl()
{
    global $_SERVER;

    /**
     * Filter php_self to avoid a security vulnerability.
     */
    $php_request_uri = htmlentities(substr($_SERVER['REQUEST_URI'], 0,
    strcspn($_SERVER['REQUEST_URI'], "\n\r")), ENT_QUOTES);

    if (isset($_SERVER['HTTPS']) && strtolower($_SERVER['HTTPS']) == 'on') {
        $protocol = 'https://';
    } else {
        $protocol = 'http://';
    }
    $host = $_SERVER['HTTP_HOST'];
    if ($_SERVER['SERVER_PORT'] != '' &&
        (($protocol == 'http://' && $_SERVER['SERVER_PORT'] != '80') ||
        ($protocol == 'https://' && $_SERVER['SERVER_PORT'] != '443'))) {
            $port = ':' . $_SERVER['SERVER_PORT'];
    } else {
        $port = '';
    }
    return $protocol . $host . $port . $php_request_uri;
}

/**
 * Returns the AuthSub URL which the user must visit to authenticate requests
 * from this application.
 *
 * Uses getCurrentUrl() to get the next URL which the user will be redirected
 * to after successfully authenticating with the Google service.
```

```
 *
 * @return string AuthSub URL
 */
function getAuthSubUrl()
{
    $next = getCurrentUrl();
    $scope = 'http://picasaweb.google.com/data';
    $secure = false;
    $session = true;
    return Zend_Gdata_AuthSub::getAuthSubTokenUri($next, $scope, $secure,
        $session);
}

/**
 * Returns a HTTP client object with the appropriate headers for communicating
 * with Google using AuthSub authentication.
 *
 * Uses the $_SESSION['sessionToken'] to store the AuthSub session token after
 * it is obtained. The single use token supplied in the URL when redirected
 * after the user succesfully authenticated to Google is retrieved from the
 * $_GET['token'] variable.
 *
 * @return Zend_Http_Client
 */
function getAuthSubHttpClient()
{
    global $_SESSION, $_GET;
    if (!isset($_SESSION['sessionToken']) && isset($_GET['token'])) {
        $_SESSION['sessionToken'] =
            Zend_Gdata_AuthSub::getAuthSubSessionToken($_GET['token']);
    }
    $client = Zend_Gdata_AuthSub::getHttpClient($_SESSION['sessionToken']);
    return $client;
}

/**
 * Create a new instance of the service, redirecting the user
 * to the AuthSub server if necessary.
 */
$service = new Zend_Gdata_Photos(getAuthSubHttpClient());
```

Finally, an unauthenticated server can be created for use with public feeds:

```
// Create an instance of the service using an unauthenticated HTTP client
$service = new Zend_Gdata_Photos();
```

11.2. Understanding and Constructing Queries

The primary method to request data from the service is by constructing a query. There are query classes for each of the following types:

- *User* is used to specify the user whose data is being searched for, and is specified as a username. If no user is provided, "default" will be used instead to indicate the currently authenticated user (if authenticated).
- *Album* is used to specify the album which is being searched for, and is specified as either an id, or an album name.
- *Photo* is used to specify the photo which is being searched for, and is specified as an id.

A new `UserQuery` can be constructed as followed:

```
$service = Zend_Gdata_Photos::AUTH_SERVICE_NAME;
$client = Zend_Gdata_ClientLogin::getHttpClient($user, $pass, $service);
$service = new Zend_Gdata_Photos($client);

$query = new Zend_Gdata_Photos_UserQuery();
$query->setUser("sample.user");
```

For each query, a number of parameters limiting the search can be requested, or specified, with get(Parameter) and set(Parameter), respectively. They are as follows:

- *Projection* sets the format of the data returned in the feed, as either "api" or "base". Normally, "api" is desired. The default is "api".
- *Type* sets the type of element to be returned, as either "feed" or "entry". The default is "feed".
- *Access* sets the visibility of items to be returned, as "all", "public", or "private". The default is "all". Non-public elements will only be returned if the query is searching for the authenticated user.
- *Tag* sets a tag filter for returned items. When a tag is set, only items tagged with this value will return.
- *Kind* sets the kind of elements to return. When kind is specified, only entries that match this value will be returned.
- *ImgMax* sets the maximum image size for entries returned. Only image entries smaller than this value will be returned.
- *Thumbsize* sets the thumbsize of entries that are returned. Any retrieved entry will have a thumbsize equal to this value.
- *User* sets the user whose data is being searched for. The default is "default".
- *AlbumId* sets the id of the album being searched for. This element only applies to album and photo queries. In the case of photo queries, this specifies the album that contains the requested photo. The album id is mutually exclusive with the album's name. Setting one unsets the other.
- *AlbumName* sets the name of the album being searched for. This element only applies to the album and photo queries. In the case of photo queries, this specifies the album that contains the requested photo. The album name is mutually exclusive with the album's id. Setting one unsets the other.
- *PhotoId* sets the id of the photo being searched for. This element only applies to photo queries.

11.3. Retrieving Feeds And Entries

The service has functions to retrieve a feed, or individual entries, for users, albums, and individual photos.

11.3.1. Retrieving A User

The service supports retrieving a user feed and list of the user's content. If the requested user is also the authenticated user, entries marked as "`hidden`" will also be returned.

The user feed can be accessed by passing the username to the `getUserFeed` method:

```
$service = Zend_Gdata_Photos::AUTH_SERVICE_NAME;
$client = Zend_Gdata_ClientLogin::getHttpClient($user, $pass, $service);
$service = new Zend_Gdata_Photos($client);

try {
    $userFeed = $service->getUserFeed("sample.user");
} catch (Zend_Gdata_App_Exception $e) {
    echo "Error: " . $e->getMessage();
}
```

Or, the feed can be accessed by constructing a query, first:

```
$service = Zend_Gdata_Photos::AUTH_SERVICE_NAME;
$client = Zend_Gdata_ClientLogin::getHttpClient($user, $pass, $service);
$service = new Zend_Gdata_Photos($client);

$query = new Zend_Gdata_Photos_UserQuery();
$query->setUser("sample.user");

try {
    $userFeed = $service->getUserFeed(null, $query);
} catch (Zend_Gdata_App_Exception $e) {
    echo "Error: " . $e->getMessage();
}
```

Constructing a query also provides the ability to request a user entry object:

```
$service = Zend_Gdata_Photos::AUTH_SERVICE_NAME;
$client = Zend_Gdata_ClientLogin::getHttpClient($user, $pass, $service);
$service = new Zend_Gdata_Photos($client);

$query = new Zend_Gdata_Photos_UserQuery();
$query->setUser("sample.user");
$query->setType("entry");

try {
    $userEntry = $service->getUserEntry($query);
} catch (Zend_Gdata_App_Exception $e) {
    echo "Error: " . $e->getMessage();
}
```

11.3.2. Retrieving An Album

The service supports retrieving an album feed and a list of the album's content.

The album feed is accessed by constructing a query object and passing it to `getAlbumFeed`:

```
$service = Zend_Gdata_Photos::AUTH_SERVICE_NAME;
$client = Zend_Gdata_ClientLogin::getHttpClient($user, $pass, $service);
$service = new Zend_Gdata_Photos($client);

$query = new Zend_Gdata_Photos_AlbumQuery();
$query->setUser("sample.user");
$query->setAlbumId("1");

try {
    $albumFeed = $service->getAlbumFeed($query);
} catch (Zend_Gdata_App_Exception $e) {
    echo "Error: " . $e->getMessage();
```

```
}
```

Alternatively, the query object can be given an album name with `setAlbumName`. Setting the album name is mutually exclusive with setting the album id, and setting one will unset the other.

Constructing a query also provides the ability to request an album entry object:

```
$service = Zend_Gdata_Photos::AUTH_SERVICE_NAME;
$client = Zend_Gdata_ClientLogin::getHttpClient($user, $pass, $service);
$service = new Zend_Gdata_Photos($client);

$query = new Zend_Gdata_Photos_AlbumQuery();
$query->setUser("sample.user");
$query->setAlbumId("1");
$query->setType("entry");

try {
    $albumEntry = $service->getAlbumEntry($query);
} catch (Zend_Gdata_App_Exception $e) {
    echo "Error: " . $e->getMessage();
}
```

11.3.3. Retrieving A Photo

The service supports retrieving a photo feed and a list of associated comments and tags.

The photo feed is accessed by constructing a query object and passing it to `getPhotoFeed`:

```
$service = Zend_Gdata_Photos::AUTH_SERVICE_NAME;
$client = Zend_Gdata_ClientLogin::getHttpClient($user, $pass, $service);
$service = new Zend_Gdata_Photos($client);

$query = new Zend_Gdata_Photos_PhotoQuery();
$query->setUser("sample.user");
$query->setAlbumId("1");
$query->setPhotoId("100");

try {
    $photoFeed = $service->getPhotoFeed($query);
} catch (Zend_Gdata_App_Exception $e) {
    echo "Error: " . $e->getMessage();
}
```

Constructing a query also provides the ability to request a photo entry object:

```
$service = Zend_Gdata_Photos::AUTH_SERVICE_NAME;
$client = Zend_Gdata_ClientLogin::getHttpClient($user, $pass, $service);
$service = new Zend_Gdata_Photos($client);

$query = new Zend_Gdata_Photos_PhotoQuery();
$query->setUser("sample.user");
$query->setAlbumId("1");
$query->setPhotoId("100");
$query->setType("entry");

try {
    $photoEntry = $service->getPhotoEntry($query);
} catch (Zend_Gdata_App_Exception $e) {
    echo "Error: " . $e->getMessage();
```

```
}
```

11.3.4. Retrieving A Comment

The service supports retrieving comments from a feed of a different type. By setting a query to return a kind of "comment", a feed request can return comments associated with a specific user, album, or photo.

Performing an action on each of the comments on a given photo can be accomplished as follows:

```
$service = Zend_Gdata_Photos::AUTH_SERVICE_NAME;
$client = Zend_Gdata_ClientLogin::getHttpClient($user, $pass, $service);
$service = new Zend_Gdata_Photos($client);

$query = new Zend_Gdata_Photos_PhotoQuery();
$query->setUser("sample.user");
$query->setAlbumId("1");
$query->setPhotoId("100");
$query->setKind("comment");

try {
    $photoFeed = $service->getPhotoFeed($query);

    foreach ($photoFeed as $entry) {
        if ($entry instanceof Zend_Gdata_Photos_CommentEntry) {
            // Do something with the comment
        }
    }
} catch (Zend_Gdata_App_Exception $e) {
    echo "Error: " . $e->getMessage();
}
```

11.3.5. Retrieving A Tag

The service supports retrieving tags from a feed of a different type. By setting a query to return a kind of "tag", a feed request can return tags associated with a specific photo.

Performing an action on each of the tags on a given photo can be accomplished as follows:

```
$service = Zend_Gdata_Photos::AUTH_SERVICE_NAME;
$client = Zend_Gdata_ClientLogin::getHttpClient($user, $pass, $service);
$service = new Zend_Gdata_Photos($client);

$query = new Zend_Gdata_Photos_PhotoQuery();
$query->setUser("sample.user");
$query->setAlbumId("1");
$query->setPhotoId("100");
$query->setKind("tag");

try {
    $photoFeed = $service->getPhotoFeed($query);

    foreach ($photoFeed as $entry) {
        if ($entry instanceof Zend_Gdata_Photos_TagEntry) {
            // Do something with the tag
        }
    }
} catch (Zend_Gdata_App_Exception $e) {
    echo "Error: " . $e->getMessage();
```

```
}
```

11.4. Creating Entries

The service has functions to create albums, photos, comments, and tags.

11.4.1. Creating An Album

The service supports creating a new album for an authenticated user:

```
$service = Zend_Gdata_Photos::AUTH_SERVICE_NAME;
$client = Zend_Gdata_ClientLogin::getHttpClient($user, $pass, $service);
$service = new Zend_Gdata_Photos($client);

$entry = new Zend_Gdata_Photos_AlbumEntry();
$entry->setTitle($service->newTitle("test album"));

$service->insertAlbumEntry($entry);
```

11.4.2. Creating A Photo

The service supports creating a new photo for an authenticated user:

```
$service = Zend_Gdata_Photos::AUTH_SERVICE_NAME;
$client = Zend_Gdata_ClientLogin::getHttpClient($user, $pass, $service);
$service = new Zend_Gdata_Photos($client);

// $photo is the name of a file uploaded via an HTML form

$fd = $service->newMediaFileSource($photo["tmp_name"]);
$fd->setContentType($photo["type"]);

$entry = new Zend_Gdata_Photos_PhotoEntry();
$entry->setMediaSource($fd);
$entry->setTitle($service->newTitle($photo["name"]));

$albumQuery = new Zend_Gdata_Photos_AlbumQuery;
$albumQuery->setUser("sample.user");
$albumQuery->setAlbumId("1");

$albumEntry = $service->getAlbumEntry($albumQuery);

$service->insertPhotoEntry($entry, $albumEntry);
```

11.4.3. Creating A Comment

The service supports creating a new comment for a photo:

```
$service = Zend_Gdata_Photos::AUTH_SERVICE_NAME;
$client = Zend_Gdata_ClientLogin::getHttpClient($user, $pass, $service);
$service = new Zend_Gdata_Photos($client);

$entry = new Zend_Gdata_Photos_CommentEntry();
$entry->setTitle($service->newTitle("comment"));
$entry->setContent($service->newContent("comment"));

$photoQuery = new Zend_Gdata_Photos_PhotoQuery;
$photoQuery->setUser("sample.user");
$photoQuery->setAlbumId("1");
```

```
$photoQuery->setPhotoId("100");
$photoQuery->setType('entry');

$photoEntry = $service->getPhotoEntry($photoQuery);

$service->insertCommentEntry($entry, $photoEntry);
```

11.4.4. Creating A Tag

The service supports creating a new tag for a photo:

```
$service = Zend_Gdata_Photos::AUTH_SERVICE_NAME;
$client = Zend_Gdata_ClientLogin::getHttpClient($user, $pass, $service);
$service = new Zend_Gdata_Photos($client);

$entry = new Zend_Gdata_Photos_TagEntry();
$entry->setTitle($service->newTitle("tag"));

$photoQuery = new Zend_Gdata_Photos_PhotoQuery;
$photoQuery->setUser("sample.user");
$photoQuery->setAlbumId("1");
$photoQuery->setPhotoId("100");
$photoQuery->setType('entry');

$photoEntry = $service->getPhotoEntry($photoQuery);

$service->insertTagEntry($entry, $photoEntry);
```

11.5. Deleting Entries

The service has functions to delete albums, photos, comments, and tags.

11.5.1. Deleting An Album

The service supports deleting an album for an authenticated user:

```
$service = Zend_Gdata_Photos::AUTH_SERVICE_NAME;
$client = Zend_Gdata_ClientLogin::getHttpClient($user, $pass, $service);
$service = new Zend_Gdata_Photos($client);

$albumQuery = new Zend_Gdata_Photos_AlbumQuery;
$albumQuery->setUser("sample.user");
$albumQuery->setAlbumId("1");
$albumQuery->setType('entry');

$entry = $service->getAlbumEntry($albumQuery);

$service->deleteAlbumEntry($entry, true);
```

11.5.2. Deleting A Photo

The service supports deleting a photo for an authenticated user:

```
$service = Zend_Gdata_Photos::AUTH_SERVICE_NAME;
$client = Zend_Gdata_ClientLogin::getHttpClient($user, $pass, $service);
$service = new Zend_Gdata_Photos($client);

$photoQuery = new Zend_Gdata_Photos_PhotoQuery;
$photoQuery->setUser("sample.user");
```

```
$photoQuery->setAlbumId("1");
$photoQuery->setPhotoId("100");
$photoQuery->setType('entry');

$entry = $service->getPhotoEntry($photoQuery);

$service->deletePhotoEntry($entry, true);
```

11.5.3. Deleting A Comment

The service supports deleting a comment for an authenticated user:

```
$service = Zend_Gdata_Photos::AUTH_SERVICE_NAME;
$client = Zend_Gdata_ClientLogin::getHttpClient($user, $pass, $service);
$service = new Zend_Gdata_Photos($client);

$photoQuery = new Zend_Gdata_Photos_PhotoQuery;
$photoQuery->setUser("sample.user");
$photoQuery->setAlbumId("1");
$photoQuery->setPhotoId("100");
$photoQuery->setType('entry');

$path = $photoQuery->getQueryUrl() . '/commentid/' . "1000";

$entry = $service->getCommentEntry($path);

$service->deleteCommentEntry($entry, true);
```

11.5.4. Deleting A Tag

The service supports deleting a tag for an authenticated user:

```
$service = Zend_Gdata_Photos::AUTH_SERVICE_NAME;
$client = Zend_Gdata_ClientLogin::getHttpClient($user, $pass, $service);
$service = new Zend_Gdata_Photos($client);

$photoQuery = new Zend_Gdata_Photos_PhotoQuery;
$photoQuery->setUser("sample.user");
$photoQuery->setAlbumId("1");
$photoQuery->setPhotoId("100");
$photoQuery->setKind("tag");
$query = $photoQuery->getQueryUrl();

$photoFeed = $service->getPhotoFeed($query);

foreach ($photoFeed as $entry) {
    if ($entry instanceof Zend_Gdata_Photos_TagEntry) {
        if ($entry->getContent() == $tagContent) {
            $tagEntry = $entry;
        }
    }
}

$service->deleteTagEntry($tagEntry, true);
```

11.5.5. Optimistic Concurrency (Notes On Deletion)

GData feeds, including those of the Picasa Web Albums service, implement optimistic concurrency, a versioning system that prevents users from overwriting changes, inadvertently.

When deleting a entry through the service class, if the entry has been modified since it was last fetched, an exception will be thrown, unless explicitly set otherwise (in which case the deletion is retried on the updated entry).

An example of how to handle versioning during a deletion is shown by `deleteAlbumEntry`:

```
// $album is the albumEntry to be deleted
try {
    $this->delete($album);
} catch (Zend_Gdata_App_HttpException $e) {
    if ($e->getMessage()->getStatus() === 409) {
        $entry =
            new Zend_Gdata_Photos_AlbumEntry($e->getMessage()->getBody());
        $this->delete($entry->getLink('edit')->href);
    } else {
        throw $e;
    }
}
```

12. Using the YouTube Data API

The YouTube Data API offers read and write access to YouTube's content. Users can perform unauthenticated requests to Google Data feeds to retrieve feeds of popular videos, comments, public information about YouTube user profiles, user playlists, favorites, subscriptions and so on.

For more information on the YouTube Data API, please refer to the official PHP Developer's Guide on code.google.com.

12.1. Authentication

The YouTube Data API allows read-only access to public data, which does not require authentication. For any write requests, a user needs to authenticate either using ClientLogin or AuthSub authentication. Please refer to the Authentication section in the PHP Developer's Guide for more detail.

12.2. Developer Keys and Client ID

A developer key identifies the YouTube developer that is submitting an API request. A client ID identifies your application for logging and debugging purposes. Please visit http://code.google.com/apis/youtube/dashboard/ to obtain a developer key and client ID. The example below demonstrates how to pass the developer key and client ID to the Zend_Gdata_YouTube service object.

Example 394. Passing a Developer Key and ClientID to Zend_Gdata_YouTube

```
$yt = new Zend_Gdata_YouTube($httpClient,
                             $applicationId,
                             $clientId,
                             $developerKey);
```

12.3. Retrieving public video feeds

The YouTube Data API provides numerous feeds that return a list of videos, such as standard feeds, related videos, video responses, user's uploads, and user's favorites. For example, the user's uploads feed returns all videos uploaded by a specific user. See the YouTube API reference guide for a detailed list of available feeds.

12.3.1. Searching for videos by metadata

You can retrieve a list of videos that match specified search criteria, using the YouTubeQuery class. The following query looks for videos which contain the word "cat" in their metadata, starting with the 10th video and displaying 20 videos per page, ordered by the view count.

Example 395. Searching for videos

```
$yt = new Zend_Gdata_YouTube();
$query = $yt->newVideoQuery();
$query->videoQuery = 'cat';
$query->startIndex = 10;
$query->maxResults = 20;
$query->orderBy = 'viewCount';

echo $query->queryUrl . "\n";
$videoFeed = $yt->getVideoFeed($query);

foreach ($videoFeed as $videoEntry) {
    echo "---------VIDEO----------\n";
    echo "Title: " . $videoEntry->getVideoTitle() . "\n";
    echo "\nDescription:\n";
    echo $videoEntry->getVideoDescription();
    echo "\n\n\n";
}
```

For more details on the different query parameters, please refer to the Reference Guide. The available helper functions in `Zend_Gdata_YouTube_VideoQuery` for each of these parameters are described in more detail in the PHP Developer's Guide.

12.3.2. Searching for videos by categories and tags/keywords

Searching for videos in specific categories is done by generating a specially formatted URL. For example, to search for comedy videos which contain the keyword dog:

Example 396. Searching for videos in specific categories

```
$yt = new Zend_Gdata_YouTube();
$query = $yt->newVideoQuery();
$query->category = 'Comedy/dog';

echo $query->queryUrl . "\n";
$videoFeed = $yt->getVideoFeed($query);
```

12.3.3. Retrieving standard feeds

The YouTube Data API has a number of standard feeds. These standard feeds can be retrieved as Zend_Gdata_YouTube_VideoFeed objects using the specified URLs, using the predefined constants within the Zend_Gdata_YouTube class (Zend_Gdata_YouTube::STANDARD_TOP_RATED_URI for example) or using the predefined helper methods (see code listing below).

To retrieve the top rated videos using the helper method:

Example 397. Retrieving a standard video feed

```
$yt = new Zend_Gdata_YouTube();
$videoFeed = $yt->getTopRatedVideoFeed();
```

There are also query parameters to specify the time period over which the standard feed is computed.

For example, to retrieve the top rated videos for today:

Example 398. Using a Zend_Gdata_YouTube_VideoQuery to Retrieve Videos

```
$yt = new Zend_Gdata_YouTube();
$query = $yt->newVideoQuery();
$query->setTime('today');
$videoFeed = $yt->getTopRatedVideoFeed($query);
```

Alternatively, you could just retrieve the feed using the URL:

Example 399. Retrieving a video feed by URL

```
$yt = new Zend_Gdata_YouTube();
$url = 'http://gdata.youtube.com/feeds/standardfeeds/top_rated?time=today'
$videoFeed = $yt->getVideoFeed($url);
```

12.3.4. Retrieving videos uploaded by a user

You can retrieve a list of videos uploaded by a particular user using a simple helper method. This example retrieves videos uploaded by the user 'liz'.

Example 400. Retrieving videos uploaded by a specific user

```
$yt = new Zend_Gdata_YouTube();
$videoFeed = $yt->getUserUploads('liz');
```

12.3.5. Retrieving videos favorited by a user

You can retrieve a list of a user's favorite videos using a simple helper method. This example retrieves videos favorited by the user 'liz'.

Example 401. Retrieving a user's favorite videos

```
$yt = new Zend_Gdata_YouTube();
$videoFeed = $yt->getUserFavorites('liz');
```

12.3.6. Retrieving video responses for a video

You can retrieve a list of a video's video responses using a simple helper method. This example retrieves video response for a video with the ID 'abc123813abc'.

Example 402. Retrieving a feed of video responses

```
$yt = new Zend_Gdata_YouTube();
$videoFeed = $yt->getVideoResponseFeed('abc123813abc');
```

12.4. Retrieving video comments

The comments for each YouTube video can be retrieved in several ways. To retrieve the comments for the video with the ID 'abc123813abc', use the following code:

Example 403. Retrieving a feed of video comments from a video ID

```
$yt = new Zend_Gdata_YouTube();
$commentFeed = $yt->getVideoCommentFeed('abc123813abc');

foreach ($commentFeed as $commentEntry) {
    echo $commentEntry->title->text . "\n";
    echo $commentEntry->content->text . "\n\n\n";
}
```

Comments can also be retrieved for a video if you have a copy of the Zend_Gdata_YouTube_VideoEntry object:

Example 404. Retrieving a Feed of Video Comments from a Zend_Gdata_YouTube_VideoEntry

```
$yt = new Zend_Gdata_YouTube();
$videoEntry = $yt->getVideoEntry('abc123813abc');
// we don't know the video ID in this example, but we do have the URL
$commentFeed = $yt->getVideoCommentFeed(null,
                                        $videoEntry->comments->href);
```

12.5. Retrieving playlist feeds

The YouTube Data API provides information about users, including profiles, playlists, subscriptions, and more.

12.5.1. Retrieving the playlists of a user

The library provides a helper method to retrieve the playlists associated with a given user. To retrieve the playlists for the user 'liz':

Example 405. Retrieving the playlists of a user

```
$yt = new Zend_Gdata_YouTube();
$playlistListFeed = $yt->getPlaylistListFeed('liz');

foreach ($playlistListFeed as $playlistEntry) {
    echo $playlistEntry->title->text . "\n";
    echo $playlistEntry->description->text . "\n";
    echo $playlistEntry->getPlaylistVideoFeedUrl() . "\n\n\n";
}
```

12.5.2. Retrieving a specific playlist

The library provides a helper method to retrieve the videos associated with a given playlist. To retrieve the playlists for a specific playlist entry:

Example 406. Retrieving a specific playlist

```
$feedUrl = $playlistEntry->getPlaylistVideoFeedUrl();
$playlistVideoFeed = $yt->getPlaylistVideoFeed($feedUrl);
```

12.6. Retrieving a list of a user's subscriptions

A user can have several types of subscriptions: channel subscription, tag subscription, or favorites subscription. A Zend_Gdata_YouTube_SubscriptionEntry is used to represent individual subscriptions.

To retrieve all subscriptions for the user 'liz':

Example 407. Retrieving all subscriptions for a user

```
$yt = new Zend_Gdata_YouTube();
$subscriptionFeed = $yt->getSubscriptionFeed('liz');

foreach ($subscriptionFeed as $subscriptionEntry) {
    echo $subscriptionEntry->title->text . "\n";
}
```

12.7. Retrieving a user's profile

You can retrieve the public profile information for any YouTube user. To retrieve the profile for the user 'liz':

Example 408. Retrieving a user's profile

```
$yt = new Zend_Gdata_YouTube();
$userProfile = $yt->getUserProfile('liz');
echo "username: " . $userProfile->username->text . "\n";
echo "age: " . $userProfile->age->text . "\n";
echo "hometown: " . $userProfile->hometown->text . "\n";
```

12.8. Uploading Videos to YouTube

Please make sure to review the diagrams in the protocol guide on code.google.com for a high-level overview of the upload process. Uploading videos can be done in one of two ways: either by uploading the video directly or by sending just the video meta-data and having a user upload the video through an HTML form.

In order to upload a video directly, you must first construct a new Zend_Gdata_YouTube_VideoEntry object and specify some required meta-data The following example shows uploading the Quicktime video "mytestmovie.mov" to YouTube with the following properties:

Table 64. Metadata used in the code-sample below

Property	Value
Title	My Test Movie
Category	Autos
Keywords	cars, funny
Description	My description
Filename	mytestmovie.mov
File MIME type	video/quicktime
Video private?	FALSE
Video location	37, -122 (lat, long)
Developer Tags	mydevelopertag, anotherdevelopertag

The code below creates a blank Zend_Gdata_YouTube_VideoEntry to be uploaded. A Zend_Gdata_App_MediaFileSource object is then used to hold the actual video file. Under the hood, the Zend_Gdata_YouTube_Extension_MediaGroup object is used to hold all of the video's

meta-data. Our helper methods detailed below allow you to just set the video meta-data without having to worry about the media group object. The $uploadUrl is the location where the new entry gets posted to. This can be specified either with the $userName of the currently authenticated user, or, alternatively, you can simply use the string 'default' to refer to the currently authenticated user.

Example 409. Uploading a video

```
$yt = new Zend_Gdata_YouTube($httpClient);
$myVideoEntry = new Zend_Gdata_YouTube_VideoEntry();

$filesource = $yt->newMediaFileSource('mytestmovie.mov');
$filesource->setContentType('video/quicktime');
$filesource->setSlug('mytestmovie.mov');

$myVideoEntry->setMediaSource($filesource);

$myVideoEntry->setVideoTitle('My Test Movie');
$myVideoEntry->setVideoDescription('My Test Movie');
// Note that category must be a valid YouTube category !
$myVideoEntry->setVideoCategory('Comedy');

// Set keywords, note that this must be a comma separated string
// and that each keyword cannot contain whitespace
$myVideoEntry->SetVideoTags('cars, funny');

// Optionally set some developer tags
$myVideoEntry->setVideoDeveloperTags(array('mydevelopertag',
                                           'anotherdevelopertag'));

// Optionally set the video's location
$yt->registerPackage('Zend_Gdata_Geo');
$yt->registerPackage('Zend_Gdata_Geo_Extension');
$where = $yt->newGeoRssWhere();
$position = $yt->newGmlPos('37.0 -122.0');
$where->point = $yt->newGmlPoint($position);
$myVideoEntry->setWhere($where);

// Upload URI for the currently authenticated user
$uploadUrl =
    'http://uploads.gdata.youtube.com/feeds/users/default/uploads';

// Try to upload the video, catching a Zend_Gdata_App_HttpException
// if availableor just a regular Zend_Gdata_App_Exception

try {
    $newEntry = $yt->insertEntry($myVideoEntry,
                                 $uploadUrl,
                                 'Zend_Gdata_YouTube_VideoEntry');
} catch (Zend_Gdata_App_HttpException $httpException) {
    echo $httpException->getRawResponseBody();
} catch (Zend_Gdata_App_Exception $e) {
    echo $e->getMessage();
}
```

To upload a video as private, simply use: $myVideoEntry->setVideoPrivate(); prior to performing the upload. $videoEntry->isVideoPrivate() can be used to check whether a video entry is private or not.

12.9. Browser-based upload

Browser-based uploading is performed almost identically to direct uploading, except that you do not attach a Zend_Gdata_App_MediaFileSource object to the Zend_Gdata_YouTube_VideoEntry you are constructing. Instead you simply submit all of your video's meta-data to receive back a token element which can be used to construct an HTML upload form.

Example 410. Browser-based upload

```
$yt = new Zend_Gdata_YouTube($httpClient);

$myVideoEntry= new Zend_Gdata_YouTube_VideoEntry();
$myVideoEntry->setVideoTitle('My Test Movie');
$myVideoEntry->setVideoDescription('My Test Movie');

// Note that category must be a valid YouTube category
$myVideoEntry->setVideoCategory('Comedy');
$myVideoEntry->SetVideoTags('cars, funny');

$tokenHandlerUrl = 'http://gdata.youtube.com/action/GetUploadToken';
$tokenArray = $yt->getFormUploadToken($myVideoEntry, $tokenHandlerUrl);
$tokenValue = $tokenArray['token'];
$postUrl = $tokenArray['url'];
```

The above code prints out a link and a token that is used to construct an HTML form to display in the user's browser. A simple example form is shown below with $tokenValue representing the content of the returned token element, as shown being retrieved from $myVideoEntry above. In order for the user to be redirected to your website after submitting the form, make sure to append a $nextUrl parameter to the $postUrl above, which functions in the same way as the $next parameter of an AuthSub link. The only difference is that here, instead of a single-use token, a status and an id variable are returned in the URL.

Example 411. Browser-based upload: Creating the HTML form

```
// place to redirect user after upload
$nextUrl = 'http://mysite.com/youtube_uploads';

$form = '<form action="'. $postUrl .'?nexturl='. $nextUrl .
        '" method="post" enctype="multipart/form-data">'.
        '<input name="file" type="file"/>'.
        '<input name="token" type="hidden" value="'. $tokenValue .'"/>'.
        '<input value="Upload Video File" type="submit" />'.
        '</form>';
```

12.10. Checking upload status

After uploading a video, it will immediately be visible in an authenticated user's uploads feed. However, it will not be public on the site until it has been processed. Videos that have been rejected or failed to upload successfully will also only be in the authenticated user's uploads feed. The following code checks the status of a Zend_Gdata_YouTube_VideoEntry to see if it is not live yet or if it has been rejected.

Example 412. Checking video upload status

```
try {
    $control = $videoEntry->getControl();
} catch (Zend_Gdata_App_Exception $e) {
    echo $e->getMessage();
}

if ($control instanceof Zend_Gdata_App_Extension_Control) {
    if ($control->getDraft() != null &&
        $control->getDraft()->getText() == 'yes') {
        $state = $videoEntry->getVideoState();

        if ($state instanceof Zend_Gdata_YouTube_Extension_State) {
            print 'Upload status: '
                  . $state->getName()
                  .' '. $state->getText();
        } else {
            print 'Not able to retrieve the video status information'
                  .' yet. ' . "Please try again shortly.\n";
        }
    }
}
```

12.11. Other Functions

In addition to the functionality described above, the YouTube API contains many other functions that allow you to modify video meta-data, delete video entries and use the full range of community features on the site. Some of the community features that can be modified through the API include: ratings, comments, playlists, subscriptions, user profiles, contacts and messages.

Please refer to the full documentation available in the PHP Developer's Guide on code.google.com.

13. Catching Gdata Exceptions

The `Zend_Gdata_App_Exception` class is a base class for exceptions thrown by `Zend_Gdata`. You can catch any exception thrown by `Zend_Gdata` by catching `Zend_Gdata_App_Exception`.

```
try {
    $client =
        Zend_Gdata_ClientLogin::getHttpClient($username, $password);
} catch(Zend_Gdata_App_Exception $ex) {
    // Report the exception to the user
    die($ex->getMessage());
}
```

The following exception subclasses are used by `Zend_Gdata`:

- `Zend_Gdata_App_AuthException` indicates that the user's account credentials were not valid.
- `Zend_Gdata_App_BadMethodCallException` indicates that a method was called for a service that does not support the method. For example, the CodeSearch service does not support `post()`.

- `Zend_Gdata_App_HttpException` indicates that an HTTP request was not successful. Provides the ability to get the full `Zend_Http_Response` object to determine the exact cause of the failure in cases where **$e->getMessage()** does not provide enough details.
- `Zend_Gdata_App_InvalidArgumentException` is thrown when the application provides a value that is not valid in a given context. For example, specifying a Calendar visibility value of "banana", or fetching a Blogger feed without specifying any blog name.
- `Zend_Gdata_App_CaptchaRequiredException` is thrown when a ClientLogin attempt receives a CAPTCHA™ challenge from the authentication service. This exception contains a token ID and a URL to a CAPTCHA™ challenge image. The image is a visual puzzle that should be displayed to the user. After collecting the user's response to the challenge image, the response can be included with the next ClientLogin attempt.The user can alternatively be directed to this website: https://www.google.com/accounts/DisplayUnlockCaptcha Further information can be found in the ClientLogin documentation.

You can use these exception subclasses to handle specific exceptions differently. See the API documentation for information on which exception subclasses are thrown by which methods in `Zend_Gdata`.

```
try {
    $client = Zend_Gdata_ClientLogin::getHttpClient($username,
                                                    $password,
                                                    $service);
} catch(Zend_Gdata_App_AuthException $authEx) {
    // The user's credentials were incorrect.
    // It would be appropriate to give the user a second try.
    ...
} catch(Zend_Gdata_App_HttpException $httpEx) {
    // Google Data servers cannot be contacted.
    die($httpEx->getMessage);}
```

Zend_Http

1. Introduction

`Zend_Http_Client` provides an easy interface for preforming Hyper-Text Transfer Protocol (HTTP) requests. `Zend_Http_Client` supports most simple features expected from an HTTP client, as well as some more complex features such as HTTP authentication and file uploads. Successful requests (and most unsuccessful ones too) return a `Zend_Http_Response` object, which provides access to the response's headers and body (see Section 5, “Zend_Http_Response”).

1.1. Using Zend_Http_Client

The class constructor optionally accepts a URL as its first parameter (can be either a string or a Zend_Uri_Http object), and an array or Zend_Config object containing configuration options. Both can be left out, and set later using the setUri() and setConfig() methods.

Example 413. Instantiating a Zend_Http_Client Object

```
$client = new Zend_Http_Client('http://example.org', array(
    'maxredirects' => 0,
    'timeout'      => 30));

// This is actually exactly the same:
$client = new Zend_Http_Client();
$client->setUri('http://example.org');
$client->setConfig(array(
    'maxredirects' => 0,
    'timeout'      => 30));

// You can also use a Zend_Config object to set the client's configuration
$config = new Zend_Config_Ini('httpclient.ini, 'secure');
$client->setConfig($config);
```

Zend_Http_Client uses Zend_Uri_Http to validate URLs. This means that some special characters like the pipe symbol ('|') or the caret symbol ('^') will not be accepted in the URL by default. This can be modified by setting the 'allow_unwise' option of Zend_Uri to 'TRUE'. See Section 1.4.1, “Allowing "Unwise" characters in URIs” for more information.

1.2. Configuration Parameters

The constructor and setConfig() method accept an associative array of configuration parameters, or a `Zend_Config` object. Setting these parameters is optional, as they all have default values.

Table 65. Zend_Http_Client configuration parameters

Parameter	Description	Expected Values	Default Value
maxredirects	Maximum number of redirections to follow (0 = none)	integer	5
strict	Whether perform validation on header	boolean	TRUE

Parameter	Description	Expected Values	Default Value
	names. When set to FALSE, validation functions will be skipped. Usually this should not be changed		
strictredirects	Whether to strictly follow the RFC when redirecting (see Section 2.1, “HTTP Redirections”)	boolean	FALSE
useragent	User agent identifier string (sent in request headers)	string	'Zend_Http_Client'
timeout	Connection timeout (seconds)	integer	10
httpversion	HTTP protocol version (usually '1.1' or '1.0')	string	'1.1'
adapter	Connection adapter class to use (see Section 3, “Zend_Http_Client - Connection Adapters”)	mixed	'Zend_Http_Client_Adapter_Socket'
keepalive	Whether to enable keep-alive connections with the server. Useful and might improve performance if several consecutive requests to the same server are performed.	boolean	FALSE
storeresponse	Whether to store last response for later retrieval with `getLastResponse()`. If set to FALSE `getLastResponse()` will return `NULL`.	boolean	TRUE

1.3. Performing Basic HTTP Requests

Performing simple HTTP requests is very easily done using the request() method, and rarely needs more than three lines of code:

Example 414. Performing a Simple GET Request

```
$client = new Zend_Http_Client('http://example.org');
$response = $client->request();
```

The request() method takes one optional parameter - the request method. This can be either GET, POST, PUT, HEAD, DELETE, TRACE, OPTIONS or CONNECT as defined by the HTTP protocol [1]. For convenience, these are all defined as class constants: Zend_Http_Client::GET, Zend_Http_Client::POST and so on.

If no method is specified, the method set by the last setMethod() call is used. If setMethod() was never called, the default request method is GET (see the above example).

Example 415. Using Request Methods Other Than GET

```
// Preforming a POST request
$response = $client->request('POST');

// Yet another way of preforming a POST request
$client->setMethod(Zend_Http_Client::POST);
$response = $client->request();
```

1.4. Adding GET and POST parameters

Adding GET parameters to an HTTP request is quite simple, and can be done either by specifying them as part of the URL, or by using the setParameterGet() method. This method takes the GET parameter's name as its first parameter, and the GET parameter's value as its second parameter. For convenience, the setParameterGet() method can also accept a single associative array of name => value GET variables - which may be more comfortable when several GET parameters need to be set.

Example 416. Setting GET Parameters

```
// Setting a get parameter using the setParameterGet method
$client->setParameterGet('knight', 'lancelot');

// This is equivalent to setting such URL:
$client->setUri('http://example.com/index.php?knight=lancelot');

// Adding several parameters with one call
$client->setParameterGet(array(
    'first_name'  => 'Bender',
    'middle_name' => 'Bending'
    'made_in'     => 'Mexico',
));
```

While GET parameters can be sent with every request method, POST parameters are only sent in the body of POST requests. Adding POST parameters to a request is very similar to adding GET parameters, and can be done with the setParameterPost() method, which is similar to the setParameterGet() method in structure.

[1] See RFC 2616 - http://www.w3.org/Protocols/rfc2616/rfc2616.html.

Example 417. Setting POST Parameters

```
// Setting a POST parameter
$client->setParameterPost('language', 'fr');

// Setting several POST parameters, one of them with several values
$client->setParameterPost(array(
    'language'  => 'es',
    'country'   => 'ar',
    'selection' => array(45, 32, 80)
));
```

Note that when sending POST requests, you can set both GET and POST parameters. On the other hand, while setting POST parameters for a non-POST request will not trigger and error, it is useless. Unless the request is a POST request, POST parameters are simply ignored.

1.5. Accessing Last Request and Response

`Zend_Http_Client` provides methods of accessing the last request sent and last response received by the client object. `Zend_Http_Client->getLastRequest()` takes no parameters and returns the last HTTP request sent by the client as a string. Similarly, `Zend_Http_Client->getLastResponse()` returns the last HTTP response received by the client as a Zend_Http_Response object.

2. Zend_Http_Client - Advanced Usage

2.1. HTTP Redirections

By default, `Zend_Http_Client` automatically handles HTTP redirections, and will follow up to 5 redirections. This can be changed by setting the 'maxredirects' configuration parameter.

According to the HTTP/1.1 RFC, HTTP 301 and 302 responses should be treated by the client by resending the same request to the specified location - using the same request method. However, most clients to not implement this and always use a GET request when redirecting. By default, Zend_Http_Client does the same - when redirecting on a 301 or 302 response, all GET and POST parameters are reset, and a GET request is sent to the new location. This behavior can be changed by setting the 'strictredirects' configuration parameter to boolean TRUE:

Example 418. Forcing RFC 2616 Strict Redirections on 301 and 302 Responses

```
// Strict Redirections
$client->setConfig(array('strictredirects' => true));

// Non-strict Redirections
$client->setConfig(array('strictredirects' => false));
```

You can always get the number of redirections done after sending a request using the getRedirectionsCount() method.

2.2. Adding Cookies and Using Cookie Persistence

Zend_Http_Client provides an easy interface for adding cookies to your request, so that no direct header modification is required. This is done using the setCookie() method. This method can be used in several ways:

Example 419. Setting Cookies Using setCookie()

```
// Easy and simple: by providing a cookie name and cookie value
$client->setCookie('flavor', 'chocolate chips');

// By directly providing a raw cookie string (name=value)
// Note that the value must be already URL encoded
$client->setCookie('flavor=chocolate%20chips');

// By providing a Zend_Http_Cookie object
$cookie = Zend_Http_Cookie::fromString('flavor=chocolate%20chips');
$client->setCookie($cookie);
```

For more information about Zend_Http_Cookie objects, see Section 4, “Zend_Http_Cookie and Zend_Http_CookieJar”.

Zend_Http_Client also provides the means for cookie stickiness - that is having the client internally store all sent and received cookies, and resend them automatically on subsequent requests. This is useful, for example when you need to log in to a remote site first and receive and authentication or session ID cookie before sending further requests.

Example 420. Enabling Cookie Stickiness

```
// To turn cookie stickiness on, set a Cookie Jar
$client->setCookieJar();

// First request: log in and start a session
$client->setUri('http://example.com/login.php');
$client->setParameterPost('user', 'h4x0r');
$client->setParameterPost('password', '1337');
$client->request('POST');

// The Cookie Jar automatically stores the cookies set
// in the response, like a session ID cookie.

// Now we can send our next request - the stored cookies
// will be automatically sent.
$client->setUri('http://example.com/read_member_news.php');
$client->request('GET');
```

For more information about the Zend_Http_CookieJar class, see Section 4.5, “The Zend_Http_CookieJar Class: Instantiation”.

2.3. Setting Custom Request Headers

Setting custom headers can be done by using the setHeaders() method. This method is quite diverse and can be used in several ways, as the following example shows:

Example 421. Setting A Single Custom Request Header

```
// Setting a single header, overwriting any previous value
$client->setHeaders('Host', 'www.example.com');

// Another way of doing the exact same thing
$client->setHeaders('Host: www.example.com');

// Setting several values for the same header
// (useful mostly for Cookie headers):
$client->setHeaders('Cookie', array(
    'PHPSESSID=1234567890abcdef1234567890abcdef',
    'language=he'
));
```

setHeader() can also be easily used to set multiple headers in one call, by providing an array of headers as a single parameter:

Example 422. Setting Multiple Custom Request Headers

```
// Setting multiple headers, overwriting any previous value
$client->setHeaders(array(
    'Host' => 'www.example.com',
    'Accept-encoding' => 'gzip,deflate',
    'X-Powered-By' => 'Zend Framework'));

// The array can also contain full array strings:
$client->setHeaders(array(
    'Host: www.example.com',
    'Accept-encoding: gzip,deflate',
    'X-Powered-By: Zend Framework'));
```

2.4. File Uploads

You can upload files through HTTP using the setFileUpload method. This method takes a file name as the first parameter, a form name as the second parameter, and data as a third optional parameter. If the third data parameter is `NULL`, the first file name parameter is considered to be a real file on disk, and Zend_Http_Client will try to read this file and upload it. If the data parameter is not `NULL`, the first file name parameter will be sent as the file name, but no actual file needs to exist on the disk. The second form name parameter is always required, and is equivalent to the "name" attribute of an >input< tag, if the file was to be uploaded through an HTML form. A fourth optional parameter provides the file's content-type. If not specified, and Zend_Http_Client reads the file from the disk, the mime_content_type function will be used to guess the file's content type, if it is available. In any case, the default MIME type will be application/octet-stream.

Example 423. Using setFileUpload to Upload Files

```
// Uploading arbitrary data as a file
$text = 'this is some plain text';
$client->setFileUpload('some_text.txt', 'upload', $text, 'text/plain');

// Uploading an existing file
$client->setFileUpload('/tmp/Backup.tar.gz', 'bufile');

// Send the files
$client->request('POST');
```

In the first example, the $text variable is uploaded and will be available as $_FILES['upload'] on the server side. In the second example, the existing file /tmp/Backup.tar.gz is uploaded to the server and will be available as $_FILES['bufile']. The content type will be guesses automatically if possible - and if not, the content type will be set to 'application/octet-stream'.

Uploading files

When uploading files, the HTTP request content-type is automatically set to multipart/form-data. Keep in mind that you must send a POST or PUT request in order to upload files. Most servers will ignore the requests body on other request methods.

2.5. Sending Raw POST Data

You can use a Zend_Http_Client to send raw POST data using the setRawData() method. This method takes two parameters: the first is the data to send in the request body. The second optional parameter is the content-type of the data. While this parameter is optional, you should usually set it before sending the request - either using setRawData(), or with another method: setEncType().

Example 424. Sending Raw POST Data

```
$xml = '<book>' .
       '  <title>Islands in the Stream</title>' .
       '  <author>Ernest Hemingway</author>' .
       '  <year>1970</year>' .
       '</book>';

$client->setRawData($xml, 'text/xml')->request('POST');

// Another way to do the same thing:
$client->setRawData($xml)->setEncType('text/xml')->request('POST');
```

The data should be available on the server side through PHP's $HTTP_RAW_POST_DATA variable or through the php://input stream.

Using raw POST data

Setting raw POST data for a request will override any POST parameters or file uploads. You should not try to use both on the same request. Keep in mind that most servers will ignore the request body unless you send a POST request.

2.6. HTTP Authentication

Currently, Zend_Http_Client only supports basic HTTP authentication. This feature is utilized using the `setAuth()` method, or by specifying a username and a password in the URI. The `setAuth()` method takes 3 parameters: The user name, the password and an optional authentication type parameter. As mentioned, currently only basic authentication is supported (digest authentication support is planned).

Example 425. Setting HTTP Authentication User and Password

```
// Using basic authentication
$client->setAuth('shahar', 'myPassword!', Zend_Http_Client::AUTH_BASIC);

// Since basic auth is default, you can just do this:
$client->setAuth('shahar', 'myPassword!');

// You can also specify username and password in the URI
$client->setUri('http://christer:secret@example.com');
```

2.7. Sending Multiple Requests With the Same Client

`Zend_Http_Client` was also designed specifically to handle several consecutive requests with the same object. This is useful in cases where a script requires data to be fetched from several places, or when accessing a specific HTTP resource requires logging in and obtaining a session cookie, for example.

When performing several requests to the same host, it is highly recommended to enable the 'keepalive' configuration flag. This way, if the server supports keep-alive connections, the connection to the server will only be closed once all requests are done and the Client object is destroyed. This prevents the overhead of opening and closing TCP connections to the server.

When you perform several requests with the same client, but want to make sure all the request-specific parameters are cleared, you should use the resetParameters() method. This ensures that GET and POST parameters, request body and request-specific headers are reset and are not reused in the next request.

Resetting parameters

Note that non-request specific headers are not reset by default when the `resetParameters()` method is used. Only the 'Content-length' and 'Content-type' headers are reset. This allows you to set-and-forget headers like 'Accept-language' and 'Accept-encoding'

To clean all headers and other data except for URI and method, use `resetParameters(true)`.

Another feature designed specifically for consecutive requests is the Cookie Jar object. Cookie Jars allow you to automatically save cookies set by the server in the first request, and send them on consecutive requests transparently. This allows, for example, going through an authentication request before sending the actual data fetching request.

If your application requires one authentication request per user, and consecutive requests might be performed in more than one script in your application, it might be a good idea to store the Cookie Jar object in the user's session. This way, you will only need to authenticate the user once every session.

Example 426. Performing consecutive requests with one client

```
// First, instantiate the client
$client = new Zend_Http_Client('http://www.example.com/fetchdata.php', array(
    'keepalive' => true
));

// Do we have the cookies stored in our session?
if (isset($_SESSION['cookiejar']) &&
    $_SESSION['cookiejar'] instanceof Zend_Http_CookieJar)) {

    $client->setCookieJar($_SESSION['cookiejar']);
} else {
    // If we don't, authenticate and store cookies
    $client->setCookieJar();
    $client->setUri('http://www.example.com/login.php');
    $client->setParameterPost(array(
        'user' => 'shahar',
        'pass' => 'somesecret'
    ));
    $client->request(Zend_Http_Client::POST);

    // Now, clear parameters and set the URI to the original one
    // (note that the cookies that were set by the server are now
    // stored in the jar)
    $client->resetParameters();
    $client->setUri('http://www.example.com/fetchdata.php');
}

$response = $client->request(Zend_Http_Client::GET);

// Store cookies in session, for next page
$_SESSION['cookiejar'] = $client->getCookieJar();
```

2.8. Data Streaming

By default, `Zend_Http_Client` accepts and returns data as PHP strings. However, in many cases there are big files to be sent or received, thus keeping them in memory might be unnecessary or too expensive. For these cases, `Zend_Http_Client` supports reading data from files (and in general, PHP streams) and writing data to files (streams).

In order to use stream to pass data to `Zend_Http_Client`, use `setRawData()` method with data argument being stream resource (e.g., result of `fopen()`).

Example 427. Sending file to HTTP server with streaming

```
$fp = fopen("mybigfile.zip", "r");
$client->setRawData($fp, 'application/zip')->request('PUT');
```

Only PUT requests currently support sending streams to HTTP server.

In order to receive data from the server as stream, use `setStream()`. Optional argument specifies the filename where the data will be stored. If the argument is just `TRUE` (default), temporary file will be used and will be deleted once response object is destroyed. Setting argument to `FALSE` disables the streaming functionality.

When using streaming, `request()` method will return object of class `Zend_Http_Client_Response_Stream`, which has two useful methods:

`getStreamName()` will return the name of the file where the response is stored, and `getStream()` will return stream from which the response could be read.

You can either write the response to pre-defined file, or use temporary file for storing it and send it out or write it to another file using regular stream functions.

Example 428. Receiving file from HTTP server with streaming

```
$client->setStreaming(); // will use temp file
$response = $client->request('GET');
// copy file
copy($response->getStreamName(), "my/downloads/file");
// use stream
$fp = fopen("my/downloads/file2", "w");
stream_copy_to_stream($response->getStream(), $fp);
// Also can write to known file
$client->setStreaming("my/downloads/myfile)->request('GET');
```

3. Zend_Http_Client - Connection Adapters

3.1. Overview

`Zend_Http_Client` is based on a connection adapter design. The connection adapter is the object in charge of performing the actual connection to the server, as well as writing requests and reading responses. This connection adapter can be replaced, and you can create and extend the default connection adapters to suite your special needs, without the need to extend or replace the entire HTTP client class, and with the same interface.

Currently, the `Zend_Http_Client` class provides four built-in connection adapters:

- `Zend_Http_Client_Adapter_Socket` (default)
- `Zend_Http_Client_Adapter_Proxy`
- `Zend_Http_Client_Adapter_Curl`
- `Zend_Http_Client_Adapter_Test`

The `Zend_Http_Client` object's adapter connection adapter is set using the 'adapter' configuration option. When instantiating the client object, you can set the 'adapter' configuration option to a string containing the adapter's name (eg. 'Zend_Http_Client_Adapter_Socket') or to a variable holding an adapter object (eg. `new Zend_Http_Client_Adapter_Test`). You can also set the adapter later, using the `Zend_Http_Client->setConfig()` method.

3.2. The Socket Adapter

The default connection adapter is the `Zend_Http_Client_Adapter_Socket` adapter - this adapter will be used unless you explicitly set the connection adapter. The Socket adapter is based on PHP's built-in fsockopen() function, and does not require any special extensions or compilation flags.

The Socket adapter allows several extra configuration options that can be set using `Zend_Http_Client->setConfig()` or passed to the client constructor.

Table 66. Zend_Http_Client_Adapter_Socket configuration parameters

Parameter	Description	Expected Type	Default Value
persistent	Whether to use persistent TCP connections	boolean	FALSE
ssltransport	SSL transport layer (eg. 'sslv2', 'tls')	string	ssl
sslcert	Path to a PEM encoded SSL certificate	string	NULL
sslpassphrase	Passphrase for the SSL certificate file	string	NULL

Persistent TCP Connections

Using persistent TCP connections can potentially speed up HTTP requests - but in most use cases, will have little positive effect and might overload the HTTP server you are connecting to.

It is recommended to use persistent TCP connections only if you connect to the same server very frequently, and are sure that the server is capable of handling a large number of concurrent connections. In any case you are encouraged to benchmark the effect of persistent connections on both the client speed and server load before using this option.

Additionally, when using persistent connections it is recommended to enable Keep-Alive HTTP requests as described in Section 1.2, “Configuration Parameters” - otherwise persistent connections might have little or no effect.

HTTPS SSL Stream Parameters

`ssltransport`, `sslcert` and `sslpassphrase` are only relevant when connecting using HTTPS.

While the default SSL settings should work for most applications, you might need to change them if the server you are connecting to requires special client setup. If so, you should read the sections about SSL transport layers and options here.

Example 429. Changing the HTTPS transport layer

```
// Set the configuration parameters
$config = array(
    'adapter'      => 'Zend_Http_Client_Adapter_Socket',
    'ssltransport' => 'tls'
);

// Instantiate a client object
$client = new Zend_Http_Client('https://www.example.com', $config);

// The following request will be sent over a TLS secure connection.
$response = $client->request();
```

The result of the example above will be similar to opening a TCP connection using the following PHP command:

```
fsockopen('tls://www.example.com', 443)
```

3.2.1. Customizing and accessing the Socket adapter stream context

Starting from Zend Framework 1.9, `Zend_Http_Client_Adapter_Socket` provides direct access to the underlying stream context used to connect to the remote server. This allows the user to pass specific options and parameters to the TCP stream, and to the SSL wrapper in case of HTTPS connections.

You can access the stream context using the following methods of `Zend_Http_Client_Adapter_Socket`:

- *`setStreamContext($context)`* Sets the stream context to be used by the adapter. Can accept either a stream context resource created using the `stream_context_create()` PHP function, or an array of stream context options, in the same format provided to this function. Providing an array will create a new stream context using these options, and set it.

- *`getStreamContext()`* Get the stream context of the adapter. If no stream context was set, will create a default stream context and return it. You can then set or get the value of different context options using regular PHP stream context functions.

Example 430. Setting stream context options for the Socket adapter

```
// Array of options
$options = array(
    'socket' => array(
        // Bind local socket side to a specific interface
        'bindto' => '10.1.2.3:50505'
    ),
    'ssl' => array(
        // Verify server side certificate,
        // do not accept invalid or self-signed SSL certificates
        'verify_peer' => true,
        'allow_self_signed' => false,

        // Capture the peer's certificate
        'capture_peer_cert' => true
    )
);

// Create an adapter object and attach it to the HTTP client
$adapter = new Zend_Http_Client_Adapter_Socket();
$client = new Zend_Http_Client();
$client->setAdapter($adapter);

// Method 1: pass the options array to setStreamContext()
$adapter->setStreamContext($options);

// Method 2: create a stream context and pass it to setStreamContext()
$context = stream_context_create($options);
$adapter->setStreamContext($context);

// Method 3: get the default stream context and set the options on it
$context = $adapter->getStreamContext();
stream_context_set_option($context, $options);

// Now, preform the request
$response = $client->request();

// If everything went well, you can now access the context again
$opts = stream_context_get_options($adapter->getStreamContext());
echo $opts['ssl']['peer_certificate'];
```

Note that you must set any stream context options before using the adapter to preform actual requests. If no context is set before preforming HTTP requests with the Socket adapter, a default stream context will be created. This context resource could be accessed after preforming any requests using the `getStreamContext()` method.

3.3. The Proxy Adapter

The `Zend_Http_Client_Adapter_Proxy` adapter is similar to the default Socket adapter - only the connection is made through an HTTP proxy server instead of a direct connection to the target server. This allows usage of `Zend_Http_Client` behind proxy servers - which is sometimes needed for security or performance reasons.

Using the Proxy adapter requires several additional configuration parameters to be set, in addition to the default 'adapter' option:

Table 67. Zend_Http_Client configuration parameters

Parameter	Description	Expected Type	Example Value
proxy_host	Proxy server address	string	'proxy.myhost.com' or '10.1.2.3'
proxy_port	Proxy server TCP port	integer	8080 (default) or 81
proxy_user	Proxy user name, if required	string	'shahar' or '' for none (default)
proxy_pass	Proxy password, if required	string	'secret' or '' for none (default)
proxy_auth	Proxy HTTP authentication type	string	Zend_Http_Client::AUTH_BASIC (default)

proxy_host should always be set - if it is not set, the client will fall back to a direct connection using `Zend_Http_Client_Adapter_Socket`. proxy_port defaults to '8080' - if your proxy listens on a different port you must set this one as well.

proxy_user and proxy_pass are only required if your proxy server requires you to authenticate. Providing these will add a 'Proxy-Authentication' header to the request. If your proxy does not require authentication, you can leave these two options out.

proxy_auth sets the proxy authentication type, if your proxy server requires authentication. Possibly values are similar to the ones accepted by the Zend_Http_Client::setAuth() method. Currently, only basic authentication (Zend_Http_Client::AUTH_BASIC) is supported.

Example 431. Using Zend_Http_Client behind a proxy server

```
// Set the configuration parameters
$config = array(
    'adapter'    => 'Zend_Http_Client_Adapter_Proxy',
    'proxy_host' => 'proxy.int.zend.com',
    'proxy_port' => 8000,
    'proxy_user' => 'shahar.e',
    'proxy_pass' => 'bananashaped'
);

// Instantiate a client object
$client = new Zend_Http_Client('http://www.example.com', $config);

// Continue working...
```

As mentioned, if proxy_host is not set or is set to a blank string, the connection will fall back to a regular direct connection. This allows you to easily write your application in a way that allows a proxy to be used optionally, according to a configuration parameter.

Since the proxy adapter inherits from `Zend_Http_Client_Adapter_Socket`, you can use the stream context access method (see Section 3.2.1, “Customizing and accessing the Socket adapter stream context”) to set stream context options on Proxy connections as demonstrated above.

3.4. The cURL Adapter

cURL is a standard HTTP client library that is distributed with many operating systems and can be used in PHP via the cURL extension. It offers functionality for many special cases which can

occur for a HTTP client and make it a perfect choice for a HTTP adapter. It supports secure connections, proxy, all sorts of authentication mechanisms and shines in applications that move large files around between servers.

Example 432. Setting cURL options

```
$config = array(
    'adapter'   => 'Zend_Http_Client_Adapter_Curl',
    'curloptions' => array(CURLOPT_FOLLOWLOCATION => true),
);
$client = new Zend_Http_Client($uri, $config);
```

By default the cURL adapter is configured to behave exactly like the Socket Adapter and it also accepts the same configuration parameters as the Socket and Proxy adapters. You can also change the cURL options by either specifying the 'curloptions' key in the constructor of the adapter or by calling `setCurlOption($name, $value)`. The `$name` key corresponds to the CURL_* constants of the cURL extension. You can get access to the Curl handle by calling `$adapter->getHandle();`

Example 433. Transfering Files by Handle

You can use cURL to transfer very large files over HTTP by filehandle.

```
$putFileSize   = filesize("filepath");
$putFileHandle = fopen("filepath", "r");

$adapter = new Zend_Http_Client_Adapter_Curl();
$client = new Zend_Http_Client();
$client->setAdapter($adapter);
$adapter->setConfig(array(
    'curloptions' => array(
        CURLOPT_INFILE => $putFileHandle,
        CURLOPT_INFILESIZE => $putFileSize
    )
));
$client->request("PUT");
```

3.5. The Test Adapter

Sometimes, it is very hard to test code that relies on HTTP connections. For example, testing an application that pulls an RSS feed from a remote server will require a network connection, which is not always available.

For this reason, the `Zend_Http_Client_Adapter_Test` adapter is provided. You can write your application to use `Zend_Http_Client`, and just for testing purposes, for example in your unit testing suite, you can replace the default adapter with a Test adapter (a mock object), allowing you to run tests without actually performing server connections.

The `Zend_Http_Client_Adapter_Test` adapter provides an additional method, setResponse() method. This method takes one parameter, which represents an HTTP response as either text or a `Zend_Http_Response` object. Once set, your Test adapter will always return this response, without even performing an actual HTTP request.

Example 434. Testing Against a Single HTTP Response Stub

```
// Instantiate a new adapter and client
$adapter = new Zend_Http_Client_Adapter_Test();
$client = new Zend_Http_Client('http://www.example.com', array(
    'adapter' => $adapter
));

// Set the expected response
$adapter->setResponse(
    "HTTP/1.1 200 OK"        . "\r\n" .
    "Content-type: text/xml" . "\r\n" .
                               "\r\n" .
    '<?xml version="1.0" encoding="UTF-8"?>' .
    '<rss version="2.0" ' .
    '     xmlns:content="http://purl.org/rss/1.0/modules/content/"' .
    '     xmlns:wfw="http://wellformedweb.org/CommentAPI/"' .
    '     xmlns:dc="http://purl.org/dc/elements/1.1/">' .
    '  <channel>' .
    '    <title>Premature Optimization</title>' .
    // and so on...
    '</rss>');

$response = $client->request('GET');
// .. continue parsing $response..
```

The above example shows how you can preset your HTTP client to return the response you need. Then, you can continue testing your own code, without being dependent on a network connection, the server's response, etc. In this case, the test would continue to check how the application parses the XML in the response body.

Sometimes, a single method call to an object can result in that object performing multiple HTTP transactions. In this case, it's not possible to use setResponse() alone because there's no opportunity to set the next response(s) your program might need before returning to the caller.

Example 435. Testing Against Multiple HTTP Response Stubs

```
// Instantiate a new adapter and client
$adapter = new Zend_Http_Client_Adapter_Test();
$client = new Zend_Http_Client('http://www.example.com', array(
    'adapter' => $adapter
));

// Set the first expected response
$adapter->setResponse(
    "HTTP/1.1 302 Found"      . "\r\n" .
    "Location: /"             . "\r\n" .
    "Content-Type: text/html" . "\r\n" .
                                "\r\n" .
    '<html>' .
    '  <head><title>Moved</title></head>' .
    '  <body><p>This page has moved.</p></body>' .
    '</html>');

// Set the next successive response
$adapter->addResponse(
    "HTTP/1.1 200 OK"         . "\r\n" .
    "Content-Type: text/html" . "\r\n" .
                                "\r\n" .
    '<html>' .
    '  <head><title>My Pet Store Home Page</title></head>' .
    '  <body><p>...</p></body>' .
    '</html>');

// inject the http client object ($client) into your object
// being tested and then test your object's behavior below
```

The setResponse() method clears any responses in the `Zend_Http_Client_Adapter_Test`'s buffer and sets the first response that will be returned. The addResponse() method will add successive responses.

The responses will be replayed in the order that they were added. If more requests are made than the number of responses stored, the responses will cycle again in order.

In the example above, the adapter is configured to test your object's behavior when it encounters a 302 redirect. Depending on your application, following a redirect may or may not be desired behavior. In our example, we expect that the redirect will be followed and we configure the test adapter to help us test this. The initial 302 response is set up with the setResponse() method and the 200 response to be returned next is added with the addResponse() method. After configuring the test adapter, inject the HTTP client containing the adapter into your object under test and test its behavior.

If you need the adapter to fail on demand you can use `setNextRequestWillFail($flag)`. The method will cause the next call to `connect()` to throw an `Zend_Http_Client_Adapter_Exception` exception. This can be useful when your application caches content from an external site (in case the site goes down) and you want to test this feature.

Example 436. Forcing the adapter to fail

```
// Instantiate a new adapter and client
$adapter = new Zend_Http_Client_Adapter_Test();
$client = new Zend_Http_Client('http://www.example.com', array(
    'adapter' => $adapter
));

// Force the next request to fail with an exception
$adapter->nextRequestWillFail(true);

try {
    // This call will result in a Zend_Http_Client_Adapter_Exception
    $client->request();
} catch (Zend_Http_Client_Adapter_Exception $e) {
    // ...
}

// Further requests will work as expected until
// you call setNextRequestWillFail(true) again
```

3.6. Creating your own connection adapters

You can create your own connection adapters and use them. You could, for example, create a connection adapter that uses persistent sockets, or a connection adapter with caching abilities, and use them as needed in your application.

In order to do so, you must create your own adapter class that implements the `Zend_Http_Client_Adapter_Interface` interface. The following example shows the skeleton of a user-implemented adapter class. All the public functions defined in this example must be defined in your adapter as well:

```
     *
     * @param array $config
     */
    public function setConfig($config = array())
    {
        // This rarely changes - you should usually copy the
        // implementation in Zend_Http_Client_Adapter_Socket.
    }

    /**
     * Connect to the remote server
     *
     * @param string  $host
     * @param int     $port
     * @param boolean $secure
     */
    public function connect($host, $port = 80, $secure = false)
    {
        // Set up the connection to the remote server
    }

    /**
     * Send request to the remote server
     *
     * @param string        $method
     * @param Zend_Uri_Http $url
     * @param string        $http_ver
     * @param array         $headers
     * @param string        $body
     * @return string Request as text
     */
    public function write($method,
                          $url,
                          $http_ver = '1.1',
                          $headers = array(),
                          $body = '')
    {
        // Send request to the remote server.
        // This function is expected to return the full request
        // (headers and body) as a string
    }

    /**
     * Read response from server
     *
     * @return string
     */
    public function read()
    {
        // Read response from remote server and return it as a string
    }

    /**
     * Close the connection to the server
     *
     */
    public function close()
    {
        // Close the connection to the remote server - called last.
    }
}

// Then, you could use this adapter:
$client = new Zend_Http_Client(array(
    'adapter' => 'MyApp_Http_Client_Adapter_BananaProtocol'
));
```

4. Zend_Http_Cookie and Zend_Http_CookieJar

4.1. Introduction

`Zend_Http_Cookie`, as expected, is a class that represents an HTTP cookie. It provides methods for parsing HTTP response strings, collecting cookies, and easily accessing their properties. It also allows checking if a cookie matches against a specific scenario, IE a request URL, expiration time, secure connection, etc.

`Zend_Http_CookieJar` is an object usually used by `Zend_Http_Client` to hold a set of `Zend_Http_Cookie` objects. The idea is that if a `Zend_Http_CookieJar` object is attached to a `Zend_Http_Client` object, all cookies going from and into the client through HTTP requests and responses will be stored by the CookieJar object. Then, when the client will send another request, it will first ask the CookieJar object for all cookies matching the request. These will be added to the request headers automatically. This is highly useful in cases where you need to maintain a user session over consecutive HTTP requests, automatically sending the session ID cookies when required. Additionally, the `Zend_Http_CookieJar` object can be serialized and stored in $_SESSION when needed.

4.2. Instantiating Zend_Http_Cookie Objects

Instantiating a Cookie object can be done in two ways:

- Through the constructor, using the following syntax: `new Zend_Http_Cookie(string $name, string $value, string $domain, [int $expires, [string $path, [boolean $secure]]]);`
  - `$name`: The name of the cookie (eg. 'PHPSESSID') (required)
  - `$value`: The value of the cookie (required)
  - `$domain`: The cookie's domain (eg. '.example.com') (required)
  - `$expires`: Cookie expiration time, as UNIX time stamp (optional, defaults to `NULL`). If not set, cookie will be treated as a 'session cookie' with no expiration time.
  - `$path`: Cookie path, eg. '/foo/bar/' (optional, defaults to '/')
  - `$secure`: Boolean, Whether the cookie is to be sent over secure (HTTPS) connections only (optional, defaults to boolean FALSE)
- By calling the fromString() static method, with a cookie string as represented in the 'Set-Cookie' HTTP response header or 'Cookie' HTTP request header. In this case, the cookie value must already be encoded. When the cookie string does not contain a 'domain' part, you must provide a reference URI according to which the cookie's domain and path will be set.

Example 438. Instantiating a Zend_Http_Cookie object

```
// First, using the constructor. This cookie will expire in 2 hours
$cookie = new Zend_Http_Cookie('foo',
                               'bar',
                               '.example.com',
                               time() + 7200,
                               '/path');

// You can also take the HTTP response Set-Cookie header and use it.
// This cookie is similar to the previous one, only it will not expire, and
// will only be sent over secure connections
$cookie = Zend_Http_Cookie::fromString('foo=bar; domain=.example.com; ' .
                                       'path=/path; secure');

// If the cookie's domain is not set, you have to manually specify it
$cookie = Zend_Http_Cookie::fromString('foo=bar; secure;',
                                       'http://www.example.com/path');
```

When instantiating a cookie object using the `Zend_Http_Cookie`::fromString() method, the cookie value is expected to be URL encoded, as cookie strings should be. However, when using the constructor, the cookie value string is expected to be the real, decoded value.

A cookie object can be transferred back into a string, using the __toString() magic method. This method will produce a HTTP request "Cookie" header string, showing the cookie's name and value, and terminated by a semicolon (';'). The value will be URL encoded, as expected in a Cookie header:

Example 439. Stringifying a Zend_Http_Cookie object

```
// Create a new cookie
$cookie = new Zend_Http_Cookie('foo',
                               'two words',
                               '.example.com',
                               time() + 7200,
                               '/path');

// Will print out 'foo=two+words;' :
echo $cookie->__toString();

// This is actually the same:
echo (string) $cookie;

// In PHP 5.2 and higher, this also works:
echo $cookie;
```

4.3. Zend_Http_Cookie getter methods

Once a `Zend_Http_Cookie` object is instantiated, it provides several getter methods to get the different properties of the HTTP cookie:

- `string getName()`: Get the name of the cookie
- `string getValue()`: Get the real, decoded value of the cookie
- `string getDomain()`: Get the cookie's domain

- `string getPath()`: Get the cookie's path, which defaults to '/'
- `int getExpiryTime()`: Get the cookie's expiration time, as UNIX time stamp. If the cookie has no expiration time set, will return `NULL`.

Additionally, several boolean tester methods are provided:

- `boolean isSecure()`: Check whether the cookie is set to be sent over secure connections only. Generally speaking, if `TRUE` the cookie should only be sent over HTTPS.
- `boolean isExpired(int $time = null)`: Check whether the cookie is expired or not. If the cookie has no expiration time, will always return `TRUE`. If $time is provided, it will override the current time stamp as the time to check the cookie against.
- `boolean isSessionCookie()`: Check whether the cookie is a "session cookie" - that is a cookie with no expiration time, which is meant to expire when the session ends.

Example 440. Using getter methods with Zend_Http_Cookie

```
// First, create the cookie
$cookie =
    Zend_Http_Cookie::fromString('foo=two+words; ' +
                                 'domain=.example.com; ' +
                                 'path=/somedir; ' +
                                 'secure; ' +
                                 'expires=Wednesday, 28-Feb-05 20:41:22 UTC');

echo $cookie->getName();   // Will echo 'foo'
echo $cookie->getValue();  // will echo 'two words'
echo $cookie->getDomain(); // Will echo '.example.com'
echo $cookie->getPath();   // Will echo '/'

echo date('Y-m-d', $cookie->getExpiryTime());
// Will echo '2005-02-28'

echo ($cookie->isExpired() ? 'Yes' : 'No');
// Will echo 'Yes'

echo ($cookie->isExpired(strtotime('2005-01-01') ? 'Yes' : 'No');
// Will echo 'No'

echo ($cookie->isSessionCookie() ? 'Yes' : 'No');
// Will echo 'No'
```

4.4. Zend_Http_Cookie: Matching against a scenario

The only real logic contained in a `Zend_Http_Cookie` object, is in the match() method. This method is used to test a cookie against a given HTTP request scenario, in order to tell whether the cookie should be sent in this request or not. The method has the following syntax and parameters: `boolean Zend_Http_Cookie->match(mixed $uri, [boolean $matchSessionCookies, [int $now]]);`

- `mixed $uri`: A `Zend_Uri_Http` object with a domain name and path to be checked. Optionally, a string representing a valid HTTP URL can be passed instead. The cookie will match if the URL's scheme (HTTP or HTTPS), domain and path all match.
- `boolean $matchSessionCookies`: Whether session cookies should be matched or not. Defaults to `TRUE`. If set to `FALSE`, cookies with no expiration time will never match.

- `int $now`: Time (represented as UNIX time stamp) to check a cookie against for expiration. If not specified, will default to the current time.

Example 441. Matching cookies

```
// Create the cookie object - first, a secure session cookie
$cookie = Zend_Http_Cookie::fromString('foo=two+words; ' +
                                       'domain=.example.com; ' +
                                       'path=/somedir; ' +
                                       'secure;');

$cookie->match('https://www.example.com/somedir/foo.php');
// Will return true

$cookie->match('http://www.example.com/somedir/foo.php');
// Will return false, because the connection is not secure

$cookie->match('https://otherexample.com/somedir/foo.php');
// Will return false, because the domain is wrong

$cookie->match('https://example.com/foo.php');
// Will return false, because the path is wrong

$cookie->match('https://www.example.com/somedir/foo.php', false);
// Will return false, because session cookies are not matched

$cookie->match('https://sub.domain.example.com/somedir/otherdir/foo.php');
// Will return true

// Create another cookie object - now, not secure, with expiration time
// in two hours
$cookie = Zend_Http_Cookie::fromString('foo=two+words; ' +
                                       'domain=www.example.com; ' +
                                       'expires='
                                       . date(DATE_COOKIE, time() + 7200));

$cookie->match('http://www.example.com/');
// Will return true

$cookie->match('https://www.example.com/');
// Will return true - non secure cookies can go over secure connections
// as well!

$cookie->match('http://subdomain.example.com/');
// Will return false, because the domain is wrong

$cookie->match('http://www.example.com/', true, time() + (3 * 3600));
// Will return false, because we added a time offset of +3 hours to
// current time
```

4.5. The Zend_Http_CookieJar Class: Instantiation

In most cases, there is no need to directly instantiate a `Zend_Http_CookieJar` object. If you want to attach a new cookie jar to your `Zend_Http_Client` object, just call the Zend_Http_Client->setCookieJar() method, and a new, empty cookie jar will be attached to your client. You could later get this cookie jar using Zend_Http_Client->getCookieJar().

If you still wish to manually instantiate a CookieJar object, you can do so by calling "new Zend_Http_CookieJar()" directly - the constructor method does not take any parameters. Another

way to instantiate a CookieJar object is to use the static Zend_Http_CookieJar::fromResponse() method. This method takes two parameters: a `Zend_Http_Response` object, and a reference URI, as either a string or a `Zend_Uri_Http` object. This method will return a new `Zend_Http_CookieJar` object, already containing the cookies set by the passed HTTP response. The reference URI will be used to set the cookie's domain and path, if they are not defined in the Set-Cookie headers.

4.6. Adding Cookies to a Zend_Http_CookieJar object

Usually, the `Zend_Http_Client` object you attached your CookieJar object to will automatically add cookies set by HTTP responses to your jar. If you wish to manually add cookies to your jar, this can be done by using two methods:

- `Zend_Http_CookieJar->addCookie($cookie[, $ref_uri])`: Add a single cookie to the jar. $cookie can be either a `Zend_Http_Cookie` object or a string, which will be converted automatically into a Cookie object. If a string is provided, you should also provide $ref_uri - which is a reference URI either as a string or `Zend_Uri_Http` object, to use as the cookie's default domain and path.

- `Zend_Http_CookieJar->addCookiesFromResponse($response, $ref_uri)`: Add all cookies set in a single HTTP response to the jar. $response is expected to be a `Zend_Http_Response` object with Set-Cookie headers. $ref_uri is the request URI, either as a string or a `Zend_Uri_Http` object, according to which the cookies' default domain and path will be set.

4.7. Retrieving Cookies From a Zend_Http_CookieJar object

Just like with adding cookies, there is usually no need to manually fetch cookies from a CookieJar object. Your `Zend_Http_Client` object will automatically fetch the cookies required for an HTTP request for you. However, you can still use 3 provided methods to fetch cookies from the jar object: `getCookie()`, `getAllCookies()`, and `getMatchingCookies()`. Additionnaly, iterating over the CookieJar will let you retrieve all the `Zend_Http_Cookie` objects from it.

It is important to note that each one of these methods takes a special parameter, which sets the return type of the method. This parameter can have 3 values:

- `Zend_Http_CookieJar::COOKIE_OBJECT`: Return a `Zend_Http_Cookie` object. If the method returns more than one cookie, an array of objects will be returned.

- `Zend_Http_CookieJar::COOKIE_STRING_ARRAY`: Return cookies as strings, in a "foo=bar" format, suitable for sending in a HTTP request "Cookie" header. If more than one cookie is returned, an array of strings is returned.

- `Zend_Http_CookieJar::COOKIE_STRING_CONCAT`: Similar to COOKIE_STRING_ARRAY, but if more than one cookie is returned, this method will concatenate all cookies into a single, long string separated by semicolons (;), and return it. This is especially useful if you want to directly send all matching cookies in a single HTTP request "Cookie" header.

The structure of the different cookie-fetching methods is described below:

- `Zend_Http_CookieJar->getCookie($uri, $cookie_name[, $ret_as])`: Get a single cookie from the jar, according to its URI (domain and path) and name. $uri is either a string or a `Zend_Uri_Http` object representing the URI. $cookie_name is a string identifying the cookie name. $ret_as specifies the return type as described above. $ret_type is optional, and defaults to COOKIE_OBJECT.

- `Zend_Http_CookieJar->getAllCookies($ret_as)`: Get all cookies from the jar. $ret_as specifies the return type as described above. If not specified, $ret_type defaults to COOKIE_OBJECT.
- `Zend_Http_CookieJar->getMatchingCookies($uri[, $matchSessionCookies[, $ret_as[, $now]]])`: Get all cookies from the jar that match a specified scenario, that is a URI and expiration time.
  - `$uri` is either a `Zend_Uri_Http` object or a string specifying the connection type (secure or non-secure), domain and path to match against.
  - `$matchSessionCookies` is a boolean telling whether to match session cookies or not. Session cookies are cookies that have no specified expiration time. Defaults to TRUE.
  - `$ret_as` specifies the return type as described above. If not specified, defaults to COOKIE_OBJECT.
  - $now is an integer representing the UNIX time stamp to consider as "now" - that is any cookies who are set to expire before this time will not be matched. If not specified, defaults to the current time.

  You can read more about cookie matching here: Section 4.4, “Zend_Http_Cookie: Matching against a scenario”.

5. Zend_Http_Response

5.1. Introduction

`Zend_Http_Response` provides easy access to an HTTP responses message, as well as a set of static methods for parsing HTTP response messages. Usually, `Zend_Http_Response` is used as an object returned by a `Zend_Http_Client` request.

In most cases, a Zend_Http_Response object will be instantiated using the fromString() method, which reads a string containing an HTTP response message, and returns a new Zend_Http_Response object:

Example 442. Instantiating a Zend Http Response Object Using the Factory Method

```
$str = '';
$sock = fsockopen('www.example.com', 80);
$req =     "GET / HTTP/1.1\r\n" .
        "Host: www.example.com\r\n" .
        "Connection: close\r\n" .
        "\r\n";

fwrite($sock, $req);
while ($buff = fread($sock, 1024))
    $str .= $sock;

$response = Zend_Http_Response::fromString($str);
```

You can also use the contractor method to create a new response object, by specifying all the parameters of the response:

```
public function __construct($code, $headers, $body = null, $version =
'1.1', $message = null)
```

- `$code`: The HTTP response code (eg. 200, 404, etc.)
- `$headers`: An associative array of HTTP response headers (eg. 'Host' => 'example.com')
- `$body`: The response body as a string
- `$version`: The HTTP response version (usually 1.0 or 1.1)
- `$message`: The HTTP response message (eg 'OK', 'Internal Server Error'). If not specified, the message will be set according to the response code

5.2. Boolean Tester Methods

Once a `Zend_Http_Response` object is instantiated, it provides several methods that can be used to test the type of the response. These all return Boolean TRUE or FALSE:

- `Boolean isSuccessful()`: Whether the request was successful or not. Returns TRUE for HTTP 1xx and 2xx response codes
- `Boolean isError()`: Whether the response code implies an error or not. Returns TRUE for HTTP 4xx (client errors) and 5xx (server errors) response codes
- `Boolean isRedirect()`: Whether the response is a redirection response or not. Returns TRUE for HTTP 3xx response codes

Example 443. Using the isError() method to validate a response

```
if ($response->isError()) {
  echo "Error transmitting data.\n"
  echo "Server reply was: " . $response->getStatus() .
       " " . $response->getMessage() . "\n";
}
// .. process the response here...
```

5.3. Accessor Methods

The main goal of the response object is to provide easy access to various response parameters.

- `int getStatus()`: Get the HTTP response status code (eg. 200, 504, etc.)
- `string  getMessage()`: Get the HTTP response status message (eg. "Not Found", "Authorization Required")
- `string getBody()`: Get the fully decoded HTTP response body
- `string getRawBody()`: Get the raw, possibly encoded HTTP response body. If the body was decoded using GZIP encoding for example, it will not be decoded.
- `array  getHeaders()`: Get the HTTP response headers as an associative array (eg. 'Content-type' => 'text/html')
- `string|array getHeader($header)`: Get a specific HTTP response header, specified by $header
- `string getHeadersAsString($status_line = true, $br = "\n")`: Get the entire set of headers as a string. If $status_line is TRUE (default), the first status line (eg. "HTTP/1.1

200 OK") will also be returned. Lines are broken with the $br parameter (Can be, for example, "<br />")

- `string asString($br = "\n")`: Get the entire response message as a string. Lines are broken with the $br parameter (Can be, for example, "<br />"). You can also use the magic method __toString() when casting the object as a string. It will then proxy to asString()

Example 444. Using Zend_Http_Response Accessor Methods

```
if ($response->getStatus() == 200) {
  echo "The request returned the following information:<br />";
  echo $response->getBody();
} else {
  echo "An error occurred while fetching data:<br />";
  echo $response->getStatus() . ": " . $response->getMessage();
}
```

Always check return value

Since a response can contain several instances of the same header, the getHeader() method and getHeaders() method may return either a single string, or an array of strings for each header. You should always check whether the returned value is a string or array.

Example 445. Accessing Response Headers

```
$ctype = $response->getHeader('Content-type');
if (is_array($ctype)) $ctype = $ctype[0];

$body = $response->getBody();
if ($ctype == 'text/html' || $ctype == 'text/xml') {
  $body = htmlentities($body);
}

echo $body;
```

5.4. Static HTTP Response Parsers

The `Zend_Http_Response` class also includes several internally-used methods for processing and parsing HTTP response messages. These methods are all exposed as static methods, which means they can be used externally, even if you do not need to instantiate a response object, and just want to extract a specific part of the response.

- `int Zend_Http_Response::extractCode($response_str)`: Extract and return the HTTP response code (eg. 200 or 404) from $response_str
- `string Zend_Http_Response::extractMessage($response_str)`: Extract and return the HTTP response message (eg. "OK" or "File Not Found") from $response_str
- `string Zend_Http_Response::extractVersion($response_str)`: : Extract and return the HTTP version (eg. 1.1 or 1.0) from $response_str
- `array Zend_Http_Response::extractHeaders($response_str)`: Extract and return the HTTP response headers from $response_str as an array
- `string Zend_Http_Response::extractBody($response_str)`: Extract and return the HTTP response body from $response_str

- `string Zend_Http_Response::responseCodeAsText($code = null, $http11 = true)`: Get the standard HTTP response message for a response code $code. For example, will return "Internal Server Error" if $code is 500. If $http11 is TRUE (default), will return HTTP/1.1 standard messages - otherwise HTTP/1.0 messages will be returned. If $code is not specified, this method will return all known HTTP response codes as an associative (code => message) array.

Apart from parser methods, the class also includes a set of decoders for common HTTP response transfer encodings:

- `string Zend_Http_Response::decodeChunkedBody($body)`: Decode a complete "Content-Transfer-Encoding: Chunked" body
- `string Zend_Http_Response::decodeGzip($body)`: Decode a "Content-Encoding: gzip" body
- `string Zend_Http_Response::decodeDeflate($body)`: Decode a "Content-Encoding: deflate" body

Zend_InfoCard

1. Introduction

The Zend_InfoCard component implements relying-party support for Information Cards. Information Cards are used for identity management on the internet and authentication of users to web sites. The web sites that the user ultimately authenticates to are called *relying-parties*.

Detailed information about information cards and their importance to the internet identity metasystem can be found on the IdentityBlog.

1.1. Basic Theory of Usage

Usage of Zend_InfoCard can be done one of two ways: either as part of the larger Zend_Auth component via the Zend_InfoCard authentication adapter or as a stand-alone component. In both cases an information card can be requested from a user by using the following HTML block in your HTML login form:

```
<form action="http://example.com/server" method="POST">
  <input type='image' src='/images/ic.png' align='center'
        width='120px' style='cursor:pointer' />
  <object type="application/x-informationCard"
          name="xmlToken">
   <param name="tokenType"
         value="urn:oasis:names:tc:SAML:1.0:assertion" />
   <param name="requiredClaims"
         value="http://.../claims/privatepersonalidentifier
         http://.../claims/givenname
         http://.../claims/surname" />
 </object>
</form>
```

In the example above, the requiredClaims <param> tag is used to identify pieces of information known as claims (i.e. person's first name, last name) which the web site (a.k.a "relying party") needs in order a user to authenticate using an information card. For your reference, the full URI (for instance the givenname claim) is as follows: http://schemas.xmlsoap.org/ws/2005/05/identity/claims/givenname

When the above HTML is activated by a user (clicks on it), the browser will bring up a card selection program which not only shows them which information cards meet the requirements of the site, but also allows them to select which information card to use if multiple meet the criteria. This information card is transmitted as an XML document to the specified POST URL and is ready to be processed by the Zend_InfoCard component.

Note, Information cards can only be HTTP POSTed to SSL-encrypted URLs. Please consult your web server's documentation on how to set up SSL encryption.

1.2. Using as part of Zend_Auth

In order to use the component as part of the Zend_Auth authentication system, you must use the provided Zend_Auth_Adapter_InfoCard to do so (not available in the standalone Zend_InfoCard distribution). An example of its usage is shown below:

```
<?php
if (isset($_POST['xmlToken'])) {

    $adapter = new Zend_Auth_Adapter_InfoCard($_POST['xmlToken']);

    $adapter->addCertificatePair('/usr/local/Zend/apache2/conf/server.key',
                                 '/usr/local/Zend/apache2/conf/server.crt');

    $auth = Zend_Auth::getInstance();

    $result = $auth->authenticate($adapter);

    switch ($result->getCode()) {
        case Zend_Auth_Result::SUCCESS:
            $claims = $result->getIdentity();
            print "Given Name: {$claims->givenname}<br />";
            print "Surname: {$claims->surname}<br />";
            print "Email Address: {$claims->emailaddress}<br />";
            print "PPI: {$claims->getCardID()}<br />";
            break;
        case Zend_Auth_Result::FAILURE_CREDENTIAL_INVALID:
            print "The Credential you provided did not pass validation";
            break;
        default:
        case Zend_Auth_Result::FAILURE:
            print "There was an error processing your credentials.";
            break;
    }

    if (count($result->getMessages()) > 0) {
        print "<pre>";
        var_dump($result->getMessages());
        print "</pre>";
    }

}
?>
<hr />
<div id="login" style="font-family: arial; font-size: 2em;">
<p>Simple Login Demo</p>
 <form method="post">
  <input type="submit" value="Login" />
   <object type="application/x-informationCard" name="xmlToken">
    <param name="tokenType"
          value="urn:oasis:names:tc:SAML:1.0:assertion" />
    <param name="requiredClaims"
          value="http://.../claims/givenname
                 http://.../claims/surname
                 http://.../claims/emailaddress
                 http://.../claims/privatepersonalidentifier" />
  </object>
 </form>
</div>
```

In the example above, we first create an instance of the `Zend_Auth_Adapter_InfoCard` and pass the XML data posted by the card selector into it. Once an instance has been created you must then provide at least one SSL certificate public/private key pair used by the web server that received the `HTTP POST`. These files are used to validate the destination of the information posted to the server and are a requirement when using Information Cards.

Once the adapter has been configured, you can then use the standard `Zend_Auth` facilities to validate the provided information card token and authenticate the user by examining the identity provided by the `getIdentity()` method.

1.3. Using the Zend_InfoCard component standalone

It is also possible to use the `Zend_InfoCard` component as a standalone component by interacting with the `Zend_InfoCard` class directly. Using the `Zend_InfoCard` class is very similar to its use with the `Zend_Auth` component. An example of its use is shown below:

```
<?php
if (isset($_POST['xmlToken'])) {
    $infocard = new Zend_InfoCard();
    $infocard->addCertificatePair('/usr/local/Zend/apache2/conf/server.key',
                                  '/usr/local/Zend/apache2/conf/server.crt');

    $claims = $infocard->process($_POST['xmlToken']);

    if($claims->isValid()) {
        print "Given Name: {$claims->givenname}<br />";
        print "Surname: {$claims->surname}<br />";
        print "Email Address: {$claims->emailaddress}<br />";
        print "PPI: {$claims->getCardID()}<br />";
    } else {
        print "Error Validating identity: {$claims->getErrorMsg()}";
    }
}
?>
<hr />
<div id="login" style="font-family: arial; font-size: 2em;">
 <p>Simple Login Demo</p>
 <form method="post">
  <input type="submit" value="Login" />
   <object type="application/x-informationCard" name="xmlToken">
    <param name="tokenType"
          value="urn:oasis:names:tc:SAML:1.0:assertion" />
    <param name="requiredClaims"
          value="http://.../claims/givenname
                 http://.../claims/surname
                 http://.../claims/emailaddress
                 http://.../claims/privatepersonalidentifier" />
   </object>
 </form>
</div>
```

In the example above, we use the `Zend_InfoCard` component independently to validate the token provided by the user. As was the case with the `Zend_Auth_Adapter_InfoCard`, we create an instance of `Zend_InfoCard` and then set one or more SSL certificate public/private key pairs used by the web server. Once configured, we can use the `process()` method to process the information card and return the results.

1.4. Working with a Claims object

Regardless of whether the `Zend_InfoCard` component is used as a standalone component or as part of `Zend_Auth` via `Zend_Auth_Adapter_InfoCard`, the ultimate result of the processing of an information card is a `Zend_InfoCard_Claims` object. This object contains the assertions (a.k.a. claims) made by the submitting user based on the data requested by

your web site when the user authenticated. As shown in the examples above, the validity of the information card can be ascertained by calling the `Zend_InfoCard_Claims::isValid()` method. Claims themselves can either be retrieved by simply accessing the identifier desired (i.e. `givenname`) as a property of the object or through the `getClaim()` method.

In most cases you will never need to use the `getClaim()` method. However, if your `requiredClaims` mandate that you request claims from multiple different sources/namespaces then you will need to extract them explicitly using this method (simply pass it the full URI of the claim to retrieve its value from within the information card). Generally speaking however, the `Zend_InfoCard` component will set the default URI for claims to be the one used the most frequently within the information card itself and the simplified property-access method can be used.

As part of the validation process, it is the developer's responsibility to examine the issuing source of the claims contained within the information card and to decide if that source is a trusted source of information. To do so, the `getIssuer()` method is provided within the `Zend_InfoCard_Claims` object which returns the URI of the issuer of the information card claims.

1.5. Attaching Information Cards to existing accounts

It is possible to add support for information cards to an existing authentication system by storing the private personal identifier (PPI) to a previously traditionally-authenticated account and including at least the `http://schemas.xmlsoap.org/ws/2005/05/identity/claims/privatepersonalidentifier` claim as part of the `requiredClaims` of the request. If this claim is requested then the `Zend_InfoCard_Claims` object will provide a unique identifier for the specific card that was submitted by calling the `getCardID()` method.

An example of how to attach an information card to an existing traditional-authentication account is shown below:

```
// ...
public function submitinfocardAction()
{
    if (!isset($_REQUEST['xmlToken'])) {
        throw new ZBlog_Exception('Expected an encrypted token ' .
                                  'but was not provided');
    }

    $infoCard = new Zend_InfoCard();
    $infoCard->addCertificatePair(SSL_CERTIFICATE_PRIVATE,
                                  SSL_CERTIFICATE_PUB);

    try {
        $claims = $infoCard->process($request['xmlToken']);
    } catch(Zend_InfoCard_Exception $e) {
        // TODO Error processing your request
        throw $e;
    }

    if ($claims->isValid()) {
        $db = ZBlog_Data::getAdapter();

        $ppi = $db->quote($claims->getCardID());
        $fullname = $db->quote("{$claims->givenname} {$claims->surname}");

        $query = "UPDATE blogusers
```

```
                    SET ppi = $ppi,
                        real_name = $fullname
                  WHERE username='administrator'";

        try {
            $db->query($query);
        } catch(Exception $e) {
            // TODO Failed to store in DB
        }

        $this->view->render();
        return;
    } else {
        throw new
            ZBlog_Exception("Infomation card failed security checks");
    }
}
```

1.6. Creating Zend_InfoCard Adapters

The `Zend_InfoCard` component was designed to allow for growth in the information card standard through the use of a modular architecture. At this time, many of these hooks are unused and can be ignored, but there is one class that should be written for any serious information card implementation: the `Zend_InfoCard` adapter.

The `Zend_InfoCard` adapter is used as a callback mechanism within the component to perform various tasks, such as storing and retrieving Assertion IDs for information cards when they are processed by the component. While storing the assertion IDs of submitted information cards is not necessary, failing to do so opens up the possibility of the authentication scheme being compromised through a replay attack.

To prevent this, one must implement the `Zend_InfoCard_Adapter_Interface` and set an instance of this interface prior to calling either the `process()` (standalone) or `authenticate()` method as a `Zend_Auth` adapter. To set this interface, the `setAdapter()` method should be used. In the example below, we set a `Zend_InfoCard` adapter and use it in our application:

```
class myAdapter implements Zend_InfoCard_Adapter_Interface
{
    public function storeAssertion($assertionURI,
                                   $assertionID,
                                   $conditions)
    {
        /* Store the assertion and its conditions by ID and URI */
    }

    public function retrieveAssertion($assertionURI, $assertionID)
    {
        /* Retrieve the assertion by URI and ID */
    }

    public function removeAssertion($assertionURI, $assertionID)
    {
        /* Delete a given assertion by URI/ID */
    }
}

$adapter  = new myAdapter();
```

```php
$infoCard = new Zend_InfoCard();
$infoCard->addCertificatePair(SSL_PRIVATE, SSL_PUB);
$infoCard->setAdapter($adapter);

$claims = $infoCard->process($_POST['xmlToken']);
```

Zend_Json

1. Introduction

Zend_Json provides convenience methods for serializing native PHP to JSON and decoding JSON to native PHP. For more information on JSON, visit the JSON project site.

JSON, JavaScript Object Notation, can be used for data interchange between JavaScript and other languages. Since JSON can be directly evaluated by JavaScript, it is a more efficient and lightweight format than XML for exchanging data with JavaScript clients.

In addition, Zend_Json provides a useful way to convert any arbitrary XML formatted string into a JSON formatted string. This built-in feature will enable PHP developers to transform the enterprise data encoded in XML format into JSON format before sending it to browser-based Ajax client applications. It provides an easy way to do dynamic data conversion on the server-side code thereby avoiding unnecessary XML parsing in the browser-side applications. It offers a nice utility function that results in easier application-specific data processing techniques.

2. Basic Usage

Usage of Zend_Json involves using the two public static methods available: Zend_Json::encode() and Zend_Json::decode().

```
// Retrieve a value:
$phpNative = Zend_Json::decode($encodedValue);

// Encode it to return to the client:
$json = Zend_Json::encode($phpNative);
```

2.1. Pretty-printing JSON

Sometimes, it may be hard to explore JSON data generated by Zend_Json::encode(), since it has no spacing or indentation. In order to make it easier, Zend_Json allows you to pretty-print JSON data in the human-readable format with Zend_Json::prettyPrint().

```
// Encode it to return to the client:
$json = Zend_Json::encode($phpNative);
if($debug) {
    echo Zend_Json::prettyPrint($json, array("indent" => " "));
}
```

Second optional argument of Zend_Json::prettyPrint() is an option array. Option indent allows to set indentation string - by default it's a single tab character.

3. Advanced Usage of Zend_Json

3.1. JSON Objects

When encoding PHP objects as JSON, all public properties of that object will be encoded in a JSON object.

JSON does not allow object references, so care should be taken not to encode objects with recursive references. If you have issues with recursion, `Zend_Json::encode()` and `Zend_Json_Encoder::encode()` allow an optional second parameter to check for recursion; if an object is serialized twice, an exception will be thrown.

Decoding JSON objects poses an additional difficulty, however, since Javascript objects correspond most closely to PHP's associative array. Some suggest that a class identifier should be passed, and an object instance of that class should be created and populated with the key/value pairs of the JSON object; others feel this could pose a substantial security risk.

By default, `Zend_Json` will decode JSON objects as associative arrays. However, if you desire an object returned, you can specify this:

```
// Decode JSON objects as PHP objects
$phpNative = Zend_Json::decode($encodedValue, Zend_Json::TYPE_OBJECT);
```

Any objects thus decoded are returned as `StdClass` objects with properties corresponding to the key/value pairs in the JSON notation.

The recommendation of Zend Framework is that the individual developer should decide how to decode JSON objects. If an object of a specified type should be created, it can be created in the developer code and populated with the values decoded using `Zend_Json`.

3.2. Encoding PHP objects

If you are encoding PHP objects by default the encoding mechanism can only access public properties of these objects. When a method `toJson()` is implemented on an object to encode, `Zend_Json` calls this method and expects the object to return a JSON representation of its internal state.

3.3. Internal Encoder/Decoder

`Zend_Json` has two different modes depending if ext/json is enabled in your PHP installation or not. If ext/json is installed by default `json_encode()` and `json_decode()` functions are used for encoding and decoding JSON. If ext/json is not installed a Zend Framework implementation in PHP code is used for en-/decoding. This is considerably slower than using the php extension, but behaves exactly the same.

Still sometimes you might want to use the internal encoder/decoder even if you have ext/json installed. You can achieve this by calling:

```
Zend_Json::$useBuiltinEncoderDecoder = true:
```

3.4. JSON Expressions

Javascript makes heavy use of anonymnous function callbacks, which can be saved within JSON object variables. Still they only work if not returned inside double qoutes, which `Zend_Json` naturally does. With the Expression support for `Zend_Json` support you can encode JSON objects with valid javascript callbacks. This works for both `json_encode()` or the internal encoder.

A javascript callback is represented using the `Zend_Json_Expr` object. It implements the value object pattern and is immutable. You can set the javascript expression as the first constructor argument. By default `Zend_Json::encode` does not encode javascript callbacks, you have to pass the option `'enableJsonExprFinder' = true` into the `encode` function. If enabled the

expression support works for all nested expressions in large object structures. A usage example would look like:

```
$data = array(
    'onClick' => new Zend_Json_Expr('function() {'
              . 'alert("I am a valid javascript callback '
              . 'created by Zend_Json"); }'),
    'other' => 'no expression',
);
$jsonObjectWithExpression = Zend_Json::encode(
    $data,
    false,
    array('enableJsonExprFinder' => true)
);
```

4. XML to JSON conversion

`Zend_Json` provides a convenience method for transforming XML formatted data into JSON format. This feature was inspired from an IBM developerWorks article.

`Zend_Json` includes a static function called `Zend_Json::fromXml()`. This function will generate JSON from a given XML input. This function takes any arbitrary XML string as an input parameter. It also takes an optional boolean input parameter to instruct the conversion logic to ignore or not ignore the XML attributes during the conversion process. If this optional input parameter is not given, then the default behavior is to ignore the XML attributes. This function call is made as shown below:

```
// fromXml function simply takes a String containing XML contents
// as input.
$jsonContents = Zend_Json::fromXml($xmlStringContents, true);
```

`Zend_Json::fromXml()` function does the conversion of the XML formatted string input parameter and returns the equivalent JSON formatted string output. In case of any XML input format error or conversion logic error, this function will throw an exception. The conversion logic also uses recursive techniques to traverse the XML tree. It supports recursion upto 25 levels deep. Beyond that depth, it will throw a `Zend_Json_Exception`. There are several XML files with varying degree of complexity provided in the tests directory of Zend Framework. They can be used to test the functionality of the xml2json feature.

The following is a simple example that shows both the XML input string passed to and the JSON output string returned as a result from the `Zend_Json::fromXml()` function. This example used the optional function parameter as not to ignore the XML attributes during the conversion. Hence, you can notice that the resulting JSON string includes a representation of the XML attributes present in the XML input string.

XML input string passed to `Zend_Json::fromXml()` function:

```
<?xml version="1.0" encoding="UTF-8"?>
<books>
    <book id="1">
        <title>Code Generation in Action</title>
        <author><first>Jack</first><last>Herrington</last></author>
        <publisher>Manning</publisher>
    </book>

    <book id="2">
```

```
        <title>PHP Hacks</title>
        <author><first>Jack</first><last>Herrington</last></author>
        <publisher>O'Reilly</publisher>
    </book>

    <book id="3">
        <title>Podcasting Hacks</title>
        <author><first>Jack</first><last>Herrington</last></author>
        <publisher>O'Reilly</publisher>
    </book>
</books>
```

JSON output string returned from `Zend_Json::fromXml()` function:

```
{
   "books" : {
      "book" : [ {
         "@attributes" : {
            "id" : "1"
         },
         "title" : "Code Generation in Action",
         "author" : {
            "first" : "Jack", "last" : "Herrington"
         },
         "publisher" : "Manning"
      }, {
         "@attributes" : {
            "id" : "2"
         },
         "title" : "PHP Hacks", "author" : {
            "first" : "Jack", "last" : "Herrington"
         },
         "publisher" : "O'Reilly"
      }, {
         "@attributes" : {
            "id" : "3"
         },
         "title" : "Podcasting Hacks", "author" : {
            "first" : "Jack", "last" : "Herrington"
         },
         "publisher" : "O'Reilly"
      }
   ]}
}
```

More details about this xml2json feature can be found in the original proposal itself. Take a look at the Zend_xml2json proposal.

5. Zend_Json_Server - JSON-RPC server

`Zend_Json_Server` is a JSON-RPC server implementation. It supports both the JSON-RPC version 1 specification as well as the version 2 specification; additionally, it provides a PHP implementation of the Service Mapping Description (SMD) specification for providing service metadata to service consumers.

JSON-RPC is a lightweight Remote Procedure Call protocol that utilizes JSON for its messaging envelopes. This JSON-RPC implementation follows PHP's SoapServer API. This means, in a typical situation, you will simply:

- Instantiate the server object
- Attach one or more functions and/or classes/objects to the server object
- handle() the request

`Zend_Json_Server` utilizes Section 2, “Zend_Server_Reflection” to perform reflection on any attached classes or functions, and uses that information to build both the SMD and enforce method call signatures. As such, it is imperative that any attached functions and/or class methods have full PHP docblocks documenting, minimally:

- All parameters and their expected variable types
- The return value variable type

`Zend_Json_Server` listens for POST requests only at this time; fortunately, most JSON-RPC client implementations in the wild at the time of this writing will only POST requests as it is. This makes it simple to utilize the same server end point to both handle requests as well as to deliver the service SMD, as is shown in the next example.

```
 * Calculator - sample class to expose via JSON-RPC
 */
class Calculator
{
    /**
     * Return sum of two variables
     *
     * @param  int $x
     * @param  int $y
     * @return int
     */
    public function add($x, $y)
    {
        return $x + $y;
    }

    /**
     * Return difference of two variables
     *
     * @param  int $x
     * @param  int $y
     * @return int
     */
    public function subtract($x, $y)
    {
        return $x - $y;
    }

    /**
     * Return product of two variables
     *
     * @param  int $x
     * @param  int $y
     * @return int
     */
    public function multiply($x, $y)
    {
        return $x * $y;
    }
```

```
$server = new Zend_Json_Server();
$server->setClass('Calculator');

if ('GET' == $_SERVER['REQUEST_METHOD']) {
    // Indicate the URL endpoint, and the JSON-RPC version used:
    $server->setTarget('/json-rpc.php')
           ->setEnvelope(Zend_Json_Server_Smd::ENV_JSONRPC_2);

    // Grab the SMD
    $smd = $server->getServiceMap();
```

```
$server = new Zend_Json_Server();
$server->setClass('Calculator');

if ('GET' == $_SERVER['REQUEST_METHOD']) {
    $server->setTarget('/json-rpc.php')
           ->setEnvelope(Zend_Json_Server_Smd::ENV_JSONRPC_2);
    $smd = $server->getServiceMap();

    // Set Dojo compatibility:
    $smd->setDojoCompatible(true);

    header('Content-Type: application/json');
    echo $smd;
    return;
}

$server->handle();
```

5.1. Advanced Details

While most functionality for `Zend_Json_Server` is spelled out in Example 446, "Zend_Json_Server Usage", more advanced functionality is available.

5.1.1. Zend_Json_Server

`Zend_Json_Server` is the core class in the JSON-RPC offering; it handles all requests and returns the response payload. It has the following methods:

- `addFunction($function)`: Specify a userland function to attach to the server.
- `setClass($class)`: Specify a class or object to attach to the server; all public methods of that item will be exposed as JSON-RPC methods.
- `fault($fault = null, $code = 404, $data = null)`: Create and return a `Zend_Json_Server_Error` object.
- `handle($request = false)`: Handle a JSON-RPC request; optionally, pass a `Zend_Json_Server_Request` object to utilize (creates one by default).
- `getFunctions()`: Return a list of all attached methods.
- `setRequest(Zend_Json_Server_Request $request)`: Specify a request object for the server to utilize.
- `getRequest()`: Retrieve the request object used by the server.
- `setResponse(Zend_Json_Server_Response $response)`: Set the response object for the server to utilize.
- `getResponse()`: Retrieve the response object used by the server.
- `setAutoEmitResponse($flag)`: Indicate whether the server should automatically emit the response and all headers; by default, this is `TRUE`.
- `autoEmitResponse()`: Determine if auto-emission of the response is enabled.
- `getServiceMap()`: Retrieve the service map description in the form of a `Zend_Json_Server_Smd` object

5.1.2. Zend_Json_Server_Request

The JSON-RPC request environment is encapsulated in the `Zend_Json_Server_Request` object. This object allows you to set necessary portions of the JSON-RPC request, including the request ID, parameters, and JSON-RPC specification version. It has the ability to load itself via JSON or a set of options, and can render itself as JSON via the `toJson()` method.

The request object has the following methods available:

- `setOptions(array $options)`: Specify object configuration. `$options` may contain keys matching any 'set' method: `setParams()`, `setMethod()`, `setId()`, and `setVersion()`.
- `addParam($value, $key = null)`: Add a parameter to use with the method call. Parameters can be just the values, or can optionally include the parameter name.
- `addParams(array $params)`: Add multiple parameters at once; proxies to `addParam()`

- `setParams(array $params)`: Set all parameters at once; overwrites any existing parameters.
- `getParam($index)`: Retrieve a parameter by position or name.
- `getParams()`: Retrieve all parameters at once.
- `setMethod($name)`: Set the method to call.
- `getMethod()`: Retrieve the method that will be called.
- `isMethodError()`: Determine whether or not the request is malformed and would result in an error.
- `setId($name)`: Set the request identifier (used by the client to match requests to responses).
- `getId()`: Retrieve the request identifier.
- `setVersion($version)`: Set the JSON-RPC specification version the request conforms to. May be either '1.0' or '2.0'.
- `getVersion()`: Retrieve the JSON-RPC specification version used by the request.
- `loadJson($json)`: Load the request object from a JSON string.
- `toJson()`: Render the request as a JSON string.

An HTTP specific version is available via `Zend_Json_Server_Request_Http`. This class will retrieve the request via `php://input`, and allows access to the raw JSON via the `getRawJson()` method.

5.1.3. Zend_Json_Server_Response

The JSON-RPC response payload is encapsulated in the `Zend_Json_Server_Response` object. This object allows you to set the return value of the request, whether or not the response is an error, the request identifier, the JSON-RPC specification version the response conforms to, and optionally the service map.

The response object has the following methods available:

- `setResult($value)`: Set the response result.
- `getResult()`: Retrieve the response result.
- `setError(Zend_Json_Server_Error $error)`: Set an error object. If set, this will be used as the response when serializing to JSON.
- `getError()`: Retrieve the error object, if any.
- `isError()`: Whether or not the response is an error response.
- `setId($name)`: Set the request identifier (so the client may match the response with the original request).
- `getId()`: Retrieve the request identifier.
- `setVersion($version)`: Set the JSON-RPC version the response conforms to.
- `getVersion()`: Retrieve the JSON-RPC version the response conforms to.

- `toJson()`: Serialize the response to JSON. If the response is an error response, serializes the error object.
- `setServiceMap($serviceMap)`: Set the service map object for the response.
- `getServiceMap()`: Retrieve the service map object, if any.

An HTTP specific version is available via `Zend_Json_Server_Response_Http`. This class will send the appropriate HTTP headers as well as serialize the response as JSON.

5.1.4. Zend_Json_Server_Error

JSON-RPC has a special format for reporting error conditions. All errors need to provide, minimally, an error message and error code; optionally, they can provide additional data, such as a backtrace.

Error codes are derived from those recommended by the XML-RPC EPI project. `Zend_Json_Server` appropriately assigns the code based on the error condition. For application exceptions, the code '-32000' is used.

`Zend_Json_Server_Error` exposes the following methods:

- `setCode($code)`: Set the error code; if the code is not in the accepted XML-RPC error code range, -32000 will be assigned.
- `getCode()`: Retrieve the current error code.
- `setMessage($message)`: Set the error message.
- `getMessage()`: Retrieve the current error message.
- `setData($data)`: Set auxiliary data further qualifying the error, such as a backtrace.
- `getData()`: Retrieve any current auxiliary error data.
- `toArray()`: Cast the error to an array. The array will contain the keys 'code', 'message', and 'data'.
- `toJson()`: Cast the error to a JSON-RPC error representation.

5.1.5. Zend_Json_Server_Smd

SMD stands for Service Mapping Description, a JSON schema that defines how a client can interact with a particular web service. At the time of this writing, the specification has not yet been formally ratified, but it is in use already within Dojo toolkit as well as other JSON-RPC consumer clients.

At its most basic, a Service Mapping Description indicates the method of transport (POST, GET, TCP/IP, etc), the request envelope type (usually based on the protocol of the server), the target URL of the service provider, and a map of services available. In the case of JSON-RPC, the service map is a list of available methods, which each method documenting the available parameters and their types, as well as the expected return value type.

`Zend_Json_Server_Smd` provides an object oriented way to build service maps. At its most basic, you pass it metadata describing the service using mutators, and specify services (methods and functions).

The service descriptions themselves are typically instances of `Zend_Json_Server_Smd_Service`; you can also pass all information as an array to the various service mutators in `Zend_Json_Server_Smd`, and it will instantiate a service object for you. The service objects contain information such as the name of the service (typically the function or method name), the parameters (names, types, and position), and the return value type. Optionally, each service can have its own target and envelope, though this functionality is rarely used.

`Zend_Json_Server` actually does all of this behind the scenes for you, by using reflection on the attached classes and functions; you should create your own service maps only if you need to provide custom functionality that class and function introspection cannot offer.

Methods available in `Zend_Json_Server_Smd` include:

- `setOptions(array $options)`: Setup an SMD object from an array of options. All mutators (methods beginning with 'set') can be used as keys.
- `setTransport($transport)`: Set the transport used to access the service; only POST is currently supported.
- `getTransport()`: Get the current service transport.
- `setEnvelope($envelopeType)`: Set the request envelope that should be used to access the service. Currently, supports the constants `Zend_Json_Server_Smd::ENV_JSONRPC_1` and `Zend_Json_Server_Smd::ENV_JSONRPC_2`.
- `getEnvelope()`: Get the current request envelope.
- `setContentType($type)`: Set the content type requests should use (by default, this is 'application/json').
- `getContentType()`: Get the current content type for requests to the service.
- `setTarget($target)`: Set the URL endpoint for the service.
- `getTarget()`: Get the URL endpoint for the service.
- `setId($id)`: Typically, this is the URL endpoint of the service (same as the target).
- `getId()`: Retrieve the service ID (typically the URL endpoint of the service).
- `setDescription($description)`: Set a service description (typically narrative information describing the purpose of the service).
- `getDescription()`: Get the service description.
- `setDojoCompatible($flag)`: Set a flag indicating whether or not the SMD is compatible with Dojo toolkit. When TRUE, the generated JSON SMD will be formatted to comply with the format that Dojo's JSON-RPC client expects.
- `isDojoCompatible()`: Returns the value of the Dojo compatibility flag (FALSE, by default).
- `addService($service)`: Add a service to the map. May be an array of information to pass to the constructor of `Zend_Json_Server_Smd_Service`, or an instance of that class.
- `addServices(array $services)`: Add multiple services at once.

- `setServices(array $services)`: Add multiple services at once, overwriting any previously set services.
- `getService($name)`: Get a service by its name.
- `getServices()`: Get all attached services.
- `removeService($name)`: Remove a service from the map.
- `toArray()`: Cast the service map to an array.
- `toDojoArray()`: Cast the service map to an array compatible with Dojo Toolkit.
- `toJson()`: Cast the service map to a JSON representation.

`Zend_Json_Server_Smd_Service` has the following methods:

- `setOptions(array $options)`: Set object state from an array. Any mutator (methods beginning with 'set') may be used as a key and set via this method.
- `setName($name)`: Set the service name (typically, the function or method name).
- `getName()`: Retrieve the service name.
- `setTransport($transport)`: Set the service transport (currently, only transports supported by `Zend_Json_Server_Smd` are allowed).
- `getTransport()`: Retrieve the current transport.
- `setTarget($target)`: Set the URL endpoint of the service (typically, this will be the same as the overall SMD to which the service is attached).
- `getTarget()`: Get the URL endpoint of the service.
- `setEnvelope($envelopeType)`: Set the service envelope (currently, only envelopes supported by `Zend_Json_Server_Smd` are allowed).
- `getEnvelope()`: Retrieve the service envelope type.
- `addParam($type, array $options = array(), $order = null)`: Add a parameter to the service. By default, only the parameter type is necessary. However, you may also specify the order, as well as options such as:
  - *name*: the parameter name
  - *optional*: whether or not the parameter is optional
  - *default*: a default value for the parameter
  - *description*: text describing the parameter
- `addParams(array $params)`: Add several parameters at once; each param should be an assoc array containing minimally the key 'type' describing the parameter type, and optionally the key 'order'; any other keys will be passed as `$options` to `addOption()`.
- `setParams(array $params)`: Set many parameters at once, overwriting any existing parameters.

- `getParams()`: Retrieve all currently set parameters.
- `setReturn($type)`: Set the return value type of the service.
- `getReturn()`: Get the return value type of the service.
- `toArray()`: Cast the service to an array.
- `toJson()`: Cast the service to a JSON representation.

Zend_Layout

1. Introduction

`Zend_Layout` implements a classic Two Step View pattern, allowing developers to wrap application content within another view, usually representing the site template. Such templates are often termed *layouts* by other projects, and Zend Framework has adopted this term for consistency.

The main goals of `Zend_Layout` are as follows:

- Automate selection and rendering of layouts when used with the Zend Framework MVC components.
- Provide separate scope for layout related variables and content.
- Allow configuration, including layout name, layout script resolution (inflection), and layout script path.
- Allow disabling layouts, changing layout scripts, and other states; allow these actions from within action controllers and view scripts.
- Follow same script resolution rules (inflection) as the ViewRenderer, but allow them to also use different rules.
- Allow usage without Zend Framework MVC components.

2. Zend_Layout Quick Start

There are two primary use cases for `Zend_Layout`: with the Zend Framework MVC, and without.

2.1. Layout scripts

In both cases, however, you'll need to create a layout script. Layout scripts simply utilize `Zend_View` (or whatever view implementation you are using). Layout variables are registered with a `Zend_Layout` placeholder, and may be accessed via the placeholder helper or by fetching them as object properties of the layout object via the layout helper.

As an example:

```
<!DOCTYPE html
    PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html>
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
    <title>My Site</title>
</head>
<body>
<?php
    // fetch 'content' key using layout helper:
    echo $this->layout()->content;
```

```
    // fetch 'foo' key using placeholder helper:
    echo $this->placeholder('Zend_Layout')->foo;

    // fetch layout object and retrieve various keys from it:
    $layout = $this->layout();
    echo $layout->bar;
    echo $layout->baz;
?>
</body>
</html>
```

Because `Zend_Layout` utilizes `Zend_View` for rendering, you can also use any view helpers registered, and also have access to any previously assigned view variables. Particularly useful are the various placeholder helpers, as they allow you to retrieve content for areas such as the <head> section, navigation, etc.:

```
<!DOCTYPE html
    PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html>
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
    <?php echo $this->headTitle() ?>
    <?php echo $this->headScript() ?>
    <?php echo $this->headStyle() ?>
</head>
<body>
    <?php echo $this->render('header.phtml') ?>
    <div id="nav"><?php echo $this->placeholder('nav') ?></div>

    <div id="content"><?php echo $this->layout()->content ?></div>

    <?php echo $this->render('footer.phtml') ?>
</body>
</html>
```

2.2. Using Zend_Layout with the Zend Framework MVC

`Zend_Controller` offers a rich set of functionality for extension via its front controller plugins and action controller helpers. `Zend_View` also has helpers. `Zend_Layout` takes advantage of these various extension points when used with the MVC components.

`Zend_Layout::startMvc()` creates an instance of `Zend_Layout` with any optional configuration you provide it. It then registers a front controller plugin that renders the layout with any application content once the dispatch loop is done, and registers an action helper to allow access to the layout object from your action controllers. Additionally, you may at any time grab the layout instance from within a view script using the `layout` view helper.

First, let's look at how to initialize `Zend_Layout` for use with the MVC:

```
// In your bootstrap:
Zend_Layout::startMvc();
```

`startMvc()` can take an optional array of options or `Zend_Config` object to customize the instance; these options are detailed in Section 3, “Zend_Layout Configuration Options”.

In an action controller, you may then access the layout instance as an action helper:

```
class FooController extends Zend_Controller_Action
{
    public function barAction()
    {
        // disable layouts for this action:
        $this->_helper->layout->disableLayout();
    }

    public function bazAction()
    {
        // use different layout script with this action:
        $this->_helper->layout->setLayout('foobaz');
    };
}
```

In your view scripts, you can then access the layout object via the `layout` view helper. This view helper is slightly different than others in that it takes no arguments, and returns an object instead of a string value. This allows you to immediately call methods on the layout object:

```
<?php $this->layout()->setLayout('foo'); // set alternate layout ?>
```

At any time, you can fetch the `Zend_Layout` instance registered with the MVC via the `getMvcInstance()` static method:

```
// Returns null if startMvc() has not first been called
$layout = Zend_Layout::getMvcInstance();
```

Finally, `Zend_Layout`'s front controller plugin has one important feature in addition to rendering the layout: it retrieves all named segments from the response object and assigns them as layout variables, assigning the 'default' segment to the variable 'content'. This allows you to access your application content and render it in your view scripts.

As an example, let's say your code first hits `FooController::indexAction()`, which renders some content to the default response segment, and then forwards to `NavController::menuAction()`, which renders content to the 'nav' response segment. Finally, you forward to `CommentController::fetchAction()` and fetch some comments, but render those to the default response segment as well (which appends content to that segment). Your view script could then render each separately:

```
<body>
    <!-- renders /nav/menu -->
    <div id="nav"><?php echo $this->layout()->nav ?></div>

    <!-- renders /foo/index + /comment/fetch -->
    <div id="content"><?php echo $this->layout()->content ?></div>
</body>
```

This feature is particularly useful when used in conjunction with the ActionStack action helper and plugin, which you can use to setup a stack of actions through which to loop, and thus create widgetized pages.

2.3. Using Zend_Layout as a Standalone Component

As a standalone component, `Zend_Layout` does not offer nearly as many features or as much convenience as when used with the MVC. However, it still has two chief benefits:

- Scoping of layout variables.
- Isolation of layout view script from other view scripts.

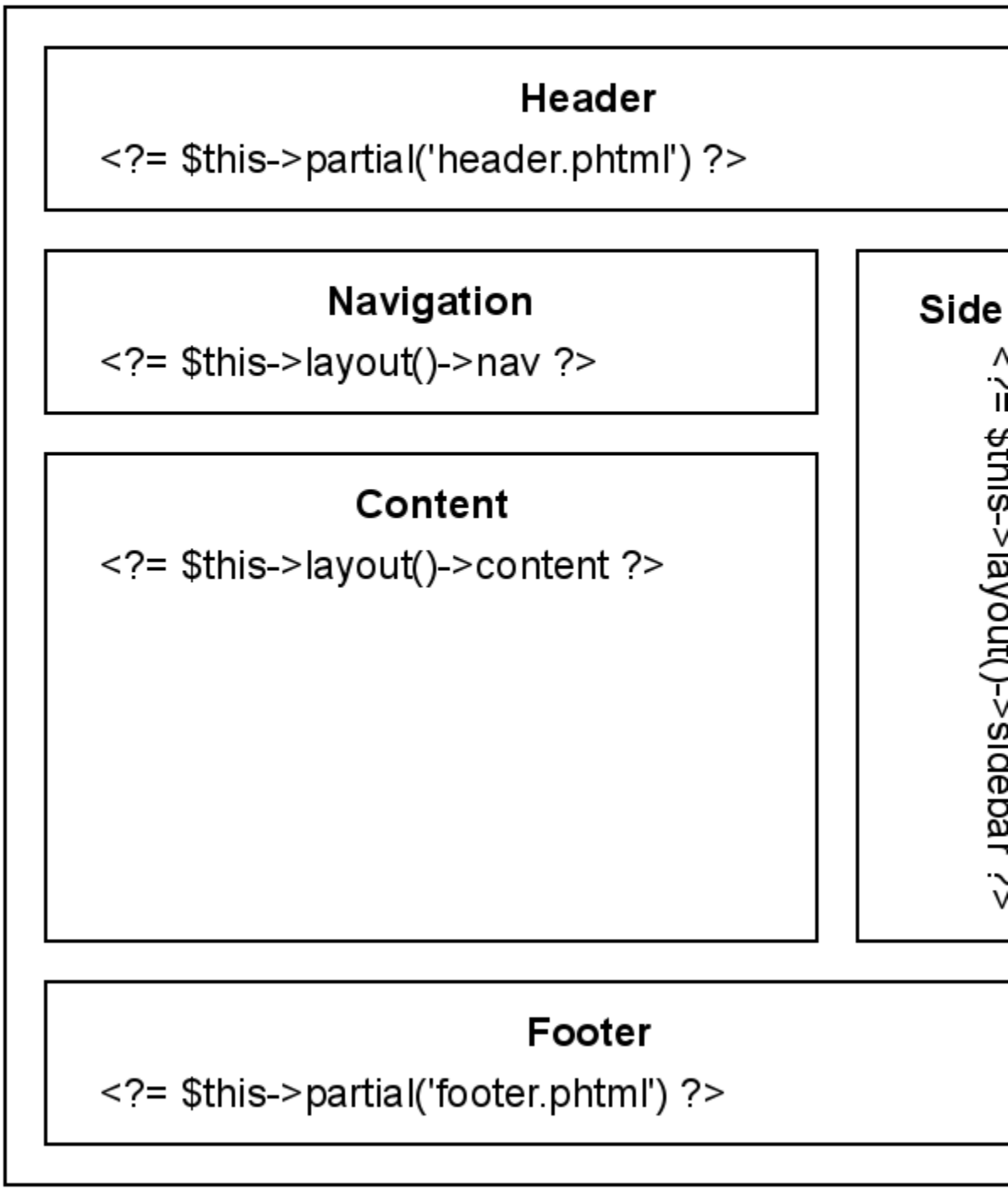
When used as a standalone component, simply instantiate the layout object, use the various accessors to set state, set variables as object properties, and render the layout:

```
$layout = new Zend_Layout();

// Set a layout script path:
$layout->setLayoutPath('/path/to/layouts');

// set some variables:
$layout->content = $content;
$layout->nav     = $nav;

// choose a different layout script:
$layout->setLayout('foo');

// render final layout
echo $layout->render();
```

2.4. Sample Layout

Sometimes a picture is worth a thousand words. The following is a sample layout script showing how it might all come together.

```
<?= $this->docType('XHTML1_STRICT') ?>
<html>
    <head>
        <?= $this->headTitle() ?>
        <?= $this->headScript() ?>
        <?= $this->headStylesheet() ?>
    </head>
    <body>
```



The actual order of elements may vary, depending on the CSS you've setup; for instance, if you're using absolute positioning, you may be able to have the navigation displayed later in the document, but still show up at the top; the same could be said for the sidebar or header. The actual mechanics of pulling the content remain the same, however.

3. Zend_Layout Configuration Options

`Zend_Layout` has a variety of configuration options. These may be set by calling the appropriate accessors, passing an array or `Zend_Config` object to the constructor or `startMvc()`, passing an array of options to `setOptions()`, or passing a `Zend_Config` object to `setConfig()`.

- *layout*: the layout to use. Uses the current inflector to resolve the name provided to the appropriate layout view script. By default, this value is 'layout' and resolves to 'layout.phtml'. Accessors are `setLayout()` and `getLayout()`.
- *layoutPath*: the base path to layout view scripts. Accessors are `setLayoutPath()` and `getLayoutPath()`.
- *contentKey*: the layout variable used for default content (when used with the MVC). Default value is 'content'. Accessors are `setContentKey()` and `getContentKey()`.
- *mvcSuccessfulActionOnly*: when using the MVC, if an action throws an exception and this flag is `TRUE`, the layout will not be rendered (this is to prevent double-rendering of the layout when the ErrorHandler plugin is in use). By default, the flat is `TRUE`. Accessors are `setMvcSuccessfulActionOnly()` and `getMvcSuccessfulActionOnly()`.
- *view*: the view object to use when rendering. When used with the MVC, `Zend_Layout` will attempt to use the view object registered with the ViewRenderer if no view object has been passed to it explicitly. Accessors are `setView()` and `getView()`.
- *helperClass*: the action helper class to use when using `Zend_Layout` with the MVC components. By default, this is `Zend_Layout_Controller_Action_Helper_Layout`. Accessors are `setHelperClass()` and `getHelperClass()`.
- *pluginClass*: the front controller plugin class to use when using `Zend_Layout` with the MVC components. By default, this is `Zend_Layout_Controller_Plugin_Layout`. Accessors are `setPluginClass()` and `getPluginClass()`.
- *inflector*: the inflector to use when resolving layout names to layout view script paths; see the `Zend_Layout` inflector documentation for more details. Accessors are `setInflector()` and `getInflector()`.

helperClass and pluginClass must be passed to startMvc()

In order for the `helperClass` and `pluginClass` settings to have effect, they must be passed in as options to `startMvc()`; if set later, they have no affect.

3.1. Examples

The following examples assume the following `$options` array and `$config` object:

```
$options = array(
    'layout'     => 'foo',
    'layoutPath' => '/path/to/layouts',
    'contentKey' => 'CONTENT',           // ignored when MVC not used
);
```

```
/**
[layout]
layout = "foo"
layoutPath = "/path/to/layouts"
contentKey = "CONTENT"
*/
$config = new Zend_Config_Ini('/path/to/layout.ini', 'layout');
```

Example 447. Passing options to the constructor or startMvc()

Both the constructor and the `startMvc()` static method can accept either an array of options or a `Zend_Config` object with options in order to configure the `Zend_Layout` instance.

First, let's look at passing an array:

```
// Using constructor:
$layout = new Zend_Layout($options);

// Using startMvc():
$layout = Zend_Layout::startMvc($options);
```

And now using a config object:

```
$config = new Zend_Config_Ini('/path/to/layout.ini', 'layout');

// Using constructor:
$layout = new Zend_Layout($config);

// Using startMvc():
$layout = Zend_Layout::startMvc($config);
```

Basically, this is the easiest way to customize your `Zend_Layout` instance.

Example 448. Using setOption() and setConfig()

Sometimes you need to configure the `Zend_Layout` object after it has already been instantiated; `setOptions()` and `setConfig()` give you a quick and easy way to do so:

```
// Using an array of options:
$layout->setOptions($options);

// Using a Zend_Config object:
$layout->setConfig($options);
```

Note, however, that certain options, such as `pluginClass` and `helperClass`, will have no affect when passed using this method; they need to be passed to the constructor or `startMvc()` method.

Example 449. Using Accessors

Finally, you can also configure your `Zend_Layout` instance via accessors. All accessors implement a fluent interface, meaning their calls may be chained:

```
$layout->setLayout('foo')
       ->setLayoutPath('/path/to/layouts')
       ->setContentKey('CONTENT');
```

4. Zend_Layout Advanced Usage

`Zend_Layout` has a number of use cases for the advanced developer who wishes to adapt it for different view implementations, file system layouts, and more.

The major points of extension are:

- *Custom view objects.* `Zend_Layout` allows you to utilize any class that implements `Zend_View_Interface`.
- *Custom front controller plugins.* `Zend_Layout` ships with a standard front controller plugin that automates rendering of layouts prior to returning the response. You can substitute your own plugin.
- *Custom action helpers.* `Zend_Layout` ships with a standard action helper that should be suitable for most needs as it is a dumb proxy to the layout object itself.
- *Custom layout script path resolution*. `Zend_Layout` allows you to use your own inflector for layout script path resolution, or simply to modify the attached inflector to specify your own inflection rules.

4.1. Custom View Objects

`Zend_Layout` allows you to use any class implementing `Zend_View_Interface` or extending `Zend_View_Abstract` for rendering your layout script. Simply pass in your custom view object as a parameter to the constructor/`startMvc()`, or set it using the `setView()` accessor:

```
$view = new My_Custom_View();
$layout->setView($view);
```

Not all Zend_View implementations are equal

While `Zend_Layout` allows you to use any class implementing `Zend_View_Interface`, you may run into issues if they can not utilize the various `Zend_View` helpers, particularly the layout and placeholder helpers. This is because `Zend_Layout` makes variables set in the object available via itself and placeholders.

If you need to use a custom `Zend_View` implementation that does not support these helpers, you will need to find a way to get the layout variables to the view. This can be done by either extending the `Zend_Layout` object and altering the `render()` method to pass variables to the view, or creating your own plugin class that passes them prior to rendering the layout.

Alternately, if your view implementation supports any sort of plugin capability, you can access the variables via the 'Zend_Layout' placeholder, using the placeholder helper:

```
$placeholders = new Zend_View_Helper_Placeholder();
$layoutVars   = $placeholders->placeholder('Zend_Layout')->getArrayCopy();
```

4.2. Custom Front Controller Plugins

When used with the MVC components, `Zend_Layout` registers a front controller plugin that renders the layout as the last action prior to exiting the dispatch loop. In most cases, the default

plugin will be suitable, but should you desire to write your own, you can specify the name of the plugin class to load by passing the `pluginClass` option to the `startMvc()` method.

Any plugin class you write for this purpose will need to extend `Zend_Controller_Plugin_Abstract`, and should accept a layout object instance as an argument to the constructor. Otherwise, the details of your implementation are up to you.

The default plugin class used is `Zend_Layout_Controller_Plugin_Layout`.

4.3. Custom Action Helpers

When used with the MVC components, `Zend_Layout` registers an action controller helper with the helper broker. The default helper, `Zend_Layout_Controller_Action_Helper_Layout`, acts as a dumb proxy to the layout object instance itself, and should be suitable for most use cases.

Should you feel the need to write custom functionality, simply write an action helper class extending `Zend_Controller_Action_Helper_Abstract` and pass the class name as the `helperClass` option to the `startMvc()` method. Details of the implementation are up to you.

4.4. Custom Layout Script Path Resolution: Using the Inflector

`Zend_Layout` uses `Zend_Filter_Inflector` to establish a filter chain for translating a layout name to a layout script path. By default, it uses the rules 'Word_CamelCaseToDash' followed by 'StringToLower', and the suffix 'phtml' to transform the name to a path. As some examples:

- 'foo' will be transformed to 'foo.phtml'.
- 'FooBarBaz' will be transformed to 'foo-bar-baz.phtml'.

You have three options for modifying inflection: modify the inflection target and/or view suffix via `Zend_Layout` accessors, modify the inflector rules and target of the inflector associated with the `Zend_Layout` instance, or create your own inflector instance and pass it to `Zend_Layout::setInflector()`.

Example 450. Using Zend_Layout accessors to modify the inflector

The default `Zend_Layout` inflector uses static references for the target and view script suffix, and has accessors for setting these values.

```
// Set the inflector target:
$layout->setInflectorTarget('layouts/:script.:suffix');

// Set the layout view script suffix:
$layout->setViewSuffix('php');
```

Example 451. Direct modification of Zend_Layout inflector

Inflectors have a target and one or more rules. The default target used with `Zend_Layout` is ':script.:suffix'; ':script' is passed the registered layout name, while ':suffix' is a static rule of the inflector.

Let's say you want the layout script to end in the suffix 'html', and that you want to separate MixedCase and camelCased words with underscores instead of dashes, and not lowercase the name. Additionally, you want it to look in a 'layouts' subdirectory for the script.

```
$layout->getInflector()->setTarget('layouts/:script.:suffix')
                       ->setStaticRule('suffix', 'html')
                       ->setFilterRule(array('Word_CamelCaseToUnderscore'));
```

Example 452. Custom inflectors

In most cases, modifying the existing inflector will be enough. However, you may have an inflector you wish to use in several places, with different objects of different types. `Zend_Layout` supports this.

```
$inflector = new Zend_Filter_Inflector('layouts/:script.:suffix');
$inflector->addRules(array(
    ':script' => array('Word_CamelCaseToUnderscore'),
    'suffix'  => 'html'
));
$layout->setInflector($inflector);
```

Inflection can be disabled

Inflection can be disabled and enabled using accessors on the `Zend_Layout` object. This can be useful if you want to specify an absolute path for a layout view script, or know that the mechanism you will be using for specifying the layout script does not need inflection. Simply use the `enableInflection()` and `disableInflection()` methods.

Zend_Ldap

1. Introduction

Zend_Ldap is a class for performing LDAP operations including but not limited to binding, searching and modifying entries in an LDAP directory.

1.1. Theory of operation

This component currently consists of the main Zend_Ldap class, that conceptually represents a binding to a single LDAP server and allows for executing operations against a LDAP server such as OpenLDAP or ActiveDirectory (AD) servers. The parameters for binding may be provided explicitly or in the form of an options array. Zend_Ldap_Node provides an object-oriented interface for single LDAP nodes and can be used to form a basis for an active-record-like interface for a LDAP-based domain model.

The component provides several helper classes to perform operations on LDAP entries (Zend_Ldap_Attribute) such as setting and retrieving attributes (date values, passwords, boolean values, ...), to create and modify LDAP filter strings (Zend_Ldap_Filter) and to manipulate LDAP distinguished names (DN) (Zend_Ldap_Dn).

Additionally the component abstracts LDAP schema browsing for OpenLDAP and ActiveDirectoy servers Zend_Ldap_Node_Schema and server information retrieval for OpenLDAP-, ActiveDirectory- and Novell eDirectory servers (Zend_Ldap_Node_RootDse).

Using the Zend_Ldap class depends on the type of LDAP server and is best summarized with some simple examples.

If you are using OpenLDAP, a simple example looks like the following (note that the *bindRequiresDn* option is important if you are *not* using AD):

```
$options = array(
    'host'              => 's0.foo.net',
    'username'          => 'CN=user1,DC=foo,DC=net',
    'password'          => 'pass1',
    'bindRequiresDn'    => true,
    'accountDomainName' => 'foo.net',
    'baseDn'            => 'OU=Sales,DC=foo,DC=net',
);
$ldap = new Zend_Ldap($options);
$acctname = $ldap->getCanonicalAccountName('abaker',
                                           Zend_Ldap::ACCTNAME_FORM_DN);
echo "$acctname\n";
```

If you are using Microsoft AD a simple example is:

```
$options = array(
    'host'                  => 'dc1.w.net',
    'useStartTls'           => true,
    'username'              => 'user1@w.net',
    'password'              => 'pass1',
    'accountDomainName'     => 'w.net',
    'accountDomainNameShort' => 'W',
    'baseDn'                => 'CN=Users,DC=w,DC=net',
```

```
);
$ldap = new Zend_Ldap($options);
$acctname = $ldap->getCanonicalAccountName('bcarter',
                                           Zend_Ldap::ACCTNAME_FORM_DN);
echo "$acctname\n";
```

Note that we use the `getCanonicalAccountName()` method to retrieve the account DN here only because that is what exercises the most of what little code is currently present in this class.

1.1.1. Automatic Username Canonicalization When Binding

If `bind()` is called with a non-DN username but *bindRequiresDN* is `TRUE` and no username in DN form was supplied as an option, the bind will fail. However, if a username in DN form is supplied in the options array, `Zend_Ldap` will first bind with that username, retrieve the account DN for the username supplied to `bind()` and then re-bind with that DN.

This behavior is critical to `Zend_Auth_Adapter_Ldap`, which passes the username supplied by the user directly to `bind()`.

The following example illustrates how the non-DN username '*abaker*' can be used with `bind()`:

```
$options = array(
        'host'              => 's0.foo.net',
        'username'          => 'CN=user1,DC=foo,DC=net',
        'password'          => 'pass1',
        'bindRequiresDn'    => true,
        'accountDomainName' => 'foo.net',
        'baseDn'            => 'OU=Sales,DC=foo,DC=net',
);
$ldap = new Zend_Ldap($options);
$ldap->bind('abaker', 'moonbike55');
$acctname = $ldap->getCanonicalAccountName('abaker',
                                           Zend_Ldap::ACCTNAME_FORM_DN);
echo "$acctname\n";
```

The `bind()` call in this example sees that the username '*abaker*' is not in DN form, finds *bindRequiresDn* is `TRUE`, uses '**CN=user1,DC=foo,DC=net**' and '*pass1*' to bind, retrieves the DN for '*abaker*', unbinds and then rebinds with the newly discovered '**CN=Alice Baker,OU=Sales,DC=foo,DC=net**'.

1.1.2. Account Name Canonicalization

The *accountDomainName* and *accountDomainNameShort* options are used for two purposes: (1) they facilitate multi-domain authentication and failover capability, and (2) they are also used to canonicalize usernames. Specifically, names are canonicalized to the form specified by the *accountCanonicalForm* option. This option may one of the following values:

Table 68. Options for accountCanonicalForm

Name	Value	Example
`ACCTNAME_FORM_DN`	1	CN=Alice Baker,CN=Users,DC=example,DC=com
`ACCTNAME_FORM_USERNAME`	2	abaker
`ACCTNAME_FORM_BACKSLASH`	3	EXAMPLE\abaker
`ACCTNAME_FORM_PRINCIPAL`	4	`abaker@example.com`

The default canonicalization depends on what account domain name options were supplied. If *accountDomainNameShort* was supplied, the default *accountCanonicalForm* value is ACCTNAME_FORM_BACKSLASH. Otherwise, if *accountDomainName* was supplied, the default is ACCTNAME_FORM_PRINCIPAL.

Account name canonicalization ensures that the string used to identify an account is consistent regardless of what was supplied to bind(). For example, if the user supplies an account name of abaker@example.com or just *abaker* and the *accountCanonicalForm* is set to 3, the resulting canonicalized name would be *EXAMPLE\abaker*.

1.1.3. Multi-domain Authentication and Failover

The Zend_Ldap component by itself makes no attempt to authenticate with multiple servers. However, Zend_Ldap is specifically designed to handle this scenario gracefully. The required technique is to simply iterate over an array of arrays of serve options and attempt to bind with each server. As described above bind() will automatically canonicalize each name, so it does not matter if the user passes abaker@foo.net or *W\bcarter* or *cdavis* - the bind() method will only succeed if the credentials were successfully used in the bind.

Consider the following example that illustrates the technique required to implement multi-domain authentication and failover:

```
$acctname = 'W\\user2';
$password = 'pass2';

$multiOptions = array(
    'server1' => array(
        'host'                 => 's0.foo.net',
        'username'             => 'CN=user1,DC=foo,DC=net',
        'password'             => 'pass1',
        'bindRequiresDn'       => true,
        'accountDomainName'    => 'foo.net',
        'accountDomainNameShort' => 'FOO',
        'accountCanonicalForm' => 4, // ACCT_FORM_PRINCIPAL
        'baseDn'               => 'OU=Sales,DC=foo,DC=net',
    ),
    'server2' => array(
        'host'                 => 'dc1.w.net',
        'useSsl'               => true,
        'username'             => 'user1@w.net',
        'password'             => 'pass1',
        'accountDomainName'    => 'w.net',
        'accountDomainNameShort' => 'W',
        'accountCanonicalForm' => 4, // ACCT_FORM_PRINCIPAL
        'baseDn'               => 'CN=Users,DC=w,DC=net',
    ),
);

$ldap = new Zend_Ldap();

foreach ($multiOptions as $name => $options) {

    echo "Trying to bind using server options for '$name'\n";

    $ldap->setOptions($options);
    try {
        $ldap->bind($acctname, $password);
        $acctname = $ldap->getCanonicalAccountName($acctname);
```

```
        echo "SUCCESS: authenticated $acctname\n";
        return;
    } catch (Zend_Ldap_Exception $zle) {
        echo '  ' . $zle->getMessage() . "\n";
        if ($zle->getCode() === Zend_Ldap_Exception::LDAP_X_DOMAIN_MISMATCH) {
            continue;
        }
    }
}
```

If the bind fails for any reason, the next set of server options is tried.

The `getCanonicalAccountName()` call gets the canonical account name that the application would presumably use to associate data with such as preferences. The *accountCanonicalForm = 4* in all server options ensures that the canonical form is consistent regardless of which server was ultimately used.

The special `LDAP_X_DOMAIN_MISMATCH` exception occurs when an account name with a domain component was supplied (e.g., `abaker@foo.net` or *FOO\abaker* and not just *abaker*) but the domain component did not match either domain in the currently selected server options. This exception indicates that the server is not an authority for the account. In this case, the bind will not be performed, thereby eliminating unnecessary communication with the server. Note that the *continue* instruction has no effect in this example, but in practice for error handling and debugging purposes, you will probably want to check for `LDAP_X_DOMAIN_MISMATCH` as well as `LDAP_NO_SUCH_OBJECT` and `LDAP_INVALID_CREDENTIALS`.

The above code is very similar to code used within `Zend_Auth_Adapter_Ldap`. In fact, we recommend that you simply use that authentication adapter for multi-domain + failover LDAP based authentication (or copy the code).

2. API overview

2.1. Configuration / options

The `Zend_Ldap` component accepts an array of options either supplied to the constructor or through the `setOptions()` method. The permitted options are as follows:

Table 69. Zend_Ldap Options

Name	Description
host	The default hostname of LDAP server if not supplied to `connect()` (also may be used when trying to canonicalize usernames in `bind()`).
port	Default port of LDAP server if not supplied to `connect()`.
useStartTls	Whether or not the LDAP client should use TLS (aka SSLv2) encrypted transport. A value of `TRUE` is strongly favored in production environments to prevent passwords from be transmitted in clear text. The default value is `FALSE`, as servers frequently require that a certificate be installed separately after installation. The *useSsl* and *useStartTls* options are mutually exclusive. The *useStartTls*

Name	Description
	option should be favored over *useSsl* but not all servers support this newer mechanism.
useSsl	Whether or not the LDAP client should use SSL encrypted transport. The *useSsl* and *useStartTls* options are mutually exclusive.
username	The default credentials username. Some servers require that this be in DN form. This must be given in DN form if the LDAP server requires a DN to bind and binding should be possible with simple usernames.
password	The default credentials password (used only with username above).
bindRequiresDn	If TRUE, this instructs Zend_Ldap to retrieve the DN for the account used to bind if the username is not already in DN form. The default value is FALSE.
baseDn	The default base DN used for searching (e.g., for accounts). This option is required for most account related operations and should indicate the DN under which accounts are located.
accountCanonicalForm	A small integer indicating the form to which account names should be canonicalized. See the *Account Name Canonicalization* section below.
accountDomainName	The FQDN domain for which the target LDAP server is an authority (e.g., example.com).
accountDomainNameShort	The 'short' domain for which the target LDAP server is an authority. This is usually used to specify the NetBIOS domain name for Windows networks but may also be used by non-AD servers.
accountFilterFormat	The LDAP search filter used to search for accounts. This string is a `sprintf()` style expression that must contain one '*%s*' to accommodate the username. The default value is '*(&(objectClass=user)(sAMAccountName=%s))*' unless *bindRequiresDn* is set to TRUE, in which case the default is '*(&(objectClass=posixAccount)(uid=%s))*'. Users of custom schemas may need to change this option.
allowEmptyPassword	Some LDAP servers can be configured to accept an empty string password as an anonymous bind. This behavior is almost always undesirable. For this reason, empty passwords are explicitly disallowed. Set this value to TRUE to allow an empty string password to be submitted during the bind.

Name	Description
optReferrals	If set to TRUE, this option indicates to the LDAP client that referrals should be followed. The default value is FALSE.
tryUsernameSplit	If set to FALSE, this option indicates that the given username should not be split at the first @ or \ character to separate the username from the domain during the binding-procedure. This allows the user to use usernames that contain an @ or \ character that do not inherit some domain-information, e.g. using email-addresses for binding. The default value is TRUE.

2.2. API Reference

Method names in *italics* are static methods.

2.2.1. Zend_Ldap

`Zend_Ldap` is the base interface into a LDAP server. It provides connection and binding methods as well as methods to operate on the LDAP tree.

Table 70. Zend_Ldap API

Method	Description
`string filterEscape(string $str)`	Escapes a value to be used in a LDAP filter according to RFC 2254. This method is *deprecated*, please use `Zend_Ldap_Filter_Abstract::escapeValue()` instead.
`boolean explodeDn($dn, array &$keys = null, array &$vals = null)`	Checks if a given DN `$dn` is malformed. If `$keys` or `$keys` and `$vals` are given, these arrays will be filled with the appropriate DN keys and values. This method is *deprecated*, please use `Zend_Ldap_Dn::checkDn()` instead.
`__construct($options)`	Constructor. The `$options` parameter is optional and can be set to an array or a `Zend_Config` instance. If no options are provided at instantiation, the connection parameters must be passed to the instance using `Zend_Ldap::setOptions()`. The allowed options are specified in Zend_Ldap Options
`resource getResource()`	Returns the raw LDAP extension (ext/ldap) resource.
`integer getLastErrorCode()`	Returns the LDAP error number of the last LDAP command.

Method	Description
`string getLastError(integer &$errorCode, array &$errorMessages)`	Returns the LDAP error message of the last LDAP command. The optional `$errorCode` parameter is set to the LDAP error number when given. The optional `$errorMessages` array will be filled with the raw error messages when given. The various LDAP error retrieval functions can return different things, so they are all collected if `$errorMessages` is given.
`Zend_Ldap setOptions($options)`	Sets the LDAP connection and binding parameters. `$options` can be an array or an instance of `Zend_Config`. The allowed options are specified in Zend_Ldap Options
`array getOptions()`	Returns the current connection and binding parameters.
`string getBaseDn()`	Returns the base DN this LDAP connection is bound to.
`string getCanonicalAccountName(string $acctname, integer $form)`	Returns the canonical account name of the given account name `$acctname`. `$form` specifies the format into which the account name is canonicalized. See Account Name Canonicalization for more details.
`Zend_Ldap disconnect()`	Disconnects the Zend_Ldap instance from the LDAP server.
`Zend_Ldap connect(string $host, integer $port, boolean $useSsl, boolean $useStartTls)`	Connects the Zend_Ldap instance to the given LDAP server. All parameters are optional and will be taken from the LDAP connection and binding parameters passed to the instance via the construtor or via `Zend_Ldap::setOptions()` when set to `NULL`.
`Zend_Ldap bind(string $username, string $password)`	Authenticates `$username` with `$password` at the LDAP server. If both paramaters are omitted the binding will be carried out with the credentials given in the connection and binding parameters. If no credentials are given in the connection and binding parameters an anonymous bind will be performed. Note that this requires anonymous binds to be allowed on the LDAP server. An empty string `''` can be passed as `$password` together with a username if, and only if, `allowEmptyPassword` is set to `TRUE` in the connection and binding parameters.
`Zend_Ldap_Collection search(string|Zend_Ldap_Filter_Abstract $filter, string|Zend_Ldap_Dn $basedn, integer $scope, array`	Searches the LDAP tree with the given `$filter` and the given search parameters. `string|Zend_Ldap_Filter_Abstract $filter` The filter string to be used in

Method	Description	
$attributes, string $sort, string $collectionClass)		the search, e.g. (objectClass=posixAcc
	string|Zend_Ldap_Dn $basedn	The search base for the search. If omitted or NULL, the baseDn from the connection and binding parameters is used.
	integer $scope	The search scope. Zend_Ldap::SEARCH_SCO searches the complete subtree including the $baseDn node. Zend_Ldap::SEARCH_SCO restricts search to one level below $baseDn. Zend_Ldap::SEARCH_SCO restricts search to the $baseDn itself; this can be used to efficiently retrieve a single entry by its DN. The default value is Zend_Ldap::SEARCH_SCO
	array $attributes	Specifies the attributes contained

Method	Description
	in the returned entries. To include all possible attributes (ACL restrictions can disallow certain attribute to be retrieved by a given user) pass either an empty array `array()` or `array('*')` to the method. On some LDAP servers you can retrieve special internal attributes by passing `array('*', '+')` to the method.
	`string $sort` If given the result collection will be sorted after the attribute `$sort`. Results can only be sorted after one single attribute as this parameter uses the

Method	Description
	ext/ldap function `ldap_sort()`. `string $collectionClass` If given the result will be wrapped in an object of type `$collectionClass`. By default an object of type `Zend_Ldap_Collection` will be returned. The custom class must extend `Zend_Ldap_Collection` and will be passed a `Zend_Ldap_Collection_` on instantiation.
`integer count(string|Zend_Ldap_Filter_Abstract $filter, string|Zend_Ldap_Dn $basedn, integer $scope)`	Counts the elements returned by the given search parameters. See `Zend_Ldap::search()` for a detailed description of the method parameters.
`integer countChildren(string|Zend_Ldap_Dn $dn)`	Counts the direct descendants (children) of the entry identified by the given `$dn`.
`boolean exists(string|Zend_Ldap_Dn $dn)`	Checks whether the entry identified by the given `$dn` exists.
`array searchEntries(string|Zend_Ldap_Filter_Abstract $filter, string|Zend_Ldap_Dn $basedn, integer $scope, array $attributes, string $sort)`	Performs a search operation and returns the result as an PHP array. This is essentially the same method as `Zend_Ldap::search()` except for the return type. See `Zend_Ldap::search()` for a detailed description of the method parameters.
`array getEntry(string|Zend_Ldap_Dn $dn, array $attributes, boolean $throwOnNotFound)`	Retrieves the LDAP entry identified by `$dn` with the attributes specified in `$attributes`. If `$attributes` is ommitted, all attributes (`array()`) are included in the result. `$throwOnNotFound` is `FALSE` by default, so the method will return `NULL` if the specified entry cannot be found. If set to `TRUE`, a `Zend_Ldap_Exception` will be thrown instead.
`void prepareLdapEntryArray(array & $entry)`	Prepare an array for the use in LDAP modification operations. This method does not

Method	Description
	need to be called by the end-user as it's implicitly called on every data modification method.
`Zend_Ldap add(string|Zend_Ldap_Dn $dn, array $entry)`	Adds the entry identified by `$dn` with its attributes `$entry` to the LDAP tree. Throws a `Zend_Ldap_Exception` if the entry could not be added.
`Zend_Ldap update(string|Zend_Ldap_Dn $dn, array $entry)`	Updates the entry identified by `$dn` with its attributes `$entry` to the LDAP tree. Throws a `Zend_Ldap_Exception` if the entry could not be modified.
`Zend_Ldap save(string|Zend_Ldap_Dn $dn, array $entry)`	Saves the entry identified by `$dn` with its attributes `$entry` to the LDAP tree. Throws a `Zend_Ldap_Exception` if the entry could not be saved. This method decides by querying the LDAP tree if the entry will be added or updated.
`Zend_Ldap delete(string|Zend_Ldap_Dn $dn, boolean $recursively)`	Deletes the entry identified by `$dn` from the LDAP tree. Throws a `Zend_Ldap_Exception` if the entry could not be deleted. `$recursively` is FALSE by default. If set to TRUE the deletion will be carried out recursively and will effectively delete a complete subtree. Deletion will fail if `$recursively` is FALSE and the entry `$dn` is not a leaf entry.
`Zend_Ldap moveToSubtree(string|Zend_Ldap_Dn $from, string|Zend_Ldap_Dn $to, boolean $recursively, boolean $alwaysEmulate)`	Moves the entry identified by `$from` to a location below `$to` keeping its RDN unchanged. `$recursively` specifies if the operation will be carried out recursively (FALSE by default) so that the entry `$from` and all its descendants will be moved. Moving will fail if `$recursively` is FALSE and the entry `$from` is not a leaf entry. `$alwaysEmulate` controls whether the ext/ldap function `ldap_rename()` should be used if available. This can only work for leaf entries and for servers and for ext/ldap supporting this function. Set to TRUE to always use an emulated rename operation.  All move-operations are carried out by copying and then deleting the corresponding entries in the LDAP tree. These operations are not *atomic* so that failures during the operation will result in an *inconsistent* state on the LDAP server. The same is true for all recursive operations. They

Method	Description
	also are by no means atomic. Please keep this in mind.
`Zend_Ldap move(string|Zend_Ldap_Dn $from, string|Zend_Ldap_Dn $to, boolean $recursively, boolean $alwaysEmulate)`	This is an alias for `Zend_Ldap::rename()`.
`Zend_Ldap rename(string|Zend_Ldap_Dn $from, string|Zend_Ldap_Dn $to, boolean $recursively, boolean $alwaysEmulate)`	Renames the entry identified by `$from` to `$to`. `$recursively` specifies if the operation will be carried out recursively (FALSE by default) so that the entry `$from` and all its descendants will be moved. Moving will fail if `$recursively` is FALSE and the entry `$from` is not a leaf entry. `$alwaysEmulate` controls whether the ext/ldap function `ldap_rename()` should be used if available. This can only work for leaf entries and for servers and for ext/ldap supporting this function. Set to TRUE to always use an emulated rename operation.
`Zend_Ldap copyToSubtree(string|Zend_Ldap_Dn $from, string|Zend_Ldap_Dn $to, boolean $recursively)`	Copies the entry identified by `$from` to a location below `$to` keeping its RDN unchanged. `$recursively` specifies if the operation will be carried out recursively (FALSE by default) so that the entry `$from` and all its descendants will be copied. Copying will fail if `$recursively` is FALSE and the entry `$from` is not a leaf entry.
`Zend_Ldap copy(string|Zend_Ldap_Dn $from, string|Zend_Ldap_Dn $to, boolean $recursively)`	Copies the entry identified by `$from` to `$to`. `$recursively` specifies if the operation will be carried out recursively (FALSE by default) so that the entry `$from` and all its descendants will be copied. Copying will fail if `$recursively` is FALSE and the entry `$from` is not a leaf entry.
`Zend_Ldap_Node getNode(string|Zend_Ldap_Dn $dn)`	Returns the entry `$dn` wrapped in a `Zend_Ldap_Node`.
`Zend_Ldap_Node getBaseNode()`	Returns the entry for the base DN `$baseDn` wrapped in a `Zend_Ldap_Node`.
`Zend_Ldap_Node_RootDse getRootDse()`	Returns the RootDSE for the current server.
`Zend_Ldap_Node_Schema getSchema()`	Returns the LDAP schema for the current server.

2.2.1.1. Zend_Ldap_Collection

`Zend_Ldap_Collection` implements `Iterator` to allow for item traversal using `foreach()` and `Countable` to be able to respond to `count()`. With its protected `_createEntry()` method it provides a simple extension point for developers needing custom result objects.

Table 71. Zend_Ldap_Collection API

Method	Description
`__construct(Zend_Ldap_Collection_Iterator_Interface $iterator)`	Constructor. The constrcutor must be provided by a `Zend_Ldap_Collection_Iterator_Interface` which does the real result iteration. `Zend_Ldap_Collection_Iterator_Default` is the default implementation for iterating ext/ldap results.
`boolean close()`	Closes the internal iterator. This is also called in the destructor.
`array toArray()`	Returns all entries as an array.
`array getFirst()`	Returns the first entry in the collection or `NULL` if the collection is empty.

2.2.2. Zend_Ldap_Attribute

`Zend_Ldap_Attribute` is a helper class providing only static methods to manipulate arrays suitable to the structure used in `Zend_Ldap` data modification methods and to the data format required by the LDAP server. PHP data types are converted the following way:

`string`	No conversion will be done.
`integer` and `float`	The value will be converted to a string.
`boolean`	TRUE will be converted to *'TRUE'* and FALSE to *'FALSE'*
`object` and `array`	The value will be converted to a string by using `serialize()`.
`resource`	If a `stream` resource is given, the data will be fetched by calling `stream_get_contents()`.
others	All other data types (namely non-stream resources) will be ommitted.

On reading attribute values the following conversion will take place:

'TRUE'	Converted to TRUE.
'FALSE'	Converted to FALSE.
others	All other strings won't be automatically converted and are passed as they are.

Table 72. Zend_Ldap_Attribute API

Method	Description
`void setAttribute(array &$data, string $attribName, mixed $value, boolean $append)`	Sets the attribute `$attribName` in `$data` to the value `$value`. If `$append` is TRUE (FALSE by default) `$value` will be appended to the attribute. `$value` can be a scalar value or an array of scalar values. Conversion will take place.
`array|mixed getAttribute(array $data, string $attribName, integer|null $index)`	Returns the attribute `$attribName` from `$data`. If `$index` is `NULL` (default) an array will be returned containing all the values for the given attribute. An empty array will be

Method	Description
	returned if the attribute does not exist in the given array. If an integer index is specified the corresponding value at the given index will be returned. If the index is out of bounds, NULL will be returned. Conversion will take place.
boolean attributeHasValue(array &$data, string $attribName, mixed|array $value)	Checks if the attribute $attribName in $data has the value(s) given in $value. The method returns TRUE only if all values in $value are present in the attribute. Comparison is done strictly (respecting the data type).
void removeDuplicatesFromAttribute(array &$data, string $attribName)	Removes all duplicates from the attribute $attribName in $data.
void removeFromAttribute(array &$data, string $attribName, mixed|array $value)	Removes the value(s) given in $value from the attribute $attribName in $data.
string|null convertToLdapValue(mixed $value)	Converts a PHP data type into its LDAP representation. See introduction for details.
mixed convertFromLdapValue(string $value)	Converts an LDAP value into its PHP data type. See introduction for details.
string|null convertToLdapDateTimeValue(integer $value, boolean $utc)	Converts a timestamp into its LDAP date/time representation. If $utc is TRUE (FALSE by default) the resulting LDAP date/time string will be in UTC, otherwise a local date/time string will be returned.
integer|null convertFromLdapDateTimeValue(string $value)	Converts LDAP date/time representation into a timestamp. The method returns NULL if $value can not be converted back into a PHP timestamp.
void setPassword(array &$data, string $password, string $hashType, string $attribName)	Sets a LDAP password for the attribute $attribName in $data. $attribName defaults to 'userPassword' which is the standard password attribute. The password hash can be specified with $hashType. The default value here is Zend_Ldap_Attribute::PASSWORD_HASH_MD5 with Zend_Ldap_Attribute::PASSWORD_HASH_SHA as the other possibilty.
string createPassword(string $password, string $hashType)	Creates a LDAP password. The password hash can be specified with $hashType. The default value here is Zend_Ldap_Attribute::PASSWORD_HASH_MD5 with Zend_Ldap_Attribute::PASSWORD_HASH_SHA as the other possibilty.
void setDateTimeAttribute(array &$data, string $attribName, integer|	Sets the attribute $attribName in $data to the date/time value $value. If $append

Method	Description
array $value, boolean $utc, boolean $append)	is TRUE (FALSE by default) $value will be appended to the attribute. $value can be an integer value or an array of integers. Date-time-conversion according to Zend_Ldap_Attribute::convertToLdapDateTimeValue will take place.
array|integer getDateTimeAttribute(array $data, string $attribName, integer|null $index)	Returns the date/time attribute $attribName from $data. If $index is NULL (default) an array will be returned containing all the date/time values for the given attribute. An empty array will be returned if the attribute does not exist in the given array. If an integer index is specified the corresponding date/time value at the given index will be returned. If the index is out of bounds, NULL will be returned. Date-time-conversion according to Zend_Ldap_Attribute::convertFromLdapDateTimeVal will take place.

2.2.3. Zend_Ldap_Dn

Zend_Ldap_Dn provides an object-oriented interface to manipulating LDAP distinguished names (DN). The parameter $caseFold that is used in several methods determines the way DN attributes are handled regarding their case. Allowed values for this paraneter are:

Zend_Ldap_Dn::ATTR_CASEFOLD_NONE: No case-folding will be done.

Zend_Ldap_Dn::ATTR_CASEFOLD_UPPER: All attributes will be converted to upper-case.

Zend_Ldap_Dn::ATTR_CASEFOLD_LOWER: All attributes will be converted to lower-case.

The default case-folding is Zend_Ldap_Dn::ATTR_CASEFOLD_NONE and can be set with Zend_Ldap_Dn::setDefaultCaseFold(). Each instance of Zend_Ldap_Dn can have its own case-folding-setting. If the $caseFold parameter is ommitted in method-calls it defaults to the instance's case-folding setting.

The class implements ArrayAccess to allow indexer-access to the different parts of the DN. The ArrayAccess-methods proxy to Zend_Ldap_Dn::get($offset, 1, null) for offsetGet(integer $offset), to Zend_Ldap_Dn::set($offset, $value) for offsetSet() and to Zend_Ldap_Dn::remove($offset, 1) for offsetUnset(). offsetExists() simply checks if the index is within the bounds.

Table 73. Zend_Ldap_Dn API

Method	Description
Zend_Ldap_Dn factory(string|array $dn, string|null $caseFold)	Creates a Zend_Ldap_Dn instance from an array or a string. The array must conform to the array structure detailed under Zend_Ldap_Dn::implodeDn().
Zend_Ldap_Dn fromString(string $dn, string|null $caseFold)	Creates a Zend_Ldap_Dn instance from a string.
Zend_Ldap_Dn fromArray(array $dn, string|null $caseFold)	Creates a Zend_Ldap_Dn instance from an array. The array must conform

Method	Description
	to the array structure detailed under `Zend_Ldap_Dn::implodeDn()`.
`array getRdn(string|null $caseFold)`	Gets the RDN of the current DN. The return value is an array with the RDN attribute names its keys and the RDN attribute values.
`string getRdnString(string|null $caseFold)`	Gets the RDN of the current DN. The return value is a string.
`Zend_Ldap_Dn getParentDn(integer $levelUp)`	Gets the DN of the current DN's ancestor `$levelUp` levels up the tree. `$levelUp` defaults to `1`.
`array get(integer $index, integer $length, string|null $caseFold)`	Returns a slice of the current DN determined by `$index` and `$length`. `$index` starts with `0` on the DN part from the left.
`Zend_Ldap_Dn set(integer $index, array $value)`	Replaces a DN part in the current DN. This operation manipulates the current instance.
`Zend_Ldap_Dn remove(integer $index, integer $length)`	Removes a DN part from the current DN. This operation manipulates the current instance. `$length` defaults to `1`
`Zend_Ldap_Dn append(array $value)`	Appends a DN part to the current DN. This operation manipulates the current instance.
`Zend_Ldap_Dn prepend(array $value)`	Prepends a DN part to the current DN. This operation manipulates the current instance.
`Zend_Ldap_Dn insert(integer $index, array $value)`	Inserts a DN part after the index `$index` to the current DN. This operation manipulates the current instance.
`void setCaseFold(string|null $caseFold)`	Sets the case-folding option to the current DN instance. If `$caseFold` is `NULL` the default case-folding setting (`Zend_Ldap_Dn::ATTR_CASEFOLD_NONE` by default or set via `Zend_Ldap_Dn::setDefaultCaseFold()` will be set for the current instance.
`string toString(string|null $caseFold)`	Returns DN as a string.
`array toArray(string|null $caseFold)`	Returns DN as an array.
`string __toString()`	Returns DN as a string - proxies to `Zend_Ldap_Dn::toString(null)`.
`void setDefaultCaseFold(string $caseFold)`	Sets the default case-folding option used by all instances on creation by default. Already existing instances are not affected by this setting.
`array escapeValue(string|array $values)`	Escapes a DN value according to RFC 2253.
`array unescapeValue(string|array $values)`	Undoes the conversion done by `Zend_Ldap_Dn::escapeValue()`.

Method	Description
array explodeDn(string $dn, array &$keys, array &$vals, string|null $caseFold)	Explodes the DN `$dn` into an array containing all parts of the given DN. `$keys` optinally receive DN keys (e.g. CN, OU, DC, ...). `$vals` optionally receive DN values. The resulting array will be of type `array( array("cn" => "name1", "uid" => "user"), array("cn" => "name2"), array("dc" => "example"), array("dc" => "org") )` for a DN of `cn=name1+uid=user,cn=name2,dc=example,dc=org`.
boolean checkDn(string $dn, array &$keys, array &$vals, string|null $caseFold)	Checks if a given DN `$dn` is malformed. If `$keys` or `$keys` and `$vals` are given, these arrays will be filled with the appropriate DN keys and values.
string implodeRdn(array $part, string|null $caseFold)	Returns a DN part in the form `$attribute=$value`
string implodeDn(array $dnArray, string|null $caseFold, string $separator)	Implodes an array in the form delivered by `Zend_Ldap_Dn::explodeDn()` to a DN string. `$separator` defaults to `','` but some LDAP servers also understand `';'`. `$dnArray` must of type `array( array("cn" => "name1", "uid" => "user"), array("cn" => "name2"), array("dc" => "example"), array("dc" => "org") )`
boolean isChildOf(string|Zend_Ldap_Dn $childDn, string|Zend_Ldap_Dn $parentDn)	Checks if given `$childDn` is beneath `$parentDn` subtree.

2.2.4. Zend_Ldap_Filter

Table 74. Zend_Ldap_Filter API

Method	Description
Zend_Ldap_Filter equals(string $attr, string $value)	Creates an 'equals' filter: `(attr=value)`.
Zend_Ldap_Filter begins(string $attr, string $value)	Creates an 'begins with' filter: `(attr=value*)`.
Zend_Ldap_Filter ends(string $attr, string $value)	Creates an 'ends with' filter: `(attr=*value)`.
Zend_Ldap_Filter contains(string $attr, string $value)	Creates an 'contains' filter: `(attr=*value*)`.

Method	Description
Zend_Ldap_Filter greater(string $attr, string $value)	Creates an 'greater' filter: (attr>value).
Zend_Ldap_Filter greaterOrEqual(string $attr, string $value)	Creates an 'greater or equal' filter: (attr>=value).
Zend_Ldap_Filter less(string $attr, string $value)	Creates an 'less' filter: (attr<value).
Zend_Ldap_Filter lessOrEqual(string $attr, string $value)	Creates an 'less or equal' filter: (attr<=value).
Zend_Ldap_Filter approx(string $attr, string $value)	Creates an 'approx' filter: (attr~=value).
Zend_Ldap_Filter any(string $attr)	Creates an 'any' filter: (attr=*).
Zend_Ldap_Filter string(string $filter)	Creates a simple custom string filter. The user is responsible for all value-escaping as the filter is used as is.
Zend_Ldap_Filter mask(string $mask, string $value,...)	Creates a filter from a string mask. All $value parameters will be escaped and substituted into $mask by using sprintf()
Zend_Ldap_Filter andFilter(Zend_Ldap_Filter_Abstract $filter,...)	Creates an 'and' filter from all arguments given.
Zend_Ldap_Filter orFilter(Zend_Ldap_Filter_Abstract $filter,...)	Creates an 'or' filter from all arguments given.
__construct(string $attr, string $value, string $filtertype, string|null $prepend, string|null $append)	Constructor. Creates an arbitrary filter according to the parameters supplied. The resulting filter will be a concatenation $attr . $filtertype . $prepend . $value . $append. Normally this constructor is not needed as all filters can be created by using the appropriate factory methods.
string toString()	Returns a string representation of the filter.
string __toString()	Returns a string representation of the filter. Proxies to Zend_Ldap_Filter::toString().
Zend_Ldap_Filter_Abstract negate()	Negates the current filter.
Zend_Ldap_Filter_Abstract addAnd(Zend_Ldap_Filter_Abstract $filter,...)	Creates an 'and' filter from the current filter and all filters passed in as the arguments.
Zend_Ldap_Filter_Abstract addOr(Zend_Ldap_Filter_Abstract $filter,...)	Creates an 'or' filter from the current filter and all filters passed in as the arguments.
string|array escapeValue(string|array $values)	Escapes the given $values according to RFC 2254 so that they can be safely used in LDAP filters. If a single string is given,

Method	Description
	a string is returned - otherwise an array is returned. Any control characters with an ASCII code < 32 as well as the characters with special meaning in LDAP filters "*", "(", ")", and "\" (the backslash) are converted into the representation of a backslash followed by two hex digits representing the hexadecimal value of the character.
string|array unescapeValue(string|array $values)	Undoes the conversion done by `Zend_Ldap_Filter::escapeValue()`. Converts any sequences of a backslash followed by two hex digits into the corresponding character.

2.2.5. Zend_Ldap_Node

`Zend_Ldap_Node` includes the magic property accessors `__set()`, `__get()`, `__unset()` and `__isset()` to access the attributes by their name. They proxy to `Zend_Ldap_Node::setAttribute()`, `Zend_Ldap_Node::getAttribute()`, `Zend_Ldap_Node::deleteAttribute()` and `Zend_Ldap_Node::existsAttribute()` respectively. Furthermore the class implements `ArrayAccess` for array-style-access to the attributes. `Zend_Ldap_Node` also implements `Iterator` and `RecursiveIterato` to allow for recursive tree-traversal.

Table 75. Zend_Ldap_Node API

Method	Description
`Zend_Ldap getLdap()`	Returns the current LDAP connection. Throws `Zend_Ldap_Exception` if current node is in detached mode (not connected to a `Zend_Ldap` instance).
`Zend_Ldap_Node attachLdap(Zend_Ldap $ldap)`	Attach the current node to the `$ldap` `Zend_Ldap` instance. Throws `Zend_Ldap_Exception` if `$ldap` is not responsible for the current node (node is not a child of the `$ldap` base DN).
`Zend_Ldap_Node detachLdap()`	Detach node from LDAP connection.
`boolean isAttached()`	Checks if the current node is attached to a LDAP connection.
Zend_Ldap_Node create(string|array|Zend_Ldap_Dn $dn, array $objectClass)	Factory method to create a new detached `Zend_Ldap_Node` for a given DN. Creates a new `Zend_Ldap_Node` with the DN `$dn` and the object-classes `$objectClass`.
Zend_Ldap_Node fromLdap(string|array|Zend_Ldap_Dn $dn, Zend_Ldap $ldap)	Factory method to create an attached `Zend_Ldap_Node` for a given DN. Loads an existing `Zend_Ldap_Node` with the DN `$dn` from the LDAP connection `$ldap`.
Zend_Ldap_Node fromArray((array $data, boolean $fromDataSource)	Factory method to create a detached `Zend_Ldap_Node` from array data `$data`. If `$fromDataSource` is `TRUE` (`FALSE` by

Method	Description
	default), the data is treated as beeing present in a LDAP tree.
`boolean isNew()`	Tells if the node is consiedered as new (not present on the server). Please note, that this doesn't tell if the node is really present on the server. Use `Zend_Ldap_Node::exists()` to see if a node is already there.
`boolean willBeDeleted()`	Tells if this node is going to be deleted once `Zend_Ldap_Node::update()` is called.
`Zend_Ldap_Node delete()`	Marks this node as to be deleted. Node will be deleted on calling `Zend_Ldap_Node::update()` if `Zend_Ldap_Node::willBeDeleted()` is TRUE.
`boolean willBeMoved()`	Tells if this node is going to be moved once `Zend_Ldap_Node::update()` is called.
`Zend_Ldap_Node update(Zend_Ldap $ldap)`	Sends all pending changes to the LDAP server. If `$ldap` is omitted the current LDAP connection is used. If the current node is detached from a LDAP connection a `Zend_Ldap_Exception` will be thrown. If `$ldap` is provided the current node will be attached to the given LDAP connection.
`Zend_Ldap_Dn getCurrentDn()`	Gets the current DN of the current node as a Zend_Ldap_Dn. This does not reflect possible rename-operations.
`Zend_Ldap_Dn getDn()`	Gets the original DN of the current node as a Zend_Ldap_Dn. This reflects possible rename-operations.
`string getDnString(string $caseFold)`	Gets the original DN of the current node as a string. This reflects possible rename-operations.
`array getDnArray(string $caseFold)`	Gets the original DN of the current node as an array. This reflects possible rename-operations.
`string getRdnString(string $caseFold)`	Gets the RDN of the current node as a string. This reflects possible rename-operations.
`array getRdnArray(string $caseFold)`	Gets the RDN of the current node as an array. This reflects possible rename-operations.
`Zend_Ldap_Node setDn(Zend_Ldap_Dn|string|array $newDn)`	Sets the new DN for this node effectively moving the node once `Zend_Ldap_Node::update()` is called.
`Zend_Ldap_Node move(Zend_Ldap_Dn|string|array $newDn)`	This is an alias for `Zend_Ldap_Node::setDn()`.
`Zend_Ldap_Node rename(Zend_Ldap_Dn|string|array $newDn)`	This is an alias for `Zend_Ldap_Node::setDn()`.

Method	Description
`array getObjectClass()`	Returns the objectClass of the node.
`Zend_Ldap_Node setObjectClass(array|string $value)`	Sets the objectClass attribute.
`Zend_Ldap_Node appendObjectClass(array|string $value)`	Appends to the objectClass attribute.
`string toLdif(array $options)`	Returns a LDIF representation of the current node. `$options` will be passed to the `Zend_Ldap_Ldif_Encoder`.
`array getChangedData()`	Gets changed node data. The array contains all changed attributes. This format can be used in `Zend_Ldap::add()` and `Zend_Ldap::update()`.
`array getChanges()`	Returns all changes made.
`string toString()`	Returns the DN of the current node - proxies to `Zend_Ldap_Dn::getDnString()`.
`string __toString()`	Casts to string representation - proxies to `Zend_Ldap_Dn::toString()`.
`array toArray(boolean $includeSystemAttributes)`	Returns an array representation of the current node. If `$includeSystemAttributes` is FALSE (defaults to TRUE) the system specific attributes are stripped from the array. Unlike `Zend_Ldap_Node::getAttributes()` the resulting array contains the DN with key `'dn'`.
`string toJson(boolean $includeSystemAttributes)`	Returns a JSON representation of the current node using `Zend_Ldap_Node::toArray()`.
`array getData(boolean $includeSystemAttributes)`	Returns the node's attributes. The array contains all attributes in its internal format (no conversion).
`boolean existsAttribute(string $name, boolean $emptyExists)`	Checks whether a given attribute exists. If `$emptyExists` is FALSE empty attributes (containing only array()) are treated as non-existent returning FALSE. If `$emptyExists` is TRUE empty attributes are treated as existent returning TRUE. In this case the method returns FALSE only if the attribute name is missing in the key-collection.
`boolean attributeHasValue(string $name, mixed|array $value)`	Checks if the given value(s) exist in the attribute. The method returns TRUE only if all values in `$value` are present in the attribute. Comparison is done strictly (respecting the data type).
`integer count()`	Returns the number of attributes in the node. Implements Countable.

Method	Description
`mixed getAttribute(string $name, integer|null $index)`	Gets a LDAP attribute. Data conversion is applied using `Zend_Ldap_Attribute::getAttribute()`.
`array getAttributes(boolean $includeSystemAttributes)`	Gets all attributes of node. If `$includeSystemAttributes` is `FALSE` (defaults to `TRUE`) the system specific attributes are stripped from the array.
`Zend_Ldap_Node setAttribute(string $name, mixed $value)`	Sets a LDAP attribute. Data conversion is applied using `Zend_Ldap_Attribute::setAttribute()`.
`Zend_Ldap_Node appendToAttribute(string $name, mixed $value)`	Appends to a LDAP attribute. Data conversion is applied using `Zend_Ldap_Attribute::setAttribute()`.
`array|integer getDateTimeAttribute(string $name, integer|null $index)`	Gets a LDAP date/time attribute. Data conversion is applied using `Zend_Ldap_Attribute::getDateTimeAttribute()`.
`Zend_Ldap_Node setDateTimeAttribute(string $name, integer|array $value, boolean $utc)`	Sets a LDAP date/time attribute. Data conversion is applied using `Zend_Ldap_Attribute::setDateTimeAttribute()`.
`Zend_Ldap_Node appendToDateTimeAttribute(string $name, integer|array $value, boolean $utc)`	Appends to a LDAP date/time attribute. Data conversion is applied using `Zend_Ldap_Attribute::setDateTimeAttribute()`.
`Zend_Ldap_Node setPasswordAttribute(string $password, string $hashType, string $attribName)`	Sets a LDAP password on `$attribName` (defaults to `'userPassword'`) to `$password` with the hash type `$hashType` (defaults to `Zend_Ldap_Attribute::PASSWORD_HASH_MD5`).
`Zend_Ldap_Node deleteAttribute(string $name)`	Deletes a LDAP attribute.
`void removeDuplicatesFromAttribute(string $name)`	Removes duplicate values from a LDAP attribute.
`void removeFromAttribute(string $attribName, mixed|array $value)`	Removes the given values from a LDAP attribute.
`boolean exists(Zend_Ldap $ldap)`	Checks if the current node exists on the given LDAP server (current server is used if `NULL` is passed).
`Zend_Ldap_Node reload(Zend_Ldap $ldap)`	Reloads the current node's attributes from the given LDAP server (current server is used if `NULL` is passed).
`Zend_Ldap_Node_Collection searchSubtree(string|Zend_Ldap_Filter_Abstract $filter, integer $scope, string $sort)`	Searches the nodes's subtree with the given `$filter` and the given search parameters. See `Zend_Ldap::search()` for details on the parameters `$scope` and `$sort`.
`integer countSubtree(string|Zend_Ldap_Filter_Abstract $filter, integer $scope)`	Count the nodes's subtree items matching the given `$filter` and the given search scope.

Method	Description
	See `Zend_Ldap::search()` for details on the `$scope` parameter.
`integer countChildren()`	Count the nodes's children.
`Zend_Ldap_Node_Collection searchChildren(string|Zend_Ldap_Filter_Abstract $filter, string $sort)`	Searches the nodes's children matching the given `$filter`. See `Zend_Ldap::search()` for details on the `$sort` parameter.
`boolean hasChildren()`	Returns whether the current node has children.
`Zend_Ldap_Node_ChildrenIterator getChildren()`	Returns all children of the current node.
`Zend_Ldap_Node getParent(Zend_Ldap $ldap)`	Returns the parent of the current node using the LDAP connection `$ldap` (uses the current LDAP connection if omitted).

2.2.6. Zend_Ldap_Node_RootDse

The following methods are available on all vendor-specific subclasses.

`Zend_Ldap_Node_RootDse` includes the magic property accessors `__get()` and `__isset()` to access the attributes by their name. They proxy to `Zend_Ldap_Node_RootDse::getAttribute()` and `Zend_Ldap_Node_RootDse::existsAttribute()` respectively. `__set()` and `__unset()` are also implemented but they throw a `BadMethodCallException` as modifications are not allowed on RootDSE nodes. Furthermore the class implements `ArrayAccess` for array-style-access to the attributes. `offsetSet()` and `offsetUnset()` also throw a `BadMethodCallException` due ro obvious reasons.

Table 76. Zend_Ldap_Node_RootDse API

Method	Description
`Zend_Ldap_Dn getDn()`	Gets the DN of the current node as a Zend_Ldap_Dn.
`string getDnString(string $caseFold)`	Gets the DN of the current node as a string.
`array getDnArray(string $caseFold)`	Gets the DN of the current node as an array.
`string getRdnString(string $caseFold)`	Gets the RDN of the current node as a string.
`array getRdnArray(string $caseFold)`	Gets the RDN of the current node as an array.
`array getObjectClass()`	Returns the objectClass of the node.
`string toString()`	Returns the DN of the current node - proxies to `Zend_Ldap_Dn::getDnString()`.
`string __toString()`	Casts to string representation - proxies to `Zend_Ldap_Dn::toString()`.
`array toArray(boolean $includeSystemAttributes)`	Returns an array representation of the current node. If `$includeSystemAttributes` is `FALSE` (defaults to `TRUE`) the system specific attributes are stripped from the array. Unlike

Method	Description
	`Zend_Ldap_Node_RootDse::getAttributes()` the resulting array contains the DN with key `'dn'`.
`string toJson(boolean $includeSystemAttributes)`	Returns a JSON representation of the current node using `Zend_Ldap_Node_RootDse::toArray()`.
`array getData(boolean $includeSystemAttributes)`	Returns the node's attributes. The array contains all attributes in its internal format (no conversion).
`boolean existsAttribute(string $name, boolean $emptyExists)`	Checks whether a given attribute exists. If `$emptyExists` is FALSE, empty attributes (containing only array()) are treated as non-existent returning FALSE. If `$emptyExists` is TRUE, empty attributes are treated as existent returning TRUE. In this case the method returns FALSE only if the attribute name is missing in the key-collection.
`boolean attributeHasValue(string $name, mixed|array $value)`	Checks if the given value(s) exist in the attribute. The method returns TRUE only if all values in `$value` are present in the attribute. Comparison is done strictly (respecting the data type).
`integer count()`	Returns the number of attributes in the node. Implements Countable.
`mixed getAttribute(string $name, integer|null $index)`	Gets a LDAP attribute. Data conversion is applied using `Zend_Ldap_Attribute::getAttribute()`.
`array getAttributes(boolean $includeSystemAttributes)`	Gets all attributes of node. If `$includeSystemAttributes` is FALSE (defaults to TRUE) the system specific attributes are stripped from the array.
`array|integer getDateTimeAttribute(string $name, integer|null $index)`	Gets a LDAP date/time attribute. Data conversion is applied using `Zend_Ldap_Attribute::getDateTimeAttribute()`.
`Zend_Ldap_Node_RootDse reload(Zend_Ldap $ldap)`	Reloads the current node's attributes from the given LDAP server.
`Zend_Ldap_Node_RootDse create(Zend_Ldap $ldap)`	Factory method to create the RootDSE.
`array getNamingContexts()`	Gets the namingContexts.
`string|null getSubschemaSubentry()`	Gets the subschemaSubentry.
`boolean supportsVersion(string|int|array $versions)`	Determines if the LDAP version is supported.
`boolean supportsSaslMechanism(string|array $mechlist)`	Determines if the sasl mechanism is supported.
`integer getServerType()`	Gets the server type. Returns

Method	Description
	`Zend_Ldap_Node_RootDse::SERVER_TYPE_GENERIC` for unknown LDAP servers `Zend_Ldap_Node_RootDse::SERVER_TYPE_OPENLDAP` for OpenLDAP servers `Zend_Ldap_Node_RootDse::SERVER_TYPE_ACTIVEDIREC` for Microsoft ActiveDirectory servers `Zend_Ldap_Node_RootDse::SERVER_TYPE_EDIRECTORY` for eDirectory servers
`Zend_Ldap_Dn getSchemaDn()`	Returns the schema DN.

2.2.6.1. OpenLDAP

Additionally the common methods above apply to instances of `Zend_Ldap_Node_RootDse_OpenLdap`.

Refer to LDAP Operational Attributes and Objectsfor information on the attributes of OpenLDAP RootDSE.

Table 77. Zend_Ldap_Node_RootDse_OpenLdap API

Method	Description
`integer getServerType()`	Gets the server type. Returns `Zend_Ldap_Node_RootDse::SERVER_TYPE_OPENLDAP`
`string|null getConfigContext()`	Gets the configContext.
`string|null getMonitorContext()`	Gets the monitorContext.
`boolean supportsControl(string|array $oids)`	Determines if the control is supported.
`boolean supportsExtension(string|array $oids)`	Determines if the extension is supported.
`boolean supportsFeature(string|array $oids)`	Determines if the feature is supported.

2.2.6.2. ActiveDirectory

Additionally the common methods above apply to instances of `Zend_Ldap_Node_RootDse_ActiveDirectory`.

Refer to RootDSEfor information on the attributes of Microsoft ActiveDirectory RootDSE.

Table 78. Zend_Ldap_Node_RootDse_ActiveDirectory API

Method	Description
`integer getServerType()`	Gets the server type. Returns `Zend_Ldap_Node_RootDse::SERVER_TYPE_ACTIVEDIREC`
`string|null getConfigurationNamingContext()`	Gets the configurationNamingContext.
`string|null getCurrentTime()`	Gets the currentTime.
`string|null getDefaultNamingContext()`	Gets the defaultNamingContext.
`string|null getDnsHostName()`	Gets the dnsHostName.
`string|null getDomainControllerFunctionality()`	Gets the domainControllerFunctionality.
`string|null getDomainFunctionality()`	Gets the domainFunctionality.
`string|null getDsServiceName()`	Gets the dsServiceName.
`string|null getForestFunctionality()`	Gets the forestFunctionality.
`string|null getHighestCommittedUSN()`	Gets the highestCommittedUSN.
`string|null getIsGlobalCatalogReady()`	Gets the isGlobalCatalogReady.
`string|null getIsSynchronized()`	Gets the isSynchronized.
`string|null getLdapServiceName()`	Gets the ldapServiceName.
`string|null getRootDomainNamingContext()`	Gets the rootDomainNamingContext.
`string|null getSchemaNamingContext()`	Gets the schemaNamingContext.
`string|null getServerName()`	Gets the serverName.
`boolean supportsCapability(string|array $oids)`	Determines if the capability is supported.
`boolean supportsControl(string|array $oids)`	Determines if the control is supported.
`boolean supportsPolicy(string|array $policies)`	Determines if the version is supported.

2.2.6.3. eDirectory

Additionally the common methods above apply to instances of `Zend_Ldap_Node_RootDse_eDirectory`.

Refer to Getting Information about the LDAP Serverfor information on the attributes of Novell eDirectory RootDSE.

Table 79. Zend_Ldap_Node_RootDse_eDirectory API

Method	Description
`integer getServerType()`	Gets the server type. Returns `Zend_Ldap_Node_RootDse::SERVER_TYPE_EDIRECTORY`
`boolean supportsExtension(string|array $oids)`	Determines if the extension is supported.
`string|null getVendorName()`	Gets the vendorName.
`string|null getVendorVersion()`	Gets the vendorVersion.
`string|null getDsaName()`	Gets the dsaName.
`string|null getStatisticsErrors()`	Gets the server statistics "errors".
`string|null getStatisticsSecurityErrors()`	Gets the server statistics "securityErrors".
`string|null getStatisticsChainings()`	Gets the server statistics "chainings".
`string|null getStatisticsReferralsReturned()`	Gets the server statistics "referralsReturned".
`string|null getStatisticsExtendedOps()`	Gets the server statistics "extendedOps".
`string|null getStatisticsAbandonOps()`	Gets the server statistics "abandonOps".
`string|null getStatisticsWholeSubtreeSearchOps()`	Gets the server statistics "wholeSubtreeSearchOps".

2.2.7. Zend_Ldap_Node_Schema

The following methods are available on all vendor-specific subclasses.

`Zend_Ldap_Node_Schema` includes the magic property accessors `__get()` and `__isset()` to access the attributes by their name. They proxy to `Zend_Ldap_Node_Schema::getAttribute()` and `Zend_Ldap_Node_Schema::existsAttribute()` respectively. `__set()` and `__unset()` are also implemented, but they throw a `BadMethodCallException` as modifications are not allowed on RootDSE nodes. Furthermore the class implements `ArrayAccess` for array-style-access to the attributes. `offsetSet()` and `offsetUnset()` also throw a `BadMethodCallException` due to obvious reasons.

Table 80. Zend_Ldap_Node_Schema API

Method	Description
`Zend_Ldap_Dn getDn()`	Gets the DN of the current node as a Zend_Ldap_Dn.
`string getDnString(string $caseFold)`	Gets the DN of the current node as a string.
`array getDnArray(string $caseFold)`	Gets the DN of the current node as an array.
`string getRdnString(string $caseFold)`	Gets the RDN of the current node as a string.
`array getRdnArray(string $caseFold)`	Gets the RDN of the current node as an array.

Method	Description
`array getObjectClass()`	Returns the objectClass of the node.
`string toString()`	Returns the DN of the current node - proxies to `Zend_Ldap_Dn::getDnString()`.
`string __toString()`	Casts to string representation - proxies to `Zend_Ldap_Dn::toString()`.
`array toArray(boolean $includeSystemAttributes)`	Returns an array representation of the current node. If `$includeSystemAttributes` is FALSE (defaults to TRUE), the system specific attributes are stripped from the array. Unlike `Zend_Ldap_Node_Schema::getAttributes()`, the resulting array contains the DN with key `'dn'`.
`string toJson(boolean $includeSystemAttributes)`	Returns a JSON representation of the current node using `Zend_Ldap_Node_Schema::toArray()`.
`array getData(boolean $includeSystemAttributes)`	Returns the node's attributes. The array contains all attributes in its internal format (no conversion).
`boolean existsAttribute(string $name, boolean $emptyExists)`	Checks whether a given attribute exists. If `$emptyExists` is FALSE, empty attributes (containing only array()) are treated as non-existent returning FALSE. If `$emptyExists` is TRUE, empty attributes are treated as existent returning TRUE. In this case the method returns FALSE only if the attribute name is missing in the key-collection.
`boolean attributeHasValue(string $name, mixed|array $value)`	Checks if the given value(s) exist in the attribute. The method returns TRUE only if all values in `$value` are present in the attribute. Comparison is done strictly (respecting the data type).
`integer count()`	Returns the number of attributes in the node. Implements Countable.
`mixed getAttribute(string $name, integer|null $index)`	Gets a LDAP attribute. Data conversion is applied using `Zend_Ldap_Attribute::getAttribute()`.
`array getAttributes(boolean $includeSystemAttributes)`	Gets all attributes of node. If `$includeSystemAttributes` is FALSE (defaults to TRUE) the system specific attributes are stripped from the array.
`array|integer getDateTimeAttribute(string $name, integer|null $index)`	Gets a LDAP date/time attribute. Data conversion is applied using `Zend_Ldap_Attribute::getDateTimeAttribute()`.
`Zend_Ldap_Node_Schema reload(Zend_Ldap $ldap)`	Reloads the current node's attributes from the given LDAP server.
`Zend_Ldap_Node_Schema create(Zend_Ldap $ldap)`	Factory method to create the Schema node.

Method	Description
`array getAttributeTypes()`	Gets the attribute types as an array of .
`array getObjectClasses()`	Gets the object classes as an array of `Zend_Ldap_Node_Schema_ObjectClass_Interface`.

Table 81. Zend_Ldap_Node_Schema_AttributeType_Interface API

Method	Description
`string getName()`	Gets the attribute name.
`string getOid()`	Gets the attribute OID.
`string getSyntax()`	Gets the attribute syntax.
`int|null getMaxLength()`	Gets the attribute maximum length.
`boolean isSingleValued()`	Returns if the attribute is single-valued.
`string getDescription()`	Gets the attribute description

Table 82. Zend_Ldap_Node_Schema_ObjectClass_Interface API

Method	Description
`string getName()`	Returns the objectClass name.
`string getOid()`	Returns the objectClass OID.
`array getMustContain()`	Returns the attributes that this objectClass must contain.
`array getMayContain()`	Returns the attributes that this objectClass may contain.
`string getDescription()`	Returns the attribute description
`integer getType()`	Returns the objectClass type. The method returns one of the following values: `Zend_Ldap_Node_Schema::OBJECTCLASS_TYPE_UNKNOWN` unknown class types `Zend_Ldap_Node_Schema::OBJECTCLASS_TYPE_STRUCTU` structural classes `Zend_Ldap_Node_Schema::OBJECTCLASS_TYPE_ABSTRAC` abstract classes `Zend_Ldap_Node_Schema::OBJECTCLASS_TYPE_AUXILIA` auxiliary classes
`array getParentClasses()`	Returns the parent objectClasses of this class. This includes structural, abstract and auxiliary objectClasses.

Classes representing attribute types and object classes extend `Zend_Ldap_Node_Schema_Item` which provides some core methods to access arbitrary attributes on the underlying LDAP node. `Zend_Ldap_Node_Schema_Item` includes the magic property accessors `__get()` and `__isset()` to access the attributes by their name.

Furthermore the class implements `ArrayAccess` for array-style-access to the attributes. `offsetSet()` and `offsetUnset()` throw a `BadMethodCallException` as modifications are not allowed on schema information nodes.

Table 83. Zend_Ldap_Node_Schema_Item API

Method	Description
`array getData()`	Gets all the underlying data from the schema information node.
`integer count()`	Returns the number of attributes in this schema information node. Implements Countable.

2.2.7.1. OpenLDAP

Additionally the common methods above apply to instances of `Zend_Ldap_Node_Schema_OpenLDAP`.

Table 84. Zend_Ldap_Node_Schema_OpenLDAP API

Method	Description
`array getLdapSyntaxes()`	Gets the LDAP syntaxes.
`array getMatchingRules()`	Gets the matching rules.
`array getMatchingRuleUse()`	Gets the matching rule use.

Table 85. Zend_Ldap_Node_Schema_AttributeType_OpenLDAP API

Method	Description
`Zend_Ldap_Node_Schema_AttributeType_OpenLDAP null getParent()`	Returns the parent attribute type in the inheritance tree if one exists.

Table 86. Zend_Ldap_Node_Schema_ObjectClass_OpenLDAP API

Method	Description
`array getParents()`	Returns the parent object classes in the inheritance tree if one exists. The returned array is an array of `Zend_Ldap_Node_Schema_ObjectClass_OpenLdap`.

2.2.7.2. ActiveDirectory

Schema browsing on ActiveDirectory servers

Due to restrictions on Microsoft ActiveDirectory servers regarding the number of entries returned by generic search routines and due to the structure of the ActiveDirectory schema repository, schema browsing is currently *not* available for Microsoft ActiveDirectory servers.

`Zend_Ldap_Node_Schema_ActiveDirectory` does not provide any additional methods.

Table 87. Zend_Ldap_Node_Schema_AttributeType_ActiveDirectory API

`Zend_Ldap_Node_Schema_AttributeType_ActiveDirectory` does not provide any additional methods.

Table 88. Zend_Ldap_Node_Schema_ObjectClass_ActiveDirectory API

`Zend_Ldap_Node_Schema_ObjectClass_ActiveDirectory` does not provide any additional methods.

2.2.8. Zend_Ldif_Encoder

Table 89. Zend_Ldif_Encoder API

Method	Description
`array decode(string $string)`	Decodes the string `$string` into an array of LDIF items.
`string encode(scalar|array|Zend_Ldap_Node $value, array $options)`	Encode `$value` into a LDIF representation. `$options` is an array that may contain the following keys: `'sort'` Sort the given attributes with `dn` following `objectClass` and following all other attributes sorted alphabetically. `TRUE` by default. `'version'` The LDIF format version. `1` by default. `'wrap'` The line-length. `78` by default to conform to the LDIF specification.

3. Usage Scenarios

3.1. Authentication scenarios

3.1.1. OpenLDAP

3.1.2. ActiveDirectory

3.2. Basic CRUD operations

3.2.1. Retrieving data from the LDAP

Example 453. Getting an entry by its DN

```
$options = array(/* ... */);
$ldap = new Zend_Ldap($options);
$ldap->bind();
$hm = $ldap->getEntry('cn=Hugo Müller,ou=People,dc=my,dc=local');
/*
$hm is an array of the following structure
array(
    'dn'          => 'cn=Hugo Müller,ou=People,dc=my,dc=local',
    'cn'          => array('Hugo Müller'),
    'sn'          => array('Müller'),
    'objectclass' => array('inetOrgPerson', 'top'),
    ...
)
*/
```

Example 454. Check for the existence of a given DN

```
$options = array(/* ... */);
$ldap = new Zend_Ldap($options);
$ldap->bind();
$isThere = $ldap->exists('cn=Hugo Müller,ou=People,dc=my,dc=local');
```

Example 455. Count children of a given DN

```
$options = array(/* ... */);
$ldap = new Zend_Ldap($options);
$ldap->bind();
$childrenCount = $ldap->countChildren(
                            'cn=Hugo Müller,ou=People,dc=my,dc=local');
```

Example 456. Searching the LDAP tree

```
$options = array(/* ... */);
$ldap = new Zend_Ldap($options);
$ldap->bind();
$result = $ldap->search('(objectclass=*)',
                        'ou=People,dc=my,dc=local',
                        Zend_Ldap_Ext::SEARCH_SCOPE_ONE);
foreach ($result as $item) {
    echo $item["dn"] . ': ' . $item['cn'][0] . PHP_EOL;
}
```

3.2.2. Adding data to the LDAP

Example 457. Add a new entry to the LDAP

```
$options = array(/* ... */);
$ldap = new Zend_Ldap($options);
$ldap->bind();
$entry = array();
Zend_Ldap_Attribute::setAttribute($entry, 'cn', 'Hans Meier');
Zend_Ldap_Attribute::setAttribute($entry, 'sn', 'Meier');
Zend_Ldap_Attribute::setAttribute($entry, 'objectClass', 'inetOrgPerson');
$ldap->add('cn=Hans Meier,ou=People,dc=my,dc=local', $entry);
```

3.2.3. Deleting from the LDAP

Example 458. Delete an existing entry from the LDAP

```
$options = array(/* ... */);
$ldap = new Zend_Ldap($options);
$ldap->bind();
$ldap->delete('cn=Hans Meier,ou=People,dc=my,dc=local');
```

3.2.4. Updating the LDAP

Example 459. Update an existing entry on the LDAP

```
$options = array(/* ... */);
$ldap = new Zend_Ldap($options);
$ldap->bind();
$hm = $ldap->getEntry('cn=Hugo Müller,ou=People,dc=my,dc=local');
Zend_Ldap_Attribute::setAttribute($hm, 'mail', 'mueller@my.local');
Zend_Ldap_Attribute::setPassword($hm,
                                 'newPa$$w0rd',
                                 Zend_Ldap_Attribute::PASSWORD_HASH_SHA1);
$ldap->update('cn=Hugo Müller,ou=People,dc=my,dc=local', $hm);
```

3.3. Extended operations

3.3.1. Copy and move entries in the LDAP

Example 460. Copy a LDAP entry recursively with all its descendants

```
$options = array(/* ... */);
$ldap = new Zend_Ldap($options);
$ldap->bind();
$ldap->copy('cn=Hugo Müller,ou=People,dc=my,dc=local',
            'cn=Hans Meier,ou=People,dc=my,dc=local',
            true);
```

Example 461. Move a LDAP entry recursively with all its descendants to a different subtree

```
$options = array(/* ... */);
$ldap = new Zend_Ldap($options);
$ldap->bind();
$ldap->moveToSubtree('cn=Hugo Müller,ou=People,dc=my,dc=local',
                     'ou=Dismissed,dc=my,dc=local',
                     true);
```

4. Tools

4.1. Creation and modification of DN strings

4.2. Using the filter API to create search filters

Example 462. Create simple LDAP filters

```
$f1  = Zend_Ldap_Filter::equals('name', 'value');         // (name=value)
$f2  = Zend_Ldap_Filter::begins('name', 'value');         // (name=value*)
$f3  = Zend_Ldap_Filter::ends('name', 'value');           // (name=*value)
$f4  = Zend_Ldap_Filter::contains('name', 'value');       // (name=*value*)
$f5  = Zend_Ldap_Filter::greater('name', 'value');        // (name>value)
$f6  = Zend_Ldap_Filter::greaterOrEqual('name', 'value'); // (name>=value)
$f7  = Zend_Ldap_Filter::less('name', 'value');           // (name<value)
$f8  = Zend_Ldap_Filter::lessOrEqual('name', 'value');    // (name<=value)
$f9  = Zend_Ldap_Filter::approx('name', 'value');         // (name~=value)
$f10 = Zend_Ldap_Filter::any('name');                     // (name=*)
```

Example 463. Create more complex LDAP filters

```
$f1 = Zend_Ldap_Filter::ends('name', 'value')->negate(); // (!(name=*value))

$f2 = Zend_Ldap_Filter::equals('name', 'value');
$f3 = Zend_Ldap_Filter::begins('name', 'value');
$f4 = Zend_Ldap_Filter::ends('name', 'value');

// (&(name=value)(name=value*)(name=*value))
$f5 = Zend_Ldap_Filter::andFilter($f2, $f3, $f4);

// (|(name=value)(name=value*)(name=*value))
$f6 = Zend_Ldap_Filter::orFilter($f2, $f3, $f4);
```

4.3. Modify LDAP entries using the Attribute API

5. Object oriented access to the LDAP tree using Zend_Ldap_Node

5.1. Basic CRUD operations

5.1.1. Retrieving data from the LDAP

5.1.1.1. Getting a node by its DN

5.1.1.2. Searching a node's subtree

5.1.2. Adding a new node to the LDAP

5.1.3. Deleting a node from the LDAP

5.1.4. Updating a node on the LDAP

5.2. Extended operations

5.2.1. Copy and move nodes in the LDAP

5.3. Tree traversal

Example 464. Traverse LDAP tree recursively

```
$options = array(/* ... */);
$ldap = new Zend_Ldap($options);
$ldap->bind();
$ri = new RecursiveIteratorIterator($ldap->getBaseNode(),
                                    RecursiveIteratorIterator::SELF_FIRST);
foreach ($ri as $rdn => $n) {
    var_dump($n);
}
```

6. Getting information from the LDAP server

6.1. RootDSE

See the following documents for more information on the attributes contained within the RootDSE for a given LDAP server.

- OpenLDAP
- Microsoft ActiveDirectory
- Novell eDirectory

Example 465. Getting hands on the RootDSE

```
$options = array(/* ... */);
$ldap = new Zend_Ldap($options);
$rootdse = $ldap->getRootDse();
$serverType = $rootdse->getServerType();
```

6.2. Schema Browsing

Example 466. Getting hands on the server schema

```
$options = array(/* ... */);
$ldap = new Zend_Ldap($options);
$schema = $ldap->getSchema();
$classes = $schema->getObjectClasses();
```

6.2.1. OpenLDAP

6.2.2. ActiveDirectory

Schema browsing on ActiveDirectory servers

Due to restrictions on Microsoft ActiveDirectory servers regarding the number of entries returned by generic search routines and due to the structure of the ActiveDirectory schema repository, schema browsing is currently *not* available for Microsoft ActiveDirectory servers.

7. Serializing LDAP data to and from LDIF

7.1. Serialize a LDAP entry to LDIF

```
$data = array(
    'dn'                          => 'uid=rogasawara,ou=###,o=Airius',
    'objectclass'                 => array('top',
                                           'person',
                                           'organizationalPerson',
                                           'inetOrgPerson'),
    'uid'                         => array('rogasawara'),
    'mail'                        => array('rogasawara@airius.co.jp'),
    'givenname;lang-ja'           => array('####'),
    'sn;lang-ja'                  => array('###'),
    'cn;lang-ja'                  => array('### ####'),
    'title;lang-ja'               => array('### ##'),
    'preferredlanguage'           => array('ja'),
    'givenname'                   => array('####'),
    'sn'                          => array('###'),
    'cn'                          => array('### ####'),
    'title'                       => array('### ##'),
    'givenname;lang-ja;phonetic' => array('####'),
    'sn;lang-ja;phonetic'         => array('#####'),
    'cn;lang-ja;phonetic'         => array('##### ####'),
    'title;lang-ja;phonetic'      => array('###### ####'),
    'givenname;lang-en'           => array('Rodney'),
    'sn;lang-en'                  => array('Ogasawara'),
    'cn;lang-en'                  => array('Rodney Ogasawara'),
    'title;lang-en'               => array('Sales, Director'),
);
$ldif = Zend_Ldap_Ldif_Encoder::encode($data, array('sort' => false,
                                                    'version' => null));
/*
$ldif contains:
dn:: dWlkPXJvZ2FzYXdhcmEsb3U95Za25qWt6YOoLG89QWlyaXVz
objectclass: top
objectclass: person
objectclass: organizationalPerson
objectclass: inetOrgPerson
uid: rogasawara
mail: rogasawara@airius.co.jp
givenname;lang-ja:: 44Ot44OJ44OL44O8
sn;lang-ja:: 5bCP56yg5Y6f
cn;lang-ja:: 5bCP56yg5Y6fIOODreODieODi+ODvA==
title;lang-ja:: 5Za25qWt6YOoIOmDqOmVtw==
preferredlanguage: ja
givenname:: 44Ot44OJ44OL44O8
sn:: 5bCP56yg5Y6f
```

```
cn:: 5bCP56yg5Y6fIOODreODieODi+ODvA==
title:: 5Za25qWt6YOoIOmDqOmVtw==
givenname;lang-ja;phonetic:: 44KN44Gp44Gr44O8
sn;lang-ja;phonetic:: 44GK44GM44GV44KP44KJ
cn;lang-ja;phonetic:: 44GK44GM44GV44KP44KJIOOCjeOBqeOBq+ODvA==
title;lang-ja;phonetic:: 44GI44GE44GO44KH44GG44G2IOOBtuOBoeOCh+OBhg==
givenname;lang-en: Rodney
sn;lang-en: Ogasawara
cn;lang-en: Rodney Ogasawara
title;lang-en: Sales, Director
*/
```

7.2. Deserialize a LDIF string into a LDAP entry

```
$ldif = "dn:: dWlkPXJvZ2FzYXdhcmEsb3U95Za25qWt6YOoLG89QWlyaXVz
objectclass: top
objectclass: person
objectclass: organizationalPerson
objectclass: inetOrgPerson
uid: rogasawara
mail: rogasawara@airius.co.jp
givenname;lang-ja:: 44Ot44OJ44OL44O8
sn;lang-ja:: 5bCP56yg5Y6f
cn;lang-ja:: 5bCP56yg5Y6fIOODreODieODi+ODvA==
title;lang-ja:: 5Za25qWt6YOoIOmDqOmVtw==
preferredlanguage: ja
givenname:: 44Ot44OJ44OL44O8
sn:: 5bCP56yg5Y6f
cn:: 5bCP56yg5Y6fIOODreODieODi+ODvA==
title:: 5Za25qWt6YOoIOmDqOmVtw==
givenname;lang-ja;phonetic:: 44KN44Gp44Gr44O8
sn;lang-ja;phonetic:: 44GK44GM44GV44KP44KJ
cn;lang-ja;phonetic:: 44GK44GM44GV44KP44KJIOOCjeOBqeOBq+ODvA==
title;lang-ja;phonetic:: 44GI44GE44GO44KH44GG44G2IOOBtuOBoeOCh+OBhg==
givenname;lang-en: Rodney
sn;lang-en: Ogasawara
cn;lang-en: Rodney Ogasawara
title;lang-en: Sales, Director";
$data = Zend_Ldap_Ldif_Encoder::decode($ldif);
/*
$data = array(
    'dn'                         => 'uid=rogasawara,ou=###,o=Airius',
    'objectclass'                => array('top',
                                          'person',
                                          'organizationalPerson',
                                          'inetOrgPerson'),
    'uid'                        => array('rogasawara'),
    'mail'                       => array('rogasawara@airius.co.jp'),
    'givenname;lang-ja'          => array('####'),
    'sn;lang-ja'                 => array('###'),
    'cn;lang-ja'                 => array('### ####'),
    'title;lang-ja'              => array('### ##'),
    'preferredlanguage'          => array('ja'),
    'givenname'                  => array('####'),
    'sn'                         => array('###'),
    'cn'                         => array('### ####'),
    'title'                      => array('### ##'),
    'givenname;lang-ja;phonetic' => array('####'),
    'sn;lang-ja;phonetic'        => array('#####'),
```

```
    'cn;lang-ja;phonetic'         => array('##### ####'),
    'title;lang-ja;phonetic'      => array('###### ####'),
    'givenname;lang-en'           => array('Rodney'),
    'sn;lang-en'                  => array('Ogasawara'),
    'cn;lang-en'                  => array('Rodney Ogasawara'),
    'title;lang-en'               => array('Sales, Director'),
);
*/
```

Zend_Loader

1. Loading Files and Classes Dynamically

The `Zend_Loader` class includes methods to help you load files dynamically.

Zend_Loader vs. require_once()

The `Zend_Loader` methods are best used if the filename you need to load is variable. For example, if it is based on a parameter from user input or method argument. If you are loading a file or a class whose name is constant, there is no benefit to using `Zend_Loader` over using traditional PHP functions such as `require_once()`.

1.1. Loading Files

The static method `Zend_Loader::loadFile()` loads a PHP file. The file loaded may contain any PHP code. The method is a wrapper for the PHP function `include()`. This method returns boolean `FALSE` on failure, for example if the specified file does not exist.

Example 467. Example of the loadFile() Method

```
Zend_Loader::loadFile($filename, $dirs=null, $once=false);
```

The `$filename` argument specifies the filename to load, which must not contain any path information. A security check is performed on `$filename`. The `$filename` may only contain alphanumeric characters, dashes ("-"), underscores ("_"), or periods ("."). No such restriction is placed on the `$dirs` argument.

The `$dirs` argument specifies which directories to search for the file in. If the value is `NULL`, only the `include_path` is searched; if the value is a string or an array, the directory or directories specified will be searched, followed by the `include_path`.

The `$once` argument is a boolean. If `TRUE`, `Zend_Loader::loadFile()` uses the PHP function `include_once()` for loading the file, otherwise the PHP function `include()` is used.

1.2. Loading Classes

The static method `Zend_Loader::loadClass($class, $dirs)` loads a PHP file and then checks for the existence of the class.

Example 468. Example of the loadClass() Method

```
Zend_Loader::loadClass('Container_Tree',
    array(
        '/home/production/mylib',
        '/home/production/myapp'
    )
);
```

The string specifying the class is converted to a relative path by substituting underscores with directory separators for your OS, and appending '.php'. In the example above, 'Container_Tree' becomes 'Container\\Tree.php' on Windows.

If `$dirs` is a string or an array, `Zend_Loader::loadClass()` searches the directories in the order supplied. The first matching file is loaded. If the file does not exist in the specified `$dirs`, then the `include_path` for the PHP environment is searched.

If the file is not found or the class does not exist after the load, `Zend_Loader::loadClass()` throws a `Zend_Exception`.

`Zend_Loader::loadFile()` is used for loading, so the class name may only contain alphanumeric characters and the hyphen ('-'), underscore ('_'), and period ('.').

Loading Classes from PHP Namespaces

Starting in version 1.10.0, Zend Framework now allows loading classes from PHP namespaces. This support follows the same guidelines and implementation as that found in the PHP Framework Interop Group PSR-0 reference implementation.

Under this guideline, the following rules apply:

- Each namespace separator is converted to a `DIRECTORY_SEPARATOR` when loading from the file system.
- Each "_" character in the *CLASS NAME* is converted to a `DIRECTORY_SEPARATOR`. The "_" character has no special meaning in the namespace.
- The fully-qualified namespace and class is suffixed with ".php" when loading from the file system.

As examples:

- `\Doctrine\Common\IsolatedClassLoader => /path/to/project/lib/vendor/Doctrine/Common/IsolatedClassLoader.php`
- `\namespace\package\Class_Name => /path/to/project/lib/vendor/namespace/package/Class/Name.php`
- `\namespace\package_name\Class_Name => /path/to/project/lib/vendor/namespace/package_name/Class/Name.php`

1.3. Testing if a File is Readable

The static method `Zend_Loader::isReadable($pathname)` returns `TRUE` if a file at the specified pathname exists and is readable, `FALSE` otherwise.

Example 469. Example of isReadable() method

```
if (Zend_Loader::isReadable($filename)) {
    // do something with $filename
}
```

The `$filename` argument specifies the filename to check. This may contain path information. This method is a wrapper for the PHP function `is_readable()`. The PHP function does not search the `include_path`, while `Zend_Loader::isReadable()` does.

1.4. Using the Autoloader

The `Zend_Loader` class contains a method you can register with the PHP SPL autoloader. `Zend_Loader::autoload()` is the callback method. As a convenience, `Zend_Loader` provides the `registerAutoload()` function to register its `autoload()` method. If the `spl_autoload` extension is not present in your PHP environment, then the `registerAutoload()` method throws a `Zend_Exception`.

Example 470. Example of registering the autoloader callback method

```
Zend_Loader::registerAutoload();
```

After registering the Zend Framework autoload callback, you can reference classes from Zend Framework without having to load them explicitly. The `autoload()` method uses `Zend_Loader::loadClass()` automatically when you reference a class.

If you have extended the `Zend_Loader` class, you can give an optional argument to `registerAutoload()`, to specify the class from which to register an `autoload()` method.

Example 471. Example of registering the autoload callback method from an extended class

Because of the semantics of static function references in PHP, you must implement code for both `loadClass()` and `autoload()`, and the `autoload()` must call `self::loadClass()`. If your `autoload()` method delegates to its parent to call `self::loadClass()`, then it calls the method of that name in the parent class, not the subclass.

```
class My_Loader extends Zend_Loader
{
    public static function loadClass($class, $dirs = null)
    {
        parent::loadClass($class, $dirs);
    }

    public static function autoload($class)
    {
        try {
            self::loadClass($class);
            return $class;
        } catch (Exception $e) {
            return false;
        }
    }
}

Zend_Loader::registerAutoload('My_Loader');
```

You can remove an autoload callback. The `registerAutoload()` has an optional second argument, which is `TRUE` by default. If this argument is `FALSE`, the autoload callback is unregistered from the SPL autoload stack.

2. The Autoloader

`Zend_Loader_Autoloader` introduces a comprehensive autoloading solution for Zend Framework. It has been designed with several goals in mind:

- Provide a true namespace autoloader. (Previous incarnations intercepted all userland namespaces.)
- Allow registering arbitrary callbacks as autoloaders, and manage them as a stack. (At the time of this writing, this overcomes some issues with `spl_autoload`, which does not allow re-registering a callback that utilizes an instance method.)
- Allow optimistic matching of namespaces to provide faster class resolution.

`Zend_Loader_Autoloader` implements a singleton, making it unversally accessible. This provides the ability to register additional autoloaders from anywhere in your code as necessary.

2.1. Using the Autoloader

The first time an instance of the autoloader is retrieved, it registers itself with `spl_autoload`. You retrieve an instance using the `getInstance()` method:

```
$autoloader = Zend_Loader_Autoloader::getInstance();
```

By default, the autoloader is configured to match the "Zend_" and "ZendX_" namespaces. If you have your own library code that uses your own namespace, you may register it with the autoloader using the `registerNamespace()` method. For instance, if your library code is prefixed with "My_", you could do so as follows:

```
$autoloader->registerNamespace('My_');
```

Namespace Prefixes

You'll note that the previous example uses "My_" and not "My". This is because `Zend_Loader_Autoloader` is intended as a general purpose autoloader, and does not make the assumption that a given class prefix namespace includes an underscore. If your class namespace *does* include one, you should include it when registering your namespace.

You can also register arbitrary autoloader callbacks, optionally with a specific namespace (or group of namespaces). `Zend_Loader_Autoloader` will attempt to match these first before using its internal autoloading mechanism.

As an example, you may want to utilize one or more eZcomponents components with your Zend Framework application. To use its autoloading capabilities, push it onto the autoloader stack using `pushAutoloader()`:

```
$autoloader->pushAutoloader(array('ezcBase', 'autoload'), 'ezc');
```

This tells the autoloader to use the eZcomponents autoloader for classes beginning with "ezc".

You can use the `unshiftAutoloader()` method to add the autoloader to the beginning of the autoloader chain.

By default, `Zend_Loader_Autoloader` does no error suppression when using its internal autoloader, which utilizes `Zend_Loader::loadClass()`. Most of the time, this is exactly what you want. However, there may be cases where you want to suppress them. You can do this using `suppressNotFoundWarnings()`:

```
$autoloader->suppressNotFoundWarnings(true);
```

Finally, there may be times when you want the autoloader to load any namespace. For instance, PEAR libraries do not share a common namespace, making specifying individual namespaces difficult when many PEAR components are in use. You can use the `setFallbackAutoloader()` method to have the autoloader act as a catch-all:

```
$autoloader->setFallbackAutoloader(true);
```

Loading Classes from PHP Namespaces

Starting in version 1.10.0, Zend Framework now allows loading classes from PHP namespaces. This support follows the same guidelines and implementation as that found in the PHP Framework Interop Group PSR-0 reference implementation.

Under this guideline, the following rules apply:

- Each namespace separator is converted to a `DIRECTORY_SEPARATOR` when loading from the file system.
- Each "_" character in the *CLASS NAME* is converted to a `DIRECTORY_SEPARATOR`. The "_" character has no special meaning in the namespace.
- The fully-qualified namespace and class is suffixed with ".php" when loading from the file system.

As examples:

- `\Doctrine\Common\IsolatedClassLoader => /path/to/project/lib/vendor/Doctrine/Common/IsolatedClassLoader.php`
- `\namespace\package\Class_Name => /path/to/project/lib/vendor/namespace/package/Class/Name.php`
- `\namespace\package_name\Class_Name => /path/to/project/lib/vendor/namespace/package_name/Class/Name.php`

2.2. Selecting a Zend Framework version

Typically, you will use the version of Zend Framework that the autoloader you instantiate came with. However, when developing a project, it's often useful to track specific versions, major or minor branches, or just the latest version. `Zend_Loader_Autoloader`, as of version 1.10, offers some features to help manage this task.

Imagine the following scenario:

- During *development*, you want to track the latest version of Zend Framework you have installed, so that you can ensure the application works when you upgrade between versions.

  When pushing to *Quality Assurance*, however, you need to have slightly more stability, so you want to use the latest installed revision of a specific minor version.

  Finally, when you push to *production*, you want to pin to a specific installed version, to ensure no breakage occurs if or when you add new versions of Zend Framework to you server.

The autoloader allows you to do this with the method `setZfPath()`. This method takes two arguments, a *path* to a set of Zend Framework installations, and a *version* to use. Once invoked, it prepends a path to the `include_path` pointing to the appropriate Zend Framework installation library.

The directory you specify as your *path* should have a tree such as the following:

```
ZendFramework/
|-- 1.9.2/
|   |-- library/
|-- ZendFramework-1.9.1-minimal/
|   |-- library/
|-- 1.8.4PL1/
|   |-- library/
|-- 1.8.4/
|   |-- library/
|-- ZendFramework-1.8.3/
|   |-- library/
|-- 1.7.8/
|   |-- library/
|-- 1.7.7/
|   |-- library/
|-- 1.7.6/
|   |-- library/
```

(where *path* points to the directory "ZendFramework" in the above example)

Note that each subdirectory should contain the directory `library`, which contains the actual Zend Framework library code. The individual subdirectory names may be version numbers, or simply be the untarred contents of a standard Zend Framework distribution tarball/zipfile.

Now, let's address the use cases. In the first use case, in *development*, we want to track the latest source install. We can do that by passing "latest" as the version:

```
$autoloader->setZfPath($path, 'latest');
```

In the example from above, this will map to the directory `ZendFramework/1.9.2/library/`; you can verify this by checking the return value of `getZfPath()`.

In the second situation, for *quality assurance*, let's say we want to pin to the 1.8 minor release, using the latest install you have for that release. You can do so as follows:

```
$autoloader->setZfPath($path, '1.8');
```

In this case, it will find the directory `ZendFramework/1.8.4PL1/library/`.

In the final case, for *production*, we'll pin to a specific version -- 1.7.7, since that was what was available when Quality Assurance tested prior to our release.

```
$autoloader->setZfPath($path, '1.7.7');
```

Predictably, it finds the directory `ZendFramework/1.7.7/library/`.

You can also specify these values in the configuration file you use with `Zend_Application`. To do so, you'd specify the following information:

```
[production]
autoloaderZfPath = "path/to/ZendFramework"
autoloaderZfVersion = "1.7.7"
```

```
[qa]
autoloaderZfVersion = "1.8"

[development]
autoloaderZfVersion = "latest"
```

Note the different environment sections, and the different version specified in each environment; these factors will allow `Zend_Application` to configure the autoloader appropriately.

Performance implications

For best performance, either do not use this feature, or specify a specific Zend Framework version (i.e., not "latest", a major revision such as "1", or a minor revision such as "1.8"). Otherwise, the autoloader will need to scan the provided path for directories matching the criteria -- a somewhat expensive operation to perform on each request.

2.3. The Autoloader Interface

Besides being able to specify arbitrary callbacks as autoloaders, Zend Framework also defines an interface autoloading classes may imlement, `Zend_Loader_Autoloader_Interface`:

```
interface Zend_Loader_Autoloader_Interface
{
    public function autoload($class);
}
```

When using this interface, you can simply pass a class instance to `Zend_Loader_Autoloader`'s `pushAutoloader()` and `unshiftAutoloader()` methods:

```
// Assume Foo_Autoloader implements Zend_Loader_Autoloader_Interface:
$foo = new Foo_Autoloader();

$autoloader->pushAutoloader($foo, 'Foo_');
```

2.4. Autoloader Reference

Below, please find a guide to the methods available in `Zend_Loader_Autoloader`.

Table 90. Zend_Loader_Autoloader Methods

Method	Return Value	Parameters	Description
`getInstance()`	`Zend_Loader_Autoloader`	N/A	Retrieve the `Zend_Loader_Autoloader` singleton instance. On first retrieval, it registers itself with `spl_autoload`. This method is static.
`resetInstance()`	`void`	N/A	Resets the state of the `Zend_Loader_Autoloader` singleton instance to it's original state, unregistering all

Method	Return Value	Parameters	Description
			autoloader callbacks and all registered namespaces.
`autoload($class)`	`string|FALSE`	• `$class`, *required*. A string class name to load.	Attempt to resolve a class name to a file and load it.
`setDefaultAutoloader(callback)`	`Zend_Loader_Autoloader`	• `$callback`, *required*.	Specify an alternate PHP callback to use for the default autoloader implementation.
`getDefaultAutoloader()`	`callback`	N/A	Retrieve the default autoloader implementation; by default, this is `Zend_Loader::loadClass()`.
`setAutoloaders(array $autoloaders)`	`Zend_Loader_Autoloader`	• `$autoloaders`, *required*.	Set a list of concrete autoloaders to use in the autoloader stack. Each item in the autoloaders array must be a PHP callback.
`getAutoloaders()`	Array	N/A	Retrieve the internal autoloader stack.
`getNamespaceAutoloaders($namespace)`	Array	• `$namespace`, *required*	Fetch all autoloaders that have registered to load a specific namespace.
`registerNamespace($namespace)`	`Zend_Loader_Autoloader`	• `$namespace`, *required*.	Register one or more namespaces with the default autoloader. If `$namespace` is a string, it registers that namespace; if it's an array of strings, registers each as a namespace.
`unregisterNamespace($namespace)`	`Zend_Loader_Autoloader`	• `$namespace`, *required*.	Unregister one or more namespaces from the default autoloader. If `$namespace` is a string, it unregisters that namespace; if it's an array of strings, unregisters each as a namespace.
`getRegisteredNamespaces()`	Array	N/A	Returns an array of all namespaces registered with the default autoloader.

Method	Return Value	Parameters	Description
`suppressNotFoundWarnings($flag = null)`	boolean|`Zend_Loader_Autoloader`	• `$flag`, *optional*.	Set or retrieve the value of the flag used to indicate whether the default autoloader implementation should suppress "file not found" warnings. If no arguments or a `NULL` value is passed, returns a boolean indicating the status of the flag; if a boolean is passed, the flag is set to that value and the autoloader instance is returned (to allow method chaining).
`setFallbackAutoloader($flag)`	`Zend_Loader_Autoloader`	• `$flag`, *required*.	Set the value of the flag used to indicate whether or not the default autoloader should be used as a fallback or catch-all autoloader for all namespaces.
`isFallbackAutoloader()`	boolean	N/A	Retrieve the value of the flag used to indicate whether or not the default autoloader should be used as a fallback or catch-all autoloader for all namespaces. By default, this is `FALSE`.
`getClassAutoloaders($class)`	array	• `$class`, *required*.	Get the list of namespaced autoloaders that could potentially match the provided class. If none match, all global (non-namespaced) autoloaders are returned.
`unshiftAutoloader($callback, $namespace = '')`	`Zend_Loader_Autoloader`	• `$callback`, *required*. A valid PHP callback • `$namespace`, *optional*. A string representing a class prefix namespace.	Add a concrete autoloader implementation to the beginning of the internal autoloader stack. If a namespace is provided, that

Method	Return Value	Parameters	Description
			namespace will be used to match optimistically; otherwise, the autoloader will be considered a global autoloader.
`pushAutoloader($callback, $namespace = '')`	`Zend_Loader_Autoloader`	• `$callback`, *required*. A valid PHP callback • `$namespace`, *optional*. A string representing a class prefix namespace.	Add a concrete autoloader implementation to the end of the internal autoloader stack. If a namespace is provided, that namespace will be used to match optimistically; otherwise, the autoloader will be considered a global autoloader.
`removeAutoloader($callback, $namespace = '')`	`Zend_Loader_Autoloader`	• `$callback`, *required*. A valid PHP callback • `$namespace`, *optional*. A string representing a class prefix namespace, or an array of namespace strings.	Remove a concrete autoloader implementation from the internal autoloader stack. If a namespace or namespaces are provided, the callback will be removed from that namespace or namespaces only.

3. Resource Autoloaders

Resource autoloaders are intended to manage namespaced library code that follow Zend Framework coding standard guidelines, but which do not have a 1:1 mapping between the class name and the directory structure. Their primary purpose is to facilitate autoloading application resource code, such as application-specific models, forms, and ACLs.

Resource autoloaders register with the autoloader on instantiation, with the namespace to which they are associated. This allows you to easily namespace code in specific directories, and still reap the benefits of autoloading.

3.1. Resource autoloader usage

Let's consider the following directory structure:

```
path/to/some/directory/
    acls/
        Site.php
    forms/
        Login.php
```

```
    models/
        User.php
```

Within this directory, all code is prefixed with the namespace "My_". Within the "acls" subdirectory, the component prefix "Acl_" is added, giving a final class name of "My_Acl_Site". Similarly, the "forms" subdirectory maps to "Form_", giving "My_Form_Login". The "models" subdirectory has no component namespace, giving "My_User".

You can use a resource autoloader to autoload these classes. To instantiate the resource autoloader, you are required to pass at the minimum the base path and namespace for the resources it will be responsible for:

```
$resourceLoader = new Zend_Loader_Autoloader_Resource(array(
    'basePath'  => 'path/to/some/directory',
    'namespace' => 'My',
));
```

Base namespace

In `Zend_Loader_Autoloader`, you are expected to provide the trailing underscore ("_") in your namespace if your autoloader will use it to match the namespace. `Zend_Loader_Autoloader_Resource` makes the assumption that all code you are autoloading will use an underscore separator between namespaces, components, and classes. As a result, you do not need to use the trailing underscore when registering a resource autoloader.

Now that we have setup the base resource autoloader, we can add some components to it to autoload. This is done using the `addResourceType()` method, which accepts three arguments: a resource "type", used internally as a reference name; the subdirectory path underneath the base path in which these resources live; and the component namespace to append to the base namespace. As an example, let's add each of our resource types.

```
$resourceLoader->addResourceType('acl', 'acls/', 'Acl')
               ->addResourceType('form', 'forms/', 'Form')
               ->addResourceType('model', 'models/');
```

Alternately, you could pass these as an array to `addResourceTypes()`; the following is equivalent to the above:

```
$resourceLoader->addResourceTypes(array(
    'acl' => array(
        'path'      => 'acls/',
        'namespace' => 'Acl',
    ),
    'form' => array(
        'path'      => 'forms/',
        'namespace' => 'Form',
    ),
    'model' => array(
        'path'      => 'models/',
    ),
));
```

Finally, you can specify all of this when instantiating the object, by simply specifying a "resourceTypes" key in the options passed and a structure like that above:

```
$resourceLoader = new Zend_Loader_Autoloader_Resource(array(
    'basePath'      => 'path/to/some/directory',
    'namespace'     => 'My',
    'resourceTypes' => array(
        'acl' => array(
            'path'      => 'acls/',
            'namespace' => 'Acl',
        ),
        'form' => array(
            'path'      => 'forms/',
            'namespace' => 'Form',
        ),
        'model' => array(
            'path'      => 'models/',
        ),
    ),
));
```

3.2. The Module Resource Autoloader

Zend Framework ships with a concrete implementation of `Zend_Loader_Autoloader_Resource` that contains resource type mappings that cover the default recommended directory structure for Zend Framework MVC applications. This loader, `Zend_Application_Module_Autoloader`, comes with the following mappings:

```
forms/       => Form
models/      => Model
    DbTable/ => Model_DbTable
    mappers/ => Model_Mapper
plugins/     => Plugin
services/    => Service
views/
    helpers  => View_Helper
    filters  => View_Filter
```

As an example, if you have a module with the prefix of "Blog_", and attempted to instantiate the class "Blog_Form_Entry", it would look in the resource directory's "forms/" subdirectory for a file named "Entry.php".

When using module bootstraps with `Zend_Application`, an instance of `Zend_Application_Module_Autoloader` will be created by default for each discrete module, allowing you to autoload module resources.

3.3. Using Resource Autoloaders as Object Factories

3.4. Resource Autoloader Reference

4. Loading Plugins

A number of Zend Framework components are pluggable, and allow loading of dynamic functionality by specifying a class prefix and path to class files that are not necessarily on the `include_path` or do not necessarily follow traditional naming conventions. `Zend_Loader_PluginLoader` provides common functionality for this process.

The basic usage of the `PluginLoader` follows Zend Framework naming conventions of one class per file, using the underscore as a directory separator when resolving paths. It allows passing an optional class prefix to prepend when determining if a particular plugin class is loaded. Additionally, paths are searched in LIFO order. Due to the LIFO search and the class prefixes, this allows you to define namespaces for your plugins, and thus override plugins from paths registered earlier.

4.1. Basic Use Case

First, let's assume the following directory structure and class files, and that the top level directory and library directory are on the include_path:

```
application/
    modules/
        foo/
            views/
                helpers/
                    FormLabel.php
                    FormSubmit.php
        bar/
            views/
                helpers/
                    FormSubmit.php
library/
    Zend/
        View/
            Helper/
                FormLabel.php
                FormSubmit.php
                FormText.php
```

Now, let's create a plugin loader to address the various view helper repositories available:

```
$loader = new Zend_Loader_PluginLoader();
$loader->addPrefixPath('Zend_View_Helper', 'Zend/View/Helper/')
       ->addPrefixPath('Foo_View_Helper',
                       'application/modules/foo/views/helpers')
       ->addPrefixPath('Bar_View_Helper',
                       'application/modules/bar/views/helpers');
```

We can then load a given view helper using just the portion of the class name following the prefixes as defined when adding the paths:

```
// load 'FormText' helper:
$formTextClass = $loader->load('FormText'); // 'Zend_View_Helper_FormText';

// load 'FormLabel' helper:
$formLabelClass = $loader->load('FormLabel'); // 'Foo_View_Helper_FormLabel'

// load 'FormSubmit' helper:
$formSubmitClass = $loader->load('FormSubmit'); // 'Bar_View_Helper_FormSubmit'
```

Once the class is loaded, we can now instantiate it.

In some cases, you may use the same prefix for multiple paths. `Zend_Loader_PluginLoader` actually registers an array of paths for each

given prefix; the last one registered will be the first one checked. This is particularly useful if you are utilizing incubator components.

Paths may be defined at instantiation

You may optionally provide an array of prefix / path pairs (or prefix / paths -- plural paths are allowed) as a parameter to the constructor:

```
$loader = new Zend_Loader_PluginLoader(array(
    'Zend_View_Helper' => 'Zend/View/Helper/',
    'Foo_View_Helper'  => 'application/modules/foo/views/helpers',
    'Bar_View_Helper'  => 'application/modules/bar/views/helpers'
));
```

`Zend_Loader_PluginLoader` also optionally allows you to share plugins across plugin-aware objects, without needing to utilize a singleton instance. It does so via a static registry. Indicate the registry name at instantiation as the second parameter to the constructor:

```
// Store plugins in static registry 'foobar':
$loader = new Zend_Loader_PluginLoader(array(), 'foobar');
```

Other components that instantiate the `PluginLoader` using the same registry name will then have access to already loaded paths and plugins.

4.2. Manipulating Plugin Paths

The example in the previous section shows how to add paths to a plugin loader. What if you want to determine the paths already loaded, or remove one or more?

- `getPaths($prefix = null)` returns all paths as prefix / path pairs if no `$prefix` is provided, or just the paths registered for a given prefix if a `$prefix` is present.
- `clearPaths($prefix = null)` will clear all registered paths by default, or only those associated with a given prefix, if the `$prefix` is provided and present in the stack.
- `removePrefixPath($prefix, $path = null)` allows you to selectively remove a specific path associated with a given prefix. If no `$path` is provided, all paths for that prefix are removed. If a `$path` is provided and exists for that prefix, only that path will be removed.

4.3. Testing for Plugins and Retrieving Class Names

Sometimes you simply want to determine if a plugin class has been loaded before you perform an action. `isLoaded()` takes a plugin name, and returns the status.

Another common use case for the `PluginLoader` is to determine fully qualified plugin class names of loaded classes; `getClassName()` provides this functionality. Typically, this would be used in conjunction with `isLoaded()`:

```
if ($loader->isLoaded('Adapter')) {
    $class   = $loader->getClassName('Adapter');
    $adapter = call_user_func(array($class, 'getInstance'));
}
```

4.4. Getting Better Performance for Plugins

Plugin loading can be an expensive operation. At its heart, it needs to loop through each prefix, then each path on the prefix, until it finds a file that matches -- and which defines the class expected. In cases where the file exists but does not define the class, an error will be added to the PHP error stack, which is also an expensive operation. The question then turns to: how can you keep the flexibility of plugins and also address performance?

`Zend_Loader_PluginLoader` offers an opt-in feature for just this situation, a class file include cache. When enabled, it will create a file that contains all successful includes which you can then call from your bootstrap. Using this strategy, you can greatly improve the performance of your production servers.

Example 472. Using the PluginLoader class file include cache

To use the class file include cache, simply drop the following code into your bootstrap:

```
$classFileIncCache = APPLICATION_PATH . '/../data/pluginLoaderCache.php';
if (file_exists($classFileIncCache)) {
    include_once $classFileIncCache;
}
Zend_Loader_PluginLoader::setIncludeFileCache($classFileIncCache);
```

Obviously, the path and filename will vary based on your needs. This code should come as early as possible, to ensure that plugin-based components can make use of it.

During development, you may wish to disable the cache. One method for doing so is to use a configuration key for determining whether or not the plugin loader should cache.

```
$classFileIncCache = APPLICATION_PATH . '/../data/pluginLoaderCache.php';
if (file_exists($classFileIncCache)) {
    include_once $classFileIncCache;
}
if ($config->enablePluginLoaderCache) {
    Zend_Loader_PluginLoader::setIncludeFileCache($classFileIncCache);
}
```

This technique allows you to keep your modifications to your configuration file rather than code.

Zend_Locale

1. Introduction

`Zend_Locale` is the Frameworks answer to the question, "How can the same application be used around the whole world?" Most people will say, "That's easy. Let's translate all our output to several languages." However, using simple translation tables to map phrases from one language to another is not sufficient. Different regions will have different conventions for first names, surnames, salutory titles, formatting of numbers, dates, times, currencies, etc.

We need Localization and complementary Internationalization . Both are often abbreviated to `L10n` and `I18n`. Internationalization refers more to support for use of systems, regardless of special needs unique to groups of users related by language, region, number format conventions, financial conventions, time and date conventions, etc. Localization involves adding explicit support to systems for special needs of these unique groups, such as language translation, and support for local customs or conventions for communicating plurals, dates, times, currencies, names, symbols, sorting and ordering, etc. `L10n` and `I18n` compliment each other. Zend Framework provides support for these through a combination of components, including `Zend_Locale`, `Zend_Date`, `Zend_Measure`, `Zend_Translate`, `Zend_Currency`, and `Zend_TimeSync`.

Zend_Locale and setLocale()

PHP's documentation states that `setlocale()` is not threadsave because it is maintained per process and not per thread. This means that, in multithreaded environments, you can have the problem that the locale changes while the script never has changed the locale itself. This can lead to unexpected behaviour when you use `setlocale()` in your scripts.

When you are using `Zend_Locale` you will not have this limitations, because `Zend_Locale` is not related to or coupled with PHP's `setlocale()`.

1.1. What is Localization

Localization means that an application (or homepage) can be used from different users which speak different languages. But as you already have expected Localization means more than only translating strings. It includes

- `Zend_Locale` - Backend support of locales available for localization support within other Zend Framework components.
- `Zend_Translate` - Translating of strings.
- `Zend_Date` - Localization of dates, times.
- `Zend_Calendar` - Localization of calendars (support for non-Gregorian calendar systems)
- `Zend_Currency` - Localization of currencies.
- `Zend_Locale_Format` - Parsing and generating localized numbers.
- `Zend_Locale_Data` - Retrieve localized standard strings as country names, language names and more from the CLDR .

- TODO - Localization of collations

1.2. What is a Locale?

Each computer user makes use of Locales, even when they don't know it. Applications lacking localization support, normally have implicit support for one particular locale (the locale of the author). When a class or function makes use of localization, we say it is `locale-aware`. How does the code know which localization the user is expecting?

A locale string or object identifying a supported locale gives `Zend_Locale` and its subclasses access to information about the language and region expected by the user. Correct formatting, normalization, and conversions are made based on this information.

1.3. How are Locales Represented?

Locale identifiers consist of information about the user's language and preferred/primary geographic region (e.g. state or province of home or workplace). The locale identifier strings used in Zend Framework are internationally defined standard abbreviations of language and region, written as `language_REGION`. Both the language and region parts are abbreviated to alphabetic, ASCII characters.

Be aware that there exist not only locales with 2 characters as most people think. Also there are languages and regions which are not only abbreviated with 2 characters. Therefor you should NOT strip the region and language yourself, but use `Zend_Locale` when you want to strip language or region from a locale string. Otherwise you could have unexpected behaviour within your code when you do this yourself.

A user from USA would expect the language `English` and the region `USA`, yielding the locale identifier "en_US". A user in Germany would expect the language `German` and the region `Germany`, yielding the locale identifier "de_DE". See the list of pre-defined locale and region combinations, if you need to select a specific locale within Zend Framework.

Example 473. Choosing a specific locale

```
$locale = new Zend_Locale('de_DE'); // German language _ Germany
```

A German user in America might expect the language `German` and the region `USA`, but these non-standard mixes are not supported directly as recognized "locales". Instead, if an invalid combination is used, then it will automatically be truncated by dropping the region code. For example, "de_IS" would be truncated to "de", and "xh_RU" would be truncated to "xh", because neither of these combinations are valid. Additionally, if the base language code is not supported (e.g. "zz_US") or does not exist, then a default "root" locale will be used. The "root" locale has default definitions for internationally recognized representations of dates, times, numbers, currencies, etc. The truncation process depends on the requested information, since some combinations of language and region might be valid for one type of data (e.g. dates), but not for another (e.g. currency format).

Beware of historical changes, as Zend Framework components do not know about or attempt to track the numerous timezone changes made over many years by many regions. For example, we can see a historical list showing dozens of changes made by governments to when and if a particular region observes Daylight Savings Time, and even which timezone a particular

geographic area belongs. Thus, when performing date math, the math performed by Zend Framework components will not adjust for these changes, but instead will give the correct time for the timezone using current, modern rules for DST and timezone assignment for geographic regions.

1.4. Selecting the Right Locale

For most situations, new Zend_Locale() will automatically select the correct locale, with preference given to information provided by the user's web browser. However, if new Zend_Locale(Zend_Locale::ENVIRONMENT) is used, then preference will be given to using the host server's environment configuration, as described below.

Example 474. Automatically selecting a locale

```
$locale  = new Zend_Locale();

// default behavior, same as above
$locale1 = new Zend_Locale(Zend_Locale::BROWSER);

// prefer settings on host server
$locale2 = new Zend_Locale(Zend_Locale::ENVIRONMENT);

// perfer framework app default settings
$locale3 = new Zend_Locale(Zend_Locale::FRAMEWORK);
```

The search algorithm used by Zend_Locale for automatic selection of a locale uses three sources of information:

1. const Zend_Locale::BROWSER - The user's Web browser provides information with each request, which is published by PHP in the global variable HTTP_ACCEPT_LANGUAGE. If no matching locale can be found, then preference is given to ENVIRONMENT and lastly FRAMEWORK.

2. const Zend_Locale::ENVIRONMENT - PHP publishes the host server's locale via the PHP internal function setlocale(). If no matching locale can be found, then preference is given to FRAMEWORK and lastly BROWSER.

3. const Zend_Locale::FRAMEWORK - When Zend Framework has a standardized way of specifying component defaults (planned, but not yet available), then using this constant during instantiation will give preference to choosing a locale based on these defaults. If no matching locale can be found, then preference is given to ENVIRONMENT and lastly BROWSER.

1.5. Usage of automatic Locales

Zend_Locale provides three additionally locales. These locales do not belong to any language or region. They are "automatic" locales which means that they have the same effect as the method getDefault() but without the negative effects like creating an instance. These "automatic" locales can be used anywhere, where also a standard locale and also the definition of a locale, its string representation, can be used. This offers simplicity for situations like working with locales which are provided by a browser.

There are three locales which have a slightly different behaviour:

1. 'browser' - Zend_Locale should work with the information which is provided by the user's Web browser. It is published by PHP in the global variable HTTP_ACCEPT_LANGUAGE.

If a user provides more than one locale within his browser, `Zend_Locale` will use the first found locale. If the user does not provide a locale or the script is being called from the command line the automatic locale `'environment'` will automatically be used and returned.

2. `'environment'` - `Zend_Locale` should work with the information which is provided by the host server. It is published by PHP via the internal function `setlocale()`.

   If a environment provides more than one locale, `Zend_Locale` will use the first found locale. If the host does not provide a locale the automatic locale `'browser'` will automatically be used and returned.

3. `'auto'` - `Zend_Locale` should automatically detect any locale which can be worked with. It will first search for a users locale and then, if not successful, search for the host locale.

   If no locale can be detected, it will throw an exception and tell you that the automatic detection has been failed.

Example 475. Using automatic locales

```
// without automatic detection
//$locale = new Zend_Locale(Zend_Locale::BROWSER);
//$date = new Zend_Date($locale);

// with automatic detection
$date = new Zend_Date('auto');
```

1.6. Using a default Locale

In some environments it is not possible to detect a locale automatically. You can expect this behaviour when you get an request from command line or the requesting browser has no language tag set and additionally your server has the default locale 'C' set or another proprietary locale.

In such cases `Zend_Locale` will normally throw an exception with a message that the automatic detection of any locale was not successful. You have two options to handle such a situation. Either through setting a new locale per hand, or defining a default locale.

Example 476. Handling locale exceptions

```
// within the bootstrap file
try {
    $locale = new Zend_Locale('auto');
} catch (Zend_Locale_Exception $e) {
    $locale = new Zend_Locale('de');
}

// within your model/controller
$date = new Zend_Date($locale);
```

But this has one big negative effect. You will have to set your locale object within every class using `Zend_Locale`. This could become very unhandy if you are using multiple classes.

Since Zend Framework Release 1.5 there is a much better way to handle this. You can set a default locale which the static `setDefault()` method. Of course, every unknown or not full

qualified locale will also throw an exception. `setDefault()` should be the first call before you initiate any class using `Zend_Locale`. See the following example for details:

Example 477. Setting a default locale

```
// within the bootstrap file
Zend_Locale::setDefault('de');

// within your model/controller
$date = new Zend_Date();
```

In the case that no locale can be detected, automatically the locale *de* will be used. Otherwise, the detected locale will be used.

1.7. ZF Locale-Aware Classes

In the Zend Framework, locale-aware classes rely on `Zend_Locale` to automatically select a locale, as explained above. For example, in a Zend Framework web application, constructing a date using `Zend_Date` without specifying a locale results in an object with a locale based on information provided by the current user's web browser.

Example 478. Dates default to correct locale of web users

```
$date = new Zend_Date('2006',Zend_Date::YEAR);
```

To override this default behavior, and force locale-aware Zend Framework components to use specific locales, regardless of the origin of your website visitors, explicitly specify a locale as the third argument to the constructor.

Example 479. Overriding default locale selection

```
$usLocale = new Zend_Locale('en_US');
$date = new Zend_Date('2006', Zend_Date::YEAR, $usLocale);
$temp = new Zend_Measure_Temperature('100,10',
                                     Zend_Measure::TEMPERATURE,
                                     $usLocale);
```

If you know many objects should all use the same default locale, explicitly specify the default locale to avoid the overhead of each object determining the default locale.

Example 480. Performance optimization when using a default locale

```
$locale = new Zend_Locale();
$date = new Zend_Date('2006', Zend_Date::YEAR, $locale);
$temp = new Zend_Measure_Temperature('100,10',
                                     Zend_Measure::TEMPERATURE,
                                     $locale);
```

1.8. Application wide locale

Zend Framework allows the usage of an application wide locale. You simply set an instance of `Zend_Locale` to the registry with the key 'Zend_Locale'. Then this instance will be used within all locale aware classes of Zend Framework. This way you set one locale within your registry and then you can forget about setting it again. It will automatically be used in all other classes. See the below example for the right usage:

Example 481. Usage of an application wide locale

```
// within your bootstrap
$locale = new Zend_Locale('de_AT');
Zend_Registry::set('Zend_Locale', $locale);

// within your model or controller
$date = new Zend_Date();
// print $date->getLocale();
echo $date->getDate();
```

1.9. Zend_Locale_Format::setOptions(array $options)

The 'precision' option of a value is used to truncate or stretch extra digits. A value of '-1' disables modification of the number of digits in the fractional part of the value. The 'locale' option helps when parsing numbers and dates using separators and month names. The date format 'format_type' option selects between CLDR/ISO date format specifier tokens and PHP's date() tokens. The 'fix_date' option enables or disables heuristics that attempt to correct invalid dates. The 'number_format' option specifies a default number format for use with `toNumber()` (see Section 3.2, “Number localization”).

The 'date_format' option can be used to specify a default date format string, but beware of using getDate(), checkdateFormat() and getTime() after using setOptions() with a 'date_format'. To use these four methods with the default date format for a locale, use array('date_format' => null, 'locale' => $locale) for their options.

Example 482. Dates default to correct locale of web users

```
Zend_Locale_Format::setOptions(array('locale' => 'en_US',
                                     'fix_date' => true,
                                     'format_type' => 'php'));
```

For working with the standard definitions of a locale the option `Zend_Locale_Format::STANDARD` can be used. Setting the option `Zend_Locale_Format::STANDARD` for `date_format` uses the standard definitions from the actual set locale. Setting it for number_format uses the standard number format for this locale. And setting it for locale uses the standard locale for this environment or browser.

Example 483. Using STANDARD definitions for setOptions()

```
Zend_Locale_Format::setOptions(array('locale' => 'en_US',
                                     'date_format' => 'dd.MMMM.YYYY'));
// overriding the global set date format
$date = Zend_Locale_Format::getDate('2007-04-20',
                                    array('date_format' =>
                                            Zend_Locale_Format::STANDARD);

// global setting of the standard locale
Zend_Locale_Format::setOptions(array('locale' => Zend_Locale_Format::STANDARD,
                                     'date_format' => 'dd.MMMM.YYYY'));
```

1.10. Speed up Zend_Locale and its subclasses

`Zend_Locale` and its subclasses can be speed up by the usage of `Zend_Cache`. Use the static method `Zend_Locale::setCache($cache)` if you are using `Zend_Locale`.

`Zend_Locale_Format` can be speed up the using the option `cache` within `Zend_Locale_Format::setOptions(array('cache' => $adapter));`. If you are using both classes you should only set the cache for `Zend_Locale`, otherwise the last set cache will overwrite the previous set cache. For convenience there are also the static methods `getCache()`, `hasCache()`, `clearCache()` and `removeCache()`.

When no cache is set, then `Zend_Locale` will automatically set a cache itself. Sometimes it is wished to prevent that a cache is set, even if this degrades performance. In this case the static `disableCache(true)` method should be used. It does not only disable the actual set cache, without erasing it, but also prevents that a cache is automatically generated when no cache is set.

2. Using Zend_Locale

`Zend_Locale` also provides localized information about locales for each locale, including localized names for other locales, days of the week, month names, etc.

2.1. Copying, Cloning, and Serializing Locale Objects

Use object cloning to duplicate a locale object exactly and efficiently. Most locale-aware methods also accept string representations of locales, such as the result of `$locale->toString()`.

Example 484. clone

```
$locale = new Zend_Locale('ar');

// Save the $locale object as a serialization
$serializedLocale = $locale->serialize();
// re-create the original object
$localeObject = unserialize($serializedLocale);

// Obtain a string identification of the locale
$stringLocale = $locale->toString();

// Make a cloned copy of the $local object
$copiedLocale = clone $locale;

print "copied: ", $copiedLocale->toString();

// PHP automatically calls toString() via __toString()
print "copied: ", $copiedLocale;
```

2.2. Equality

`Zend_Locale` also provides a convenience function to compare two locales. All locale-aware classes should provide a similar equality check.

Example 485. Check for equal locales

```
$locale = new Zend_Locale();
$mylocale = new Zend_Locale('en_US');

// Check if locales are equal
if ($locale->equals($mylocale)) {
    print "Locales are equal";
}
```

2.3. Default locales

The method `getDefault()` returns an array of relevant locales using information from the user's web browser (if available), information from the environment of the host server, and Zend Framework settings. As with the constructor for `Zend_Locale`, the first parameter selects a preference of which information to consider (`BROWSER`, `ENVIRONMENT`, or `FRAMEWORK`) first. The second parameter toggles between returning all matching locales or only the first/best match. Locale-aware components normally use only the first locale. A quality rating is included, when available.

Example 486. Get default locales

```
$locale = new Zend_Locale();

// Return all default locales
$found = $locale->getDefault();
print_r($found);

// Return only browser locales
$found2 = $locale->getDefault(Zend_Locale::BROWSER,TRUE);
print_r($found2);
```

To obtain only the default locales relevant to the `BROWSER`, `ENVIRONMENT`, or `FRAMEWORK` , use the corresponding method:

- `getEnvironment()`
- `getBrowser()`
- `getLocale()`

2.4. Set a new locale

A new locale can be set with the function `setLocale()`. This function takes a locale string as parameter. If no locale is given, a locale is automatically selected. Since `Zend_Locale` objects are "light", this method exists primarily to cause side-effects for code that have references to the existing instance object.

Example 487. setLocale

```
$locale = new Zend_Locale();

// Actual locale
print $locale->toString();

// new locale
$locale->setLocale('aa_DJ');
print $locale->toString();
```

2.5. Getting the language and region

Use `getLanguage()` to obtain a string containing the two character language code from the string locale identifier. Use `getRegion()` to obtain a string containing the two character region code from the string locale identifier.

Example 488. getLanguage and getRegion

```
$locale = new Zend_Locale();

// if locale is 'de_AT' then 'de' will be returned as language
print $locale->getLanguage();

// if locale is 'de_AT' then 'AT' will be returned as region
print $locale->getRegion();
```

2.6. Obtaining localized strings

`getTranslationList()` gives you access to localized informations of several types. These information are useful if you want to display localized data to a customer without the need of translating it. They are already available for your usage.

The requested list of information is always returned as named array. If you want to give more than one value to a explicit type where you wish to receive values from, you have to give an array instead of multiple values.

Example 489. getTranslationList

```
$list = Zend_Locale::getTranslationList('language', 'de_AT');

print_r ($list);
// example key -> value pairs...
// [de] -> Deutsch
// [en] -> Englisch

// use one of the returned key as value for the getTranslation() method
// of another language
print Zend_Locale::getTranslation('de', 'language', 'zh');
// returns the translation for the language 'de' in chinese
```

You can receive this informations for all languages. But not all of the informations are completely available for all languages. Some of these types are also available through an own function for simplicity. See this list for detailed informations.

Table 91. Details for getTranslationList($type = null, $locale = null, $value = null)

Type	Description
Language	Returns a localized list of all languages. The language part of the locale is returned as key and the translation as value
Script	Returns a localized list of all scripts. The script is returned as key and the translation as value
Territory	Returns a localized list of all territories. This contains countries, continents and territories. To get only territories and continents use '1' as value. To get only countries use '2' as value. The country part of the locale is used as key where applicable. In the other case the official ISO code for this territory is used. The translated territory is returned as value. When you omit the value you will get a list with both.

Type	Description
Variant	Returns a localized list of known variants of scripts. The variant is returned as key and the translation as value
Key	Returns a localized list of known keys. This keys are generic values used in translation. These are normally calendar, collation and currency. The key is returned as array key and the translation as value
Type	Returns a localized list of known types of keys. These are variants of types of calendar representations and types of collations. When you use 'collation' as value you will get all types of collations returned. When you use 'calendar' as value you will get all types of calendars returned. When you omit the value you will get a list all both returned. The type is used as key and the translation as value
Layout	Returns a list of rules which describes how to format special text parts
Characters	Returns a list of allowed characters within this locale
Delimiters	Returns a list of allowed quoting characters for this locale
Measurement	Returns a list of known measurement values. This list is depreciated
Months	Returns a list of all month representations within this locale. There are several different representations which are all returned as sub array. If you omit the value you will get a list of all months from the 'gregorian' calendar returned. You can give any known calendar as value to get a list of months from this calendar returned. Use Zend_Date for simplicity
Month	Returns a localized list of all month names for this locale. If you omit the value you will get the normally used gregorian full name of the months where each month number is used as key and the translated month is returned as value. You can get the months for different calendars and formats if you give an array as value. The first array entry has to be the calendar, the second the used context and the third the width to return. Use Zend_Date for simplicity
Days	Returns a list of all day representations within this locale. There are several different representations which are all returned as sub array. If you omit the value you will get a list of all days from the 'gregorian' calendar returned.

Type	Description
	You can give any known calendar as value to get a list of days from this calendar returned. Use Zend_Date for simplicity
Day	Returns a localized list of all day names for this locale. If you omit the value you will get the normally used gregorian full name of the days where the english day abbreviation is used as key and the translated day is returned as value. You can get the days for different calendars and formats if you give an array as value. The first array entry has to be the calendar, the second the used context and the third the width to return. Use Zend_Date for simplicity
Week	Returns a list of values used for proper week calculations within a locale. Use Zend_Date for simplicity
Quarters	Returns a list of all quarter representations within this locale. There are several different representations which are all returned as sub array. If you omit the value you will get a list of all quarters from the 'gregorian' calendar returned. You can give any known calendar as value to get a list of quarters from this calendar returned
Quarter	Returns a localized list of all quarter names for this locale. If you omit the value you will get the normally used gregorian full name of the quarters where each quarter number is used as key and the translated quarter is returned as value. You can get the quarters for different calendars and formats if you give an array as value. The first array entry has to be the calendar, the second the used context and the third the width to return
Eras	Returns a list of all era representations within this locale. If you omit the value you will get a list of all eras from the 'gregorian' calendar returned. You can give any known calendar as value to get a list of eras from this calendar returned
Era	Returns a localized list of all era names for this locale. If you omit the value you will get the normally used gregorian full name of the eras where each era number is used as key and the translated era is returned as value. You can get the eras for different calendars and formats if you give an array as value. The first array entry has to be the calendar and the second the width to return

Type	Description
Date	Returns a localized list of all date formats for this locale. The name of the dateformat is used as key and the format itself as value.If you omit the value you will get the date formats for the gregorian calendar returned. You can get the date formats for different calendars if you give the wished calendar as string. Use Zend_Date for simplicity
Time	Returns a localized list of all time formats for this locale. The name of the timeformat is used as key and the format itself as value. If you omit the value you will get the time formats for the gregorian calendar returned. You can get the time formats for different calendars if you give the wished calendar as string. Use Zend_Date for simplicity
DateTime	Returns a localized list of all known date-time formats for this locale. The name of the date-time format is used as key and the format itself as value. If you omit the value you will get the date-time formats for the gregorian calendar returned. You can get the date-time formats for different calendars if you give the wished calendar as string. Use Zend_Date for simplicity
DateItem	Returns a list of default formats for given date or time items
DateInterval	Returns a list of date or time formats which are used when you want to display intervals. The list is a multidimentional array where the first dimension is the interval format, and the second dimension is the token with the greatest difference.
Field	Returns a localized list of date fields which can be used to display calendars or date strings like 'month' or 'year' in a wished language. If you omit the value you will get this list for the gregorian calendar returned. You can get the list for different calendars if you give the wished calendar as string
Relative	Returns a localized list of relative dates which can be used to display textual relative dates like 'yesterday' or 'tomorrow' in a wished language. If you omit the value you will get this list for the gregorian calendar returned. You can get the list for different calendars if you give the wished calendar as string
Symbols	Returns a localized list of characters used for number representations

Type	Description
NameToCurrency	Returns a localized list of names for currencies. The currency is used as key and the translated name as value. Use Zend_Currency for simplicity
CurrencyToName	Returns a list of currencies for localized names. The translated name is used as key and the currency as value. Use Zend_Currency for simplicity
CurrencySymbol	Returns a list of known localized currency symbols for currencies. The currency is used as key and the symbol as value. Use Zend_Currency for simplicity
Question	Returns a list of localized strings for acceptance ('yes') and negotation ('no'). Use Zend_Locale's getQuestion method for simplicity
CurrencyFraction	Returns a list of fractions for currency values. The currency is used as key and the fraction as integer value. Use Zend_Currency for simplicity
CurrencyRounding	Returns a list of how to round which currency. The currency is used as key and the rounding as integer value. Use Zend_Currency for simplicity
CurrencyToRegion	Returns a list of currencies which are known to be used within a region. The `ISO3166` value ('region') is used as array key and the `ISO4217` value ('currency') as array value. Use Zend_Currency for simplicity
RegionToCurrency	Returns a list of regions where a currency is used . The `ISO4217` value ('currency') is used as array key and the `ISO3166` value ('region') as array value. When a currency is used in several regions these regions are separated with a whitespace. Use Zend_Currency for simplicity
RegionToTerritory	Returns a list of territories with the countries or sub territories which are included within that territory. The ISO territory code ('territory') is used as array key and the `ISO3166` value ('region') as array value. When a territory contains several regions these regions are separated with a whitespace
TerritoryToRegion	Returns a list of regions and the territories where these regions are located. The `ISO3166` code ('region') is used as array key and the ISO territory code ('territory') as array value. When a region is located in several territories these territories are separated with a whitespace

Type	Description
ScriptToLanguage	Returns a list of scripts which are used within a language. The language code is used as array key and the script code as array value. When a language contains several scripts these scripts are separated with a whitespace
LanguageToScript	Returns a list of languages which are using a script. The script code is used as array key and the language code as array value. When a script is used in several languages these languages are separated with a whitespace
TerritoryToLanguage	Returns a list of countries which are using a language. The country code is used as array key and the language code as array value. When a language is used in several countries these countries are separated with a whitespace
LanguageToTerritory	Returns a list of countries and the languages spoken within these countries. The country code is used as array key and the language code as array value. When a territory is using several languages these languages are separated with a whitespace
TimezoneToWindows	Returns a list of windows timezones and the related ISO timezone. The windows timezone is used as array key and the ISO timezone as array value
WindowsToTimezone	Returns a list of ISO timezones and the related windows timezone. The ISO timezone is used as array key and the windows timezone as array value
TerritoryToTimezone	Returns a list of regions or territories and the related ISO timezone. The ISO timezone is used as array key and the territory code as array value
TimezoneToTerritory	Returns a list of timezones and the related region or territory code. The region or territory code is used as array key and the ISO timezone as array value
CityToTimezone	Returns a localized list of cities which can be used as translation for a related timezone. Not for all timezones is a translation available, but for a user is the real city written in his languages more accurate than the ISO name of this timezone. The ISO timezone is used as array key and the translated city as array value
TimezoneToCity	Returns a list of timezones for localized city names. The localized city is used as array key and the ISO timezone name as array value

Type	Description
PhoneToTerritory	Returns a list of phone codes which are known to be used within a territory. The territory (region) is used as array key and the telephone code as array value
TerritoryToPhone	Returns a list of territories where a phone is used . The phone code is used as array key and the territory (region) as array value. When a phone code is used in several territories these territories are separated with a whitespace
NumericToTerritory	Returns a list of 3 digit number codes for territories. The territory (region) is used as array key and the 3 digit number code as array value
TerritoryToNumeric	Returns a list of territories with their 3 digit number code. The 3 digit number code is used as array key and the territory (region) as array value
Alpha3ToTerritory	Returns a list of 3 sign character codes for territories. The territory (region) is used as array key and the 3 sign character code as array value
TerritoryToAlpha3	Returns a list of territories with their 3 sign character code. The 3 sign character code is used as array key and the territory (region) as array value
PostalToTerritory	Returns a list of territories with a regex for postal codes which are included within that territory. The ISO territory code ('territory') is used as array key and the regex as array value.
NumberingSystem	Returns a list of scripts with the notation for digits used within the script
FallbackToChar	Returns a list of replacement characters for often used unicode characters. This can be used to replace "©" with "(C)" for example
CharToFallback	Returns a list of unicode characters for often used replacement characters. This can be used to replace "(C)" with "©" for example
LocaleUpgrade	Returns a list of locale dependencies which can be used to upgrade a language to a full qualified locale
Unit	Returns a list of localized calendar units. This can be used to translate the strings "day", "month" and so on automatically

If you are in need of a single translated value, you can use the `getTranslation()` method. It returns always a string but it accepts some different types than the `getTranslationList()` method. Also value is the same as before with one difference. You have to give the detail you want to get returned as additional value.

Because you have almost always give a value as detail this parameter has to be given as first parameter. This differs from the `getTranslationList()` method.

See the following table for detailed information:

Table 92. Details for getTranslation($value = null, $type = null, $locale = null)

Type	Description
Language	Returns a translation for a language. To select the wished translation you must give the language code as value
Script	Returns a translation for a script. To select the wished translation you must give the script code as value
Territory or *Country*	Returns a translation for a territory. This can be countries, continents and territories. To select the wished variant you must give the territory code as value
Variant	Returns a translation for a script variant. To select the wished variant you must give the variant code as value
Key	Returns translation for a known keys. This keys are generic values used in translation. These are normally calendar, collation and currency. To select the wished key you must give the key code as value
DefaultCalendar	Returns the default calendar for the given locale. For most locales this will be 'gregorian'. Use Zend_Date for simplicity
MonthContext	Returns the default context for months which is used within the given calendar. If you omit the value the 'gregorian' calendar will be used. Use Zend_Date for simplicity
DefaultMonth	Returns the default format for months which is used within the given calendar. If you omit the value the 'gregorian' calendar will be used. Use Zend_Date for simplicity
Month	Returns a translation for a month. You have to give the number of the month as integer value. It has to be between 1 and 12. If you want to receive data for other calendars, contexts or formats, then you must give an array instead of an integer with the expected values. The array has to look like this: `array( 'calendar', 'context', 'format', 'month number')`. If you give only an integer then the default values are the 'gregorian' calendar, the context 'format' and the format 'wide'. Use Zend_Date for simplicity

Type	Description
DayContext	Returns the default context for ´days which is used within the given calendar. If you omit the value the 'gregorian' calendar will be used. Use Zend_Date for simplicity
DefaultDay	Returns the default format for days which is used within the given calendar. If you omit the value the 'gregorian' calendar will be used. Use Zend_Date for simplicity
Day	Returns a translation for a day. You have to give the english abbreviation of the day as string value ('sun', 'mon', etc.). If you want to receive data for other calendars, contexts or format, then you must give an array instead of an integer with the expected values. The array has to look like this: `array('calendar', 'context', 'format', 'day abbreviation')`. If you give only an string then the default values are the 'gregorian' calendar, the context 'format' and the format 'wide'. Use Zend_Date for simplicity
Quarter	Returns a translation for a quarter. You have to give the number of the quarter as integer and it has to be between 1 and 4. If you want to receive data for other calendars, contexts or formats, then you must give an array instead of an integer with the expected values. The array has to look like this: `array('calendar', 'context', 'format', 'quarter number')`. If you give only an string then the default values are the 'gregorian' calendar, the context 'format' and the format 'wide'
Am	Returns a translation for 'AM' in a expected locale. If you want to receive data for other calendars an string with the expected calendar. If you omit the value then the 'gregorian' calendar will be used. Use Zend_Date for simplicity
Pm	Returns a translation for 'PM' in a expected locale. If you want to receive data for other calendars an string with the expected calendar. If you omit the value then the 'gregorian' calendar will be used. Use Zend_Date for simplicity
Era	Returns a translation for an era within a locale. You have to give the era number as string or integer. If you want to receive data for other calendars or formats, then you must give an array instead of the era number with the

Type	Description
	expected values. The array has to look like this: `array('calendar', 'format', 'era number')`. If you give only an string then the default values are the 'gregorian' calendar and the 'abbr' format
DefaultDate	Returns the default date format which is used within the given calendar. If you omit the value the 'gregorian' calendar will be used. Use Zend_Date for simplicity
Date	Returns the date format for an given calendar or format within a locale. If you omit the value then the 'gregorian' calendar will be used with the 'medium' format. If you give a string then the 'gregorian' calendar will be used with the given format. Or you can also give an array which will have to look like this: `array('calendar', 'format')`. Use Zend_Date for simplicity
DefaultTime	Returns the default time format which is used within the given calendar. If you omit the value the 'gregorian' calendar will be used. Use Zend_Date for simplicity
Time	Returns the time format for an given calendar or format within a locale. If you omit the value then the 'gregorian' calendar will be used with the 'medium' format. If you give a string then the 'gregorian' calendar will be used with the given format. Or you can also give an array which will have to look like this: `array('calendar', 'format')`. Use Zend_Date for simplicity
DateTime	Returns the datetime format for the given locale which indicates how to display date with times in the same string within the given calendar. If you omit the value the 'gregorian' calendar will be used. Use Zend_Date for simplicity
DateItem	Returns the default format for a given date or time item
DateInterval	Returns the interval format for a given date or time format. The first value is the calendar format, normally 'gregorian'. The second value is the interval format and the third value the token with the greatest difference. For example: array('gregorian', 'yMMMM', 'y') returns the interval format for the date format 'yMMMM' where 'y' has the greatest difference.
Field	Returns a translated date field which can be used to display calendars or date strings like 'month' or 'year' in a wished language. You must give the field which has to be returned as string. In this case the 'gregorian' calendar

Type	Description
	will be used. You can get the field for other calendar formats if you give an array which has to look like this: `array('calendar', 'date field')`
Relative	Returns a translated date which is relative to today which can include date strings like 'yesterday' or 'tomorrow' in a wished language. You have to give the number of days relative to tomorrow to receive the expected string. Yesterday would be '-1', tomorrow '1' and so on. This will use the 'gregorian' calendar. If you want to get relative dates for other calendars you will have to give an array which has to look like this: `array('calendar', 'relative days')`. Use Zend_Date for simplicity
DecimalNumber	Returns the format for decimal numbers within a given locale. Use Zend_Locale_Format for simplicity
ScientificNumber	Returns the format for scientific numbers within a given locale
PercentNumber	Returns the format for percentage numbers within a given locale
CurrencyNumber	Returns the format for displaying currency numbers within a given locale. Use Zend_Currency for simplicity
NameToCurrency	Returns the translated name for a given currency. The currency has to be given in ISO format which is for example 'EUR' for the currency 'euro'. Use Zend_Currency for simplicity
CurrencyToName	Returns a currency for a given localized name. Use Zend_Currency for simplicity
CurrencySymbol	Returns the used symbol for a currency within a given locale. Not for all currencies exists a symbol. Use Zend_Currency for simplicity
Question	Returns a localized string for acceptance ('yes') and negotation ('no'). You have to give either 'yes' or 'no' as value to receive the expected string. Use Zend_Locale's getQuestion method for simplicity
CurrencyFraction	Returns the fraction to use for a given currency. You must give the currency as ISO value. Use Zend_Currency for simplicity
CurrencyRounding	Returns how to round a given currency. You must give the currency as ISO value. If you omit the currency then the 'DEFAULT' rounding will be returned. Use Zend_Currency for simplicity

Type	Description
CurrencyToRegion	Returns the currency for a given region. The region code has to be given as `ISO3166` string for example 'AT' for austria. Use Zend_Currency for simplicity
RegionToCurrency	Returns the regions where a currency is used. The currency has to be given as `ISO4217` code for example 'EUR' for euro. When a currency is used in multiple regions, these regions are separated with a whitespace character. Use Zend_Currency for simplicity
RegionToTerritory	Returns the regions for a given territory. The territory has to be given as `ISO4217` string for example '001' for world. The regions within this territory are separated with a whitespace character
TerritoryToRegion	Returns the territories where a given region is located. The region has to be given in `ISO3166` string for example 'AT' for austria. When a region is located in multiple territories then these territories are separated with a whitespace character
ScriptToLanguage	Returns the scripts which are used within a given language. The language has to be given as ISO language code for example 'en' for english. When multiple scripts are used within a language then these scripts are separated with a whitespace character
LanguageToScript	Returns the languages which are used within a given script. The script has to be given as ISO script code for example 'Latn' for latin. When a script is used in multiple languages then these languages are separated with a whitespace character
TerritoryToLanguage	Returns the territories where a given language is used. The language has to be given as ISO language code for example 'en' for english. When multiple territories exist where this language is used then these territories are separated with a whitespace character
LanguageToTerritory	Returns the languages which are used within a given territory. The territory has to be given as `ISO3166` code for example 'IT' for italia. When a language is used in multiple territories then these territories are separated with a whitespace character
TimezoneToWindows	Returns a ISO timezone for a given windows timezone

Type	Description
WindowsToTimezone	Returns a windows timezone for a given ISO timezone
TerritoryToTimezone	Returns the territory for a given ISO timezone
TimezoneToTerritory	Returns the ISO timezone for a given territory
CityToTimezone	Returns the localized city for a given ISO timezone. Not for all timezones does a city translation exist
TimezoneToCity	Returns the ISO timezone for a given localized city name. Not for all cities does a timezone exist
PhoneToTerritory	Returns the telephone code for a given territory (region). The territory code has to be given as `ISO3166` string for example 'AT' for austria
TerritoryToPhone	Returns the territory (region) where a telephone code is used. The telephone code has to be given as plain integer code for example '43' for +43. When a telephone code is used in multiple territories (regions), these territories are separated with a whitespace character
NumericToTerritory	Returns the 3 digit number code for a given territory (region). The territory code has to be given as `ISO3166` string for example 'AT' for austria
TerritoryToNumeric	Returns the territory (region) for a 3 digit number code. The 3 digit number code has to be given as plain integer code for example '43'
Alpha3ToTerritory	Returns the 3 sign character code for a given territory (region). The territory code has to be given as `ISO3166` string for example 'AT' for austria
TerritoryToAlpha3	Returns the territory (region) for a 3 sign character code
PostalToTerritory	Returns the a regex for postal codes for a given territory. The territory has to be given as `ISO4217` string for example '001' for world
NumberingSystem	Returns a scripts with the notation for digits used within this script
FallbackToChar	Returns a replacement character for a often used unicode character. This can be used to replace "©" with "(C)" for example
CharToFallback	Returns a unicode character for a often used replacement character. This can be used to replace "(C)" with "©" for example
LocaleUpgrade	Returns a locale dependencies for a given language which can be used to upgrade this language to a full qualified locale

Type	Description
Unit	Returns a localized calendar unit. This can be used to translate the strings "day", "month" and so on automatically. The first parameter has to be the type, and the second parameter has to be the count

With Zend Framework 1.5 several old types have been renamed. This has to be done because of several new types, some misspelling and to increase the usability. See this table for a list of old to new types:

Table 93. Differences between Zend Framework 1.0 and 1.5

Old type	New type
Country	Territory (with value '2')
Calendar	Type (with value 'calendar')
Month_Short	Month (with array('gregorian', 'format', 'abbreviated')
Month_Narrow	Month (with array('gregorian', 'stand-alone', 'narrow')
Month_Complete	Months
Day_Short	Day (with array('gregorian', 'format', 'abbreviated')
Day_Narrow	Day (with array('gregorian', 'stand-alone', 'narrow')
DateFormat	Date
TimeFormat	Time
Timezones	CityToTimezone
Currency	NameToCurrency
Currency_Sign	CurrencySymbol
Currency_Detail	CurrencyToRegion
Territory_Detail	TerritoryToRegion
Language_Detail	LanguageToTerritory

The example below demonstrates how to obtain the names of things in different languages.

Example 490. getTranslationList

```
// prints the names of all countries in German language
print_r(Zend_Locale::getTranslationList('country', 'de'));
```

The next example shows how to find the name of a language in another language, when the two letter iso country code is not known.

Example 491. Converting country name in one language to another

```
$code2name = Zend_Locale::getLanguageTranslationList('en_US');
$name2code = array_flip($code2name);
$frenchCode = $name2code['French'];
echo Zend_Locale::getLanguageTranslation($frenchCode, 'de_AT');
// output is the German name of the French language
```

To generate a list of all languages known by `Zend_Locale`, with each language name shown in its own language, try the example below in a web page. Similarly, `getCountryTranslationList()` and `getCountryTranslation()` could be used to create a table mapping your native language names for regions to the names of the regions shown in another language. Use a `try .. catch` block to handle exceptions that occur when using a locale that does not exist. Not all languages are also locales. In the example, below exceptions are ignored to prevent early termination.

Example 492. All Languages written in their native language

```
$list = Zend_Locale::getLanguageTranslationList('auto');

foreach($list as $language => $content) {
    try {
        $output = Zend_Locale::getLanguageTranslation($language, $language);
        if (is_string($output)) {
            print "\n<br>[".$language."] ".$output;
        }
    } catch (Exception $e) {
        continue;
    }
}
```

2.7. Obtaining translations for "yes" and "no"

Frequently, programs need to solicit a "yes" or "no" response from the user. Use `getQuestion()` to obtain an array containing the correct word(s) or regex strings to use for prompting the user in a particular $locale (defaults to the current object's locale). The returned array will contain the following informations :

- *yes and no*: A generic string representation for yes and no responses. This will contain the first and most generic response from yesarray and noarray.

  yesarray and noarray: An array with all known yes and no responses. Several languages have more than just two responses. In general this is the full string and its abbreviation.

  yesexpr and noexpr: An generated regex which allows you to handle user response, and search for yes or no.

All of this informations are of course localized and depend on the set locale. See the following example for the informations you can receive:

Example 493. getQuestion()

```
$locale = new Zend_Locale();
// Question strings
print_r($locale->getQuestion('de'));

- - - Output - - -

Array
(
    [yes] => ja
    [no] => nein
    [yesarray] => Array
        (
            [0] => ja
            [1] => j
        )

    [noarray] => Array
        (
            [0] => nein
            [1] => n
        )

    [yesexpr] => ^([jJ][aA]?)|([jJ]?)
    [noexpr] => ^([nN]([eE][iI][nN])?)|([nN]?)
)
```

Until 1.0.3 *yesabbr* from the underlaying locale data was also available. Since 1.5 this information is no longer standalone available, but you will find the information from it within *yesarray*.

2.8. Get a list of all known locales

Sometimes you will want to get a list of all known locales. This can be used for several tasks like the creation of a selectbox. For this purpose you can use the static `getLocaleList()` method which will return a list of all known locales.

Example 494. getLocaleList()

```
$localelist = Zend_Locale::getLocaleList();
```

Note that the locales are returned as key of the array you will receive. The value is always a boolean TRUE.

2.9. Detecting locales

When you want to detect if a given input, regardless of its source, is a locale you should use the static `isLocale()` method. The first parameter of this method is the string which you want to check.

Example 495. Simple locale detection

```
$input = 'to_RU';
if (Zend_Locale::isLocale($input)) {
    print "'{$input}' is a locale";
} else {
    print "Sorry... the given input is no locale";
}
```

As you can see, the output of this method is always a boolean. There is only one reason you could get an exception when calling this method. When your system does not provide any locale and Zend Framework is not able to detect it automatically. Normally this shows that there is a problem with your OS in combination with PHP's `setlocale()`.

You should also note that any given locale string will automatically be degraded if the region part does not exist for this locale. In our previous example the language `'to'` does not exist in the region `'RU'`, but you will still get TRUE returned as `Zend_Locale` can work with the given input.

Still it's sometimes useful to prevent this automatic degrading, and this is where the second parameter of `isLocale()` comes in place. The `strict` parameter defaults to FALSE and can be used to prevent degrading when set to TRUE.

Example 496. Strict locale detection

```
$input = 'to_RU';
if (Zend_Locale::isLocale($input, true)) {
    print "'{$input}' is a locale";
} else {
    print "Sorry... the given input is no locale";
}
```

Now that you are able to detect if a given string is a locale you could add locale aware behaviour to your own classes. But you will soon detect that this will always leads to the same 15 lines of code. Something like the following example:

Example 497. Implement locale aware behaviour

```
if ($locale === null) {
    $locale = new Zend_Locale();
}

if (!Zend_Locale::isLocale($locale, true, false)) {
    if (!Zend_Locale::isLocale($locale, false, false)) {
        throw new Zend_Locale_Exception(
            "The locale '$locale' is no known locale");
    }

    $locale = new Zend_Locale($locale);
}

if ($locale instanceof Zend_Locale) {
    $locale = $locale->toString();
}
```

With Zend Framework 1.8 we added a static `findLocale()` method which returns you a locale string which you can work with. It processes the following tasks:

- Detects if a given string is a locale

- Degrades the locale if it does not exist in the given region
- Returns a previous set application wide locale if no input is given
- Detects the locale from browser when the previous detections failed
- Detects the locale from environment when the previous detections failed
- Detects the locale from framework when the previous detections failed
- Returns always a string which represents the found locale.

The following example shows how these checks and the above code can be simplified with one single call:

Example 498. Locale aware behaviour as with Zend Framework 1.8

```
$locale = Zend_Locale::findLocale($inputstring);
```

3. Normalization and Localization

`Zend_Locale_Format` is a internal component used by `Zend_Locale`. All locale aware classes use `Zend_Locale_Format` for normalization and localization of numbers and dates. Normalization involves parsing input from a variety of data representations, like dates, into a standardized, structured representation, such as a PHP array with year, month, and day elements.

The exact same string containing a number or a date might mean different things to people with different customs and conventions. Disambiguation of numbers and dates requires rules about how to interpret these strings and normalize the values into a standardized data structure. Thus, all methods in `Zend_Locale_Format` require a locale in order to parse the input data.

Default "root" Locale

If no locale is specified, then normalization and localization will use the standard "root" locale, which might yield unexpected behavior, if the input originated in a different locale, or output for a specific locale was expected.

3.1. Number normalization: getNumber($input, Array $options)

There are many number systems different from the common decimal system (e.g. "3.14"). Numbers can be normalized with the `getNumber()` function to obtain the standard decimal representation. for all number-related discussions in this manual, Arabic/European numerals (0,1,2,3,4,5,6,7,8,9) are implied, unless explicitly stated otherwise. The options array may contain a 'locale' to define grouping and decimal characters. The array may also have a 'precision' to truncate excess digits from the result.

Example 499. Number normalization

```
$locale = new Zend_Locale('de_AT');
$number = Zend_Locale_Format::getNumber('13.524,678',
                                        array('locale' => $locale,
                                              'precision' => 3)
                                       );

print $number; // will return 13524.678
```

3.1.1. Precision and Calculations

Since `getNumber($value, array $options = array())` can normalize extremely large numbers, check the result carefully before using finite precision calculations, such as ordinary PHP math operations. For example, `if ((string)int_val($number) != $number) { use BCMath or GMP` . Most PHP installations support the BCMath extension.

Also, the precision of the resulting decimal representation can be rounded to a desired length with `getNumber()` with the option `'precision'`. If no precision is given, no rounding occurs. Use only PHP integers to specify the precision.

If the resulting decimal representation should be truncated to a desired length instead of rounded the option `'number_format'` can be used instead. Define the length of the decimal representation with the desired length of zeros. The result will then not be rounded. So if the defined precision within `number_format` is zero the value "1.6" will return "1", not "2. See the example nearby:

Example 500. Number normalization with precision

```
$locale = new Zend_Locale('de_AT');
$number = Zend_Locale_Format::getNumber('13.524,678',
                                        array('precision' => 1,
                                              'locale' => $locale)
                                       );
print $number; // will return 13524.7

$number = Zend_Locale_Format::getNumber('13.524,678',
                                        array('number_format' => '#.00',
                                              'locale' => $locale)
                                       );
print $number; // will return 13524.67
```

3.2. Number localization

`toNumber($value, array $options = array())` can localize numbers to the following supported locales . This function will return a localized string of the given number in a conventional format for a specific locale. The 'number_format' option explicitly specifies a non-default number format for use with `toNumber()`.

Example 501. Number localization

```
$locale = new Zend_Locale('de_AT');
$number = Zend_Locale_Format::toNumber(13547.36,
                                       array('locale' => $locale));

// will return 13.547,36
print $number;
```

Unlimited length

`toNumber()` can localize numbers with unlimited length. It is not related to integer or float limitations.

The same way as within `getNumber()`, `toNumber()` handles precision. If no precision is given, the complete localized number will be returned.

Example 502. Number localization with precision

```
$locale = new Zend_Locale('de_AT');
$number = Zend_Locale_Format::toNumber(13547.3678,
                                       array('precision' => 2,
                                             'locale' => $locale));

// will return 13.547,37
print $number;
```

Using the option 'number_format' a self defined format for generating a number can be defined. The format itself has to be given in CLDR format as described below. The locale is used to get separation, precision and other number formatting signs from it. German for example defines ',' as precision separation and in English the '.' sign is used.

Table 94. Format tokens for self generated number formats

Token	Description	Example format	Generated output
#0	Generates a number without precision and separation	#0	1234567
,	Generates a separation with the length from separation to next separation or to 0	#,##0	1,234,567
#,##,##0	Generates a standard separation of 3 and all following separations with 2	#,##,##0	12,34,567
.	Generates a precision	#0.#	1234567.1234
0	Generates a precision with a defined length	#0.00	1234567.12

Example 503. Using a self defined number format

```
$locale = new Zend_Locale('de_AT');
$number = Zend_Locale_Format::toNumber(13547.3678,
                                       array('number_format' => '#,#0.00',
                                             'locale' => 'de')
                                      );

// will return 1.35.47,36
print $number;

$number = Zend_Locale_Format::toNumber(13547.3,
                                       array('number_format' => '#,##0.00',
                                             'locale' => 'de')
                                       );

// will return 13.547,30
print $number;
```

3.3. Number testing

`isNumber($value, array $options = array())` checks if a given string is a number and returns TRUE or FALSE.

Example 504. Number testing

```
$locale = new Zend_Locale();
if (Zend_Locale_Format::isNumber('13.445,36', array('locale' => 'de_AT')) {
    print "Number";
} else {
    print "not a Number";
}
```

3.4. Float value normalization

Floating point values can be parsed with the `getFloat($value, array $options = array())` function. A floating point value will be returned.

Example 505. Floating point value normalization

```
$locale = new Zend_Locale('de_AT');
$number = Zend_Locale_Format::getFloat('13.524,678',
                                       array('precision' => 2,
                                             'locale' => $locale)
                                      );

// will return 13524.68
print $number;
```

3.5. Floating point value localization

`toFloat()` can localize floating point values. This function will return a localized string of the given number.

Example 506. Floating point value localization

```
$locale = new Zend_Locale('de_AT');
$number = Zend_Locale_Format::toFloat(13547.3655,
                                      array('precision' => 1,
                                            'locale' => $locale)
                                     );

// will return 13.547,4
print $number;
```

3.6. Floating point value testing

`isFloat($value, array $options = array())` checks if a given string is a floating point value and returns TRUE or FALSE.

Example 507. Floating point value testing

```
$locale = new Zend_Locale('de_AT');
if (Zend_Locale_Format::isFloat('13.445,36', array('locale' => $locale)) {
    print "float";
} else {
    print "not a float";
}
```

3.7. Integer value normalization

Integer values can be parsed with the `getInteger()` function. A integer value will be returned.

Example 508. Integer value normalization

```
$locale = new Zend_Locale('de_AT');
$number = Zend_Locale_Format::getInteger('13.524,678',
                                         array('locale' => $locale));

// will return 13524
print $number;
```

3.8. Integer point value localization

`toInteger($value, array $options = array())` can localize integer values. This function will return a localized string of the given number.

Example 509. Integer value localization

```
$locale = new Zend_Locale('de_AT');
$number = Zend_Locale_Format::toInteger(13547.3655,
                                        array('locale' => $locale));

// will return 13.547
print $number;
```

3.9. Integer value testing

`isInteger($value, array $options = array())` checks if a given string is a integer value and returns TRUE or FALSE.

Example 510. Integer value testing

```
$locale = new Zend_Locale('de_AT');
if (Zend_Locale_Format::isInteger('13.445', array('locale' => $locale)) {
    print "integer";
} else {
    print "not a integer";
}
```

3.10. Numeral System Conversion

`Zend_Locale_Format::convertNumerals()` converts digits between different numeral systems , including the standard Arabic/European/Latin numeral system (0,1,2,3,4,5,6,7,8,9),

not to be confused with Eastern Arabic numerals sometimes used with the Arabic language to express numerals. Attempts to use an unsupported numeral system will result in an exception, to avoid accidentally performing an incorrect conversion due to a spelling error. All characters in the input, which are not numerals for the selected numeral system, are copied to the output with no conversion provided for unit separator characters. `Zend_Locale*` components rely on the data provided by CLDR (see their list of scripts grouped by language).

In CLDR and hereafter, the Europena/Latin numerals will be referred to as "Latin" or by the assigned 4-letter code "Latn". Also, the CLDR refers to this numeral systems as "scripts".

Suppose a web form collected a numeric input expressed using Eastern Arabic digits "####". Most software and PHP functions expect input using Arabic numerals. Fortunately, converting this input to its equivalent Latin numerals "100" requires little effort using `convertNumerals($inputNumeralString, $sourceNumeralSystem, $destNumeralSystem)` , which returns the `$input` with numerals in the script `$sourceNumeralSystem` converted to the script `$destNumeralSystem`.

Example 511. Converting numerals from Eastern Arabic scripts to European/Latin scripts

```
$arabicScript = "####";   // Arabic for "100" (one hundred)
$latinScript = Zend_Locale_Format::convertNumerals($arabicScript,
                                                   'Arab',
                                                   'Latn');

print "\nOriginal:   " . $arabicScript;
print "\nNormalized: " . $latinScript;
```

Similarly, any of the supported numeral systems may be converted to any other supported numeral system.

Example 512. Converting numerals from Latin script to Eastern Arabic script

```
$latinScript = '123';
$arabicScript = Zend_Locale_Format::convertNumerals($latinScript,
                                                    'Latn',
                                                    'Arab');

print "\nOriginal:  " . $latinScript;
print "\nLocalized: " . $arabicScript;
```

Example 513. Getting 4 letter CLDR script code using a native-language name of the script

```
function getScriptCode($scriptName, $locale)
{
    $scripts2names = Zend_Locale_Data::getList($locale, 'script');
    $names2scripts = array_flip($scripts2names);
    return $names2scripts[$scriptName];
}
echo getScriptCode('Latin', 'en'); // outputs "Latn"
echo getScriptCode('Tamil', 'en'); // outputs "Taml"
echo getScriptCode('tamoul', 'fr'); // outputs "Taml"
```

For a list of supported numeral systems call `Zend_Locale::getTranslationList('numberingsystem', 'en')`.

4. Working with Dates and Times

`Zend_Locale_Format` provides several methods for working with dates and times to help convert and normalize between different formats for different locales. Use `Zend_Date` for manipulating dates, and working with date strings that already conform to one of the many internationally recognized standard formats, or one of the localized date formats supported by `Zend_Date` . Using an existing, pre-defined format offers advantages, including the use of well-tested code, and the assurance of some degree of portability and interoperability (depending on the standard used). The examples below do not follow these recommendations, since using non-standard date formats would needlessly increase the difficulty of understanding these examples.

4.1. Normalizing Dates and Times

The `getDate()` method parses strings containing dates in localized formats. The results are returned in a structured array, with well-defined keys for each part of the date. In addition, the array will contain a key 'date_format' showing the format string used to parse the input date string. Since a localized date string may not contain all parts of a date/time, the key-value pairs are optional. For example, if only the year, month, and day is given, then all time values are suppressed from the returned array, and vice-versa if only hour, minute, and second were given as input. If no date or time can be found within the given input, an exception will be thrown.

If `setOption(array('fix_date' => true))` is set the `getDate()` method adds a key 'fixed' with a whole number value indicating if the input date string required "fixing" by rearranging the day, month, or year in the input to fit the format used.

Table 95. Key values for getDate() with option 'fix_date'

value	meaning
0	nothing to fix
1	fixed false month
2	swapped day and year
3	swapped month and year
4	swapped month and day

For those needing to specify explicitly the format of the date string, the following format token specifiers are supported. If an invalid format specifier is used, such as the PHP 'i' specifier when in ISO format mode, then an error will be thrown by the methods in `Zend_Locale_Format` that support user-defined formats.

These specifiers (below) are a small subset of the full "ISO" set supported by `Zend_Date`'s `toString()`. If you need to use PHP `date()` compatible format specifiers, then first call `setOptions(array('format_type' => 'php'))`. And if you want to convert only one special format string from PHP `date()` compatible format to "ISO" format use `convertPhpToIsoFormat()`. Currently, the only practical difference relates to the specifier for minutes ('m' using the ISO default, and 'i' using the PHP date format).

Table 96. Return values

getDate() format character	Array key	Returned value	Minimum	Maximum
d	day	integer	1	31

getDate() format character	Array key	Returned value	Minimum	Maximum
M	month	integer	1	12
y	year	integer	no limit	PHP integer's maximum
h	hour	integer	0	PHP integer's maximum
m	minute	integer	0	PHP integer's maximum
s	second	integer	0	PHP integer's maximum

Example 514. Normalizing a date

```
$dateString = Zend_Locale_Format::getDate('13.04.2006',
                                          array('date_format' =>
                                                'dd.MM.yyyy')
                                         );

// creates a Zend_Date object for this date
$dateObject = Zend_Date('13.04.2006',
                        array('date_format' => 'dd.MM.yyyy'));

print_r($dateString); // outputs:

Array
(
    [format] => dd.MM.yyyy
    [day] => 13
    [month] => 4
    [year] => 2006
)

// alternatively, some types of problems with input data can be
// automatically corrected
$date = Zend_Locale_Format::getDate('04.13.2006',
                                    array('date_format' => 'dd.MM.yyyy',
                                          'fix_date' => true)
                                   );

print_r($date); // outputs:

Array
(
    [format] => dd.MM.yyyy
    [day] => 13
    [month] => 4
    [year] => 2006
    [fixed] => 4
)
```

Since `getDate()` is "locale-aware", specifying the `$locale` is sufficient for date strings adhering to that locale's format. The option '`fix_date`' uses simple tests to determine if the day or month is not valid, and then applies heuristics to try and correct any detected problems. Note the use of '`Zend_Locale_Format::STANDARD`' as the value for '`date_format`' to prevent the

use of a class-wide default date format set using `setOptions()`. This forces getDate to use the default date format for `$locale`.

Example 515. Normalizing a date by locale

```
$locale = new Zend_Locale('de_AT');
$date = Zend_Locale_Format::getDate('13.04.2006',
                                    array('date_format' =>
                                              Zend_Locale_Format::STANDARD,
                                          'locale' => $locale)
                                   );

print_r ($date);
```

A complete date and time is returned when the input contains both a date and time in the expected format.

Example 516. Normalizing a date with time

```
$locale = new Zend_Locale('de_AT');
$date = Zend_Locale_Format::getDate('13.04.2005 22:14:55',
                                    array('date_format' =>
                                              Zend_Locale_Format::STANDARD,
                                          'locale' => $locale)
                                    );

print_r ($date);
```

If a specific format is desired, specify the `$format` argument, without giving a `$locale`. Only single-letter codes (H, m, s, y, M, d), and MMMM and EEEE are supported in the `$format`.

Example 517. Normalizing a userdefined date

```
$date = Zend_Locale_Format::getDate('13200504T551422',
                                    array('date_format' =>
                                              'ddyyyyMM ssmmHH')
                                   );

print_r ($date);
```

The format can include the following signs :

Table 97. Format definition

Format Letter	Description
d or dd	1 or 2 digit day
M or MM	1 or 2 digit month
y or yy	1 or 2 digit year
yyyy	4 digit year
h	1 or 2 digit hour
m	1 or 2 digit minute
s	1 or 2 digit second

Examples for proper formats are

Table 98. Example formats

Formats	Input	Output
dd.MM.yy	1.4.6	['day'] => 1, ['month'] => 4, ['year'] => 6
dd.MM.yy	01.04.2006	['day'] => 1, ['month'] => 4, ['year'] => 2006
yyyyMMdd	1.4.6	['day'] => 6, ['month'] => 4, ['year'] => 1

Database date format

To parse a database date value (f.e. MySql or MsSql), use `Zend_Date`'s ISO_8601 format instead of getDate().

The option '`fix_date`' uses simple tests to determine if the day or month is not valid, and then applies heuristics to try and correct any detected problems. `getDate()` automatically detects and corrects some kinds of problems with input, such as misplacing the year:

Example 518. Automatic correction of input dates

```
$date = Zend_Locale_Format::getDate('41.10.20',
                                    array('date_format' => 'ddMMyy',
                                          'fix_date' => true)
                                   );

// instead of 41 for the day, the 41 will be returned as year value
print_r ($date);
```

4.2. Testing Dates

Use `checkDateFormat($inputString, array('date_format' => $format, $locale))` to check if a given string contains all expected date parts. The `checkDateFormat()` method uses `getDate()`, but without the option `'fixdate'` to avoid returning TRUE when the input fails to conform to the date format. If errors are detected in the input, such as swapped values for months and days, the option `'fixdate'` method will apply heuristics to "correct" dates before determining their validity.

Example 519. Date testing

```
$locale = new Zend_Locale('de_AT');
// using the default date format for 'de_AT', is this a valid date?
if (Zend_Locale_Format::checkDateFormat('13.Apr.2006',
                                        array('date_format' =>
                                              Zend_Locale_Format::STANDARD,
                                            $locale)
                                       ) {
    print "date";
} else {
    print "not a date";
}
```

4.3. Normalizing a Time

Normally, a time will be returned with a date, if the input contains both. If the proper format is not known, but the locale relevant to the user input is known, then `getTime()` should be used, because it uses the default time format for the selected locale.

Example 520. Normalize an unknown time

```
$locale = new Zend_Locale('de_AT');
if (Zend_Locale_Format::getTime('13:44:42',
                                array('date_format' =>
                                      Zend_Locale_Format::STANDARD,
                                      'locale' => $locale)) {
    print "time";
} else {
    print "not a time";
}
```

4.4. Testing Times

Use `checkDateFormat()` to check if a given string contains a proper time. The usage is exact the same as with checking Dates, only `date_format` should contain the parts which you expect to have.

Example 521. Testing a time

```
$locale = new Zend_Locale('de_AT');
if (Zend_Locale_Format::checkDateFormat('13:44:42',
                                        array('date_format' => 'HH:mm:ss',
                                              'locale' => $locale)) {
    print "time";
} else {
    print "not a time";
}
```

5. Supported locales

`Zend_Locale` provides information on several locales. The following table shows all languages and their related locales, sorted by language:

Table 99. List of all supported languages

Language	Locale	Region
Afar	aa	---
	aa_DJ	Djibouti
	aa_ER	Eritrea
	aa_ET	Ethiopia
Afrikaans	af	---
	af_NA	Namibia
	af_ZA	South Africa
Akan	ak	---
	ak_GH	Ghana
Amharic	am	---

Language	Locale	Region
	am_ET	Ethiopia
Arabic	ar	---
	ar_AE	United Arab Emirates
	ar_BH	Bahrain
	ar_DZ	Algeria
	ar_EG	Egypt
	ar_IQ	Iraq
	ar_JO	Jordan
	ar_KW	Kuwait
	ar_LB	Lebanon
	ar_LY	Libya
	ar_MA	Morocco
	ar_OM	Oman
	ar_QA	Qatar
	ar_SA	Saudi Arabia
	ar_SD	Sudan
	ar_SY	Syria
	ar_TN	Tunisia
	ar_YE	Yemen
Assamese	as	---
	as_IN	India
Azerbaijani	az	---
	az_AZ	Azerbaijan
Belarusian	be	---
	be_BY	Belarus
Bulgarian	bg	---
	bg_BG	Bulgaria
Bengali	bn	---
	bn_BD	Bangladesh
	bn_IN	India
Tibetan	bo	---
	bo_CN	China
	bn_IN	India
Bosnian	bs	---
	bs_BA	Bosnia and Herzegovina
Blin	byn	---
	byn_ER	Eritrea
Catalan	ca	---

Language	Locale	Region
	ca_ES	Spain
Atsam	cch	---
	cch_NG	Nigeria
Coptic	cop	---
Czech	cs	---
	cs_CZ	Czech Republic
Welsh	cy	---
	cy_GB	United Kingdom
Danish	da	---
	da_DK	Denmark
German	de	---
	de_AT	Austria
	de_BE	Belgium
	de_CH	Switzerland
	de_DE	Germany
	de_LI	Liechtenstein
	de_LU	Luxembourg
Divehi	dv	---
	dv_MV	Maldives
Dzongkha	dz	---
	dz_BT	Bhutan
Ewe	ee	---
	ee_GH	Ghana
	ee_TG	Togo
Greek	el	---
	el_CY	Cyprus
	el_GR	Greece
English	en	---
	en_AS	American Samoa
	en_AU	Australia
	en_BE	Belgium
	en_BW	Botswana
	en_BZ	Belize
	en_CA	Canada
	en_GB	United Kingdom
	en_GU	Guam
	en_HK	Hong Kong
	en_IE	Ireland

Language	Locale	Region
	en_IN	India
	en_JM	Jamaica
	en_MH	Marshall Islands
	en_MP	Northern Mariana Islands
	en_MT	Malta
	en_NA	Namibia
	en_NZ	New Zealand
	en_PH	Philippines
	en_PK	Pakistan
	en_SG	Singapore
	en_TT	Trinidad and Tobago
	en_UM	United States Minor Outlying Islands
	en_US	United States
	en_VI	U.S. Virgin Islands
	en_ZA	South Africa
	en_ZW	Zimbabwe
Esperanto	eo	---
Spanish	es	---
	es_AR	Argentina
	es_BO	Bolivia
	es_CL	Chile
	es_CO	Colombia
	es_CR	Costa Rica
	es_DO	Dominican Republic
	es_EC	Ecuador
	es_ES	Spain
	es_GT	Guatemala
	es_HN	Honduras
	es_MX	Mexico
	es_NI	Nicaragua
	es_PA	Panama
	es_PE	Peru
	es_PR	Puerto Rico
	es_PY	Paraguay
	es_SV	El Salvador
	es_US	United States
	es_UY	Uruguay

Language	Locale	Region
	es_VE	Venezuela
Estonian	et	---
	et_EE	Estonia
Basque	eu	---
	eu_ES	Spain
Persian	fa	---
	fa_AF	Afghanistan
	fa_IR	Iran
Finnish	fi	---
	fi_FI	Finland
Filipino	fil	---
	fil_PH	Philippines
Faroese	fo	---
	fo_FO	Faroe Islands
French	fr	---
	fr_BE	Belgium
	fr_CA	Canada
	fr_CH	Switzerland
	fr_FR	France
	fr_LU	Luxembourg
	fr_MC	Monaco
	fr_SN	Senegal
Friulian	fur	---
	fur_IT	Italy
Irish	ga	---
	ga_IE	Ireland
Ga	gaa	---
	gaa_GH	Ghana
Geez	gez	---
	gez_ER	Eritrea
	gez_ET	Ethiopia
Gallegan	gl	---
	gl_ES	Spain
Swiss German	gsw	---
	gsw_CH	Swiss
Gujarati	gu	---
	gu_IN	India
Manx	gv	---

Language	Locale	Region
	gv_GB	United Kingdom
Hausa	ha	---
	ha_GH	Ghana
	ha_NE	Niger
	ha_NG	Nigeria
	ha_SD	Sudan
Hawaiian	haw	---
	haw_US	United States
Hebrew	he	---
	he_IL	Israel
Hindi	hi	---
	hi_IN	India
Croatian	hr	---
	hr_HR	Croatia
Hungarian	hu	---
	hu_HU	Hungary
Armenian	hy	---
Interlingua	ia	---
Indonesian	id	---
	id_ID	Indonesia
Igbo	ig	---
	ig_NG	Nigeria
Sichuan Yi	ii	---
	ii_CN	China
Indonesian	in	---
Icelandic	is	---
	is_IS	Iceland
Italian	it	---
	it_CH	Switzerland
	it_IT	Italy
Inuktitut	iu	---
Hebrew	iw	---
Japanese	ja	---
	ja_JP	Japan
Georgian	ka	---
	ka_GE	Georgia
Jju	kaj	---
	kaj_NG	Nigeria

Language	Locale	Region
Kamba	kam	---
	kam_KE	Kenya
Tyap	kcg	---
	kcg_NG	Nigeria
Koro	kfo	---
	kfo_CI	Ivory Coast
Kazakh	kk	---
	kk_KZ	Kazakhstan
Kalaallisut	kl	---
	kl_GL	Greenland
Khmer	km	---
	km_KH	Cambodia
Kannada	kn	---
	kn_IN	India
Korean	ko	---
	ko_KR	South Korea
Konkani	kok	---
	kok_IN	India
Kpelle	kpe	---
	kpe_GN	Guinea
	kpe_LR	Liberia
Kurdish	ku	---
	ku_IQ	Iraq
ku_IR	Iran	
ku_SY	Syria	
ku_TR	Turkey	
Cornish	kw	---
	kw_GB	United Kingdom
Kirghiz	ky	---
	ky_KG	Kyrgyzstan
Lingala	ln	---
	ln_CD	Congo - Kinshasa
	ln_CG	Congo - Brazzaville
Lao	lo	---
	lo_LA	Laos
Lithuanian	lt	---
	lt_LT	Lithuania
Latvian	lv	---

Language	Locale	Region
	lv_LV	Latvia
Macedonian	mk	---
	mk_MK	Macedonia
Malayalam	ml	---
	ml_IN	India
Mongolian	mn	---
	mn_CN	China
	mn_MN	Mongolia
Romanian	mo	---
Marathi	mr	---
	mr_IN	India
Malay	ms	---
	ms_BN	Brunei
	ms_MY	Malaysia
Maltese	mt	---
	mt_MT	Malta
Burmese	my	---
	my_MM	Myanmar
Norwegian Bokmal	nb	---
	nb_NO	Norway
Low German	nds	---
	nds_DE	Germany
Nepali	ne	---
	ne_IN	India
	ne_NP	Nepal
Dutch	nl	---
	nl_BE	Belgium
	nl_NL	Netherlands
Norwegian Nynorsk	nn	---
	nn_NO	Norway
Norwegian	no	---
South Ndebele	nr	---
	nr_ZA	South Africa
Northern Sotho	nso	---
	nso_ZA	South Africa
Nyanja	ny	---
	ny_MW	Malawi
Occitan	oc	---

Language	Locale	Region
	oc_FR	France
Oromo	om	---
	om_ET	Ethiopia
	om_KE	Kenya
Oriya	or	---
	or_IN	India
Punjabi	pa	---
	pa_IN	India
	pa_PK	Pakistan
Polish	pl	---
	pl_PL	Poland
Pashto	ps	---
	ps_AF	Afghanistan
Portuguese	pt	---
	pt_BR	Brazil
	pt_PT	Portugal
Romanian	ro	---
	ro_MD	Moldova
	ro_RO	Romania
Russian	ru	---
	ru_RU	Russia
	ru_UA	Ukraine
Kinyarwanda	rw	---
	rw_RW	Rwanda
Sanskrit	sa	---
	sa_IN	India
Northern Sami	se	---
	se_FI	Finland
	se_NO	Norway
Serbo-Croatian	sh	---
	sh_BA	Bosnia and Herzegovina
	sh_CS	Serbia and Montenegro
	sh_YU	Serbia
Sinhala	si	---
	si_LK	Sri Lanka
Sidamo	sid	---
	sid_ET	Ethiopia
Slovak	sk	---

Language	Locale	Region
	sk_SK	Slovakia
Slovenian	sl	---
	sl_SI	Slovenia
Somali	so	---
	so_DJ	Djibouti
	so_ET	Ethiopia
	so_KE	Kenya
	so_SO	Somalia
Albanian	sq	---
	sq_AL	Albania
Serbian	sr	---
	sr_BA	Bosnia and Herzegovina
	sr_CS	Serbia and Montenegro
	sr_ME	Montenegro
	sr_RS	Serbia
	sr_YU	Serbia
Swati	ss	---
	ss_SZ	Swaziland
	ss_ZA	South Africa
Southern Sotho	st	---
	st_LS	Lesotho
	st_ZA	South Africa
Swedish	sv	---
	sv_FI	Finland
	sv_SE	Sweden
Swahili	sw	---
	sw_KE	Kenya
	sw_TZ	Tanzania
Syriac	syr	---
	syr_SY	Syria
Tamil	ta	---
	ta_IN	India
Telugu	te	---
	te_IN	India
Tajik	tg	---
	tg_TJ	Tajikistan
Thai	th	---
	th_TH	Thailand

Language	Locale	Region
Tigrinya	ti	---
	ti_ER	Eritrea
	ti_ET	Ethiopia
Tigre	tig	---
	tig_ER	Eritrea
Tagalog	tl	---
Tswana	tn	---
	tn_ZA	South Africa
Tonga	to	---
	to_TO	Tonga
Turkish	tr	---
	tr_TR	Turkey
Taroko	trv	---
	trv_TW	Taiwan
Tsonga	ts	---
	ts_ZA	South Africa
Tatar	tt	---
	tt_RU	Russia
Uighur	ug	---
	ug_CN	China
Ukrainian	uk	---
	uk_UA	Ukraine
Urdu	ur	---
	ur_IN	India
	ur_PK	Pakistan
Uzbek	uz	---
	uz_AF	Afghanistan
	uz_UZ	Uzbekistan
Venda	ve	---
	ve_ZA	South Africa
Vietnamese	vi	---
	vi_VN	Vietnam
Walamo	wal	---
	wal_ET	Ethiopia
Wolof	wo	---
	wo_SN	Senegal
Xhosa	xh	---
	xh_ZA	South Africa

Language	Locale	Region
Yoruba	yo	---
	yo_NG	Nigeria
Chinese	zh	---
	zh_CN	China
	zh_HK	Hong Kong
	zh_MO	Macau
	zh_SG	Singapore
	zh_TW	Taiwan
Zulu	zu	---
	zu_ZA	South Africa

Zend_Log

1. Overview

Zend_Log is a component for general purpose logging. It supports multiple log backends, formatting messages sent to the log, and filtering messages from being logged. These functions are divided into the following objects:

- A Log (instance of Zend_Log) is the object that your application uses the most. You can have as many Log objects as you like; they do not interact. A Log object must contain at least one Writer, and can optionally contain one or more Filters.
- A Writer (inherits from Zend_Log_Writer_Abstract) is responsible for saving data to storage.
- A Filter (implements Zend_Log_Filter_Interface) blocks log data from being saved. A filter may be applied to an individual Writer, or to a Log where it is applied before all Writers. In either case, filters may be chained.
- A Formatter (implements Zend_Log_Formatter_Interface) can format the log data before it is written by a Writer. Each Writer has exactly one Formatter.

1.1. Creating a Log

To get started logging, instantiate a Writer and then pass it to a Log instance:

```
$logger = new Zend_Log();
$writer = new Zend_Log_Writer_Stream('php://output');

$logger->addWriter($writer);
```

It is important to note that the Log must have at least one Writer. You can add any number of Writers using the Log's addWriter() method.

Alternatively, you can pass a Writer directly to constructor of Log as a shortcut:

```
$writer = new Zend_Log_Writer_Stream('php://output');
$logger = new Zend_Log($writer);
```

The Log is now ready to use.

1.2. Logging Messages

To log a message, call the log() method of a Log instance and pass it the message with a corresponding priority:

```
$logger->log('Informational message', Zend_Log::INFO);
```

The first parameter of the log() method is a string message and the second parameter is an integer priority. The priority must be one of the priorities recognized by the Log instance. This is explained in the next section.

A shortcut is also available. Instead of calling the log() method, you can call a method by the same name as the priority:

```
$logger->log('Informational message', Zend_Log::INFO);
$logger->info('Informational message');

$logger->log('Emergency message', Zend_Log::EMERG);
$logger->emerg('Emergency message');
```

1.3. Destroying a Log

If the Log object is no longer needed, set the variable containing it to `NULL` to destroy it. This will automatically call the `shutdown()` instance method of each attached Writer before the Log object is destroyed:

```
$logger = null;
```

Explicitly destroying the log in this way is optional and is performed automatically at PHP shutdown.

1.4. Using Built-in Priorities

The `Zend_Log` class defines the following priorities:

```
EMERG   = 0;  // Emergency: system is unusable
ALERT   = 1;  // Alert: action must be taken immediately
CRIT    = 2;  // Critical: critical conditions
ERR     = 3;  // Error: error conditions
WARN    = 4;  // Warning: warning conditions
NOTICE  = 5;  // Notice: normal but significant condition
INFO    = 6;  // Informational: informational messages
DEBUG   = 7;  // Debug: debug messages
```

These priorities are always available, and a convenience method of the same name is available for each one.

The priorities are not arbitrary. They come from the BSD `syslog` protocol, which is described in RFC-3164. The names and corresponding priority numbers are also compatible with another PHP logging system, PEAR Log, which perhaps promotes interoperability between it and `Zend_Log`.

Priority numbers descend in order of importance. `EMERG` (0) is the most important priority. `DEBUG` (7) is the least important priority of the built-in priorities. You may define priorities of lower importance than `DEBUG`. When selecting the priority for your log message, be aware of this priority hierarchy and choose appropriately.

1.5. Adding User-defined Priorities

User-defined priorities can be added at runtime using the Log's `addPriority()` method:

```
$logger->addPriority('FOO', 8);
```

The snippet above creates a new priority, `FOO`, whose value is `8`. The new priority is then available for logging:

```
$logger->log('Foo message', 8);
$logger->foo('Foo Message');
```

New priorities cannot overwrite existing ones.

1.6. Understanding Log Events

When you call the `log()` method or one of its shortcuts, a log event is created. This is simply an associative array with data describing the event that is passed to the writers. The following keys are always created in this array: `timestamp`, `message`, `priority`, and `priorityName`.

The creation of the `event` array is completely transparent. However, knowledge of the `event` array is required for adding an item that does not exist in the default set above.

To add a new item to every future event, call the `setEventItem()` method giving a key and a value:

```
$logger->setEventItem('pid', getmypid());
```

The example above sets a new item named `pid` and populates it with the PID of the current process. Once a new item has been set, it is available automatically to all writers along with all of the other data event data during logging. An item can be overwritten at any time by calling the `setEventItem()` method again.

Setting a new event item with `setEventItem()` causes the new item to be sent to all writers of the logger. However, this does not guarantee that the writers actually record the item. This is because the writers won't know what to do with it unless a formatter object is informed of the new item. Please see the section on Formatters to learn more.

2. Writers

A Writer is an object that inherits from `Zend_Log_Writer_Abstract`. A Writer's responsibility is to record log data to a storage backend.

2.1. Writing to Streams

`Zend_Log_Writer_Stream` sends log data to a PHP stream.

To write log data to the PHP output buffer, use the URL `php://output`. Alternatively, you can send log data directly to a stream like STDERR (`php://stderr`).

```
$writer = new Zend_Log_Writer_Stream('php://output');
$logger = new Zend_Log($writer);

$logger->info('Informational message');
```

To write data to a file, use one of the Filesystem URLs:

```
$writer = new Zend_Log_Writer_Stream('/path/to/logfile');
$logger = new Zend_Log($writer);

$logger->info('Informational message');
```

By default, the stream opens in the append mode (`"a"`). To open it with a different mode, the `Zend_Log_Writer_Stream` constructor accepts an optional second parameter for the mode.

The constructor of `Zend_Log_Writer_Stream` also accepts an existing stream resource:

```
$stream = @fopen('/path/to/logfile', 'a', false);
if (! $stream) {
    throw new Exception('Failed to open stream');
}
```

```
$writer = new Zend_Log_Writer_Stream($stream);
$logger = new Zend_Log($writer);

$logger->info('Informational message');
```

You cannot specify the mode for existing stream resources. Doing so causes a `Zend_Log_Exception` to be thrown.

2.2. Writing to Databases

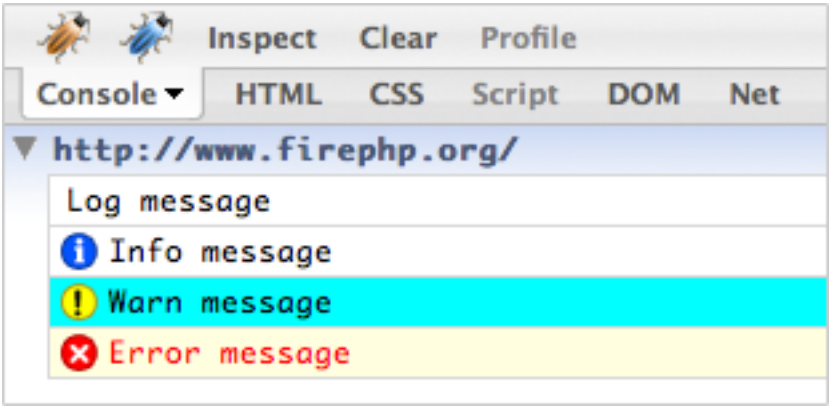
`Zend_Log_Writer_Db` writes log information to a database table using `Zend_Db`. The constructor of `Zend_Log_Writer_Db` receives a `Zend_Db_Adapter` instance, a table name, and a mapping of database columns to event data items:

```
$params = array ('host'     => '127.0.0.1',
                 'username' => 'malory',
                 'password' => '******',
                 'dbname'   => 'camelot');
$db = Zend_Db::factory('PDO_MYSQL', $params);

$columnMapping = array('lvl' => 'priority', 'msg' => 'message');
$writer = new Zend_Log_Writer_Db($db, 'log_table_name', $columnMapping);

$logger = new Zend_Log($writer);

$logger->info('Informational message');
```

The example above writes a single row of log data to the database table named `log_table_name` table. The database column named `lvl` receives the priority number and the column named `msg` receives the log message.

2.3. Writing to Firebug

`Zend_Log_Writer_Firebug` sends log data to the Firebug Console.



All data is sent via the `Zend_Wildfire_Channel_HttpHeaders` component which uses HTTP headers to ensure the page content is not disturbed. Debugging AJAX requests that require clean JSON and XML responses is possible with this approach.

Requirements:

- Firefox Browser ideally version 3 but version 2 is also supported.

- Firebug Firefox Extension which you can download from https://addons.mozilla.org/en-US/firefox/addon/1843.
- FirePHP Firefox Extension which you can download from https://addons.mozilla.org/en-US/firefox/addon/6149.

Example 522. Logging with Zend_Controller_Front

```
// Place this in your bootstrap file before dispatching your front controller
$writer = new Zend_Log_Writer_Firebug();
$logger = new Zend_Log($writer);

// Use this in your model, view and controller files
$logger->log('This is a log message!', Zend_Log::INFO);
```

Example 523. Logging without Zend_Controller_Front

```
$writer = new Zend_Log_Writer_Firebug();
$logger = new Zend_Log($writer);

$request = new Zend_Controller_Request_Http();
$response = new Zend_Controller_Response_Http();
$channel = Zend_Wildfire_Channel_HttpHeaders::getInstance();
$channel->setRequest($request);
$channel->setResponse($response);

// Start output buffering
ob_start();

// Now you can make calls to the logger

$logger->log('This is a log message!', Zend_Log::INFO);

// Flush log data to browser
$channel->flush();
$response->sendHeaders();
```

2.3.1. Setting Styles for Priorities

Built-in and user-defined priorities can be styled with the `setPriorityStyle()` method.

```
$logger->addPriority('FOO', 8);
$writer->setPriorityStyle(8, 'TRACE');
$logger->foo('Foo Message');
```

The default style for user-defined priorities can be set with the `setDefaultPriorityStyle()` method.

```
$writer->setDefaultPriorityStyle('TRACE');
```

The supported styles are as follows:

Table 100. Firebug Logging Styles

Style	Description
`LOG`	Displays a plain log message
`INFO`	Displays an info log message

Style	Description
WARN	Displays a warning log message
ERROR	Displays an error log message that increments Firebug's error count
TRACE	Displays a log message with an expandable stack trace
EXCEPTION	Displays an error long message with an expandable stack trace
TABLE	Displays a log message with an expandable table

2.3.2. Preparing data for Logging

While any PHP variable can be logged with the built-in priorities, some special formatting is required if using some of the more specialized log styles.

The `LOG`, `INFO`, `WARN`, `ERROR` and `TRACE` styles require no special formatting.

2.3.3. Exception Logging

To log a `Zend_Exception` simply pass the exception object to the logger. It does not matter which priority or style you have set as the exception is automatically recognized.

```
$exception = new Zend_Exception('Test exception');
$logger->err($exception);
```

2.3.4. Table Logging

You can also log data and format it in a table style. Columns are automatically recognized and the first row of data automatically becomes the header.

```
$writer->setPriorityStyle(8, 'TABLE');
$logger->addPriority('TABLE', 8);

$table = array('Summary line for the table',
             array(
                 array('Column 1', 'Column 2'),
                 array('Row 1 c 1',' Row 1 c 2'),
                 array('Row 2 c 1',' Row 2 c 2')
             )
            );
$logger->table($table);
```

2.4. Writing to Email

`Zend_Log_Writer_Mail` writes log entries in an email message by using `Zend_Mail`. The `Zend_Log_Writer_Mail` constructor takes a `Zend_Mail` object, and an optional `Zend_Layout` object.

The primary use case for `Zend_Log_Writer_Mail` is notifying developers, systems administrators, or any concerned parties of errors that might be occurring with PHP-based scripts. `Zend_Log_Writer_Mail` was born out of the idea that if something is broken, a human being needs to be alerted of it immediately so they can take corrective action.

Basic usage is outlined below:

```
$mail = new Zend_Mail();
$mail->setFrom('errors@example.org')
     ->addTo('project_developers@example.org');

$writer = new Zend_Log_Writer_Mail($mail);

// Set subject text for use; summary of number of errors is appended to the
// subject line before sending the message.
$writer->setSubjectPrependText('Errors with script foo.php');

// Only email warning level entries and higher.
$writer->addFilter(Zend_Log::WARN);

$log = new Zend_Log();
$log->addWriter($writer);

// Something bad happened!
$log->error('unable to connect to database');

// On writer shutdown, Zend_Mail::send() is triggered to send an email with
// all log entries at or above the Zend_Log filter level.
```

`Zend_Log_Writer_Mail` will render the email body as plain text by default.

One email is sent containing all log entries at or above the filter level. For example, if warning-level entries an up are to be emailed, and two warnings and five errors occur, the resulting email will contain a total of seven log entries.

2.4.1. Zend_Layout Usage

A `Zend_Layout` instance may be used to generate the HTML portion of a multipart email. If a `Zend_Layout` instance is in use, `Zend_Log_Writer_Mail` assumes that it is being used to render HTML and sets the body HTML for the message as the `Zend_Layout`-rendered value.

When using `Zend_Log_Writer_Mail` with a `Zend_Layout` instance, you have the option to set a custom formatter by using the `setLayoutFormatter()` method. If no `Zend_Layout`-specific entry formatter was specified, the formatter currently in use will be used. Full usage of `Zend_Layout` with a custom formatter is outlined below.

```
$mail = new Zend_Mail();
$mail->setFrom('errors@example.org')
     ->addTo('project_developers@example.org');
// Note that a subject line is not being set on the Zend_Mail instance!

// Use a simple Zend_Layout instance with its defaults.
$layout = new Zend_Layout();

// Create a formatter that wraps the entry in a listitem tag.
$layoutFormatter = new Zend_Log_Formatter_Simple(
    '<li>' . Zend_Log_Formatter_Simple::DEFAULT_FORMAT . '</li>'
);

$writer = new Zend_Log_Writer_Mail($mail, $layout);

// Apply the formatter for entries as rendered with Zend_Layout.
$writer->setLayoutFormatter($layoutFormatter);
$writer->setSubjectPrependText('Errors with script foo.php');
$writer->addFilter(Zend_Log::WARN);
```

```
$log = new Zend_Log();
$log->addWriter($writer);

// Something bad happened!
$log->error('unable to connect to database');

// On writer shutdown, Zend_Mail::send() is triggered to send an email with
// all log entries at or above the Zend_Log filter level. The email will
// contain both plain text and HTML parts.
```

2.4.2. Subject Line Error Level Summary

The `setSubjectPrependText()` method may be used in place of `Zend_Mail::setSubject()` to have the email subject line dynamically written before the email is sent. For example, if the subject prepend text reads "Errors from script", the subject of an email generated by `Zend_Log_Writer_Mail` with two warnings and five errors would be "Errors from script (warn = 2; error = 5)". If subject prepend text is not in use via `Zend_Log_Writer_Mail`, the `Zend_Mail` subject line, if any, is used.

2.4.3. Caveats

Sending log entries via email can be dangerous. If error conditions are being improperly handled by your script, or if you're misusing the error levels, you might find yourself in a situation where you are sending hundreds or thousands of emails to the recipients depending on the frequency of your errors.

At this time, `Zend_Log_Writer_Mail` does not provide any mechanism for throttling or otherwise batching up the messages. Such functionality should be implemented by the consumer if necessary.

Again, `Zend_Log_Writer_Mail`'s primary goal is to proactively notify a human being of error conditions. If those errors are being handled in a timely fashion, and safeguards are being put in place to prevent those circumstances in the future, then email-based notification of errors can be a valuable tool.

2.5. Writing to the System Log

`Zend_Log_Writer_Syslog` writes log entries to the system log (syslog). Internally, it proxies to PHP's `openlog()`, `closelog()`, and `syslog()` functions.

One useful case for `Zend_Log_Writer_Syslog` is for aggregating logs from clustered machines via the system log functionality. Many systems allow remote logging of system events, which allows system administrators to monitor a cluster of machines from a single log file.

By default, all syslog messages generated are prefixed with the string "Zend_Log". You may specify a different "application" name by which to identify such log messages by either passing the application name to the constructor or the application accessor:

```
// At instantiation, pass the "application" key in the options:
$writer = new Zend_Log_Writer_Syslog(array('application' => 'FooBar'));

// Any other time:
$writer->setApplicationName('BarBaz');
```

The system log also allows you to identify the "facility," or application type, logging the message; many system loggers will actually generate different log files per facility, which again aids administrators monitoring server activity.

You may specify the log facility either in the constructor or via an accessor. It should be one of the `openlog()` constants defined on the openlog() manual page.

```
// At instantiation, pass the "facility" key in the options:
$writer = new Zend_Log_Writer_Syslog(array('facility' => LOG_AUTH));

// Any other time:
$writer->setFacility(LOG_USER);
```

When logging, you may continue to use the default `Zend_Log` priority constants; internally, they are mapped to the appropriate `syslog()` priority constants.

2.6. Writing to the Zend Server Monitor

`Zend_Log_Writer_ZendMonitor` allows you to log events via Zend Server's Monitor API. This allows you to aggregate log messages for your entire application environment in a single location. Internally, it simply uses the `monitor_custom_event()` function from the Zend Monitor API.

One particularly useful feature of the Monitor API is that it allows you to specify arbitrary custom information alongside the log message. For instance, if you wish to log an exception, you can log not just the exception message, but pass the entire exception object to the function, and then inspect the object within the Zend Server event monitor.

Zend Monitor must be installed and enabled

In order to use this log writer, Zend Monitor must be both installed and enabled. However, it is designed such that if Zend Monitor is not detected, it will simply act as a `NULL` logger.

Instantiating the `ZendMonitor` log writer is trivial:

```
$writer = new Zend_Log_Writer_ZendMonitor();
$log    = new Zend_Log($writer);
```

Then, simply log messages as usual:

```
$log->info('This is a message');
```

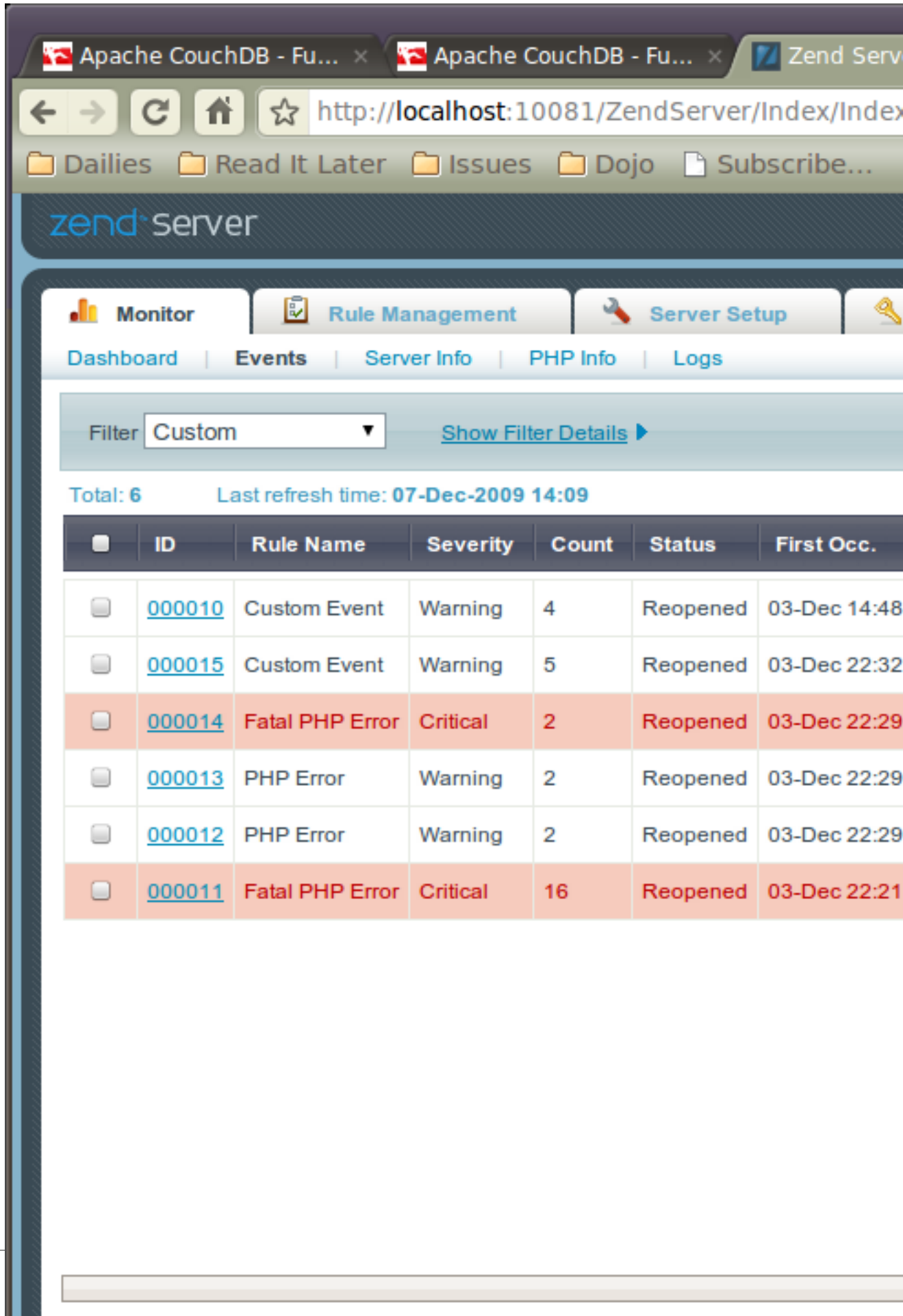
If you want to specify additional information to log with the event, pass that information in a second parameter:

```
$log->info('Exception occurred', $e);
```

The second parameter may be a scalar, object, or array; if you need to pass multiple pieces of information, the best way to do so is to pass an associative array.

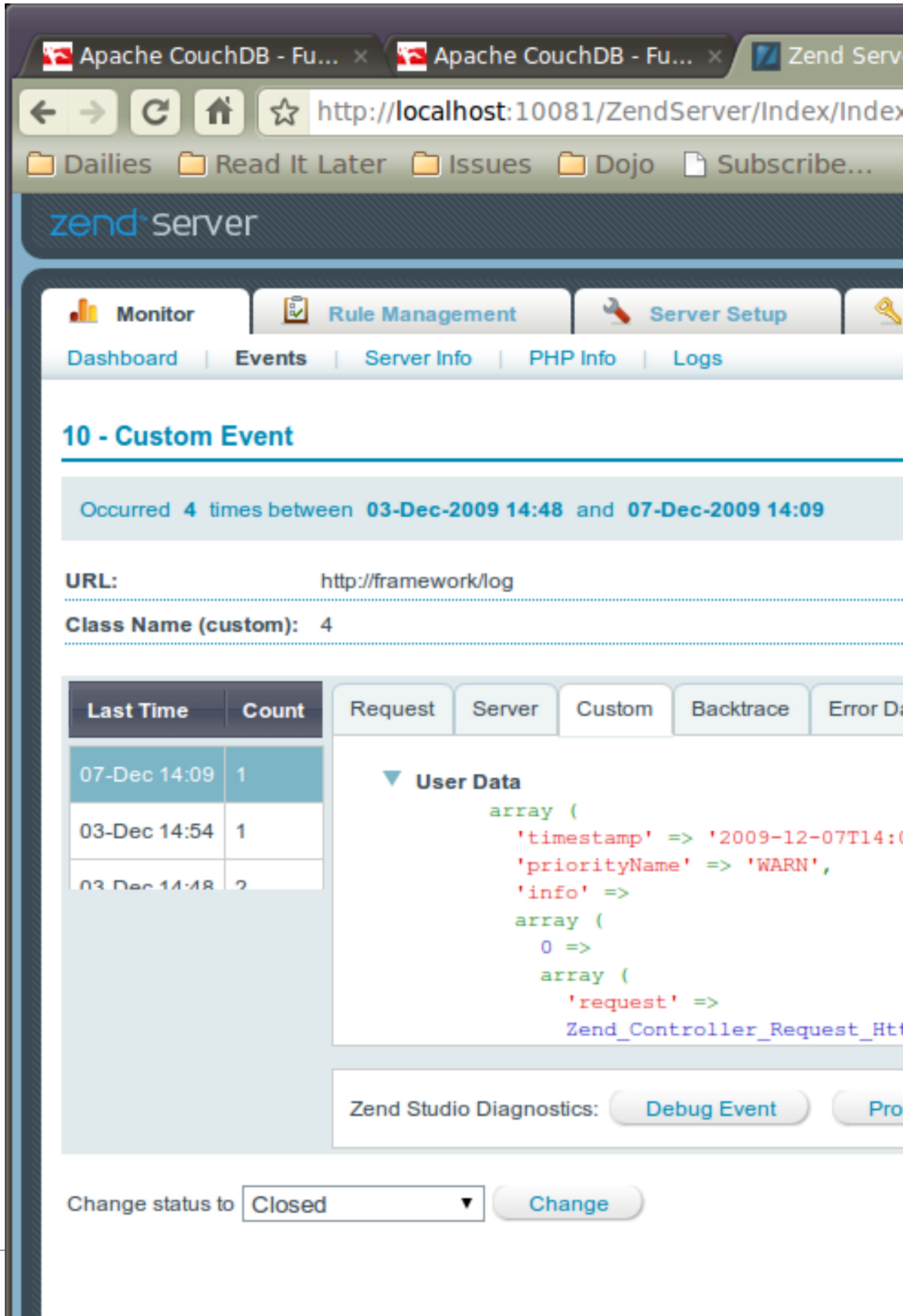
```
$log->info('Exception occurred', array(
    'request'   => $request,
    'exception' => $e,
));
```

Within Zend Server, your event is logged as a "custom event". From the "Monitor" tab, select the "Events" sub-item, and then filter on "Custom" to see custom events.


Apache CouchDB - Fu...
Apache CouchDB - Fu...
Zend Serv
http://localhost:10081/ZendServer/Index/Index
Dailies
Read It Later
Issues
Dojo
Subscribe...
zend server
Monitor
Rule Management
Server Setup
Dashboard
Events
Server Info
PHP Info
Logs
Filter
Custom
Show Filter Details
Total: 6
Last refresh time: 07-Dec-2009 14:09
ID
Rule Name
Severity
Count
Status
First Occ.
000010 Custom Event Warning 4 Reopened 03-Dec 14:48
000015 Custom Event Warning 5 Reopened 03-Dec 22:32
000014 Fatal PHP Error Critical 2 Reopened 03-Dec 22:29
000013 PHP Error Warning 2 Reopened 03-Dec 22:29
000012 PHP Error Warning 2 Reopened 03-Dec 22:29
000011 Fatal PHP Error Critical 16 Reopened 03-Dec 22:21

Events in Zend Server's Monitor dashboard

In this screenshot, the first two events listed are custom events logged via the `ZendMonitor` log writer. You may then click on an event to view all information related to it.


Apache CouchDB - Fu...
Apache CouchDB - Fu...
Zend Serv
http://localhost:10081/ZendServer/Index/Index
Dailies
Read It Later
Issues
Dojo
Subscribe...
zend server
Monitor
Rule Management
Server Setup
Dashboard
Events
Server Info
PHP Info
Logs
10 - Custom Event
Occurred 4 times between 03-Dec-2009 14:48 and 07-Dec-2009 14:09
URL: http://framework/log
Class Name (custom): 4
Last Time
Count
07-Dec 14:09
1
03-Dec 14:54
1
Request
Server
Custom
Backtrace
User Data
array (
'timestamp' => '2009-12-07T14:
'priorityName' => 'WARN',
'info' =>
array (
0 =>
array (
'request' =>
Zend_Controller_Request_Ht
Zend Studio Diagnostics:
Debug Event
Change status to
Closed
Change

Event detail in Zend Server's Monitor

Clicking on the "Custom" sub tab will detail any extra information you logged by passing the second argument to the logging method. This information will be logged as the `info` subkey; you can see that the request object was logged in this example.

Integration with Zend_Application

By default, the **zf.sh** and **zf.bat** commands add configuration for the Zend_Application log resource, which includes configuration for the `ZendMonitor` log writer. Additionally, the `ErrorController` uses the configured logger to log application exceptions -- providing you with Zend Monitor event integration by default.

As noted previously, if the Monitor API is not detected in your PHP installation, the logger will simply act as a `NULL` logger.

2.7. Stubbing Out the Writer

The `Zend_Log_Writer_Null` is a stub that does not write log data to anything. It is useful for disabling logging or stubbing out logging during tests:

```
$writer = new Zend_Log_Writer_Null;
$logger = new Zend_Log($writer);

// goes nowhere
$logger->info('Informational message');
```

2.8. Testing with the Mock

The `Zend_Log_Writer_Mock` is a very simple writer that records the raw data it receives in an array exposed as a public property.

```
$mock = new Zend_Log_Writer_Mock;
$logger = new Zend_Log($mock);

$logger->info('Informational message');

var_dump($mock->events[0]);

// Array
// (
//    [timestamp] => 2007-04-06T07:16:37-07:00
//    [message] => Informational message
//    [priority] => 6
//    [priorityName] => INFO
// )
```

To clear the events logged by the mock, simply set `$mock->events = array()`.

2.9. Compositing Writers

There is no composite Writer object. However, a Log instance can write to any number of Writers. To do this, use the `addWriter()` method:

```
$writer1 = new Zend_Log_Writer_Stream('/path/to/first/logfile');
$writer2 = new Zend_Log_Writer_Stream('/path/to/second/logfile');
```

```
$logger = new Zend_Log();
$logger->addWriter($writer1);
$logger->addWriter($writer2);

// goes to both writers
$logger->info('Informational message');
```

3. Formatters

A Formatter is an object that is responsible for taking an `event` array describing a log event and outputting a string with a formatted log line.

Some Writers are not line-oriented and cannot use a Formatter. An example is the Database Writer, which inserts the event items directly into database columns. For Writers that cannot support a Formatter, an exception is thrown if you attempt to set a Formatter.

3.1. Simple Formatting

`Zend_Log_Formatter_Simple` is the default formatter. It is configured automatically when you specify no formatter. The default configuration is equivalent to the following:

```
$format = '%timestamp% %priorityName% (%priority%): %message%' . PHP_EOL;
$formatter = new Zend_Log_Formatter_Simple($format);
```

A formatter is set on an individual Writer object using the Writer's `setFormatter()` method:

```
$writer = new Zend_Log_Writer_Stream('php://output');
$formatter = new Zend_Log_Formatter_Simple('hello %message%' . PHP_EOL);
$writer->setFormatter($formatter);

$logger = new Zend_Log();
$logger->addWriter($writer);

$logger->info('there');

// outputs "hello there"
```

The constructor of `Zend_Log_Formatter_Simple` accepts a single parameter: the format string. This string contains keys surrounded by percent signs (e.g. `%message%`). The format string may contain any key from the event data array. You can retrieve the default keys by using the DEFAULT_FORMAT constant from `Zend_Log_Formatter_Simple`.

3.2. Formatting to XML

`Zend_Log_Formatter_Xml` formats log data into XML strings. By default, it automatically logs all items in the event data array:

```
$writer = new Zend_Log_Writer_Stream('php://output');
$formatter = new Zend_Log_Formatter_Xml();
$writer->setFormatter($formatter);

$logger = new Zend_Log();
$logger->addWriter($writer);

$logger->info('informational message');
```

The code above outputs the following XML (space added for clarity):

```
<logEntry>
  <timestamp>2007-04-06T07:24:37-07:00</timestamp>
  <message>informational message</message>
  <priority>6</priority>
  <priorityName>INFO</priorityName>
</logEntry>
```

It's possible to customize the root element as well as specify a mapping of XML elements to the items in the event data array. The constructor of `Zend_Log_Formatter_Xml` accepts a string with the name of the root element as the first parameter and an associative array with the element mapping as the second parameter:

```
$writer = new Zend_Log_Writer_Stream('php://output');
$formatter = new Zend_Log_Formatter_Xml('log',
                                        array('msg' => 'message',
                                              'level' => 'priorityName')
                                       );
$writer->setFormatter($formatter);

$logger = new Zend_Log();
$logger->addWriter($writer);

$logger->info('informational message');
```

The code above changes the root element from its default of `logEntry` to `log`. It also maps the element `msg` to the event data item `message`. This results in the following output:

```
<log>
  <msg>informational message</msg>
  <level>INFO</level>
</log>
```

4. Filters

A Filter object blocks a message from being written to the log.

4.1. Filtering for All Writers

To filter before all writers, you can add any number of Filters to a Log object using the `addFilter()` method:

```
$logger = new Zend_Log();

$writer = new Zend_Log_Writer_Stream('php://output');
$logger->addWriter($writer);

$filter = new Zend_Log_Filter_Priority(Zend_Log::CRIT);
$logger->addFilter($filter);

// blocked
$logger->info('Informational message');

// logged
$logger->emerg('Emergency message');
```

When you add one or more Filters to the Log object, the message must pass through all of the Filters before any Writers receives it.

4.2. Filtering for a Writer Instance

To filter only on a specific Writer instance, use the `addFilter` method of that Writer:

```
$logger = new Zend_Log();

$writer1 = new Zend_Log_Writer_Stream('/path/to/first/logfile');
$logger->addWriter($writer1);

$writer2 = new Zend_Log_Writer_Stream('/path/to/second/logfile');
$logger->addWriter($writer2);

// add a filter only to writer2
$filter = new Zend_Log_Filter_Priority(Zend_Log::CRIT);
$writer2->addFilter($filter);

// logged to writer1, blocked from writer2
$logger->info('Informational message');

// logged by both writers
$logger->emerg('Emergency message');
```

5. Using the Factory to Create a Log

In addition to direct instantiation, you may also use the static `factory()` method to instantiate a Log instance, as well as to configure attached writers and their filters. Using the factory, you can attach zero or more writers. Configuration may be passed as either an array or a `Zend_Config` instance.

As an example:

```
$logger = Zend_Log::factory(array(
    array(
        'writerName'   => 'Stream',
        'writerParams' => array(
            'stream'   => '/tmp/zend.log',
        ),
        'filterName'   => 'Priority',
        'filterParams' => array(
            'priority' => Zend_Log::WARN,
        ),
    ),
    array(
        'writerName'   => 'Firebug',
        'filterName'   => 'Priority',
        'filterParams' => array(
            'priority' => Zend_Log::INFO,
        ),
    ),
));
```

The above will instantiate a logger with two writers, one for writing to a local file, another for sending data to Firebug. Each has an attached priority filter, with different maximum priorities.

Each writer can be defined with the following keys:

writerName (required)	The "short" name of a log writer; the name of the log writer minus the leading class prefix/namespace. See the "writerNamespace" entry below for more details. Examples: "Mock", "Stream", "Firebug".
writerParams (optional)	An associative array of parameters to use when instantiating the log writer. Each log writer's `factory()` method will map these to constructor arguments, as noted below.
writerNamespace (optional)	The class prefix/namespace to use when constructing the final log writer classname. By default, if this is not provided, "Zend_Log_Writer" is assumed; however, you can pass your own namespace if you are using a custom log writer.
filterName (optional)	The "short" name of a filter to use with the given log writer; the name of the filter minus the leading class prefix/namespace. See the "filterNamespace" entry below for more details. Examples: "Message", "Priority".
filterParams (optional)	An associative array of parameters to use when instantiating the log filter. Each log filter's `factory()` method will map these to constructor arguments, as noted below.
filterNamespace (optional)	The class prefix/namespace to use when constructing the final log filter classname. By default, if this is not provided, "Zend_Log_Filter" is assumed; however, you can pass your own namespace if you are using a custom log filter.

Each writer and each filter has specific options.

5.1. Writer Options

5.1.1. Zend_Log_Writer_Db Options

db	A `Zend_Db_Adapter` instance.
table	The name of the table in the RDBMS that will contain log entries.
columnMap	An associative array mapping database table column names to log event fields.

5.1.2. Zend_Log_Writer_Firebug Options

This log writer takes no options; any provided will be ignored.

5.1.3. Zend_Log_Writer_Mail Options

`Zend_Log_Writer_Mail` currently (as of 1.10) does not implement a factory, and will raise an exception if you attempt to instantiate it via `Zend_Log::factory()`.

5.1.4. Zend_Log_Writer_Mock Options

This log writer takes no options; any provided will be ignored.

5.1.5. Zend_Log_Writer_Null Options

This log writer takes no options; any provided will be ignored.

5.1.6. Zend_Log_Writer_Stream Options

stream|url A valid PHP stream identifier to which to log.

mode The I/O mode with which to log; defaults to "a", for "append".

5.1.7. Zend_Log_Writer_Syslog Options

application Application name used by the syslog writer.

facility Facility used by the syslog writer.

5.1.8. Zend_Log_Writer_ZendMonitor Options

This log writer takes no options; any provided will be ignored.

5.2. Filter Options

5.2.1. Zend_Log_Filter_Message Options

regexp Regular expression that must be matched in order to log a message.

5.2.2. Zend_Log_Filter_Priority Options

priority The maximum priority level by which messages will be logged.

operator The comparison operator by which to do priority comparisons; defaults to "<=".

5.2.3. Zend_Log_Writer_Suppress Options

This log filter takes no options; any provided will be ignored.

5.3. Creating Configurable Writers and Filters

If you find yourself needing to write your own log writers and/or filters, you can make them compatible with `Zend_Log::factory()` very easily.

At the minimum, you need to implement `Zend_Log_FactoryInterface`, which expects a static `factory()` method that accepts a single argument, `$config`, which may be either an array or `Zend_Config` object. If your log writer extends `Zend_Log_Writer_Abstract`, or your log filter extends `Zend_Log_Filter_Abstract`, you will pick this up for free.

Then, simply define mappings between the accepted configuration and any constructor arguments. As an example:

```
class My_Log_Writer_Foo extends Zend_Log_Writer_Abstract
{
    public function __construct($bar, $baz)
    {
        // ...
    }

    public static function factory($config)
    {
        if ($config instanceof Zend_Config) {
            $config = $config->toArray();
        }
        if (!is_array($config)) {
```

```
            throw new Exception(
                'factory expects an array or Zend_Config instance'
            );
        }

        $default = array(
            'bar' => null,
            'baz' => null,
        );
        $config = array_merge($default, $config);

        return new self(
            $config['bar'],
            $config['baz']
        );
    }
}
```

Alternately, you could call appropriate setters after instantiation, but prior to returning the instance:

```
class My_Log_Writer_Foo extends Zend_Log_Writer_Abstract
{
    public function __construct($bar = null, $baz = null)
    {
        // ...
    }

    public function setBar($value)
    {
        // ...
    }

    public function setBaz($value)
    {
        // ...
    }

    public static function factory($config)
    {
        if ($config instanceof Zend_Config) {
            $config = $config->toArray();
        }
        if (!is_array($config)) {
            throw new Exception(
                'factory expects an array or Zend_Config instance'
            );
        }

        $writer = new self();
        if (isset($config['bar'])) {
            $writer->setBar($config['bar']);
        }
        if (isset($config['baz'])) {
            $writer->setBaz($config['baz']);
        }
        return $writer;
    }
}
```

Zend_Mail

1. Introduction

1.1. Getting started

`Zend_Mail` provides generalized functionality to compose and send both text and MIME-compliant multipart e-mail messages. Mail can be sent with `Zend_Mail` via the default `Zend_Mail_Transport_Sendmail` transport or via `Zend_Mail_Transport_Smtp`.

Example 524. Simple E-Mail with Zend_Mail

A simple e-mail consists of some recipients, a subject, a body and a sender. To send such a mail using `Zend_Mail_Transport_Sendmail`, do the following:

```
$mail = new Zend_Mail();
$mail->setBodyText('This is the text of the mail.');
$mail->setFrom('somebody@example.com', 'Some Sender');
$mail->addTo('somebody_else@example.com', 'Some Recipient');
$mail->setSubject('TestSubject');
$mail->send();
```

Minimum definitions

In order to send an e-mail with `Zend_Mail` you have to specify at least one recipient, a sender (e.g., with `setFrom()`), and a message body (text and/or HTML).

For most mail attributes there are "get" methods to read the information stored in the mail object. For further details, please refer to the API documentation. A special one is `getRecipients()`. It returns an array with all recipient e-mail addresses that were added prior to the method call.

For security reasons, `Zend_Mail` filters all header fields to prevent header injection with newline (`\n`) characters. Double quotation is changed to single quotation and angle brackets to square brackets in the name of sender and recipients. If the marks are in email address, the marks will be removed.

You also can use most methods of the `Zend_Mail` object with a convenient fluent interface.

```
$mail = new Zend_Mail();
$mail->setBodyText('This is the text of the mail.')
    ->setFrom('somebody@example.com', 'Some Sender')
    ->addTo('somebody_else@example.com', 'Some Recipient')
    ->setSubject('TestSubject')
    ->send();
```

1.2. Configuring the default sendmail transport

The default transport for a `Zend_Mail` instance is `Zend_Mail_Transport_Sendmail`. It is essentially a wrapper to the PHP `mail()` function. If you wish to pass additional parameters to the `mail()` function, simply create a new transport instance and pass your parameters to the

constructor. The new transport instance can then act as the default `Zend_Mail` transport, or it can be passed to the `send()` method of `Zend_Mail`.

Example 525. Passing additional parameters to the Zend_Mail_Transport_Sendmail transport

This example shows how to change the Return-Path of the `mail()` function.

```
$tr = new Zend_Mail_Transport_Sendmail('-freturn_to_me@example.com');
Zend_Mail::setDefaultTransport($tr);

$mail = new Zend_Mail();
$mail->setBodyText('This is the text of the mail.');
$mail->setFrom('somebody@example.com', 'Some Sender');
$mail->addTo('somebody_else@example.com', 'Some Recipient');
$mail->setSubject('TestSubject');
$mail->send();
```

Safe mode restrictions

The optional additional parameters will be cause the `mail()` function to fail if PHP is running in safe mode.

Sendmail Transport and Windows

As the PHP manual states the `mail()` function has different behaviour on Windows and on *nix based systems. Using the Sendmail Transport on Windows will not work in combination with `addBcc()`. The `mail()` function will sent to the BCC recipient such that all the other recipients can see him as recipient!

Therefore if you want to use BCC on a windows server, use the SMTP transport for sending!

2. Sending via SMTP

To send mail via SMTP, `Zend_Mail_Transport_Smtp` needs to be created and registered with `Zend_Mail` before the `send()` method is called. For all remaining `Zend_Mail::send()` calls in the current script, the SMTP transport will then be used:

Example 526. Sending E-Mail via SMTP

```
$tr = new Zend_Mail_Transport_Smtp('mail.example.com');
Zend_Mail::setDefaultTransport($tr);
```

The `setDefaultTransport()` method and the constructor of `Zend_Mail_Transport_Smtp` are not expensive. These two lines can be processed at script setup time (e.g., config.inc or similar) to configure the behavior of the `Zend_Mail` class for the rest of the script. This keeps configuration information out of the application logic - whether mail is sent via SMTP or `mail()`, what mail server is used, etc.

3. Sending Multiple Mails per SMTP Connection

By default, a single SMTP transport creates a single connection and re-uses it for the lifetime of the script execution. You may send multiple e-mails through this SMTP connection. A RSET command is issued before each delivery to ensure the correct SMTP handshake is followed.

Optionally, you can also define a default From email address and name, as well as a default reply-to header. This can be done through the static methods `setDefaultFrom()` and `setDefaultReplyTo()`. These defaults will be used when you don't specify a From/Reply-to Address or -Name until the defaults are reset (cleared). Resetting the defaults can be done through the use of the `clearDefaultFrom()` and `clearDefaultReplyTo`.

Example 527. Sending Multiple Mails per SMTP Connection

```
// Create transport
$config = array('name' => 'sender.example.com');
$transport = new Zend_Mail_Transport_Smtp('mail.example.com', $config);

// Set From & Reply-To address and name for all emails to send.
Zend_Mail::setDefaultFrom('sender@example.com', 'John Doe');
Zend_Mail::setDefaultReplyTo('replyto@example.com','Jane Doe');

// Loop through messages
for ($i = 0; $i < 5; $i++) {
    $mail = new Zend_Mail();
    $mail->addTo('studio@example.com', 'Test');

    $mail->setSubject(
        'Demonstration - Sending Multiple Mails per SMTP Connection'
    );
    $mail->setBodyText('...Your message here...');
    $mail->send($transport);
}

// Reset defaults
Zend_Mail::clearDefaultFrom();
Zend_Mail::clearDefaultReplyTo();
```

If you wish to have a separate connection for each mail delivery, you will need to create and destroy your transport before and after each `send()` method is called. Or alternatively, you can manipulate the connection between each delivery by accessing the transport's protocol object.

Example 528. Manually controlling the transport connection

```
// Create transport
$transport = new Zend_Mail_Transport_Smtp();

$protocol = new Zend_Mail_Protocol_Smtp('mail.example.com');
$protocol->connect();
$protocol->helo('sender.example.com');

$transport->setConnection($protocol);

// Loop through messages
for ($i = 0; $i < 5; $i++) {
    $mail = new Zend_Mail();
    $mail->addTo('studio@example.com', 'Test');
    $mail->setFrom('studio@example.com', 'Test');
    $mail->setSubject(
        'Demonstration - Sending Multiple Mails per SMTP Connection'
    );
    $mail->setBodyText('...Your message here...');

    // Manually control the connection
    $protocol->rset();
    $mail->send($transport);
}

$protocol->quit();
$protocol->disconnect();
```

4. Using Different Transports

In case you want to send different e-mails through different connections, you can also pass the transport object directly to `send()` without a prior call to `setDefaultTransport()`. The passed object will override the default transport for the actual `send()` request.

Example 529. Using Different Transports

```
$mail = new Zend_Mail();
// build message...
$tr1 = new Zend_Mail_Transport_Smtp('server@example.com');
$tr2 = new Zend_Mail_Transport_Smtp('other_server@example.com');
$mail->send($tr1);
$mail->send($tr2);
$mail->send();  // use default again
```

Additional transports

Additional transports can be written by implementing `Zend_Mail_Transport_Interface`.

5. HTML E-Mail

To send an e-mail in HTML format, set the body using the method `setBodyHTML()` instead of `setBodyText()`. The MIME content type will automatically be set to `text/html` then. If you use both HTML and Text bodies, a multipart/alternative MIME message will automatically be generated:

Example 530. Sending HTML E-Mail

```
$mail = new Zend_Mail();
$mail->setBodyText('My Nice Test Text');
$mail->setBodyHtml('My Nice <b>Test</b> Text');
$mail->setFrom('somebody@example.com', 'Some Sender');
$mail->addTo('somebody_else@example.com', 'Some Recipient');
$mail->setSubject('TestSubject');
$mail->send();
```

6. Attachments

Files can be attached to an e-mail using the `createAttachment()` method. The default behavior of `Zend_Mail` is to assume the attachment is a binary object (application/octet-stream), that it should be transferred with base64 encoding, and that it is handled as an attachment. These assumptions can be overridden by passing more parameters to `createAttachment()`:

Example 531. E-Mail Messages with Attachments

```
$mail = new Zend_Mail();
// build message...
$mail->createAttachment($someBinaryString);
$mail->createAttachment($myImage,
                        'image/gif',
                        Zend_Mime::DISPOSITION_INLINE,
                        Zend_Mime::ENCODING_8BIT);
```

If you want more control over the MIME part generated for this attachment you can use the return value of `createAttachment()` to modify its attributes. The `createAttachment()` method returns a `Zend_Mime_Part` object:

```
$mail = new Zend_Mail();

$at = $mail->createAttachment($myImage);
$at->type        = 'image/gif';
$at->disposition = Zend_Mime::DISPOSITION_INLINE;
$at->encoding    = Zend_Mime::ENCODING_8BIT;
$at->filename    = 'test.gif';

$mail->send();
```

An alternative is to create an instance of `Zend_Mime_Part` and add it with `addAttachment()`:

```
$mail = new Zend_Mail();

$at = new Zend_Mime_Part($myImage);
$at->type        = 'image/gif';
$at->disposition = Zend_Mime::DISPOSITION_INLINE;
$at->encoding    = Zend_Mime::ENCODING_8BIT;
$at->filename    = 'test.gif';

$mail->addAttachment($at);

$mail->send();
```

7. Adding Recipients

Recipients can be added in three ways:

- `addTo()`: Adds a recipient to the mail with a "To" header
- `addCc()`: Adds a recipient to the mail with a "Cc" header
- `addBcc()`: Adds a recipient to the mail not visible in the header

`getRecipients()` serves list of the recipients. `clearRecipients()` clears the list.

Additional parameter

`addTo()` and `addCc()` accept a second optional parameter that is used as a human-readable name of the recipient for the header. Double quotation is changed to single quotation and angle brackets to square brackets in the parameter.

Optional Usage

All three of these methods can also accept an array of email addresses to add instead of one at a time. In the case of `addTo()` and `addCc()`, they can be associative arrays where the key is the human readable name for the recipient.

8. Controlling the MIME Boundary

In a multipart message, a MIME boundary for separating the different parts of the message is normally generated at random. In some cases, however, you might want to specify the MIME boundary that is used. This can be done using the `setMimeBoundary()` method, as in the following example:

Example 532. Changing the MIME Boundary

```
$mail = new Zend_Mail();
$mail->setMimeBoundary('=_' . md5(microtime(1) . $someId++));
// build message...
```

9. Additional Headers

`Zend_Mail` provides several methods to set additional Mail Headers:

- `setReplyTo($email, $name=null)`: sets the Reply-To: header.
- `setDate($date = null)`: sets the Date: header. This method uses current time stamp by default. Or You can pass time stamp, date string or `Zend_Date` instance to this method.
- `setMessageId($id = true)`: sets the Message-Id: header. This method can generate message ID automatically by default. Or You can pass your message ID string to this method. This method call `createMessageId()` internally.

Return-Path

If you set Return-Path on your mail, see Configuring sendmail transport. Unfortunately, `setReturnPath($email)` method does not perform this purpose.

Furthermore, arbitrary mail headers can be set by using the `addHeader()` method. It requires two parameters containing the name and the value of the header field. A third optional parameter determines if the header should have only one or multiple values:

Example 533. Adding E-Mail Message Headers

```
$mail = new Zend_Mail();
$mail->addHeader('X-MailGenerator', 'MyCoolApplication');
$mail->addHeader('X-greetingsTo', 'Mom', true); // multiple values
$mail->addHeader('X-greetingsTo', 'Dad', true);
```

10. Character Sets

`Zend_Mail` does not check for the correct character set of the mail parts. When instantiating `Zend_Mail`, a charset for the e-mail itself may be given. It defaults to `iso-8859-1`. The application has to make sure that all parts added to that mail object have their content encoded in the correct character set. When creating a new mail part, a different charset can be given for each part.

Only in text format

Character sets are only applicable for message parts in text format.

Example 534. Usage in CJK languages

The following example is how to use `Zend_Mail` in Japanese. This is one of CJK (aka CJKV) languages. If you use Chinese, you may use HZ-GB-2312 instead of ISO-2022-JP.

```
//We suppose that character encoding of strings is UTF-8 on PHP script.
function myConvert($string) {
    return mb_convert_encoding($string, 'ISO-2022-JP', 'UTF-8');
}

$mail = new Zend_Mail('ISO-2022-JP');
//In this case, You can use ENCODING_7BIT because the ISO-2022-JP does not use MSB.
$mail->setBodyText(myConvert('This is the text of the mail.'), null, Zend_Mime::ENCODINC
$mail->setHeaderEncoding(Zend_Mime::ENCODING_BASE64);
$mail->setFrom('somebody@example.com', myConvert('Some Sender'));
$mail->addTo('somebody_else@example.com', myConvert('Some Recipient'));
$mail->setSubject(myConvert('TestSubject'));
$mail->send();
```

11. Encoding

Text and HTML message bodies are encoded with the quotedprintable mechanism by default. Message headers are also encoded with the quotedprintable mechanism if you do not specify base64 in `setHeaderEncoding()`. If you use language that is not Roman letters-based, the base64 would be more suitable. All other attachments are encoded via base64 if no other encoding is given in the `addAttachment()` call or assigned to the MIME part object later. 7Bit and 8Bit encoding currently only pass on the binary content data.

Header Encoding, especially the encoding of the subject, is a tricky topic. `Zend_Mime` currently implements its own algorithm to encode quoted printable headers according to RFC-2045. This due to the problems of `iconv_mime_encode` and `mb_encode_mimeheader` with regards to

certain charsets. This algorithm only breaks the header at spaces, which might lead to headers that far exceed the suggested length of 76 chars. For this cases it is suggested to switch to BASE64 header encoding same as the following example describes:

```
// By default Zend_Mime::ENCODING_QUOTEDPRINTABLE
$mail = new Zend_Mail('KOI8-R');

// Reset to Base64 Encoding because Russian expressed in KOI8-R is
// different from Roman letters-based languages greatly.
$mail->setHeaderEncoding(Zend_Mime::ENCODING_BASE64);
```

`Zend_Mail_Transport_Smtp` encodes lines starting with one dot or two dots so that the mail does not violate the SMTP protocol.

12. SMTP Authentication

`Zend_Mail` supports the use of SMTP authentication, which can be enabled be passing the 'auth' parameter to the configuration array in the `Zend_Mail_Transport_Smtp` constructor. The available built-in authentication methods are PLAIN, LOGIN and CRAM-MD5 which all expect a 'username' and 'password' value in the configuration array.

Example 535. Enabling authentication within Zend_Mail_Transport_Smtp

```
$config = array('auth' => 'login',
                'username' => 'myusername',
                'password' => 'password');

$transport = new Zend_Mail_Transport_Smtp('mail.server.com', $config);

$mail = new Zend_Mail();
$mail->setBodyText('This is the text of the mail.');
$mail->setFrom('sender@test.com', 'Some Sender');
$mail->addTo('recipient@test.com', 'Some Recipient');
$mail->setSubject('TestSubject');
$mail->send($transport);
```

Authentication types

The authentication type is case-insensitive but has no punctuation. E.g. to use CRAM-MD5 you would pass 'auth' => 'crammd5' in the `Zend_Mail_Transport_Smtp` constructor.

13. Securing SMTP Transport

`Zend_Mail` also supports the use of either TLS or SSL to secure a SMTP connection. This can be enabled be passing the 'ssl' parameter to the configuration array in the `Zend_Mail_Transport_Smtp` constructor with a value of either 'ssl' or 'tls'. A port can optionally be supplied, otherwise it defaults to 25 for TLS or 465 for SSL.

Example 536. Enabling a secure connection within Zend_Mail_Transport_Smtp

```
$config = array('ssl' => 'tls',
                'port' => 25); // Optional port number supplied

$transport = new Zend_Mail_Transport_Smtp('mail.server.com', $config);

$mail = new Zend_Mail();
$mail->setBodyText('This is the text of the mail.');
$mail->setFrom('sender@test.com', 'Some Sender');
$mail->addTo('recipient@test.com', 'Some Recipient');
$mail->setSubject('TestSubject');
$mail->send($transport);
```

14. Reading Mail Messages

`Zend_Mail` can read mail messages from several local or remote mail storages. All of them have the same basic API to count and fetch messages and some of them implement additional interfaces for not so common features. For a feature overview of the implemented storages see the following table.

Table 101. Mail Read Feature Overview

Feature	Mbox	Maildir	Pop3	IMAP
Storage type	local	local	remote	remote
Fetch message	Yes	Yes	Yes	Yes
Fetch MIME-part	emulated	emulated	emulated	emulated
Folders	Yes	Yes	No	Yes
Create message/ folder	No	todo	No	todo
Flags	No	Yes	No	Yes
Quota	No	Yes	No	No

14.1. Simple example using Pop3

```
$mail = new Zend_Mail_Storage_Pop3(array('host'     => 'localhost',
                                         'user'     => 'test',
                                         'password' => 'test'));

echo $mail->countMessages() . " messages found\n";
foreach ($mail as $message) {
    echo "Mail from '{$message->from}': {$message->subject}\n";
}
```

14.2. Opening a local storage

Mbox and Maildir are the two supported formats for local mail storages, both in their most simple formats.

If you want to read from a Mbox file you only need to give the filename to the constructor of `Zend_Mail_Storage_Mbox`:

```
$mail = new Zend_Mail_Storage_Mbox(array('filename' =>
```

```
                                                '/home/test/mail/inbox'));
```

Maildir is very similar but needs a dirname:

```
$mail = new Zend_Mail_Storage_Maildir(array('dirname' =>
                                                   '/home/test/mail/'));
```

Both constructors throw a `Zend_Mail_Exception` if the storage can't be read.

14.3. Opening a remote storage

For remote storages the two most popular protocols are supported: Pop3 and Imap. Both need at least a host and a user to connect and login. The default password is an empty string, the default port as given in the protocol RFC.

```
// connecting with Pop3
$mail = new Zend_Mail_Storage_Pop3(array('host'     => 'example.com',
                                         'user'     => 'test',
                                         'password' => 'test'));

// connecting with Imap
$mail = new Zend_Mail_Storage_Imap(array('host'     => 'example.com',
                                         'user'     => 'test',
                                         'password' => 'test'));

// example for a none standard port
$mail = new Zend_Mail_Storage_Pop3(array('host'     => 'example.com',
                                         'port'     => 1120
                                         'user'     => 'test',
                                         'password' => 'test'));
```

For both storages SSL and TLS are supported. If you use SSL the default port changes as given in the RFC.

```
// examples for Zend_Mail_Storage_Pop3, same works for Zend_Mail_Storage_Imap

// use SSL on different port (default is 995 for Pop3 and 993 for Imap)
$mail = new Zend_Mail_Storage_Pop3(array('host'     => 'example.com',
                                         'user'     => 'test',
                                         'password' => 'test',
                                         'ssl'      => 'SSL'));

// use TLS
$mail = new Zend_Mail_Storage_Pop3(array('host'     => 'example.com',
                                         'user'     => 'test',
                                         'password' => 'test',
                                         'ssl'      => 'TLS'));
```

Both constructors can throw `Zend_Mail_Exception` or `Zend_Mail_Protocol_Exception` (extends `Zend_Mail_Exception`), depending on the type of error.

14.4. Fetching messages and simple methods

Messages can be fetched after you've opened the storage . You need the message number, which is a counter starting with 1 for the first message. To fetch the message, you use the method `getMessage()`:

```
$message = $mail->getMessage($messageNum);
```

Array access is also supported, but this access method won't supported any additional parameters that could be added to `getMessage()`. As long as you don't mind, and can live with the default values, you may use:

```
$message = $mail[$messageNum];
```

For iterating over all messages the Iterator interface is implemented:

```
foreach ($mail as $messageNum => $message) {
    // do stuff ...
}
```

To count the messages in the storage, you can either use the method `countMessages()` or use array access:

```
// method
$maxMessage = $mail->countMessages();

// array access
$maxMessage = count($mail);
```

To remove a mail, you use the method `removeMessage()` or again array access:

```
// method
$mail->removeMessage($messageNum);

// array access
unset($mail[$messageNum]);
```

14.5. Working with messages

After you fetch the messages with `getMessage()` you want to fetch headers, the content or single parts of a multipart message. All headers can be accessed via properties or the method `getHeader()` if you want more control or have unusual header names. The header names are lower-cased internally, thus the case of the header name in the mail message doesn't matter. Also headers with a dash can be written in camel-case. If no header is found for both notations an exception is thrown. To encounter this the method `headerExists()` can be used to check the existance of a header.

```
// get the message object
$message = $mail->getMessage(1);

// output subject of message
echo $message->subject . "\n";

// get content-type header
$type = $message->contentType;

// check if CC isset:
if( isset($message->cc) ) { // or $message->headerExists('cc');
    $cc = $message->cc;
}
```

If you have multiple headers with the same name- i.e. the Received headers- you might want an array instead of a string. In this case, use the `getHeader()` method.

```
// get header as property - the result is always a string,
// with new lines between the single occurrences in the message
$received = $message->received;

// the same via getHeader() method
$received = $message->getHeader('received', 'string');

// better an array with a single entry for every occurrences
$received = $message->getHeader('received', 'array');
foreach ($received as $line) {
    // do stuff
}

// if you don't define a format you'll get the internal representation
// (string for single headers, array for multiple)
$received = $message->getHeader('received');
if (is_string($received)) {
    // only one received header found in message
}
```

The method `getHeaders()` returns all headers as array with the lower-cased name as key and the value as and array for multiple headers or as string for single headers.

```
// dump all headers
foreach ($message->getHeaders() as $name => $value) {
    if (is_string($value)) {
        echo "$name: $value\n";
        continue;
    }
    foreach ($value as $entry) {
        echo "$name: $entry\n";
    }
}
```

If you don't have a multipart message, fetching the content is easily done via `getContent()`. Unlike the headers, the content is only fetched when needed (aka late-fetch).

```
// output message content for HTML
echo '<pre>';
echo $message->getContent();
echo '</pre>';
```

Checking for multipart messages is done with the method `isMultipart()`. If you have multipart message you can get an instance of `Zend_Mail_Part` with the method `getPart()`. `Zend_Mail_Part` is the base class of `Zend_Mail_Message`, so you have the same methods: `getHeader()`, `getHeaders()`, `getContent()`, `getPart()`, `isMultipart` and the properties for headers.

```
// get the first none multipart part
$part = $message;
while ($part->isMultipart()) {
    $part = $message->getPart(1);
}
echo 'Type of this part is ' . strtok($part->contentType, ';') . "\n";
```

```
echo "Content:\n";
echo $part->getContent();
```

`Zend_Mail_Part` also implements `RecursiveIterator`, which makes it easy to scan through all parts. And for easy output, it also implements the magic method `__toString()`, which returns the content.

```
// output first text/plain part
$foundPart = null;
foreach (new RecursiveIteratorIterator($mail->getMessage(1)) as $part) {
    try {
        if (strtok($part->contentType, ';') == 'text/plain') {
            $foundPart = $part;
            break;
        }
    } catch (Zend_Mail_Exception $e) {
        // ignore
    }
}
if (!$foundPart) {
    echo 'no plain text part found';
} else {
    echo "plain text part: \n" . $foundPart;
}
```

14.6. Checking for flags

Maildir and IMAP support storing flags. The class `Zend_Mail_Storage` has constants for all known maildir and IMAP system flags, named `Zend_Mail_Storage::FLAG_<flagname>`. To check for flags `Zend_Mail_Message` has a method called `hasFlag()`. With `getFlags()` you'll get all set flags.

```
// find unread messages
echo "Unread mails:\n";
foreach ($mail as $message) {
    if ($message->hasFlag(Zend_Mail_Storage::FLAG_SEEN)) {
        continue;
    }
    // mark recent/new mails
    if ($message->hasFlag(Zend_Mail_Storage::FLAG_RECENT)) {
        echo '! ';
    } else {
        echo '  ';
    }
    echo $message->subject . "\n";
}

// check for known flags
$flags = $message->getFlags();
echo "Message is flagged as: ";
foreach ($flags as $flag) {
    switch ($flag) {
        case Zend_Mail_Storage::FLAG_ANSWERED:
            echo 'Answered ';
            break;
        case Zend_Mail_Storage::FLAG_FLAGGED:
            echo 'Flagged ';
            break;
```

```
        // ...
        // check for other flags
        // ...

        default:
            echo $flag . '(unknown flag) ';
    }
}
```

As IMAP allows user or client defined flags, you could get flags that don't have a constant in `Zend_Mail_Storage`. Instead, they are returned as strings and can be checked the same way with `hasFlag()`.

```
// check message for client defined flags $IsSpam, $SpamTested
if (!$message->hasFlag('$SpamTested')) {
    echo 'message has not been tested for spam';
} else if ($message->hasFlag('$IsSpam')) {
    echo 'this message is spam';
} else {
    echo 'this message is ham';
}
```

14.7. Using folders

All storages, except Pop3, support folders, also called mailboxes. The interface implemented by all storages supporting folders is called `Zend_Mail_Storage_Folder_Interface`. Also all of these classes have an additional optional parameter called `folder`, which is the folder selected after login, in the constructor.

For the local storages you need to use separate classes called `Zend_Mail_Storage_Folder_Mbox` or `Zend_Mail_Storage_Folder_Maildir`. Both need one parameter called `dirname` with the name of the base dir. The format for maildir is as defined in maildir++ (with a dot as default delimiter), Mbox is a directory hierarchy with Mbox files. If you don't have a Mbox file called INBOX in your Mbox base dir you need to set another folder in the constructor.

`Zend_Mail_Storage_Imap` already supports folders by default. Examples for opening these storages:

```
// mbox with folders
$mail = new Zend_Mail_Storage_Folder_Mbox(array('dirname' =>
                                                '/home/test/mail/'));

// mbox with a default folder not called INBOX, also works
// with Zend_Mail_Storage_Folder_Maildir and Zend_Mail_Storage_Imap
$mail = new Zend_Mail_Storage_Folder_Mbox(array('dirname' =>
                                                '/home/test/mail/',
                                            'folder'  =>
                                                'Archive'));

// maildir with folders
$mail = new Zend_Mail_Storage_Folder_Maildir(array('dirname' =>
                                                   '/home/test/mail/'));

// maildir with colon as delimiter, as suggested in Maildir++
$mail = new Zend_Mail_Storage_Folder_Maildir(array('dirname' =>
```

```
                                                    '/home/test/mail/',
                                                'delim'   => ':'));

// imap is the same with and without folders
$mail = new Zend_Mail_Storage_Imap(array('host'     => 'example.com',
                                         'user'     => 'test',
                                         'password' => 'test'));
```

With the method getFolders($root = null) you can get the folder hierarchy starting with the root folder or the given folder. It's returned as an instance of `Zend_Mail_Storage_Folder`, which implements `RecursiveIterator` and all children are also instances of `Zend_Mail_Storage_Folder`. Each of these instances has a local and a global name returned by the methods `getLocalName()` and `getGlobalName()`. The global name is the absolute name from the root folder (including delimiters), the local name is the name in the parent folder.

Table 102. Mail Folder Names

Global Name	Local Name
/INBOX	INBOX
/Archive/2005	2005
List.ZF.General	General

If you use the iterator, the key of the current element is the local name. The global name is also returned by the magic method `__toString()`. Some folders may not be selectable, which means they can't store messages and selecting them results in an error. This can be checked with the method `isSelectable()`. So it's very easy to output the whole tree in a view:

```
$folders = new RecursiveIteratorIterator($this->mail->getFolders(),
                                         RecursiveIteratorIterator::SELF_FIRST);
echo '<select name="folder">';
foreach ($folders as $localName => $folder) {
    $localName = str_pad('', $folders->getDepth(), '-', STR_PAD_LEFT) .
                 $localName;
    echo '<option';
    if (!$folder->isSelectable()) {
        echo ' disabled="disabled"';
    }
    echo ' value="' . htmlspecialchars($folder) . '">'
        . htmlspecialchars($localName) . '</option>';
}
echo '</select>';
```

The current selected folder is returned by the method `getSelectedFolder()`. Changing the folder is done with the method `selectFolder()`, which needs the global name as parameter. If you want to avoid to write delimiters you can also use the properties of a `Zend_Mail_Storage_Folder` instance:

```
// depending on your mail storage and its settings $rootFolder->Archive->2005
// is the same as:
//   /Archive/2005
//  Archive:2005
//  INBOX.Archive.2005
//  ...
$folder = $mail->getFolders()->Archive->2005;
echo 'Last folder was '
```

```
    . $mail->getSelectedFolder()
    . "new folder is $folder\n";
$mail->selectFolder($folder);
```

14.8. Advanced Use

14.8.1. Using NOOP

If you're using a remote storage and have some long tasks you might need to keep the connection alive via noop:

```
foreach ($mail as $message) {

    // do some calculations ...

    $mail->noop(); // keep alive

    // do something else ...

    $mail->noop(); // keep alive
}
```

14.8.2. Caching instances

`Zend_Mail_Storage_Mbox`, `Zend_Mail_Storage_Folder_Mbox`, `Zend_Mail_Storage_Maildir` and `Zend_Mail_Storage_Folder_Maildir` implement the magic methods `__sleep()` and `__wakeup()`, which means they are serializable. This avoids parsing the files or directory tree more than once. The disadvantage is that your Mbox or Maildir storage should not change. Some easy checks may be done, like reparsing the current Mbox file if the modification time changes, or reparsing the folder structure if a folder has vanished (which still results in an error, but you can search for another folder afterwards). It's better if you have something like a signal file for changes and check it before using the cached instance.

```
// there's no specific cache handler/class used here,
// change the code to match your cache handler
$signal_file = '/home/test/.mail.last_change';
$mbox_basedir = '/home/test/mail/';
$cache_id = 'example mail cache ' . $mbox_basedir . $signal_file;

$cache = new Your_Cache_Class();
if (!$cache->isCached($cache_id) ||
    filemtime($signal_file) > $cache->getMTime($cache_id)) {
    $mail = new Zend_Mail_Storage_Folder_Pop3(array('dirname' =>
                                                    $mbox_basedir));
} else {
    $mail = $cache->get($cache_id);
}

// do stuff ...

$cache->set($cache_id, $mail);
```

14.8.3. Extending Protocol Classes

Remote storages use two classes: `Zend_Mail_Storage_<Name>` and `Zend_Mail_Protocol_<Name>`. The protocol class translates the protocol commands and

responses from and to PHP, like methods for the commands or variables with different structures for data. The other/main class implements the common interface.

If you need additional protocol features, you can extend the protocol class and use it in the constructor of the main class. As an example, assume we need to knock different ports before we can connect to POP3.

```
class Example_Mail_Exception extends Zend_Mail_Exception
{
}

class Example_Mail_Protocol_Exception extends Zend_Mail_Protocol_Exception
{
}

class Example_Mail_Protocol_Pop3_Knock extends Zend_Mail_Protocol_Pop3
{
    private $host, $port;

    public function __construct($host, $port = null)
    {
        // no auto connect in this class
        $this->host = $host;
        $this->port = $port;
    }

    public function knock($port)
    {
        $sock = @fsockopen($this->host, $port);
        if ($sock) {
            fclose($sock);
        }
    }

    public function connect($host = null, $port = null, $ssl = false)
    {
        if ($host === null) {
            $host = $this->host;
        }
        if ($port === null) {
            $port = $this->port;
        }
        parent::connect($host, $port);
    }
}

class Example_Mail_Pop3_Knock extends Zend_Mail_Storage_Pop3
{
    public function __construct(array $params)
    {
        // ... check $params here! ...
        $protocol = new Example_Mail_Protocol_Pop3_Knock($params['host']);

        // do our "special" thing
        foreach ((array)$params['knock_ports'] as $port) {
            $protocol->knock($port);
        }

        // get to correct state
```

```
        $protocol->connect($params['host'], $params['port']);
        $protocol->login($params['user'], $params['password']);

        // initialize parent
        parent::__construct($protocol);
    }
}

$mail = new Example_Mail_Pop3_Knock(array('host'        => 'localhost',
                                          'user'        => 'test',
                                          'password'    => 'test',
                                          'knock_ports' =>
                                              array(1101, 1105, 1111)));
```

As you see, we always assume we're connected, logged in and, if supported, a folder is selected in the constructor of the main class. Thus if you assign your own protocol class, you always need to make sure that's done or the next method will fail if the server doesn't allow it in the current state.

14.8.4. Using Quota (since 1.5)

`Zend_Mail_Storage_Writable_Maildir` has support for Maildir++ quotas. It's disabled by default, but it's possible to use it manually, if the automatic checks are not desired (this means `appendMessage()`, `removeMessage()` and `copyMessage()` do no checks and do not add entries to the maildirsize file). If enabled, an exception is thrown if you try to write to the maildir and it's already over quota.

There are three methods used for quotas: `getQuota()`, `setQuota()` and `checkQuota()`:

```
$mail = new Zend_Mail_Storage_Writable_Maildir(array('dirname' =>
                                                     '/home/test/mail/'));
$mail->setQuota(true); // true to enable, false to disable
echo 'Quota check is now ', $mail->getQuota() ? 'enabled' : 'disabled', "\n";
// check quota can be used even if quota checks are disabled
echo 'You are ', $mail->checkQuota() ? 'over quota' : 'not over quota', "\n";
```

`checkQuota()` can also return a more detailed response:

```
$quota = $mail->checkQuota(true);
echo 'You are ', $quota['over_quota'] ? 'over quota' : 'not over quota', "\n";
echo 'You have ',
     $quota['count'],
     ' of ',
     $quota['quota']['count'],
     ' messages and use ';
echo $quota['size'], ' of ', $quota['quota']['size'], ' octets';
```

If you want to specify your own quota instead of using the one specified in the maildirsize file you can do with `setQuota()`:

```
// message count and octet size supported, order does matter
$quota = $mail->setQuota(array('size' => 10000, 'count' => 100));
```

To add your own quota checks use single letters as keys, and they will be preserved (but obviously not checked). It's also possible to extend `Zend_Mail_Storage_Writable_Maildir` to define your own quota only if the maildirsize file is missing (which can happen in Maildir++):

```
class Example_Mail_Storage_Maildir extends Zend_Mail_Storage_Writable_Maildir {
    // getQuota is called with $fromStorage = true by quota checks
    public function getQuota($fromStorage = false) {
        try {
            return parent::getQuota($fromStorage);
        } catch (Zend_Mail_Storage_Exception $e) {
            if (!$fromStorage) {
                // unknown error:
                throw $e;
            }
            // maildirsize file must be missing

            list($count, $size) = get_quota_from_somewhere_else();
            return array('count' => $count, 'size' => $size);
        }
    }
}
```

Zend_Markup

1. Introduction

The Zend_Markup component provides an extensible way for parsing text and rendering lightweight markup languages like BBcode and Textile. It is available as of Zend Framework version 1.10.

Zend_Markup uses a factory method to instantiate an instance of a renderer that extends Zend_Markup_Renderer_Abstract. The factory method accepts three arguments. The first one is the parser used to tokenize the text (e.g. BbCode). The second (optional) parameter is the renderer to use, Html by default. Thirdly an array with options to use for the renderer can be specified.

2. Getting Started With Zend_Markup

This guide to get you started with Zend_Markup uses the BBCode parser and HTML renderer. The priciples discussed can be adapted to other parsers and renderers.

Example 537. Basic Zend_Markup Usage

We will first instantiate a Zend_Markup_Renderer_Html object using the Zend_Markup::factory() method. This will also create a Zend_Markup_Parser_Bbcode object which will be added to the renderer object.

Afther that, we will use the render() method to convert a piece of BBCode to HTML.

```
// Creates instance of Zend_Markup_Renderer_Html,
// with Zend_Markup_Parser_BbCode as its parser
$bbcode = Zend_Markup::factory('Bbcode');

echo $bbcode->render('[b]bold text[/b] and [i]cursive text[/i]');
// Outputs: '<strong>bold text</strong> and <em>cursive text</em>'
```

Example 538. A more complicated example of Zend_Markup

This time, we will do exactly the same as above, but with more complicated BBCode markup.

```
$bbcode = Zend_Markup::factory('Bbcode');

$input = <<<EOT
[list]
[*]Zend Framework
[*]Foobar
[/list]
EOT;

echo $bbcode->render($input);
/*
Should output something like:
<ul>
<li>Zend Framework</li>
<li>Foobar</li>
</ul>
*/
```

Example 539. Processing incorrect input

Besides simply parsing and rendering markup such as BBCode, Zend_Markup is also able to handle incorrect input. Most BBCode processors are not able to render all input to XHTML valid output. Zend_Markup corrects input that is nested incorrectly, and also closes tags that were not closed:

```
$bbcode = Zend_Markup::factory('Bbcode');

echo $bbcode->render('some [i]wrong [b]sample [/i] text');
// Note that the '[b]' tag is never closed, and is also incorrectly
// nested; regardless, Zend_Markup renders it correctly as:
// some <em>wrong <strong>sample </strong></em><strong> text</strong>
```

3. Zend_Markup Parsers

Zend_Markup is currently shipped with two parsers, a BBCode parser and a Textile parser.

3.1. Theory of Parsing

The parsers of Zend_Markup are classes that convert text with markup to a token tree. Although we are using the BBCode parser as example here, the idea of the token tree remains the same across all parsers. We will start with this piece of BBCode for example:

```
[b]foo[i]bar[/i][/b]baz
```

Then the BBCode parser will take that value, tear it apart and create the following tree:

- [b]
  - foo
  - [i]
    - bar
- baz

You will notice that the closing tags are gone, they don't show up as content in the tree structure. This is because the closing tag isn't part of the actual content. Although, this does not mean that the closing tag is just lost, it is stored inside the tag information for the tag itself. Also, please note that this is just a simplified view of the tree itself. The actual tree contains a lot more information, like the tag's attributes and its name.

3.2. The BBCode parser

The BBCode parser is a Zend_Markup parser that converts BBCode to a token tree. The syntax of all BBCode tags is:

```
[name(=(value|"value"))( attribute=(value|"value"))*]
```

Some examples of valid BBCode tags are:

```
[b]
[list=1]
[code file=Zend/Markup.php]
```

```
[url="http://framework.zend.com/" title="Zend Framework!"]
```

By default, all tags are closed by using the format '[/tagname]'.

3.3. The Textile parser

The Textile parser is a `Zend_Markup` parser that converts Textile to a token tree. Because Textile doesn't have a tag structure, the following is a list of example tags:

Table 103. List of basic Textile tags

Sample input	Sample output
foo	foo
foo	foo
??foo??	<cite>foo</cite>
-foo-	~~foo~~
+foo+	<ins>foo</ins>
^foo^	^{foo}
~foo~	_{foo}
%foo%	foo
PHP(PHP Hypertext Preprocessor)	<acronym title="PHP Hypertext Preprocessor">PHP</acronym>
"Zend Framework":http://framework.zend.com/	Zend Framework
h1. foobar	foobar
h6. foobar	foobar
!http://framework.zend.com/images/logo.gif!	

Also, the Textile parser wraps all tags into paragraphs; a paragraph ends with two newlines, and if there are more tags, a new paragraph will be added.

3.3.1. Lists

The Textile parser also supports two types of lists. The numeric type, using the "#" character and bullit-lists using the "*" character. An example of both lists:

```
# Item 1
# Item 2

* Item 1
* Item 2
```

The above will generate two lists: the first, numbered; and the second, bulleted. Inside list items, you can use normal tags like strong (*), and emphasized (_). Tags that need to start on a new line (like 'h1' etc.) cannot be used inside lists.

4. Zend_Markup Renderers

`Zend_Markup` is currently shipped with one renderer, the HTML renderer.

4.1. Adding your own tags

By adding your own tags, you can add your own functionality to the Zend_Markup renderers. With the tag structure, you can add about any functionality you want. From simple tags, to complicated tag structures. A simple example for a 'foo' tag:

```
// Creates instance of Zend_Markup_Renderer_Html,
// with Zend_Markup_Parser_BbCode as its parser
$bbcode = Zend_Markup::factory('Bbcode');

// this will create a simple 'foo' tag
// The first parameter defines the tag's name.
// The second parameter takes an integer that defines the tags type.
// The third parameter is an array that defines other things about a
// tag, like the tag's group, and (in this case) a start and end tag.
$bbcode->addTag(
    'foo',
    Zend_Markup_Renderer_RendererAbstract::TYPE_REPLACE,
    array(
        'start' => '-bar-',
        'end'   => '-baz-',
        'group' => 'inline'
    )
);

// now, this will output: 'my -bar-tag-baz-'
echo $bbcode->render('my [foo]tag[/foo]');
```

Please note that creating your own tags only makes sense when your parser also supports it with a tag structure. Currently, only BBCode supports this. Textile doesn't have support for custom tags.

Some renderers (like the HTML renderer) also have support for a 'tag' parameter. This replaces the 'start' and 'end' parameters, and it renders the tags including some default attributes and the closing tag.

4.1.1. Add a callback tag

By adding a callback tag, you can do a lot more then just a simple replace of the tags. For instance, you can change the contents, use the parameters to influence the output etc.

A callback is a class that implements the Zend_Markup_Renderer_TokenInterface interface. An example of a callback class:

```
class My_Markup_Renderer_Html_Upper extends Zend_Markup_Renderer_TokenConverterInterface
{

    public function convert(Zend_Markup_Token $token, $text)
    {
        return '!up!' . strtoupper($text) . '!up!';
    }

}
```

Now you can add the 'upper' tag, with as callback, an instance of the My_Markup_Renderer_Html_Upper class. A simple example:

```php
// Creates instance of Zend_Markup_Renderer_Html,
// with Zend_Markup_Parser_BbCode as its parser
$bbcode = Zend_Markup::factory('Bbcode');

// this will create a simple 'foo' tag
// The first parameter defines the tag's name.
// The second parameter takes an integer that defines the tags type.
// The third parameter is an array that defines other things about a
// tag, like the tag's group, and (in this case) a start and end tag.
$bbcode->addTag(
    'upper',
    Zend_Markup_Renderer_RendererAbstract::TYPE_REPLACE,
    array(
        'callback' => new My_Markup_Renderer_Html_Upper(),
        'group'    => 'inline'
    )
);

// now, this will output: 'my !up!TAG!up!'
echo $bbcode->render('my [upper]tag[/upper]');
```

4.2. List of tags

Table 104. List of tags

Sample input (bbcode)	Sample output
[b]foo[/b]	foo
[i]foo[/i]	foo
[cite]foo[/cite]	<cite>foo</cite>
[del]foo[/del]	~~foo~~
[ins]foo[/ins]	<ins>foo</ins>
[sup]foo[/sup]	^{foo}
[sub]foo[/sub]	_{foo}
[span]foo[/span]	foo
[acronym title="PHP Hypertext Preprocessor]PHP[/acronym]	<acronym title="PHP Hypertext Preprocessor">PHP</acronym>
[url=http://framework.zend.com/]Zend Framework[/url]	Zend Framework
[h1]foobar[/h1]	foobar
[img]http://framework.zend.com/images/logo.gif[/img]	

Zend_Measure

1. Introduction

`Zend_Measure_*` classes provide a generic and easy way for working with measurements. Using `Zend_Measure_*` classes, you can convert measurements into different units of the same type. They can be added, subtracted and compared against each other. From a given input made in the user's native language, the unit of measurement can be automatically extracted. Numerous units of measurement are supported.

Example 540. Converting measurements

The following introductory example shows automatic conversion of units of measurement. To convert a measurement, its value and its type have to be known. The value can be an integer, a float, or even a string containing a number. Conversions are only possible for units of the same type (mass, area, temperature, velocity, etc.), not between types.

```
$locale = new Zend_Locale('en');
$unit = new Zend_Measure_Length(100, Zend_Measure_Length::METER, $locale);

// Convert meters to yards
echo $unit->convertTo(Zend_Measure_Length::YARD);
```

`Zend_Measure_*` includes support for many different units of measurement. The units of measurement all have a unified notation: `Zend_Measure_<TYPE>::NAME_OF_UNIT`, where <TYPE> corresponds to a well-known physical or numerical property. . Every unit of measurement consists of a conversion factor and a display unit. A detailed list can be found in the chapter `Types of measurements` .

Example 541. The meter measurement

The `meter` is used for measuring lengths, so its type constant can be found in the `Length` class. To refer to this unit of measurement, the notation `Length::METER` must be used. The display unit is `m`.

```
echo Zend_Measure_Length::STANDARD;  // outputs 'Length::METER'
echo Zend_Measure_Length::KILOMETER; // outputs 'Length::KILOMETER'

$unit = new Zend_Measure_Length(100,'METER');
echo $unit;
// outputs '100 m'
```

2. Creation of Measurements

When creating a measurement object, `Zend_Measure_*` methods expect the input/original measurement data value as the first parameter. This can be a `numeric argument` , a String without units, or a `localized string with unit(s) specified.` The second parameter defines the type of the measurement. Both parameters are mandatory. The language may optionally be specified as the third parameter.

2.1. Creating measurements from integers and floats

In addition to integer data values, floating point types may be used, but "simple decimal fractions like 0.1 or 0.7 cannot be converted into their internal binary counterparts without a little loss of

precision," sometimes giving surprising results. Also, do not compare two "float" type numbers for equality.

Example 542. Creation using integer and floating values

```
$measurement = 1234.7;
$unit = new Zend_Measure_Length((integer)$measurement,
                                Zend_Measure_Length::STANDARD);
echo $unit;
// outputs '1234 m' (meters)

$unit = new Zend_Measure_Length($measurement, Zend_Measure_Length::STANDARD);
echo $unit;
// outputs '1234.7 m' (meters)
```

2.2. Creating measurements from strings

Many measurements received as input to Zend Framework applications can only be passed to `Zend_Measure_*` classes as strings, such as numbers written using roman numerals or extremely large binary values that exceed the precision of PHP's native integer and float types. Since integers can be denoted using strings, if there is any risk of losing precision due to limitations of PHP's native integer and float types, using strings instead. `Zend_Measure_Number` uses the BCMath extension to support arbitrary precision, as shown in the example below, to avoid limitations in many PHP functions, such as `bin2dec()`.

Example 543. Creation using strings

```
$mystring = "10010100111010111010100001011011101010001";
$unit = new Zend_Measure_Number($mystring, Zend_Measure_Number::BINARY);

echo $unit;
```

2.3. Measurements from localized strings

When a string is entered in a localized notation, the correct interpretation can not be determined without knowing the intended locale. The division of decimal digits with "." and grouping of thousands with "," is common in the English language, but not so in other languages. For example, the English number "1,234.50" would be interpreted as meaning "1.2345" in German. To deal with such problems, the locale-aware `Zend_Measure_*` family of classes offer the possibility to specify a language or region to disambiguate the input data and properly interpret the intended semantic value.

Example 544. Localized string

```
$locale = new Zend_Locale('de');
$mystring = "1,234.50";
$unit = new Zend_Measure_Length($mystring,
                                Zend_Measure_Length::STANDARD,
                                $locale);
echo $unit; // outputs "1.234 m"

$mystring = "1,234.50";
$unit = new Zend_Measure_Length($mystring,
                                Zend_Measure_Length::STANDARD,
                                'en_US');
echo $unit; // outputs "1234.50 m"
```

Since Zend Framework 1.7.0 `Zend_Measure` does also support the usage of an application wide locale. You can simply set a `Zend_Locale` instance to the registry like shown below. With this notation you can forget about setting the locale manually with each instance when you want to use the same locale multiple times.

```
// in your bootstrap file
$locale = new Zend_Locale('de_AT');
Zend_Registry::set('Zend_Locale', $locale);

// somewhere in your application
$length = new Zend_Measure_Length(Zend_Measure_Length::METER();
```

3. Outputting measurements

Measurements can be output in a number of different ways.

`Automatic output`

`Outputting values`

`Output with unit of measurement`

`Output as localized string`

3.1. Automatic output

`Zend_Measure` supports outputting of strings automatically.

Example 545. Automatic output

```
$locale = new Zend_Locale('de');
$mystring = "1.234.567,89";
$unit = new Zend_Measure_Length($mystring,
                                Zend_Measure_Length::STANDARD,
                                $locale);

echo $unit;
```

Measurement output

Output can be achieved simply by using `echo` or `print` .

3.2. Outputting values

The value of a measurement can be output using `getValue()`.

Example 546. Output a value

```
$locale = new Zend_Locale('de');
$mystring = "1.234.567,89";
$unit = new Zend_Measure_Length($mystring,
                                Zend_Measure_Length::STANDARD,
                                $locale);

echo $unit->getValue();
```

The `getValue()` method accepts an optional parameter '`round`' which allows to define a precision for the generated output. The standard precision is '2'.

3.3. Output with unit of measurement

The function `getType()` returns the current unit of measurement.

Example 547. Outputting units

```
$locale = new Zend_Locale('de');
$mystring = "1.234.567,89";
$unit = new Zend_Measure_Weight($mystring,
                                Zend_Measure_Weight::POUND,
                                $locale);

echo $unit->getType();
```

3.4. Output as localized string

Outputting a string in a format common in the users' country is usually desirable. For example, the measurement "1234567.8" would become "1.234.567,8" for Germany. This functionality will be supported in a future release.

4. Manipulating Measurements

Parsing and normalization of input, combined with output to localized notations makes data accessible to users in different locales. Many additional methods exist in `Zend_Measure_*` components to manipulate and work with this data, after it has been normalized.

- Convert
- Add and subtract
- Compare to boolean
- Compare to greater/smaller
- Manually change values
- Manually change types

4.1. Convert

Probably the most important feature is the conversion into different units of measurement. The conversion of a unit can be done any number of times using the method `convertTo()`. Units of measurement can only be converted to other units of the same type (class). Therefore, it is not possible to convert (e.g.) a length into a weight, which would might encourage poor programming practices and allow errors to propagate without exceptions.

The `convertTo` method accepts an optional parameter. With this parameter you can define an precision for the returned output. The standard precision is '2'.

Example 548. Convert

```
$locale = new Zend_Locale('de');
$mystring = "1.234.567,89";
$unit = new Zend_Measure_Weight($mystring,'POND', $locale);

print "Kilo:".$unit->convertTo('KILOGRAM');

// constants are considered "better practice" than strings
print "Ton:".$unit->convertTo(Zend_Measure_Weight::TON);

// define a precision for the output
print "Ton:".$unit->convertTo(Zend_Measure_Weight::TON, 3);
```

4.2. Add and subtract

Measurements can be added together using `add()` and subtracted using `sub()`. Each addition will create a new object for the result. The actual object will never be changed by the class. The new object will be of the same type as the originating object. Dynamic objects support a fluid style of programming, where complex sequences of operations can be nested without risk of side-effects altering the input objects.

Example 549. Adding units

```
// Define objects
$unit = new Zend_Measure_Length(200, Zend_Measure_Length::CENTIMETER);
$unit2 = new Zend_Measure_Length(1, Zend_Measure_Length::METER);

// Add $unit2 to $unit
$sum = $unit->add($unit2);

echo $sum; // outputs "300 cm"
```

Automatic conversion

Adding one object to another will automatically convert it to the correct unit. It is not necessary to call `convertTo()` before adding different units.

Example 550. Subtract

Subtraction of measurements works just like addition.

```
// Define objects
$unit = new Zend_Measure_Length(200, Zend_Measure_Length::CENTIMETER);
$unit2 = new Zend_Measure_Length(1, Zend_Measure_Length::METER);

// Subtract $unit2 from $unit
$sum = $unit->sub($unit2);

echo $sum;
```

4.3. Compare

Measurements can also be compared, but without automatic unit conversion. Thus, `equals()` returns TRUE, only if both the value and the unit of measure are identical.

Example 551. Different measurements

```
// Define measurements
$unit = new Zend_Measure_Length(100, Zend_Measure_Length::CENTIMETER);
$unit2 = new Zend_Measure_Length(1, Zend_Measure_Length::METER);

if ($unit->equals($unit2)) {
    print "Both measurements are identical";
} else {
    print "These are different measurements";
}
```

Example 552. Identical measurements

```
// Define measurements
$unit = new Zend_Measure_Length(100, Zend_Measure_Length::CENTIMETER);
$unit2 = new Zend_Measure_Length(1, Zend_Measure_Length::METER);

$unit2->setType(Zend_Measure_Length::CENTIMETER);

if ($unit->equals($unit2)) {
    print "Both measurements are identical";
} else {
    print "These are different measurements";
}
```

4.4. Compare

To determine if a measurement is less than or greater than another, use `compare()`, which returns 0, -1 or 1 depending on the difference between the two objects. Identical measurements will return 0. Lesser ones will return a negative, greater ones a positive value.

Example 553. Difference

```
$unit = new Zend_Measure_Length(100, Zend_Measure_Length::CENTIMETER);
$unit2 = new Zend_Measure_Length(1, Zend_Measure_Length::METER);
$unit3 = new Zend_Measure_Length(1.2, Zend_Measure_Length::METER);

print "Equal:".$unit2->compare($unit);
print "Lesser:".$unit2->compare($unit3);
print "Greater:".$unit3->compare($unit2);
```

4.5. Manually change values

To change the value of a measurement explicitly, use `setValue()`. to overwrite the current value. The parameters are the same as the constructor.

Example 554. Changing a value

```
$locale = new Zend_Locale('de_AT');
$unit = new Zend_Measure_Length(1,Zend_Measure_Length::METER);

$unit->setValue(1.2);
echo $unit;

$unit->setValue(1.2, Zend_Measure_Length::KILOMETER);
echo $unit;

$unit->setValue("1.234,56", Zend_Measure_Length::MILLIMETER,$locale);
echo $unit;
```

4.6. Manually change types

To change the type of a measurement without altering its value use `setType()`.

Example 555. Changing the type

```
$unit = new Zend_Measure_Length(1,Zend_Measure_Length::METER);
echo $unit; // outputs "1 m"

$unit->setType(Zend_Measure_Length::KILOMETER);
echo $unit; // outputs "1000 km"
```

5. Types of measurements

All supported measurement types are listed below, each with an example of the standard usage for such measurements.

Table 105. List of measurement types

Typ	Class	Standardunit	Description
Acceleration	Zend_Measure_Acceleration	Meter per square second | m/s²	Zend_Measure_Acceleration covers the physical factor of acceleration.
Angle	Zend_Measure_Angle	Radiant | rad	Zend_Measure_Angle covers angular dimensions.
Area	Zend_Measure_Area	Square meter | m²	Zend_Measure_Area covers square measures.
Binary	Zend_Measure_Binary	Byte | b	Zend_Measure_Binary covers binary conversions.
Capacitance	Zend_Measure_Capacitance	Farad | F	Zend_Measure_Capacitance covers physical factor of capacitance.
Cooking volumes	Zend_Measure_Cooking_Volume	Cubic meter | m³	Zend_Measure_Cooking_Volume covers volumes which are used for cooking or written in cookbooks.

Typ	Class	Standardunit	Description
Cooking weights	Zend_Measure_Cooking_Weight	Gram	Zend_Measure_Cooking_Weight covers the weights which are used for cooking or written in cookbooks.
Current	Zend_Measure_Current	Ampere | A	Zend_Measure_Current covers the physical factor of current.
Density	Zend_Measure_Density	Kilogram per cubic meter | kg/m³	Zend_Measure_Density covers the physical factor of density.
Energy	Zend_Measure_Energy	Joule | J	Zend_Measure_Energy covers the physical factor of energy.
Force	Zend_Measure_Force	Newton | N	Zend_Measure_Force covers the physical factor of force.
Flow (mass)	Zend_Measure_Flow_Mass	Kilogram per second | kg/s	Zend_Measure_Flow_Mass covers the physical factor of flow rate. The weight of the flowing mass is used as reference point within this class.
Flow (mole)	Zend_Measure_Flow_Mole	Mole per second | mol/s	Zend_Measure_Flow_Mole covers the physical factor of flow rate. The density of the flowing mass is used as reference point within this class.
Flow (volume)	Zend_Measure_Flow_Volume	Cubic meter per second | m³/s	Zend_Measure_Flow_Volume covers the physical factor of flow rate. The volume of the flowing mass is used as reference point within this class.
Frequency	Zend_Measure_Frequency	Hertz | Hz	Zend_Measure_Frequency covers the physical factor of frequency.
Illumination	Zend_Measure_Illumination	Lux | lx	Zend_Measure_Illumination covers the physical factor of light density.
Length	Zend_Measure_Length	Meter | m	Zend_Measure_Length covers the physical factor of length.

Typ	Class	Standardunit	Description
Lightness	Zend_Measure_Lightness	Candela per square meter | cd/m²	Zend_Measure_Lightness covers the physical factor of light energy.
Number	Zend_Measure_Number	Decimal | (10)	Zend_Measure_Number converts between number formats.
Power	Zend_Measure_Power	Watt | W	Zend_Measure_Power covers the physical factor of power.
Pressure	Zend_Measure_Pressure	Newton per square meter | N/m²	Zend_Measure_Pressure covers the physical factor of pressure.
Speed	Zend_Measure_Speed	Meter per second | m/s	Zend_Measure_Speed covers the physical factor of speed.
Temperature	Zend_Measure_Temperature	Kelvin | K	Zend_Measure_Temperature covers the physical factor of temperature.
Time	Zend_Measure_Time	Second | s	Zend_Measure_Time covers the physical factor of time.
Torque	Zend_Measure_Torque	Newton meter | Nm	Zend_Measure_Torque covers the physical factor of torque.
Viscosity (dynamic)	Zend_Measure_Viscosity_Dynamic	Kilogram per meter second | kg/ms	Zend_Measure_Viscosity_Dynamic covers the physical factor of viscosity. The weight of the fluid is used as reference point within this class.
Viscosity (kinematic)	Zend_Measure_Viscosity_Kinematic	Square meter per second | m²/s	Zend_Measure_Viscosity_Kinematic covers the physical factor of viscosity. The distance of the flown fluid is used as reference point within this class.
Volume	Zend_Measure_Volume	Cubic meter | m³	Zend_Measure_Volume covers the physical factor of volume (content).
Weight	Zend_Measure_Weight	Kilogram | kg	Zend_Measure_Weight covers the physical factor of weight.

5.1. Hints for Zend_Measure_Binary

Some popular binary conventions, include terms like kilo-, mega-, giga, etc. in normal language use imply base 10, such as 1000 or 10^3. However, in the binary format for computers these terms have to be seen for a conversion factor of 1024 instead of 1000. To preclude confusions a few years ago the notation BI was introduced. Instead of kilobyte, kibibyte for kilo-binary-byte should be used.

In the class BINARY both notations can be found, such as `KILOBYTE = 1024 - binary conputer conversion KIBIBYTE = 1024 - new notation KILO_BINARY_BYTE = 1024 - new`, or the notation, long format `KILOBYTE_SI = 1000 - SI notation for kilo (1000)`. DVDs for example are marked with the SI-notation, but almost all harddisks are marked in computer binary notation.

5.2. Hints for Zend_Measure_Number

The best known number format is the decimal system. Additionally this class supports the octal system, the hexadecimal system, the binary system, the roman number system and some other less popular systems. Note that only the decimal part of numbers is handled. Any fractional part will be stripped.

5.3. Roman numbers

For the roman number system digits greater 4000 are supported. In reality these digits are shown with a crossbeam on top of the digit. As the crossbeam can not be shown within the computer, an underline has to be used instead of it.

```
$great = '_X';
$locale = new Zend_Locale('en');
$unit = new Zend_Measure_Number($great,Zend_Measure_Number::ROMAN, $locale);

// convert to the decimal system
echo $unit->convertTo(Zend_Measure_Number::DECIMAL);
```

Zend_Memory

1. Overview

1.1. Introduction

The Zend_Memory component is intended to manage data in an environment with limited memory.

Memory objects (memory containers) are generated by memory manager by request and transparently swapped/loaded when it's necessary.

For example, if creating or loading a managed object would cause the total memory usage to exceed the limit you specify, some managed objects are copied to cache storage outside of memory. In this way, the total memory used by managed objects does not exceed the limit you need to enforce.

The memory manager uses Zend_Cache backends as storage providers.

Example 556. Using Zend_Memory component

Zend_Memory::factory() instantiates the memory manager object with specified backend options.

```
$backendOptions = array(
    'cache_dir' => './tmp/' // Directory where to put the swapped memory blocks
);

$memoryManager = Zend_Memory::factory('File', $backendOptions);

$loadedFiles = array();

for ($count = 0; $count < 10000; $count++) {
    $f = fopen($fileNames[$count], 'rb');
    $data = fread($f, filesize($fileNames[$count]));
    $fclose($f);

    $loadedFiles[] = $memoryManager->create($data);
}

echo $loadedFiles[$index1]->value;

$loadedFiles[$index2]->value = $newValue;

$loadedFiles[$index3]->value[$charIndex] = '_';
```

1.2. Theory of Operation

Zend_Memory component operates with the following concepts:

- Memory manager
- Memory container
- Locked memory object

- Movable memory object

1.2.1. Memory manager

The memory manager generates memory objects (locked or movable) by request of user application and returns them wrapped into a memory container object.

1.2.2. Memory container

The memory container has a virtual or actual `value` attribute of string type. This attribute contains the data value specified at memory object creation time.

You can operate with this `value` attribute as an object property:

```
$memObject = $memoryManager->create($data);

echo $memObject->value;

$memObject->value = $newValue;

$memObject->value[$index] = '_';

echo ord($memObject->value[$index1]);

$memObject->value = substr($memObject->value, $start, $length);
```

If you are using a PHP version earlier than 5.2, use the getRef() method instead of accessing the value property directly.

1.2.3. Locked memory

Locked memory objects are always stored in memory. Data stored in locked memory are never swapped to the cache backend.

1.2.4. Movable memory

Movable memory objects are transparently swapped and loaded to/from the cache backend by `Zend_Memory` when it's necessary.

The memory manager doesn't swap objects with size less than the specified minimum, due to performance considerations. See Section 2.3.2, “MinSize” for more details.

2. Memory Manager

2.1. Creating a Memory Manager

You can create new a memory manager (`Zend_Memory_Manager` object) using the `Zend_Memory::factory($backendName [, $backendOprions])` method.

The first argument `$backendName` is a string that names one of the backend implementations supported by `Zend_Cache`.

The second argument `$backendOptions` is an optional backend options array.

```
$backendOptions = array(
```

```
    'cache_dir' => './tmp/' // Directory where to put the swapped memory blocks
);

$memoryManager = Zend_Memory::factory('File', $backendOptions);
```

Zend_Memory uses Zend_Cache backends as storage providers.

You may use the special name 'None' as a backend name, in addition to standard Zend_Cache backends.

```
$memoryManager = Zend_Memory::factory('None');
```

If you use 'None' as the backend name, then the memory manager never swaps memory blocks. This is useful if you know that memory is not limited or the overall size of objects never reaches the memory limit.

The 'None' backend doesn't need any option specified.

2.2. Managing Memory Objects

This section describes creating and destroying objects in the managed memory, and settings to control memory manager behavior.

2.2.1. Creating Movable Objects

Create movable objects (objects, which may be swapped) using the Zend_Memory_Manager::create([$data]) method:

```
$memObject = $memoryManager->create($data);
```

The $data argument is optional and used to initialize the object value. If the $data argument is omitted, the value is an empty string.

2.2.2. Creating Locked Objects

Create locked objects (objects, which are not swapped) using the Zend_Memory_Manager::createLocked([$data]) method:

```
$memObject = $memoryManager->createLocked($data);
```

The $data argument is optional and used to initialize the object value. If the $data argument is omitted, the value is an empty string.

2.2.3. Destroying Objects

Memory objects are automatically destroyed and removed from memory when they go out of scope:

```
function foo()
{
    global $memoryManager, $memList;

    ...

    $memObject1 = $memoryManager->create($data1);
    $memObject2 = $memoryManager->create($data2);
    $memObject3 = $memoryManager->create($data3);
```

```
    ...

    $memList[] = $memObject3;

    ...

    unset($memObject2); // $memObject2 is destroyed here

    ...
    // $memObject1 is destroyed here
    // but $memObject3 object is still referenced by $memList
    // and is not destroyed
}
```

This applies to both movable and locked objects.

2.3. Memory Manager Settings

2.3.1. Memory Limit

Memory limit is a number of bytes allowed to be used by loaded movable objects.

If loading or creation of an object causes memory usage to exceed of this limit, then the memory manager swaps some other objects.

You can retrieve or set the memory limit setting using the `getMemoryLimit()` and `setMemoryLimit($newLimit)` methods:

```
$oldLimit = $memoryManager->getMemoryLimit();  // Get memory limit in bytes
$memoryManager->setMemoryLimit($newLimit);     // Set memory limit in bytes
```

A negative value for memory limit means 'no limit'.

The default value is two-thirds of the value of '`memory_limit`' in php.ini or 'no limit' (-1) if '`memory_limit`' is not set in php.ini.

2.3.2. MinSize

MinSize is a minimal size of memory objects, which may be swapped by memory manager. The memory manager does not swap objects that are smaller than this value. This reduces the number of swap/load operations.

You can retrieve or set the minimum size using the `getMinSize()` and `setMinSize($newSize)` methods:

```
$oldMinSize = $memoryManager->getMinSize();  // Get MinSize in bytes
$memoryManager->setMinSize($newSize);        // Set MinSize limit in bytes
```

The default minimum size value is 16KB (16384 bytes).

3. Memory Objects

3.1. Movable

Create movable memory objects using the `create([$data])` method of the memory manager:

```
$memObject = $memoryManager->create($data);
```

"Movable" means that such objects may be swapped and unloaded from memory and then loaded when application code accesses the object.

3.2. Locked

Create locked memory objects using the `createLocked([$data])` method of the memory manager:

```
$memObject = $memoryManager->createLocked($data);
```

"Locked" means that such objects are never swapped and unloaded from memory.

Locked objects provides the same interface as movable objects (`Zend_Memory_Container_Interface`). So locked object can be used in any place instead of movable objects.

It's useful if an application or developer can decide, that some objects should never be swapped, based on performance considerations.

Access to locked objects is faster, because the memory manager doesn't need to track changes for these objects.

The locked objects class (`Zend_Memory_Container_Locked`) guarantees virtually the same performance as working with a string variable. The overhead is a single dereference to get the class property.

3.3. Memory container 'value' property

Use the memory container (movable or locked) '`value`' property to operate with memory object data:

```
$memObject = $memoryManager->create($data);

echo $memObject->value;

$memObject->value = $newValue;

$memObject->value[$index] = '_';

echo ord($memObject->value[$index1]);

$memObject->value = substr($memObject->value, $start, $length);
```

An alternative way to access memory object data is to use the `getRef()` method. This method *must* be used for PHP versions before 5.2. It also may have to be used in some other cases for performance reasons.

3.4. Memory container interface

Memory container provides the following methods:

3.4.1. getRef() method

```
public function &getRef();
```

The `getRef()` method returns reference to the object value.

Movable objects are loaded from the cache at this moment if the object is not already in memory. If the object is loaded from the cache, this might cause swapping of other objects if the memory limit would be exceeded by having all the managed objects in memory.

The `getRef()` method *must* be used to access memory object data for PHP versions before 5.2.

Tracking changes to data needs additional resources. The `getRef()` method returns reference to string, which is changed directly by user application. So, it's a good idea to use the `getRef()` method for value data processing:

```
$memObject = $memoryManager->create($data);

$value = &$memObject->getRef();

for ($count = 0; $count < strlen($value); $count++) {
    $char = $value[$count];
    ...
}
```

3.4.2. touch() method

```
public function touch();
```

The `touch()` method should be used in common with `getRef()`. It signals that object value has been changed:

```
$memObject = $memoryManager->create($data);
...

$value = &$memObject->getRef();

for ($count = 0; $count < strlen($value); $count++) {
    ...
    if ($condition) {
        $value[$count] = $char;
    }
    ...
}

$memObject->touch();
```

3.4.3. lock() method

```
public function lock();
```

The `lock()` methods locks object in memory. It should be used to prevent swapping of some objects you choose. Normally, this is not necessary, because the memory manager uses an intelligent algorithm to choose candidates for swapping. But if you exactly know, that at this part of code some objects should not be swapped, you may lock them.

Locking objects in memory also guarantees that reference returned by the `getRef()` method is valid until you unlock the object:

```
$memObject1 = $memoryManager->create($data1);
$memObject2 = $memoryManager->create($data2);
```

```
...

$memObject1->lock();
$memObject2->lock();

$value1 = &$memObject1->getRef();
$value2 = &$memObject2->getRef();

for ($count = 0; $count < strlen($value2); $count++) {
    $value1 .= $value2[$count];
}

$memObject1->touch();
$memObject1->unlock();
$memObject2->unlock();
```

3.4.4. unlock() method

```
public function unlock();
```

`unlock()` method unlocks object when it's no longer necessary to be locked. See the example above.

3.4.5. isLocked() method

```
public function isLocked();
```

The `isLocked()` method can be used to check if object is locked. It returns TRUE if the object is locked, or FALSE if it is not locked. This is always TRUE for "locked" objects, and may be either TRUE or FALSE for "movable" objects.

Zend_Mime

1. Zend_Mime

1.1. Introduction

`Zend_Mime` is a support class for handling multipart MIME messages. It is used by `Zend_Mail` and `Zend_Mime_Message` and may be used by applications requiring MIME support.

1.2. Static Methods and Constants

`Zend_Mime` provides a simple set of static helper methods to work with MIME:

- `Zend_Mime::isPrintable()`: Returns `TRUE` if the given string contains no unprintable characters, FALSE otherwise.
- `Zend_Mime::encode()`: Encodes a string with specified encoding.
- `Zend_Mime::encodeBase64()`: Encodes a string into base64 encoding.
- `Zend_Mime::encodeQuotedPrintable()`: Encodes a string with the quoted-printable mechanism.
- `Zend_Mime::encodeBase64Header()`: Encodes a string into base64 encoding for Mail Headers.
- `Zend_Mime::encodeQuotedPrintableHeader()`: Encodes a string with the quoted-printable mechanism for Mail Headers.

`Zend_Mime` defines a set of constants commonly used with MIME Messages:

- `Zend_Mime::TYPE_OCTETSTREAM`: 'application/octet-stream'
- `Zend_Mime::TYPE_TEXT`: 'text/plain'
- `Zend_Mime::TYPE_HTML`: 'text/html'
- `Zend_Mime::ENCODING_7BIT`: '7bit'
- `Zend_Mime::ENCODING_8BIT`: '8bit'
- `Zend_Mime::ENCODING_QUOTEDPRINTABLE`: 'quoted-printable'
- `Zend_Mime::ENCODING_BASE64`: 'base64'
- `Zend_Mime::DISPOSITION_ATTACHMENT`: 'attachment'
- `Zend_Mime::DISPOSITION_INLINE`: 'inline'
- `Zend_Mime::MULTIPART_ALTERNATIVE`: 'multipart/alternative'
- `Zend_Mime::MULTIPART_MIXED`: 'multipart/mixed'
- `Zend_Mime::MULTIPART_RELATED`: 'multipart/related'

1.3. Instantiating Zend_Mime

When Instantiating a Zend_Mime Object, a MIME boundary is stored that is used for all subsequent non-static method calls on that object. If the constructor is called with a string parameter, this value is used as a MIME boundary. If not, a random MIME boundary is generated during construction time.

A Zend_Mime object has the following Methods:

- boundary(): Returns the MIME boundary string.
- boundaryLine(): Returns the complete MIME boundary line.
- mimeEnd(): Returns the complete MIME end boundary line.

2. Zend_Mime_Message

2.1. Introduction

Zend_Mime_Message represents a MIME compliant message that can contain one or more separate Parts (Represented as Zend_Mime_Part objects). With Zend_Mime_Message, MIME compliant multipart messages can be generated from Zend_Mime_Part objects. Encoding and Boundary handling are handled transparently by the class. Zend_Mime_Message objects can also be reconstructed from given strings (experimental). Used by Zend_Mail.

2.2. Instantiation

There is no explicit constructor for Zend_Mime_Message.

2.3. Adding MIME Parts

Zend_Mime_Part Objects can be added to a given Zend_Mime_Message object by calling ->addPart($part)

An array with all Zend_Mime_Part objects in the Zend_Mime_Message is returned from the method ->getParts(). The Zend_Mime_Part objects can then be changed since they are stored in the array as references. If parts are added to the array or the sequence is changed, the array needs to be given back to the Zend_Mime_Part object by calling ->setParts($partsArray).

The function ->isMultiPart() will return TRUE if more than one part is registered with the Zend_Mime_Message object and thus the object would generate a Multipart-Mime-Message when generating the actual output.

2.4. Boundary handling

Zend_Mime_Message usually creates and uses its own Zend_Mime Object to generate a boundary. If you need to define the boundary or want to change the behaviour of the Zend_Mime object used by Zend_Mime_Message, you can instantiate the Zend_Mime object yourself and then register it to Zend_Mime_Message. Usually you will not need to do this. ->setMime(Zend_Mime $mime) sets a special instance of Zend_Mime to be used by this Zend_Mime_Message

->getMime() returns the instance of Zend_Mime that will be used to render the message when generateMessage() is called.

`->generateMessage()` renders the `Zend_Mime_Message` content to a string.

2.5. parsing a string to create a Zend_Mime_Message object (experimental)

A given MIME compliant message in string form can be used to reconstruct a `Zend_Mime_Message` Object from it. `Zend_Mime_Message` has a static factory Method to parse this String and return a `Zend_Mime_Message` Object.

`Zend_Mime_Message::createFromMessage($str, $boundary)` decodes the given string and returns a `Zend_Mime_Message` Object that can then be examined using `->getParts()`

3. Zend_Mime_Part

3.1. Introduction

This class represents a single part of a MIME message. It contains the actual content of the message part plus information about its encoding, content type and original filename. It provides a method for generating a string from the stored data. `Zend_Mime_Part` objects can be added to `Zend_Mime_Message` to assemble a complete multipart message.

3.2. Instantiation

`Zend_Mime_Part` is instantiated with a string that represents the content of the new part. The type is assumed to be OCTET-STREAM, encoding is 8Bit. After instantiating a `Zend_Mime_Part`, meta information can be set by accessing its attributes directly:

```
public $type = Zend_Mime::TYPE_OCTETSTREAM;
public $encoding = Zend_Mime::ENCODING_8BIT;
public $id;
public $disposition;
public $filename;
public $description;
public $charset;
public $boundary;
public $location;
public $language;
```

3.3. Methods for rendering the message part to a string

`getContent()` returns the encoded content of the MimePart as a string using the encoding specified in the attribute $encoding. Valid values are Zend_Mime::ENCODING_* Characterset conversions are not performed.

`getHeaders()` returns the Mime-Headers for the MimePart as generated from the information in the publicly accessible attributes. The attributes of the object need to be set correctly before this method is called.

- `$charset` has to be set to the actual charset of the content if it is a text type (Text or HTML).
- `$id` may be set to identify a content-id for inline images in a HTML mail.
- `$filename` contains the name the file will get when downloading it.

- `$disposition` defines if the file should be treated as an attachment or if it is used inside the (HTML-) mail (inline).
- `$description` is only used for informational purposes.
- `$boundary` defines string as boundary.
- `$location` can be used as resource URI that has relation to the content.
- `$language` defines languages in the content.

Zend_Navigation

1. Introduction

`Zend_Navigation` is a component for managing trees of pointers to web pages. Simply put: It can be used for creating menus, breadcrumbs, links, and sitemaps, or serve as a model for other navigation related purposes.

1.1. Pages and Containers

There are two main concepts in `Zend_Navigation`:

1.1.1. Pages

A page (`Zend_Navigation_Page`) in `Zend_Navigation` – in its most basic form – is an object that holds a pointer to a web page. In addition to the pointer itself, the page object contains a number of other properties that are typically relevant for navigation, such as `label`, `title`, etc.

Read more about pages in the pages section.

1.1.2. Containers

A navigation container (`Zend_Navigation_Container`) is a container class for pages. It has methods for adding, retrieving, deleting and iterating pages. It implements the SPL interfaces `RecursiveIterator` and `Countable`, and can thus be iterated with SPL iterators such as `RecursiveIteratorIterator`.

Read more about containers in the containers section.

`Zend_Navigation_Page` extends `Zend_Navigation_Container`, which means that a page can have sub pages.

1.2. Separation of data (model) and rendering (view)

Classes in the `Zend_Navigation` namespace do not deal with rendering of navigational elements. Rendering is done with navigational view helpers. However, pages contain information that is used by view helpers when rendering, such as; label, CSS class, title, `lastmod` and `priority` properties for sitemaps, etc.

Read more about rendering navigational elements in the manual section on navigation helpers.

2. Pages

`Zend_Navigation` ships with two page types:

- MVC pages – using the class `Zend_Navigation_Page_Mvc`
- URI pages – using the class `Zend_Navigation_Page_Uri`

MVC pages are link to on-site web pages, and are defined using MVC parameters (`action`, `controller`, `module`, `route`, `params`). URI pages are defined by a single property `uri`, which give you the full flexibility to link off-site pages or do other things with the generated links (e.g. an URI that turns into `<a href="#">foo<a>`).

2.1. Common page features

All page classes must extend `Zend_Navigation_Page`, and will thus share a common set of features and properties. Most notably they share the options in the table below and the same initialization process.

Option keys are mapped to `set` methods. This means that the option `order` maps to the method `setOrder()`, and `reset_params` maps to the method `setResetParams()`. If there is no setter method for the option, it will be set as a custom property of the page.

Read more on extending `Zend_Navigation_Page` in Creating custom page types.

Table 106. Common page options

Key	Type	Default	Description
`label`	String	`NULL`	A page label, such as 'Home' or 'Blog'.
`id`	String | `int`	`NULL`	An id tag/attribute that may be used when rendering the page, typically in an anchor element.
`class`	String	`NULL`	A CSS class that may be used when rendering the page, typically in an anchor element.
`title`	String	`NULL`	A short page description, typically for using as the `title` attribute in an anchor.
`target`	String	`NULL`	Specifies a target that may be used for the page, typically in an anchor element.
`rel`	Array	`array()`	Specifies forward relations for the page. Each element in the array is a key-value pair, where the key designates the relation/link type, and the value is a pointer to the linked page. An example of a key-value pair is `'alternate' => 'format/plain.html'`. To allow full flexbility, there are no restrictions on relation values. The value does not have to be a string.

Key	Type	Default	Description
			Read more about `rel` and `rev` in the section on the Links helper..
`rev`	Array	`array()`	Specifies reverse relations for the page. Works exactly like `rel`.
`order`	String | `int` | `NULL`	`NULL`	Works like order for elements in `Zend_Form`. If specified, the page will be iterated in a specific order, meaning you can force a page to be iterated before others by setting the `order` attribute to a low number, e.g. -100. If a String is given, it must parse to a valid `int`. If `NULL` is given, it will be reset, meaning the order in which the page was added to the container will be used.
`resource`	String | `Zend_Acl_Resource_Interface` | `NULL`	`NULL`	ACL resource to associate with the page. Read more in the section on ACL integration in view helpers..
`privilege`	String | `NULL`	`NULL`	ACL privilege to associate with the page. Read more in the section on ACL integration in view helpers..
`active`	`bool`	`FALSE`	Whether the page should be considered active for the current request. If active is `FALSE` or not given, MVC pages will check its properties against the request object upon calling `$page->isActive()`.
`visible`	`bool`	`TRUE`	Whether page should be visible for the user, or just be a part of

Key	Type	Default	Description
			the structure. Invisible pages are skipped by view helpers.
pages	Array | Zend_Config | NULL	NULL	Child pages of the page. This could be an Array or Zend_Config object containing either page options that can be passed to the factory() method, or actual Zend_Navigation_Page instances, or a mixture of both.

Custom properties

All pages support setting and getting of custom properties by use of the magic methods `__set($name, $value)`, `__get($name)`, `__isset($name)` and `__unset($name)`. Custom properties may have any value, and will be included in the array that is returned from `$page->toArray()`, which means that pages can be serialized/deserialized successfully even if the pages contains properties that are not native in the page class.

Both native and custom properties can be set using `$page->set($name, $value)` and retrieved using `$page->get($name)`, or by using magic methods.

Example 557. Custom page properties

This example shows how custom properties can be used.

```
$page = new Zend_Navigation_Page_Mvc();
$page->foo = 'bar';
$page->meaning = 42;

echo $page->foo;

if ($page->meaning != 42) {
    // action should be taken
}
```

2.2. Zend_Navigation_Page_Mvc

MVC pages are defined using MVC parameters known from the `Zend_Controller` component. An MVC page will use `Zend_Controller_Action_Helper_Url` internally in the `getHref()` method to generate hrefs, and the `isActive()` method will intersect the `Zend_Controller_Request_Abstract` params with the page's params to determine if the page is active.

Table 107. MVC page options

Key	Type	Default	Description
`action`	String	`NULL`	Action name to use when generating href to the page.
`controller`	String	`NULL`	Controller name to use when generating href to the page.
`module`	String	`NULL`	Module name to use when generating href to the page.
`params`	Array	`array()`	User params to use when generating href to the page.
`route`	String	`NULL`	Route name to use when generating href to the page.
`reset_params`	`bool`	`TRUE`	Whether user params should be reset when generating href to the page.

The three examples below assume a default MVC setup with the `default` route in place.

The URI returned is relative to the `baseUrl` in `Zend_Controller_Front`. In the examples, the baseUrl is '/' for simplicity.

Example 558. getHref() generates the page URI

This example show that MVC pages use `Zend_Controller_Action_Helper_Url` internally to generate URIs when calling `$page->getHref()`.

```
// getHref() returns /
$page = new Zend_Navigation_Page_Mvc(array(
    'action'     => 'index',
    'controller' => 'index'
));

// getHref() returns /blog/post/view
$page = new Zend_Navigation_Page_Mvc(array(
    'action'     => 'view',
    'controller' => 'post',
    'module'     => 'blog'
));

// getHref() returns /blog/post/view/id/1337
$page = new Zend_Navigation_Page_Mvc(array(
    'action'     => 'view',
    'controller' => 'post',
    'module'     => 'blog',
    'params'     => array('id' => 1337)
));
```

Example 559. isActive() determines if page is active

This example show that MVC pages determine whether they are active by using the params found in the request object.

```
/*
 * Dispatched request:
 * - module:     default
 * - controller: index
 * - action:     index
 */
$page1 = new Zend_Navigation_Page_Mvc(array(
    'action'     => 'index',
    'controller' => 'index'
));

$page2 = new Zend_Navigation_Page_Mvc(array(
    'action'     => 'bar',
    'controller' => 'index'
));

$page1->isActive(); // returns true
$page2->isActive(); // returns false

/*
 * Dispatched request:
 * - module:     blog
 * - controller: post
 * - action:     view
 * - id:         1337
 */
$page = new Zend_Navigation_Page_Mvc(array(
    'action'     => 'view',
    'controller' => 'post',
    'module'     => 'blog'
));

// returns true, because request has the same module, controller and action
$page->isActive();

/*
 * Dispatched request:
 * - module:     blog
 * - controller: post
 * - action:     view
 */
$page = new Zend_Navigation_Page_Mvc(array(
    'action'     => 'view',
    'controller' => 'post',
    'module'     => 'blog',
    'params'     => array('id' => null)
));

// returns false, because page requires the id param to be set in the request
$page->isActive(); // returns false
```

Example 560. Using routes

Routes can be used with MVC pages. If a page has a route, this route will be used in `getHref()` to generate the URL for the page.

Note that when using the `route` property in a page, you should also specify the default params that the route defines (module, controller, action, etc.), otherwise the `isActive()` method will not be able to determine if the page is active. The reason for this is that there is currently no way to get the default params from a `Zend_Controller_Router_Route_Interface` object, nor to retrieve the current route from a `Zend_Controller_Router_Interface` object.

```
// the following route is added to the ZF router
Zend_Controller_Front::getInstance()->getRouter()->addRoute(
    'article_view', // route name
    new Zend_Controller_Router_Route(
        'a/:id',
        array(
            'module'     => 'news',
            'controller' => 'article',
            'action'     => 'view',
            'id'         => null
        )
    )
);

// a page is created with a 'route' option
$page = new Zend_Navigation_Page_Mvc(array(
    'label'      => 'A news article',
    'route'      => 'article_view',
    'module'     => 'news',    // required for isActive(), see note above
    'controller' => 'article', // required for isActive(), see note above
    'action'     => 'view',    // required for isActive(), see note above
    'params'     => array('id' => 42)
));

// returns: /a/42
$page->getHref();
```

2.3. Zend_Navigation_Page_Uri

Pages of type `Zend_Navigation_Page_Uri` can be used to link to pages on other domains or sites, or to implement custom logic for the page. URI pages are simple; in addition to the common page options, a URI page takes only one option — `uri`. The `uri` will be returned when calling `$page->getHref()`, and may be a String or `NULL`.

`Zend_Navigation_Page_Uri` will not try to determine whether it should be active when calling `$page->isActive()`. It merely returns what currently is set, so to make a URI page active you have to manually call `$page->setActive()` or specifying `active` as a page option when constructing.

Table 108. URI page options

Key	Type	Default	Description
`uri`	String	`NULL`	URI to page. This can be any string or `NULL`.

2.4. Creating custom page types

When extending `Zend_Navigation_Page`, there is usually no need to override the constructor or the methods `setOptions()` or `setConfig()`. The page constructor takes a single parameter, an Array or a `Zend_Config` object, which is passed to `setOptions()` or `setConfig()` respectively. Those methods will in turn call `set()` method, which will map options to native or custom properties. If the option `internal_id` is given, the method will first look for a method named `setInternalId()`, and pass the option to this method if it exists. If the method does not exist, the option will be set as a custom property of the page, and be accessible via `$internalId = $page->internal_id;` or `$internalId = $page->get('internal_id');`.

Example 561. The most simple custom page

The only thing a custom page class needs to implement is the `getHref()` method.

```
class My_Simple_Page extends Zend_Navigation_Page
{
    public function getHref()
    {
        return 'something-completely-different';
    }
}
```

Example 562. A custom page with properties

When adding properties to an extended page, there is no need to override/modify `setOptions()` or `setConfig()`.

```
class My_Navigation_Page extends Zend_Navigation_Page
{
    private $_foo;
    private $_fooBar;

    public function setFoo($foo)
    {
        $this->_foo = $foo;
    }

    public function getFoo()
    {
        return $this->_foo;
    }

    public function setFooBar($fooBar)
    {
        $this->_fooBar = $fooBar;
    }

    public function getFooBar()
    {
        return $this->_fooBar;
    }

    public function getHref()
    {
        return $this->foo . '/' . $this->fooBar;
    }
}

// can now construct using
$page = new My_Navigation_Page(array(
    'label'   => 'Property names are mapped to setters',
    'foo'     => 'bar',
    'foo_bar' => 'baz'
));

// ...or
$page = Zend_Navigation_Page::factory(array(
    'type'    => 'My_Navigation_Page',
    'label'   => 'Property names are mapped to setters',
    'foo'     => 'bar',
    'foo_bar' => 'baz'
));
```

2.5. Creating pages using the page factory

All pages (also custom classes), can be created using the page factory, `Zend_Navigation_Page::factory()`. The factory can take an array with options, or a `Zend_Config` object. Each key in the array/config corresponds to a page option, as seen in the section on Pages. If the option `uri` is given and no MVC options are given (`action`,

`controller, module, route`), an URI page will be created. If any of the MVC options are given, an MVC page will be created.

If `type` is given, the factory will assume the value to be the name of the class that should be created. If the value is `mvc` or `uri` and MVC/URI page will be created.

Example 563. Creating an MVC page using the page factory

```
$page = Zend_Navigation_Page::factory(array(
    'label'  => 'My MVC page',
    'action' => 'index'
));

$page = Zend_Navigation_Page::factory(array(
    'label'      => 'Search blog',
    'action'     => 'index',
    'controller' => 'search',
    'module'     => 'blog'
));

$page = Zend_Navigation_Page::factory(array(
    'label'      => 'Home',
    'action'     => 'index',
    'controller' => 'index',
    'module'     => 'index',
    'route'      => 'home'
));

$page = Zend_Navigation_Page::factory(array(
    'type'   => 'mvc',
    'label'  => 'My MVC page'
));
```

Example 564. Creating a URI page using the page factory

```
$page = Zend_Navigation_Page::factory(array(
    'label' => 'My URI page',
    'uri'   => 'http://www.example.com/'
));

$page = Zend_Navigation_Page::factory(array(
    'label'  => 'Search',
    'uri'    => 'http://www.example.com/search',
    'active' => true
));

$page = Zend_Navigation_Page::factory(array(
    'label' => 'My URI page',
    'uri'   => '#'
));

$page = Zend_Navigation_Page::factory(array(
    'type'   => 'uri',
    'label'  => 'My URI page'
));
```

Example 565. Creating a custom page type using the page factory

To create a custom page type using the factory, use the option `type` to specify a class name to instantiate.

```
class My_Navigation_Page extends Zend_Navigation_Page
{
    protected $_fooBar = 'ok';

    public function setFooBar($fooBar)
    {
        $this->_fooBar = $fooBar;
    }
}

$page = Zend_Navigation_Page::factory(array(
    'type'    => 'My_Navigation_Page',
    'label'   => 'My custom page',
    'foo_bar' => 'foo bar'
));
```

3. Containers

Containers have methods for adding, retrieving, deleting and iterating pages. Containers implement the SPL interfaces `RecursiveIterator` and `Countable`, meaning that a container can be iterated using the SPL `RecursiveIteratorIterator` class.

3.1. Creating containers

`Zend_Navigation_Container` is abstract, and can not be instantiated directly. Use `Zend_Navigation` if you want to instantiate a container.

`Zend_Navigation` can be constructed entirely empty, or take an array or a `Zend_Config` object with pages to put in the container. Each page in the given array/config will eventually be passed to the `addPage()` method of the container class, which means that each element in the array/config can be an array or a config object, or a `Zend_Navigation_Page` instance.

```
                'uri' => '#',
                'pages' => array(
                    array(
                        'label' => 'Page 5.1.1',
                        'uri' => '#',
                        'pages' => array(
                            array(
                                'label' => 'Page 5.1.2',
                                'uri' => '#',
                                // let's say this page is active
                                'active' => true
                            )
                        )
                    )
                )
            )
        ),
        array(
            'label' => 'ACL page 1 (guest)',
            'uri' => '#acl-guest',
            'resource' => 'nav-guest',
            'pages' => array(
                array(
                    'label' => 'ACL page 1.1 (foo)',
                    'uri' => '#acl-foo',
                    'resource' => 'nav-foo'
                ),
                array(
                    'label' => 'ACL page 1.2 (bar)',
                    'uri' => '#acl-bar',
                    'resource' => 'nav-bar'
                ),
                array(
                    'label' => 'ACL page 1.3 (baz)',
                    'uri' => '#acl-baz',
                    'resource' => 'nav-baz'
                ),
                array(
                    'label' => 'ACL page 1.4 (bat)',
                    'uri' => '#acl-bat',
                    'resource' => 'nav-bat'
                )
            )
        ),
        array(
            'label' => 'ACL page 2 (member)',
            'uri' => '#acl-member',
            'resource' => 'nav-member'
        ),
        array(
            'label' => 'ACL page 3 (admin',
            'uri' => '#acl-admin',
            'resource' => 'nav-admin',
            'pages' => array(
                array(
                    'label' => 'ACL page 3.1 (nothing)',
                    'uri' => '#acl-nada'
                )
            )
        ),
        array(
            'label' => 'Zend Framework',
            'route' => 'zf-route'
        )
));
```

```
            <pages>

                <page3_1>
                    <label>Page 3.1</label>
                    <uri>page3/page3_1</uri>
                    <resource>guest</resource>
                </page3_1>

                <page3_2>
                    <label>Page 3.2</label>
                    <uri>page3/page3_2</uri>
                    <resource>member</resource>
                    <pages>

                        <page3_2_1>
                            <label>Page 3.2.1</label>
                            <uri>page3/page3_2/page3_2_1</uri>
                        </page3_2_1>

                        <page3_2_2>
                            <label>Page 3.2.2</label>
                            <uri>page3/page3_2/page3_2_2</uri>
                            <resource>admin</resource>
                        </page3_2_2>

                    </pages>
                </page3_2>

                <page3_3>
                    <label>Page 3.3</label>
                    <uri>page3/page3_3</uri>
                    <resource>special</resource>
                    <pages>

                        <page3_3_1>
                            <label>Page 3.3.1</label>
                            <uri>page3/page3_3/page3_3_1</uri>
                            <visible>0</visible>
                        </page3_3_1>

                        <page3_3_2>
                            <label>Page 3.3.2</label>
                            <uri>page3/page3_3/page3_3_2</uri>
                            <resource>admin</resource>
                        </page3_3_2>

                    </pages>
                </page3_3>

            </pages>
        </page3>

        <home>
            <label>Home</label>
            <order>-100</order>
            <module>default</module>
            <controller>index</controller>
            <action>index</action>
        </home>

    </nav>
</config>
 */

$config = new Zend_Config_Xml('/path/to/navigation.xml', 'nav');
$container = new Zend_Navigation($config);
```

3.2. Adding pages

Adding pages to a container can be done with the methods `addPage()`, `addPages()`, or `setPages()`. See examples below for explanation.

Example 568. Adding pages to a container

```
// create container
$container = new Zend_Navigation();

// add page by giving a page instance
$container->addPage(Zend_Navigation_Page::factory(array(
    'uri' => 'http://www.example.com/'
)))

// add page by giving an array
$container->addPage(array(
    'uri' => 'http://www.example.com/'
)))

// add page by giving a config object
$container->addPage(new Zend_Config(array(
    'uri' => 'http://www.example.com/'
)))

$pages = array(
    array(
        'label'  => 'Save'
        'action' => 'save',
    ),
    array(
        'label' =>  'Delete',
        'action' => 'delete'
    )
);

// add two pages
$container->addPages($pages);

// remove existing pages and add the given pages
$container->setPages($pages);
```

3.3. Removing pages

Removing pages can be done with `removePage()` or `removePages()`. The first method accepts a an instance of a page, or an integer. The integer corresponds to the `order` a page has. The latter method will remove all pages in the container.

Example 569. Removing pages from a container

```
$container = new Zend_Navigation(array(
    array(
        'label'  => 'Page 1',
        'action' => 'page1'
    ),
    array(
        'label'  => 'Page 2',
        'action' => 'page2',
        'order'  => 200
    ),
    array(
        'label'  => 'Page 3',
        'action' => 'page3'
    )
));

// remove page by implicit page order
$container->removePage(0);      // removes Page 1

// remove page by instance
$page3 = $container->findOneByAction('Page 3');
$container->removePage($page3); // removes Page 3

// remove page by explicit page order
$container->removePage(200);    // removes Page 2

// remove all pages
$container->removePages();      // removes all pages
```

3.4. Finding pages

Containers have finder methods for retrieving pages. They are `findOneBy($property, $value)`, `findAllBy($property, $value)`, and `findBy($property, $value, $all = false)`. Those methods will recursively search the container for pages matching the given `$page->$property == $value`. The first method, `findOneBy()`, will return a single page matching the property with the given value, or `NULL` if it cannot be found. The second method will return all pages with a property matching the given value. The third method will call one of the two former methods depending on the `$all` flag.

The finder methods can also be used magically by appending the property name to `findBy`, `findOneBy`, or `findAllBy`, e.g. `findOneByLabel('Home')` to return the first matching page with label `Home`. Other combinations are `findByLabel(...)`, `findOnyByTitle(...)`, `findAllByController(...)`, etc. Finder methods also work on custom properties, such as `findByFoo('bar')`.

```
                'label' => 'Page 1.2',
                'uri'   => 'page-1.2',
                'class' => 'my-class',
            ),
            array(
                'type'   => 'uri',
                'label'  => 'Page 1.3',
                'uri'    => 'page-1.3',
                'action' => 'about'
            )
        )
    ),
    array(
        'label'      => 'Page 2',
        'id'         => 'page_2_and_3',
        'class'      => 'my-class',
        'module'     => 'page2',
        'controller' => 'index',
        'action'     => 'page1'
    ),
    array(
        'label'      => 'Page 3',
        'id'         => 'page_2_and_3',
        'module'     => 'page3',
        'controller' => 'index'
    )
));

// The 'id' is not required to be unique, but be aware that
// having two pages with the same id will render the same id attribute
// in menus and breadcrumbs.
$found = $container->findBy('id',
                            'page_2_and_3');      // returns Page 2
$found = $container->findOneBy('id',
                               'page_2_and_3');   // returns Page 2
$found = $container->findBy('id',
                            'page_2_and_3',
                            true);                // returns Page 2 and Page 3
$found = $container->findById('page_2_and_3');    // returns Page 2
$found = $container->findOneById('page_2_and_3'); // returns Page 2
$found = $container->findAllById('page_2_and_3'); // returns Page 2 and Page 3

// Find all matching CSS class my-class
$found = $container->findAllBy('class',
                               'my-class');       // returns Page 1.2 and Page 2
$found = $container->findAllByClass('my-class');  // returns Page 1.2 and Page 2

// Find first matching CSS class my-class
$found = $container->findOneByClass('my-class');  // returns Page 1.2

// Find all matching CSS class non-existant
$found = $container->findAllByClass('non-existant'); // returns array()

// Find first matching CSS class non-existant
$found = $container->findOneByClass('non-existant'); // returns null

// Find all pages with custom property 'foo' = 'bar'
$found = $container->findAllBy('foo', 'bar'); // returns Page 1 and Page 1.1

// To achieve the same magically, 'foo' must be in lowercase.
// This is because 'foo' is a custom property, and thus the
// property name is not normalized to 'Foo'
$found = $container->findAllByfoo('bar');

// Find all with controller = 'index'
$found = $container->findAllByController('index'); // returns Page 2 and Page 3
```

3.5. Iterating containers

`Zend_Navigation_Container` implements `RecursiveIteratorIterator`, and can be iterated using any `Iterator` class. To iterate a container recursively, use the `RecursiveIteratorIterator` class.

Example 571. Iterating a container

```
/*
 * Create a container from an array
 */
$container = new Zend_Navigation(array(
    array(
        'label' => 'Page 1',
        'uri'   => '#'
    ),
    array(
        'label' => 'Page 2',
        'uri'   => '#',
        'pages' => array(
            array(
                'label' => 'Page 2.1',
                'uri'   => '#'
            ),
            array(
                'label' => 'Page 2.2',
                'uri'   => '#'
            )
        )
    )
    array(
        'label' => 'Page 3',
        'uri'   => '#'
    )
));

// Iterate flat using regular foreach:
// Output: Page 1, Page 2, Page 3
foreach ($container as $page) {
    echo $page->label;
}

// Iterate recursively using RecursiveIteratorIterator
$it = new RecursiveIteratorIterator(
        $container, RecursiveIteratorIterator::SELF_FIRST);

// Output: Page 1, Page 2, Page 2.1, Page 2.2, Page 3
foreach ($it as $page) {
    echo $page->label;
}
```

3.6. Other operations

The method `hasPage(Zend_Navigation_Page $page)` checks if the container has the given page. The method `hasPages()` checks if there are any pages in the container, and is equivalent to `count($container) > 1`.

The `toArray()` method converts the container and the pages in it to an array. This can be useful for serializing and debugging.

```
    }
    ["uri"]=> string(1) "#"
  }
  [1]=> array(15) {
    ["label"]=> string(6) "Page 2"
    ["id"]=> NULL
    ["class"]=> NULL
    ["title"]=> NULL
    ["target"]=> NULL
    ["rel"]=> array(0) {
    }
    ["rev"]=> array(0) {
    }
    ["order"]=> NULL
    ["resource"]=> NULL
    ["privilege"]=> NULL
    ["active"]=> bool(false)
    ["visible"]=> bool(true)
    ["type"]=> string(23) "Zend_Navigation_Page_Uri"
    ["pages"]=> array(2) {
      [0]=> array(15) {
        ["label"]=> string(8) "Page 2.1"
        ["id"]=> NULL
        ["class"]=> NULL
        ["title"]=> NULL
        ["target"]=> NULL
        ["rel"]=> array(0) {
        }
        ["rev"]=> array(0) {
        }
        ["order"]=> NULL
        ["resource"]=> NULL
        ["privilege"]=> NULL
        ["active"]=> bool(false)
        ["visible"]=> bool(true)
        ["type"]=> string(23) "Zend_Navigation_Page_Uri"
        ["pages"]=> array(0) {
        }
        ["uri"]=> string(1) "#"
      }
      [1]=>
      array(15) {
        ["label"]=> string(8) "Page 2.2"
        ["id"]=> NULL
        ["class"]=> NULL
        ["title"]=> NULL
        ["target"]=> NULL
        ["rel"]=> array(0) {
        }
        ["rev"]=> array(0) {
        }
        ["order"]=> NULL
        ["resource"]=> NULL
        ["privilege"]=> NULL
        ["active"]=> bool(false)
        ["visible"]=> bool(true)
        ["type"]=> string(23) "Zend_Navigation_Page_Uri"
        ["pages"]=> array(0) {
        }
        ["uri"]=> string(1) "#"
      }
    }
    ["uri"]=> string(1) "#"
  }
}
*/
```

Zend_Oauth

1. Introduction to OAuth

OAuth allows you to approve access by any application to your private data stored a website without being forced to disclose your username or password. If you think about it, the practice of handing over your username and password for sites like Yahoo Mail or Twitter has been endemic for quite a while. This has raised some serious concerns because there's nothing to prevent other applications from misusing this data. Yes, some services may appear trustworthy but that is never guaranteed. OAuth resolves this problem by eliminating the need for any username and password sharing, replacing it with a user controlled authorization process.

This authorization process is token based. If you authorize an application (and by application we can include any web based or desktop application) to access your data, it will be in receipt of an Access Token associated with your account. Using this Access Token, the application can access your private data without continually requiring your credentials. In all this authorization delegation style of protocol is simply a more secure solution to the problem of accessing private data via any web service API.

OAuth is not a completely new idea, rather it is a standardized protocol building on the existing properties of protocols such as Google AuthSub, Yahoo BBAuth, Flickr API, etc. These all to some extent operate on the basis of exchanging user credentials for an Access Token of some description. The power of a standardized specification like OAuth is that it only requires a single implementation as opposed to many disparate ones depending on the web service. This standardization has not occurred independently of the major players, and indeed many now support OAuth as an alternative and future replacement for their own solutions.

Zend Framework's `Zend_Oauth` currently implements a full OAuth Consumer conforming to the OAuth Core 1.0 Revision A Specification (24 June 2009) via the `Zend_Oauth_Consumer` class.

1.1. Protocol Workflow

Before implementing OAuth it makes sense to understand how the protocol operates. To do so we'll take the example of Twitter which currently implements OAuth based on the OAuth Core 1.0 Revision A Specification. This example looks at the protocol from the perspectives of the User (who will approve access), the Consumer (who is seeking access) and the Provider (who holds the User's private data). Access may be read-only or read and write.

By chance, our User has decided that they want to utilise a new service called TweetExpress which claims to be capable of reposting your blog posts to Twitter in a manner of seconds. TweetExpress is a registered application on Twitter meaning that it has access to a Consumer Key and a Consumer Secret (all OAuth applications must have these from the Provider they will be accessing) which identify its requests to Twitter and that ensure all requests can be signed using the Consumer Secret to verify their origin.

To use TweetExpress you are asked to register for a new account, and after your registration is confirmed you are informed that TweetExpress will seek to associate your Twitter account with the service.

In the meantime TweetExpress has been busy. Before gaining your approval from Twitter, it has sent a HTTP request to Twitter's service asking for a new unauthorized Request Token. This token is not User specific from Twitter's perspective, but TweetExpress may use it specifically for the current User and should associate it with their account and store it for future use.

TweetExpress now redirects the User to Twitter so they can approve TweetExpress' access. The URL for this redirect will be signed using TweetExpress' Consumer Secret and it will contain the unauthorized Request Token as a parameter.

At this point the User may be asked to log into Twitter and will now be faced with a Twitter screen asking if they approve this request by TweetExpress to access Twitter's API on the User's behalf. Twitter will record the response which we'll assume was positive. Based on the User's approval, Twitter will record the current unauthorized Request Token as having been approved by the User (thus making it User specific) and will generate a new value in the form of a verification code. The User is now redirected back to a specific callback URL used by TweetExpress (this callback URL may be registered with Twitter or dynamically set using an oauth_callback parameter in requests). The redirect URL will contain the newly generated verification code.

TweetExpress' callback URL will trigger an examination of the response to determine whether the User has granted their approval to Twitter. Assuming so, it may now exchange it's unauthorized Request Token for a fully authorized Access Token by sending a request back to Twitter including the Request Token and the received verification code. Twitter should now send back a response containing this Access Token which must be used in all requests used to access Twitter's API on behalf of the User. Twitter will only do this once they have confirmed the attached Request Token has not already been used to retrieve another Access Token. At this point, TweetExpress may confirm the receipt of the approval to the User and delete the original Request Token which is no longer needed.

From this point forward, TweetExpress may use Twitter's API to post new tweets on the User's behalf simply by accessing the API endpoints with a request that has been digitally signed (via HMAC-SHA1) with a combination of TweetExpress' Consumer Secret and the Access Key being used.

Although Twitter do not currently expire Access Tokens, the User is free to deauthorize TweetExpress from their Twitter account settings. Once deauthorized, TweetExpress' access will be cut off and their Access Token rendered invalid.

1.2. Security Architecture

OAuth was designed specifically to operate over an insecure HTTP connection and so the use of HTTPS is not required though obviously it would be desireable if available. Should a HTTPS connection be feasible, OAuth offers a signature method implementation called PLAINTEXT which may be utilised. Over a typical unsecured HTTP connection, the use of PLAINTEXT must be avoided and an alternate scheme using. The OAuth specification defines two such signature methods: HMAC-SHA1 and RSA-SHA1. Both are fully supported by `Zend_Oauth`.

These signature methods are quite easy to understand. As you can imagine, a PLAINTEXT signature method does nothing that bears mentioning since it relies on HTTPS. If you were to use PLAINTEXT over HTTP, you are left with a significant problem: there's no way to be sure that the content of any OAuth enabled request (which would include the OAuth Access Token) was altered en route. This is because unsecured HTTP requests are always at risk of eavesdropping, Man In The Middle (MITM) attacks, or other risks whereby a request can be retooled so to speak to perform tasks on behalf of the attacker by masquerading as the origin application without being noticed by the service provider.

HMAC-SHA1 and RSA-SHA1 alleviate this risk by digitally signing all OAuth requests with the original application's registered Consumer Secret. Assuming only the Consumer and the Provider know what this secret is, a middle-man can alter requests all they wish - but they will not be able to validly sign them and unsigned or invalidly signed requests would be discarded by both parties. Digital signatures therefore offer a guarantee that validly signed requests do come from

the expected party and have not been altered en route. This is the core of why OAuth can operate over an unsecure connection.

How these digital signatures operate depends on the method used, i.e. HMAC-SHA1, RSA-SHA1 or perhaps another method defined by the service provider. HMAC-SHA1 is a simple mechanism which generates a Message Authentication Code (MAC) using a cryptographic hash function (i.e. SHA1) in combination with a secret key known only to the message sender and receiver (i.e. the OAuth Consumer Secret and the authorized Access Key combined). This hashing mechanism is applied to the parameters and content of any OAuth requests which are concatenated into a "base signature string" as defined by the OAuth specification.

RSA-SHA1 operates on similar principles except that the shared secret is, as you would expect, each parties' RSA private key. Both sides would have the other's public key with which to verify digital signatures. This does pose a level of risk compared to HMAC-SHA1 since the RSA method does not use the Access Key as part of the shared secret. This means that if the RSA private key of any Consumer is compromised, then all Access Tokens assigned to that Consumer are also. RSA imposes an all or nothing scheme. In general, the majority of service providers offering OAuth authorization have therefore tended to use HMAC-SHA1 by default, and those who offer RSA-SHA1 may offer fallback support to HMAC-SHA1.

While digital signatures add to OAuth's security they are still vulnerable to other forms of attack, such as replay attacks which copy earlier requests which were intercepted and validly signed at that time. An attacker can now resend the exact same request to a Provider at will at any time and intercept its results. This poses a significant risk but it is quiet simple to defend against - add a unique string (i.e. a nonce) to all requests which changes per request (thus continually changing the signature string) but which can never be reused because Providers actively track used nonces within the a certain window defined by the timestamp also attached to a request. You might first suspect that once you stop tracking a particular nonce, the replay could work but this ignore the timestamp which can be used to determine a request's age at the time it was validly signed. One can assume that a week old request used in an attempted replay should be summarily discarded!

As a final point, this is not an exhaustive look at the security architecture in OAuth. For example, what if HTTP requests which contain both the Access Token and the Consumer Secret are eavesdropped? The system relies on at one in the clear transmission of each unless HTTPS is active, so the obvious conclusion is that where feasible HTTPS is to be preferred leaving unsecured HTTP in place only where it is not possible or affordable to do so.

1.3. Getting Started

With the OAuth protocol explained, let's show a simple example of it with source code. Our new Consumer will be handling Twitter Status submissions. To do so, it will need to be registered with Twitter in order to receive an OAuth Consumer Key and Consumer Secret. This are utilised to obtain an Access Token before we use the Twitter API to post a status message.

Assuming we have obtained a key and secret, we can start the OAuth workflow by setting up a `Zend_Oauth_Consumer` instance as follows passing it a configuration (either an array or `Zend_Config` object).

```
$config = array(
    'callbackUrl' => 'http://example.com/callback.php',
    'siteUrl' => 'http://twitter.com/oauth',
    'consumerKey' => 'gg3DsFTW9OU9eWPnbuPzQ',
    'consumerSecret' => 'tFB0fyWLSMf74lkEu9FTyoHXcazOWpbrAjTCCK48A'
);
$consumer = new Zend_Oauth_Consumer($config);
```

The callbackUrl is the URI we want Twitter to request from our server when sending information. We'll look at this later. The siteUrl is the base URI of Twitter's OAuth API endpoints. The full list of endpoints include http://twitter.com/oauth/request_token, http://twitter.com/oauth/access_token, and http://twitter.com/oauth/authorize. The base siteUrl utilises a convention which maps to these three OAuth endpoints (as standard) for requesting a request token, access token or authorization. If the actual endpoints of any service differ from the standard set, these three URIs can be separately set using the methods `setRequestTokenUrl()`, `setAccessTokenUrl()`, and `setAuthorizeUrl()` or the configuration fields requestTokenUrl, accessTokenUrl and authorizeUrl.

The consumerKey and consumerSecret are retrieved from Twitter when your application is registered for OAuth access. These also apply to any OAuth enabled service, so each one will provide a key and secret for your application.

All of these configuration options may be set using method calls simply by converting from, e.g. callbackUrl to setCallbackUrl().

In addition, you should note several other configuration values not explicitly used: requestMethod and requestScheme. By default, `Zend_Oauth_Consumer` sends requests as POST (except for a redirect which uses GET). The customised client (see later) also includes its authorization by way of a header. Some services may, at their discretion, require alternatives. You can reset the requestMethod (which defaults to Zend_Oauth::POST) to Zend_Oauth::GET, for example, and reset the requestScheme from its default of Zend_Oauth::REQUEST_SCHEME_HEADER to one of Zend_Oauth::REQUEST_SCHEME_POSTBODY or Zend_Oauth::REQUEST_SCHEME_QUERYSTRING. Typically the defaults should work fine apart from some exceptional cases. Please refer to the service provider's documentation for more details.

The second area of customisation is how HMAC operates when calculating/comparing them for all requests. This is configured using the signatureMethod configuration field or `setSignatureMethod()` . By default this is HMAC-SHA1. You can set it also to a provider's preferred method including RSA-SHA1. For RSA-SHA1, you should also configure RSA private and public keys via the rsaPrivateKey and rsaPublicKey configuration fields or the `setRsaPrivateKey()` and `setRsaPublicKey()` methods.

The first part of the OAuth workflow is obtaining a request token. This is accomplished using:

```
$config = array(
    'callbackUrl' => 'http://example.com/callback.php',
    'siteUrl' => 'http://twitter.com/oauth',
    'consumerKey' => 'gg3DsFTW9OU9eWPnbuPzQ',
    'consumerSecret' => 'tFB0fyWLSMf74lkEu9FTyoHXcazOWpbrAjTCCK48A'
);
$consumer = new Zend_Oauth_Consumer($config);

// fetch a request token
$token = $consumer->getRequestToken();
```

The new request token (an instance of `Zend_Oauth_Token_Request`) is unauthorized. In order to exchange it for an authorized token with which we can access the Twitter API, we need the user to authorize it. We accomplish this by redirecting the user to Twitter's authorize endpoint via:

```
$config = array(
```

```
    'callbackUrl' => 'http://example.com/callback.php',
    'siteUrl' => 'http://twitter.com/oauth',
    'consumerKey' => 'gg3DsFTW9OU9eWPnbuPzQ',
    'consumerSecret' => 'tFB0fyWLSMf74lkEu9FTyoHXcazOWpbrAjTCCK48A'
);
$consumer = new Zend_Oauth_Consumer($config);

// fetch a request token
$token = $consumer->getRequestToken();

// persist the token to storage
$_SESSION['TWITTER_REQUEST_TOKEN'] = serialize($token);

// redirect the user
$consumer->redirect();
```

The user will now be redirected to Twitter. They will be asked to authorize the request token attached to the redirect URI's query string. Assuming they agree, and complete the authorization, they will be again redirected, this time to our Callback URL as previously set (note that the callback URL is also registered with Twitter when we registered our application).

Before redirecting the user, we should persist the request token to storage. For simplicity I'm just using the user's session, but you can easily use a database for the same purpose, so long as you tie the request token to the current user so it can be retrieved when they return to our application.

The redirect URI from Twitter will contain an authorized Access Token. We can include code to parse out this access token as follows - this source code would exist within the executed code of our callback URI. Once parsed we can discard the previous request token, and instead persist the access token for future use with the Twitter API. Again, we're simply persisting to the user session, but in reality an access token can have a long lifetime so it should really be stored to a database.

```
$config = array(
    'callbackUrl' => 'http://example.com/callback.php',
    'siteUrl' => 'http://twitter.com/oauth',
    'consumerKey' => 'gg3DsFTW9OU9eWPnbuPzQ',
    'consumerSecret' => 'tFB0fyWLSMf74lkEu9FTyoHXcazOWpbrAjTCCK48A'
);
$consumer = new Zend_Oauth_Consumer($config);

if (!empty($_GET) && isset($_SESSION['TWITTER_REQUEST_TOKEN'])) {
    $token = $consumer->getAccessToken($_GET, unserialize($_SESSION['TWITTER_REQUEST_TOKEN'
    $_SESSION['TWITTER_ACCESS_TOKEN'] = serialize($token);

    // Now that we have an Access Token, we can discard the Request Token
    $_SESSION['TWITTER_REQUEST_TOKEN'] = null;
} else {
    // Mistaken request? Some malfeasant trying something?
    exit('Invalid callback request. Oops. Sorry.');
}
```

Success! We have an authorized access token - so it's time to actually use the Twitter API. Since the access token must be included with every single API request, Zend_Oauth_Consumer offers a ready-to-go HTTP client (a subclass of `Zend_Http_Client`) to use either by itself or by passing it as a custom HTTP Client to another library or component. Here's an example of using it standalone. This can be done from anywhere in your application, so long as you can access the OAuth configuration and retrieve the final authorized access token.

```
$config = array(
    'callbackUrl' => 'http://example.com/callback.php',
    'siteUrl' => 'http://twitter.com/oauth',
    'consumerKey' => 'gg3DsFTW9OU9eWPnbuPzQ',
    'consumerSecret' => 'tFB0fyWLSMf74lkEu9FTyoHXcazOWpbrAjTCCK48A'
);

$statusMessage = 'I\'m posting to Twitter using Zend_Oauth!';

$token = unserialize($_SESSION['TWITTER_ACCESS_TOKEN']);
$client = $token->getHttpClient($configuration);
$client->setUri('http://twitter.com/statuses/update.json');
$client->setMethod(Zend_Http_Client::POST);
$client->setParameterPost('status', $statusMessage);
$response = $client->request();

$data = Zend_Json::decode($response->getBody());
$result = $response->getBody();
if (isset($data->text)) {
    $result = 'true';
}
echo $result;
```

As a note on the customised client, this can be passed to most Zend Framework service or other classes using `Zend_Http_Client` displacing the default client they would otherwise use.

Zend_OpenId

1. Introduction

`Zend_OpenId` is a Zend Framework component that provides a simple API for building OpenID-enabled sites and identity providers.

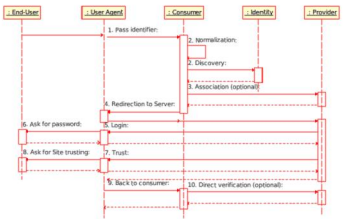
1.1. What is OpenID?

OpenID is a set of protocols for user-centric digital identities. These protocols allows users to create an identity online, using an identity provider. This identity can be used on any site that supports OpenID. Using OpenID-enabled sites, users do not need to remember traditional authentication tokens such as usernames and passwords for each site. All OpenID-enabled sites accept a single OpenID identity. This identity is typically a URL. It may be the URL of the user's personal page, blog or other resource that may provide additional information about them. That mean a user needs just one identifier for all sites he or she uses. services. OpenID is an open, decentralized, and free user-centric solution. Users may choose which OpenID provider to use, or even create their own personal identity server. No central authority is required to approve or register OpenID-enabled sites or identity providers.

For more information about OpenID visit the OpenID official site.

1.2. How Does it Work?

The purpose of the `Zend_OpenId` component is to implement the OpenID authentication protocol as described in the following sequence diagram:



1. Authentication is initiated by the end user, who passes their OpenID identifier to the OpenID consumer through a User-Agent.

2. The OpenID consumer performs normalization and discovery on the user-supplied identifier. Through this process, the consumer obtains the claimed identifier, the URL of the OpenID provider and an OpenID protocol version.

3. The OpenID consumer establishes an optional association with the provider using Diffie-Hellman keys. As a result, both parties have a common "shared secret" that is used for signing and verification of the subsequent messages.

4. The OpenID consumer redirects the User-Agent to the URL of the OpenID provider with an OpenID authentication request.

5. The OpenID provider checks if the User-Agent is already authenticated and, if not, offers to do so.

6. The end user enters the required password.

7. The OpenID provider checks if it is allowed to pass the user identity to the given consumer, and asks the user if necessary.
8. The user allows or disallows passing his identity.
9. The OpenID Provider redirects the User-Agent back to the OpenID consumer with an "authentication approved" or "failed" request.
10. The OpenID consumer verifies the information received from the provider by using the shared secret it got in step 3 or by sending an additional direct request to the OpenID provider.

1.3. Zend_OpenId Structure

`Zend_OpenId` consists of two sub-packages. The first one is `Zend_OpenId_Consumer` for developing OpenID-enabled sites, and the second is `Zend_OpenId_Provider` for developing OpenID servers. They are completely independent of each other and may be used separately.

The only common code used by these sub-packages are the OpenID Simple Registration Extension implemented by `Zend_OpenId_Extension_Sreg` class and a set of utility functions implemented by the `Zend_OpenId` class.

`Zend_OpenId` takes advantage of the GMP extension, where available. Consider enabling the GMP extension for enhanced performance when using `Zend_OpenId`.

1.4. Supported OpenID Standards

The `Zend_OpenId` component supports the following standards:

- OpenID Authentication protocol version 1.1
- OpenID Authentication protocol version 2.0 draft 11
- OpenID Simple Registration Extension version 1.0
- OpenID Simple Registration Extension version 1.1 draft 1

2. Zend_OpenId_Consumer Basics

`Zend_OpenId_Consumer` can be used to implement OpenID authentication for web sites.

2.1. OpenID Authentication

From a web site developer's point of view, the OpenID authentication process consists of three steps:

1. Show OpenID authentication form
2. Accept OpenID identity and pass it to the OpenID provider
3. Verify response from the OpenID provider

The OpenID authentication protocol actually requires more steps, but many of them are encapsulated inside `Zend_OpenId_Consumer` and are therefore transparent to the developer.

The end user initiates the OpenID authentication process by submitting his or her identification credentials with the appropriate form. The following example shows a simple form that accepts an OpenID identifier. Note that the example only demonstrates a login.

Example 573. The Simple OpenID Login form

```
<html><body>
<form method="post" action="example-1_2.php"><fieldset>
<legend>OpenID Login</legend>
<input type="text" name="openid_identifier">
<input type="submit" name="openid_action" value="login">
</fieldset></form></body></html>
```

This form passes the OpenID identity on submission to the following PHP script that performs the second step of authentication. The PHP script need only call the `Zend_OpenId_Consumer::login()` method in this step. The first argument of this method is an accepted OpenID identity, and the second is the URL of a script that handles the third and last step of authentication.

Example 574. The Authentication Request Handler

```
$consumer = new Zend_OpenId_Consumer();
if (!$consumer->login($_POST['openid_identifier'], 'example-1_3.php')) {
    die("OpenID login failed.");
}
```

The `Zend_OpenId_Consumer::login()` method performs discovery on a given identifier, and, if successful, obtains the address of the identity provider and its local identifier. It then creates an association to the given provider so that both the site and provider share a secret that is used to sign the subsequent messages. Finally, it passes an authentication request to the provider. This request redirects the end user's web browser to an OpenID server site, where the user can continue the authentication process.

An OpenID provider usually asks users for their password (if they weren't previously logged-in), whether the user trusts this site and what information may be returned to the site. These interactions are not visible to the OpenID consumer, so it can not obtain the user's password or other information that the user did not has not directed the OpenID provider to share with it.

On success, `Zend_OpenId_Consumer::login()` does not return, instead performing an HTTP redirection. However, if there is an error it may return FALSE. Errors may occur due to an invalid identity, unresponsive provider, communication error, etc.

The third step of authentication is initiated by the response from the OpenID provider, after it has authenticated the user's password. This response is passed indirectly, as an HTTP redirection using the end user's web browser. The consumer must now simply check that this response is valid.

Example 575. The Authentication Response Verifier

```
$consumer = new Zend_OpenId_Consumer();
if ($consumer->verify($_GET, $id)) {
    echo "VALID " . htmlspecialchars($id);
} else {
    echo "INVALID " . htmlspecialchars($id);
}
```

This check is performed using the `Zend_OpenId_Consumer::verify` method, which takes an array of the HTTP request's arguments and checks that this response is properly signed by the OpenID provider. It may assign the claimed OpenID identity that was entered by end user in the first step using a second, optional argument.

2.2. Combining all Steps in One Page

The following example combines all three steps in one script. It doesn't provide any new functionality. The advantage of using just one script is that the developer need not specify URL's for a script to handle the next step. By default, all steps use the same URL. However, the script now includes some dispatch code to execute the appropriate code for each step of authentication.

Example 576. The Complete OpenID Login Script

```
<?php
$status = "";
if (isset($_POST['openid_action']) &&
    $_POST['openid_action'] == "login" &&
    !empty($_POST['openid_identifier'])) {

    $consumer = new Zend_OpenId_Consumer();
    if (!$consumer->login($_POST['openid_identifier'])) {
        $status = "OpenID login failed.";
    }
} else if (isset($_GET['openid_mode'])) {
    if ($_GET['openid_mode'] == "id_res") {
        $consumer = new Zend_OpenId_Consumer();
        if ($consumer->verify($_GET, $id)) {
            $status = "VALID " . htmlspecialchars($id);
        } else {
            $status = "INVALID " . htmlspecialchars($id);
        }
    } else if ($_GET['openid_mode'] == "cancel") {
        $status = "CANCELLED";
    }
}
?>
<html><body>
<?php echo "$status<br>" ?>
<form method="post">
<fieldset>
<legend>OpenID Login</legend>
<input type="text" name="openid_identifier" value=""/>
<input type="submit" name="openid_action" value="login"/>
</fieldset>
</form>
</body></html>
```

In addition, this code differentiates between cancelled and invalid authentication responses. The provider returns a cancelled response if the identity provider is not aware of the supplied identity, the user is not logged in, or the user doesn't trust the site. An invalid response indicates that the response is not conformant to the OpenID protocol or is incorrectly signed.

2.3. Consumer Realm

When an OpenID-enabled site passes authentication requests to a provider, it identifies itself with a realm URL. This URL may be considered a root of a trusted site. If the user trusts the realm URL, he or she should also trust matched and subsequent URLs.

By default, the realm URL is automatically set to the URL of the directory in which the login script resides. This default value is useful for most, but not all, cases. Sometimes an entire domain, and not a directory should be trusted. Or even a combination of several servers in one domain.

To override the default value, developers may pass the realm URL as a third argument to the `Zend_OpenId_Consumer::login` method. In the following example, a single interaction asks for trusted access to all php.net sites.

Example 577. Authentication Request for Specified Realm

```
$consumer = new Zend_OpenId_Consumer();
if (!$consumer->login($_POST['openid_identifier'],
                      'example-3_3.php',
                      'http://*.php.net/')) {
    die("OpenID login failed.");
}
```

This example implements only the second step of authentication; the first and third steps are similar to the examples above.

2.4. Immediate Check

In some cases, an application need only check if a user is already logged in to a trusted OpenID server without any interaction with the user. The `Zend_OpenId_Consumer::check` method does precisely that. It is executed with the same arguments as `Zend_OpenId_Consumer::login`, but it doesn't display any OpenID server pages to the user. From the users point of view this process is transparent, and it appears as though they never left the site. The third step succeeds if the user is already logged in and trusted by the site, otherwise it will fail.

Example 578. Immediate Check without Interaction

```
$consumer = new Zend_OpenId_Consumer();
if (!$consumer->check($_POST['openid_identifier'], 'example-4_3.php')) {
    die("OpenID login failed.");
}
```

This example implements only the second step of authentication; the first and third steps are similar to the examples above.

2.5. Zend_OpenId_Consumer_Storage

There are three steps in the OpenID authentication procedure, and each step is performed by a separate HTTP request. To store information between requests, `Zend_OpenId_Consumer` uses internal storage.

Developers do not necessarily have to be aware of this storage because by default `Zend_OpenId_Consumer` uses file-based storage under the temporary directory- similar to PHP sessions. However, this storage may be not suitable in all cases. Some developers may want to store information in a database, while others may need to use common storage suitable for server farms. Fortunately, developers may easily replace the default storage with their own. To specify a custom storage mechanism, one need only extend the `Zend_OpenId_Consumer_Storage` class and pass this subclass to the `Zend_OpenId_Consumer` constructor in the first argument.

The following example demonstrates a simple storage mechanism that uses `Zend_Db` as its backend and exposes three groups of functions. The first group contains functions for working with associations, while the second group caches discovery information, and the third group can

be used to check whether a response is unique. This class can easily be used with existing or new databases; if the required tables don't exist, it will create them.

```
                                    $expires)
    {
        $table = $this->_discovery_table;
        $this->_db->insert($table, array(
            'id'      => $id,
            'realId'  => $realId,
            'server'  => $server,
            'version' => $version,
            'expires' => $expires,
        ));

        return true;
    }

    public function getDiscoveryInfo($id,
                                     &$realId,
                                     &$server,
                                     &$version,
                                     &$expires)
    {
        $table = $this->_discovery_table;
        $this->_db->delete($table, $this->quoteInto('expires < ?', time()));
        $select = $this->_db->select()
                ->from($table, array('realId', 'server', 'version', 'expires'))
                ->where('id = ?', $id);
        $res = $this->_db->fetchRow($select);

        if (is_array($res)) {
            $realId  = $res['realId'];
            $server  = $res['server'];
            $version = $res['version'];
            $expires = $res['expires'];
            return true;
        }
        return false;
    }

    public function delDiscoveryInfo($id)
    {
        $table = $this->_discovery_table;
        $this->_db->delete($table, $this->_db->quoteInto('id = ?', $id));
        return true;
    }

    public function isUniqueNonce($nonce)
    {
        $table = $this->_nonce_table;
        try {
            $ret = $this->_db->insert($table, array(
                'nonce' => $nonce,
            ));
        } catch (Zend_Db_Statement_Exception $e) {
            return false;
        }
        return true;
    }

    public function purgeNonces($date=null)
    {
    }
}

$db = Zend_Db::factory('Pdo_Sqlite',
    array('dbname'=>'/tmp/openid_consumer.db'));
$storage = new DbStorage($db);
$consumer = new Zend_OpenId_Consumer($storage);
```

This example doesn't list the OpenID authentication code itself, but this code would be the same as that for other examples in this chapter. examples.

2.6. Simple Registration Extension

In addition to authentication, the OpenID standard can be used for lightweight profile exchange to make information about a user portable across multiple sites. This feature is not covered by the OpenID authentication specification, but by the OpenID Simple Registration Extension protocol. This protocol allows OpenID-enabled sites to ask for information about end users from OpenID providers. Such information may include:

- *nickname* - any UTF-8 string that the end user uses as a nickname
- *email* - the email address of the user as specified in section 3.4.1 of RFC2822
- *fullname* - a UTF-8 string representation of the user's full name
- *dob* - the user's date of birth in the format 'YYYY-MM-DD'. Any values whose representation uses fewer than the specified number of digits in this format should be zero-padded. In other words, the length of this value must always be 10. If the end user does not want to reveal any particular part of this value (i.e., year, month or day), it must be set to zero. For example, if the user wants to specify that his date of birth falls in 1980, but not specify the month or day, the value returned should be '1980-00-00'.
- *gender* - the user's gender: "M" for male, "F" for female
- *postcode* - a UTF-8 string that conforms to the postal system of the user's country
- *country* - the user's country of residence as specified by ISO3166
- *language* - the user's preferred language as specified by ISO639
- *timezone* - an ASCII string from a TimeZone database. For example, "Europe/Paris" or "America/Los_Angeles".

An OpenID-enabled web site may ask for any combination of these fields. It may also strictly require some information and allow users to provide or hide additional information. The following example instantiates the `Zend_OpenId_Extension_Sreg` class, requiring a *nickname* and optionally requests an *email* and a *fullname*.

Example 580. Sending Requests with a Simple Registration Extension

```
$sreg = new Zend_OpenId_Extension_Sreg(array(
    'nickname'=>true,
    'email'=>false,
    'fullname'=>false), null, 1.1);
$consumer = new Zend_OpenId_Consumer();
if (!$consumer->login($_POST['openid_identifier'],
                      'example-6_3.php',
                      null,
                      $sreg)) {
    die("OpenID login failed.");
}
```

As you can see, the `Zend_OpenId_Extension_Sreg` constructor accepts an array of OpenID fields. This array has the names of fields as indexes to a flag indicating whether the field

is required; TRUE means the field is required and FALSE means the field is optional. The Zend_OpenId_Consumer::login method accepts an extension or an array of extensions as its fourth argument.

On the third step of authentication, the Zend_OpenId_Extension_Sreg object should be passed to Zend_OpenId_Consumer::verify. Then on successful authentication the Zend_OpenId_Extension_Sreg::getProperties method will return an associative array of requested fields.

Example 581. Verifying Responses with a Simple Registration Extension

```
$sreg = new Zend_OpenId_Extension_Sreg(array(
    'nickname'=>true,
    'email'=>false,
    'fullname'=>false), null, 1.1);
$consumer = new Zend_OpenId_Consumer();
if ($consumer->verify($_GET, $id, $sreg)) {
    echo "VALID " . htmlspecialchars($id) ."<br>\n";
    $data = $sreg->getProperties();
    if (isset($data['nickname'])) {
        echo "nickname: " . htmlspecialchars($data['nickname']) . "<br>\n";
    }
    if (isset($data['email'])) {
        echo "email: " . htmlspecialchars($data['email']) . "<br>\n";
    }
    if (isset($data['fullname'])) {
        echo "fullname: " . htmlspecialchars($data['fullname']) . "<br>\n";
    }
} else {
    echo "INVALID " . htmlspecialchars($id);
}
```

If the Zend_OpenId_Extension_Sreg object was created without any arguments, the user code should check for the existence of the required data itself. However, if the object is created with the same list of required fields as on the second step, it will automatically check for the existence of required data. In this case, Zend_OpenId_Consumer::verify will return FALSE if any of the required fields are missing.

Zend_OpenId_Extension_Sreg uses version 1.0 by default, because the specification for version 1.1 is not yet finalized. However, some libraries don't fully support version 1.0. For example, www.myopenid.com requires an SREG namespace in requests which is only available in 1.1. To work with such a server, you must explicitly set the version to 1.1 in the Zend_OpenId_Extension_Sreg constructor.

The second argument of the Zend_OpenId_Extension_Sreg constructor is a policy URL, that should be provided to the user by the identity provider.

2.7. Integration with Zend_Auth

Zend Framework provides a special class to support user authentication: Zend_Auth. This class can be used together with Zend_OpenId_Consumer. The following example shows how OpenIdAdapter implements the Zend_Auth_Adapter_Interface with the authenticate method. This performs an authentication query and verification.

The big difference between this adapter and existing ones, is that it works on two HTTP requests and includes a dispatch code to perform the second or third step of OpenID authentication.

```
<?php
class OpenIdAdapter implements Zend_Auth_Adapter_Interface {
    private $_id = null;

    public function __construct($id = null) {
        $this->_id = $id;
    }

    public function authenticate() {
        $id = $this->_id;
        if (!empty($id)) {
            $consumer = new Zend_OpenId_Consumer();
            if (!$consumer->login($id)) {
                $ret = false;
                $msg = "Authentication failed.";
            }
        } else {
            $consumer = new Zend_OpenId_Consumer();
            if ($consumer->verify($_GET, $id)) {
                $ret = true;
                $msg = "Authentication successful";
            } else {
                $ret = false;
                $msg = "Authentication failed";
            }
        }
        return new Zend_Auth_Result($ret, $id, array($msg));
    }
}

$status = "";
$auth = Zend_Auth::getInstance();
if ((isset($_POST['openid_action']) &&
     $_POST['openid_action'] == "login" &&
     !empty($_POST['openid_identifier'])) ||
    isset($_GET['openid_mode'])) {
    $adapter = new OpenIdAdapter(@$_POST['openid_identifier']);
    $result = $auth->authenticate($adapter);
    if ($result->isValid()) {
        Zend_OpenId::redirect(Zend_OpenId::selfURL());
    } else {
        $auth->clearIdentity();
        foreach ($result->getMessages() as $message) {
            $status .= "$message<br>\n";
        }
    }
} else if ($auth->hasIdentity()) {
    if (isset($_POST['openid_action']) &&
        $_POST['openid_action'] == "logout") {
        $auth->clearIdentity();
    } else {
        $status = "You are logged in as " . $auth->getIdentity() . "<br>\n";
    }
}
?>
<html><body>
<?php echo htmlspecialchars($status);?>
<form method="post"><fieldset>
<legend>OpenID Login</legend>
<input type="text" name="openid_identifier" value="">
<input type="submit" name="openid_action" value="login">
<input type="submit" name="openid_action" value="logout">
</fieldset></form></body></html>
```

With Zend_Auth the end-user's identity is saved in the session's data. It may be checked with Zend_Auth::hasIdentity and Zend_Auth::getIdentity.

2.8. Integration with Zend_Controller

Finally a couple of words about integration into Model-View-Controller applications: such Zend Framework applications are implemented using the Zend_Controller class and they use objects of the Zend_Controller_Response_Http class to prepare HTTP responses and send them back to the user's web browser.

Zend_OpenId_Consumer doesn't provide any GUI capabilities but it performs HTTP redirections on success of Zend_OpenId_Consumer::login and Zend_OpenId_Consumer::check. These redirections may work incorrectly or not at all if some data was already sent to the web browser. To properly perform HTTP redirection in MVC code the real Zend_Controller_Response_Http should be sent to Zend_OpenId_Consumer::login or Zend_OpenId_Consumer::check as the last argument.

3. Zend_OpenId_Provider

Zend_OpenId_Provider can be used to implement OpenID servers. This chapter provides examples that demonstrate how to build a very basic server. However, for implementation of a production OpenID server (such as www.myopenid.com) you may have to deal with more complex issues.

3.1. Quick Start

The following example includes code for creating a user account using Zend_OpenId_Provider::register. The link element with rel="openid.server" points to our own server script. If you submit this identity to an OpenID-enabled site, it will perform authentication on this server.

The code before the <html> tag is just a trick that automatically creates a user account. You won't need such code when using real identities.

Example 583. The Identity

```
<?php
// Set up test identity
define("TEST_SERVER", Zend_OpenId::absoluteURL("example-8.php"));
define("TEST_ID", Zend_OpenId::selfURL());
define("TEST_PASSWORD", "123");
$server = new Zend_OpenId_Provider();
if (!$server->hasUser(TEST_ID)) {
    $server->register(TEST_ID, TEST_PASSWORD);
}
?>
<html><head>
<link rel="openid.server" href="<?php echo TEST_SERVER;?>" />
</head><body>
<?php echo TEST_ID;?>
</body></html>
```

The following identity server script handles two kinds of requests from OpenID-enabled sites (for association and authentication). Both of them are handled by the same method:

`Zend_OpenId_Provider::handle`. The two arguments to the `Zend_OpenId_Provider` constructor are URLs of login and trust pages, which ask for input from the end user.

On success, the method `Zend_OpenId_Provider::handle` returns a string that should be passed back to the OpenID-enabled site. On failure, it returns FALSE. This example will return an HTTP 403 response if `Zend_OpenId_Provider::handle` fails. You will get this response if you open this script with a web browser, because it sends a non-OpenID conforming request.

Example 584. Simple Identity Provider

```
$server = new Zend_OpenId_Provider("example-8-login.php",
                                   "example-8-trust.php");
$ret = $server->handle();
if (is_string($ret)) {
    echo $ret;
} else if ($ret !== true) {
    header('HTTP/1.0 403 Forbidden');
    echo 'Forbidden';
}
```

It is a good idea to use a secure connection (HTTPS) for these scripts- especially for the following interactive scripts- to prevent password disclosure.

The following script implements a login screen for an identity server using `Zend_OpenId_Provider` and redirects to this page when a required user has not yet logged in. On this page, a user will enter his password to login.

You should use the password "123" that was used in the identity script above.

On submit, the script calls `Zend_OpenId_Provider::login` with the accepted user's identity and password, then redirects back to the main identity provider's script. On success, the `Zend_OpenId_Provider::login` establishes a session between the user and the identity provider and stores the information about the user, who is now logged in. All following requests from the same user won't require a login procedure- even if they come from another OpenID enabled web site.

Note that this session is between end-user and identity provider only. OpenID enabled sites know nothing about it.

Example 585. Simple Login Screen

```
<?php
$server = new Zend_OpenId_Provider();

if ($_SERVER['REQUEST_METHOD'] == 'POST' &&
    isset($_POST['openid_action']) &&
    $_POST['openid_action'] === 'login' &&
    isset($_POST['openid_identifier']) &&
    isset($_POST['openid_password'])) {
    $server->login($_POST['openid_identifier'],
                   $_POST['openid_password']);
    Zend_OpenId::redirect("example-8.php", $_GET);
}
?>
<html>
<body>
<form method="post">
<fieldset>
<legend>OpenID Login</legend>
<table border=0>
<tr>
<td>Name:</td>
<td>
<input type="text"
       name="openid_identifier"
       value="<?php echo htmlspecialchars($_GET['openid_identity']);?>">
</td>
</tr>
<tr>
<td>Password:</td>
<td>
<input type="text"
       name="openid_password"
       value="">
</td>
</tr>
<tr>
<td>&amp;nbsp;</td>
<td>
<input type="submit"
       name="openid_action"
       value="login">
</td>
</tr>
</table>
</fieldset>
</form>
</body>
</html>
```

The fact that the user is now logged in doesn't mean that the authentication must necessarily succeed. The user may decide not to trust particular OpenID enabled sites. The following trust screen allows the end user to make that choice. This choice may either be made only for current requests or forever. In the second case, information about trusted/untrusted sites is stored in an internal database, and all following authentication requests from this site will be handled automatically without user interaction.

Example 586. Simple Trust Screen

```
<?php
$server = new Zend_OpenId_Provider();

if ($_SERVER['REQUEST_METHOD'] == 'POST' &&
    isset($_POST['openid_action']) &&
    $_POST['openid_action'] === 'trust') {

    if (isset($_POST['allow'])) {
        if (isset($_POST['forever'])) {
            $server->allowSite($server->getSiteRoot($_GET));
        }
        $server->respondToConsumer($_GET);
    } else if (isset($_POST['deny'])) {
        if (isset($_POST['forever'])) {
            $server->denySite($server->getSiteRoot($_GET));
        }
        Zend_OpenId::redirect($_GET['openid_return_to'],
                              array('openid.mode'=>'cancel'));
    }
}
?>
<html>
<body>
<p>A site identifying as
<a href="<?php echo htmlspecialchars($server->getSiteRoot($_GET));?>">
<?php echo htmlspecialchars($server->getSiteRoot($_GET));?>
</a>
has asked us for confirmation that
<a href="<?php echo htmlspecialchars($server->getLoggedInUser());?>">
<?php echo htmlspecialchars($server->getLoggedInUser());?>
</a>
is your identity URL.
</p>
<form method="post">
<input type="checkbox" name="forever">
<label for="forever">forever</label><br>
<input type="hidden" name="openid_action" value="trust">
<input type="submit" name="allow" value="Allow">
<input type="submit" name="deny" value="Deny">
</form>
</body>
</html>
```

Production OpenID servers usually support the Simple Registration Extension that allows consumers to request some information about the user from the provider. In this case, the trust page can be extended to allow entering requested fields or selecting a specific user profile.

3.2. Combined Provide Scripts

It is possible to combine all provider functionality in one script. In this case login and trust URLs are omitted, and `Zend_OpenId_Provider` assumes that they point to the same page with the additional "openid.action" GET argument.

The following example is not complete. It doesn't provide GUI code for the user, instead performing an automatic login and trust relationship instead. This is done

just to simplify the example; a production server should include some code from previous examples.

Example 587. Everything Together

```
$server = new Zend_OpenId_Provider();

define("TEST_ID", Zend_OpenId::absoluteURL("example-9-id.php"));
define("TEST_PASSWORD", "123");

if ($_SERVER['REQUEST_METHOD'] == 'GET' &&
    isset($_GET['openid_action']) &&
    $_GET['openid_action'] === 'login') {
    $server->login(TEST_ID, TEST_PASSWORD);
    unset($_GET['openid_action']);
    Zend_OpenId::redirect(Zend_OpenId::selfUrl(), $_GET);
} else if ($_SERVER['REQUEST_METHOD'] == 'GET' &&
    isset($_GET['openid_action']) &&
    $_GET['openid_action'] === 'trust') {
    unset($_GET['openid_action']);
    $server->respondToConsumer($_GET);
} else {
    $ret = $server->handle();
    if (is_string($ret)) {
        echo $ret;
    } else if ($ret !== true) {
        header('HTTP/1.0 403 Forbidden');
        echo 'Forbidden';
    }
}
```

If you compare this example with previous examples split in to separate pages, you will see only the one difference besides the dispatch code: `unset($_GET['openid_action'])`. This call to `unset` is necessary to route the next request to main handler.

3.3. Simple Registration Extension

Again, the code before the <html> tag is just a trick to demonstrate functionality. It creates a new user account and associates it with a profile (nickname and password). Such tricks aren't needed in deployed providers where end users register on OpenID servers and fill in their profiles. Implementing this GUI is out of scope for this manual.

Example 588. Identity with Profile

```
<?php
define("TEST_SERVER", Zend_OpenId::absoluteURL("example-10.php"));
define("TEST_ID", Zend_OpenId::selfURL());
define("TEST_PASSWORD", "123");
$server = new Zend_OpenId_Provider();
if (!$server->hasUser(TEST_ID)) {
    $server->register(TEST_ID, TEST_PASSWORD);
    $server->login(TEST_ID, TEST_PASSWORD);
    $sreg = new Zend_OpenId_Extension_Sreg(array(
        'nickname' =>'test',
        'email' => 'test@test.com'
    ));
    $root = Zend_OpenId::absoluteURL(".");
    Zend_OpenId::normalizeUrl($root);
    $server->allowSite($root, $sreg);
    $server->logout();
}
?>
<html>
<head>
<link rel="openid.server" href="<?php echo TEST_SERVER;?>" />
</head>
<body>
<?php echo TEST_ID;?>
</body>
</html>
```

You should now pass this identity to the OpenID-enabled web site (use the Simple Registration Extension example from the previous section), and it should use the following OpenID server script.

This script is a variation of the script in the "Everything Together" example. It uses the same automatic login mechanism, but doesn't contain any code for a trust page. The user already trusts the example scripts forever. This trust was established by calling the `Zend_OpenId_Provider::allowSite()` method in the identity script. The same method associates the profile with the trusted URL. This profile will be returned automatically for a request from the trusted URL.

To make Simple Registration Extension work, you must simply pass an instance of `Zend_OpenId_Extension_Sreg` as the second argument to the `Zend_OpenId_Provider::handle()` method.

Example 589. Provider with SREG

```
$server = new Zend_OpenId_Provider();
$sreg = new Zend_OpenId_Extension_Sreg();

define("TEST_ID", Zend_OpenId::absoluteURL("example-10-id.php"));
define("TEST_PASSWORD", "123");

if ($_SERVER['REQUEST_METHOD'] == 'GET' &&
    isset($_GET['openid_action']) &&
    $_GET['openid_action'] === 'login') {
    $server->login(TEST_ID, TEST_PASSWORD);
    unset($_GET['openid_action']);
    Zend_OpenId::redirect(Zend_OpenId::selfUrl(), $_GET);
} else if ($_SERVER['REQUEST_METHOD'] == 'GET' &&
    isset($_GET['openid_action']) &&
    $_GET['openid_action'] === 'trust') {
   echo "UNTRUSTED DATA" ;
} else {
    $ret = $server->handle(null, $sreg);
    if (is_string($ret)) {
        echo $ret;
    } else if ($ret !== true) {
        header('HTTP/1.0 403 Forbidden');
        echo 'Forbidden';
    }
}
```

3.4. Anything Else?

Building OpenID providers is much less common than building OpenID-enabled sites, so this manual doesn't cover all `Zend_OpenId_Provider` features exhaustively, as was done for `Zend_OpenId_Consumer`.

To summamize, `Zend_OpenId_Provider` contains:

- A set of methods to build an end-user GUI that allows users to register and manage their trusted sites and profiles
- An abstract storage layer to store information about users, their sites and their profiles. It also stores associations between the provider and OpenID-enabled sites. This layer is very similar to that of the `Zend_OpenId_Consumer` class. It also uses file storage by default, but may used with another backend.
- An abstract user-association layer that may associate a user's web browser with a logged-in identity

The `Zend_OpenId_Provider` class doesn't attempt to cover all possible features that can be implemented by OpenID servers, e.g. digital certificates, but it can be extended easily using `Zend_OpenId_Extensions` or by standard object-oriented extension.

Zend_Paginator

1. Introduction

`Zend_Paginator` is a flexible component for paginating collections of data and presenting that data to users.

The primary design goals of `Zend_Paginator` are as follows:

- Paginate arbitrary data, not just relational databases
- Fetch only the results that need to be displayed
- Do not force users to adhere to only one way of displaying data or rendering pagination controls
- Loosely couple `Zend_Paginator` to other Zend Framework components so that users who wish to use it independently of `Zend_View`, `Zend_Db`, etc. can do so

2. Usage

2.1. Paginating data collections

In order to paginate items into pages, `Zend_Paginator` must have a generic way of accessing that data. For that reason, all data access takes place through data source adapters. Several adapters ship with Zend Framework by default:

Table 109. Adapters for Zend_Paginator

Adapter	Description
Array	Use a PHP array
DbSelect	Use a `Zend_Db_Select` instance, which will return an array
DbTableSelect	Use a `Zend_Db_Table_Select` instance, which will return an instance of `Zend_Db_Table_Rowset_Abstract`. This provides additional information about the result set, such as column names.
Iterator	Use an `Iterator` instance
Null	Do not use `Zend_Paginator` to manage data pagination. You can still take advantage of the pagination control feature.

Instead of selecting every matching row of a given query, the DbSelect and DbTableSelect adapters retrieve only the smallest amount of data necessary for displaying the current page.

Because of this, a second query is dynamically generated to determine the total number of matching rows. However, it is possible to directly supply a count or count query yourself. See the `setRowCount()` method in the DbSelect adapter for more information.

To create an instance of `Zend_Paginator`, you must supply an adapter to the constructor:

```
$paginator = new Zend_Paginator(new Zend_Paginator_Adapter_Array($array));
```

For convenience, you may take advantage of the static `factory()` method for the adapters packaged with Zend Framework:

```
$paginator = Zend_Paginator::factory($array);
```

In the case of the `Null` adapter, in lieu of a data collection you must supply an item count to its constructor.

Although the instance is technically usable in this state, in your controller action you'll need to tell the paginator what page number the user requested. This allows him to advance through the paginated data.

```
$paginator->setCurrentPageNumber($page);
```

The simplest way to keep track of this value is through a URL. Although we recommend using a `Zend_Controller_Router_Interface`-compatible router to handle this, it is not a requirement.

The following is an example route you might use in an INI configuration file:

```
routes.example.route = articles/:articleName/:page
routes.example.defaults.controller = articles
routes.example.defaults.action = view
routes.example.defaults.page = 1
routes.example.reqs.articleName = \w+
routes.example.reqs.page = \d+
```

With the above route (and using Zend Framework MVC components), you might set the current page number like this:

```
$paginator->setCurrentPageNumber($this->_getParam('page'));
```

There are other options available; see Configuration for more on them.

Finally, you'll need to assign the paginator instance to your view. If you're using `Zend_View` with the ViewRenderer action helper, the following will work:

```
$this->view->paginator = $paginator;
```

2.2. The DbSelect and DbTableSelect adapter

The usage of most adapters is pretty straight-forward. However, the database adapters require a more detailed explanation regarding the retrieval and count of the data from the database.

To use the DbSelect and DbTableSelect adapters you don't have to retrieve the data upfront from the database. Both adapters do the retrieval for you, aswell as the counting of the total pages. If additional work has to be done on the database results the adapter `getItems()` method has to be extended in your application.

Additionally these adapters do *not* fetch all records from the database in order to count them. Instead, the adapters manipulates the original query to produce the corresponding COUNT

query. Paginator then executes that COUNT query to get the number of rows. This does require an extra round-trip to the database, but this is many times faster than fetching an entire result set and using `count()`. Especially with large collections of data.

The database adapters will try and build the most efficient query that will execute on pretty much all modern databases. However, depending on your database or even your own schema setup, there might be more efficient ways to get a rowcount. For this scenario the database adapters allow you to set a custom COUNT query. For example, if you keep track of the count of blog posts in a separate table, you could achieve a faster count query with the following setup:

```
$adapter = new Zend_Paginator_Adapter_DbSelect($db->select()->from('posts'));
$adapter->setRowCount(
    $db->select()
       ->from(
            'item_counts',
            array(
               Zend_Paginator_Adapter_DbSelect::ROW_COUNT_COLUMN => 'post_count'
            )
         )
);

$paginator = new Zend_Paginator($adapter);
```

This approach will probably not give you a huge performance gain on small collections and/or simple select queries. However, with complex queries and large collections, a similar approach could give you a significant performance boost.

2.3. Rendering pages with view scripts

The view script is used to render the page items (if you're using `Zend_Paginator` to do so) and display the pagination control.

Because `Zend_Paginator` implements the SPL interface `IteratorAggregate`, looping over your items and displaying them is simple.

```
<html>
<body>
<h1>Example</h1>
<?php if (count($this->paginator)): ?>
<ul>
<?php foreach ($this->paginator as $item): ?>
  <li><?php echo $item; ?></li>
<?php endforeach; ?>
</ul>
<?php endif; ?>
<?php echo $this->paginationControl($this->paginator,
                                    'Sliding',
                                    'my_pagination_control.phtml'); ?>
</body>
</html>
```

Notice the view helper call near the end. PaginationControl accepts up to four parameters: the paginator instance, a scrolling style, a view partial, and an array of additional parameters.

The second and third parameters are very important. Whereas the view partial is used to determine how the pagination control should *look*, the scrolling style is used to control how it should *behave*. Say the view partial is in the style of a search pagination control, like the one below:

Results Page:
Prev 2 3 4 5 6 7 8 9 10 11 Next

What happens when the user clicks the "next" link a few times? Well, any number of things could happen. The current page number could stay in the middle as you click through (as it does on Yahoo!), or it could advance to the end of the page range and then appear again on the left when the user clicks "next" one more time. The page numbers might even expand and contract as the user advances (or "scrolls") through them (as they do on Google).

There are four scrolling styles packaged with Zend Framework:

Table 110. Scrolling styles for Zend_Paginator

Scrolling style	Description
All	Returns every page. This is useful for dropdown menu pagination controls with relatively few pages. In these cases, you want all pages available to the user at once.
Elastic	A Google-like scrolling style that expands and contracts as a user scrolls through the pages.
Jumping	As users scroll through, the page number advances to the end of a given range, then starts again at the beginning of the new range.
Sliding	A Yahoo!-like scrolling style that positions the current page number in the center of the page range, or as close as possible. This is the default style.

The fourth and final parameter is reserved for an optional associative array of additional variables that you want available in your view partial (available via `$this`). For instance, these values could include extra URL parameters for pagination links.

By setting the default view partial, default scrolling style, and view instance, you can eliminate the calls to PaginationControl completely:

```
Zend_Paginator::setDefaultScrollingStyle('Sliding');
Zend_View_Helper_PaginationControl::setDefaultViewPartial(
    'my_pagination_control.phtml'
);
$paginator->setView($view);
```

When all of these values are set, you can render the pagination control inside your view script with a simple echo statement:

```
<?php echo $this->paginator; ?>
```

Of course, it's possible to use `Zend_Paginator` with other template engines. For example, with Smarty you might do the following:

```
$smarty->assign('pages', $paginator->getPages());
```

You could then access paginator values from a template like so:

```
{$pages->pageCount}
```

2.3.1. Example pagination controls

The following example pagination controls will hopefully help you get started:

Search pagination:

```
<!--
See http://developer.yahoo.com/ypatterns/pattern.php?pattern=searchpagination
-->

<?php if ($this->pageCount): ?>
<div class="paginationControl">
<!-- Previous page link -->
<?php if (isset($this->previous)): ?>
  <a href="<?php echo $this->url(array('page' => $this->previous)); ?>">
     Previous
  </a> |
<?php else: ?>
  <span class="disabled"> Previous</span> |
<?php endif; ?>
<!-- Numbered page links -->
<?php foreach ($this->pagesInRange as $page): ?>
  <?php if ($page != $this->current): ?>
    <a href="<?php echo $this->url(array('page' => $page)); ?>">
        <?php echo $page; ?>
    </a> |
  <?php else: ?>
    <?php echo $page; ?> |
  <?php endif; ?>
<?php endforeach; ?>
<!-- Next page link -->
<?php if (isset($this->next)): ?>
  <a href="<?php echo $this->url(array('page' => $this->next)); ?>">
    Next
  </a>
<?php else: ?>
  <span class="disabled">Next </span>
<?php endif; ?>
</div>
<?php endif; ?>
```

Item pagination:

```
<!--
See http://developer.yahoo.com/ypatterns/pattern.php?pattern=itempagination
-->

<?php if ($this->pageCount): ?>
<div class="paginationControl">
<?php echo $this->firstItemNumber; ?> - <?php echo $this->lastItemNumber; ?>
of <?php echo $this->totalItemCount; ?>
<!-- First page link -->
<?php if (isset($this->previous)): ?>
  <a href="<?php echo $this->url(array('page' => $this->first)); ?>">
```

```
    First
  </a> |
<?php else: ?>
  <span class="disabled">First</span> |
<?php endif; ?>
<!-- Previous page link -->
<?php if (isset($this->previous)): ?>
  <a href="<?php echo $this->url(array('page' => $this->previous)); ?>">
     Previous
  </a> |
<?php else: ?>
  <span class="disabled"> Previous</span> |
<?php endif; ?>
<!-- Next page link -->
<?php if (isset($this->next)): ?>
  <a href="<?php echo $this->url(array('page' => $this->next)); ?>">
    Next
  </a> |
<?php else: ?>
  <span class="disabled">Next </span> |
<?php endif; ?>
<!-- Last page link -->
<?php if (isset($this->next)): ?>
  <a href="<?php echo $this->url(array('page' => $this->last)); ?>">
    Last
  </a>
<?php else: ?>
  <span class="disabled">Last</span>
<?php endif; ?>
</div>
<?php endif; ?>
```

Dropdown pagination:

```
<?php if ($this->pageCount): ?>
<select id="paginationControl" size="1">
<?php foreach ($this->pagesInRange as $page): ?>
  <?php $selected = ($page == $this->current) ? ' selected="selected"' : ''; ?>
  <option value="<?php
        echo $this->url(array('page' => $page));?>"<?php echo $selected ?>>
    <?php echo $page; ?>
  </option>
<?php endforeach; ?>
</select>
<?php endif; ?>
<script type="text/javascript"
     src="http://ajax.googleapis.com/ajax/libs/prototype/1.6.0.2/prototype.js">
</script>
<script type="text/javascript">
$('paginationControl').observe('change', function() {
    window.location = this.options[this.selectedIndex].value;
})
</script>
```

2.3.2. Listing of properties

The following options are available to pagination control view partials:

Table 111. Properties available to view partials

Property	Type	Description
first	integer	First page number (i.e., 1)
firstItemNumber	integer	Absolute number of the first item on this page
firstPageInRange	integer	First page in the range returned by the scrolling style
current	integer	Current page number
currentItemCount	integer	Number of items on this page
itemCountPerPage	integer	Maximum number of items available to each page
last	integer	Last page number
lastItemNumber	integer	Absolute number of the last item on this page
lastPageInRange	integer	Last page in the range returned by the scrolling style
next	integer	Next page number
pageCount	integer	Number of pages
pagesInRange	array	Array of pages returned by the scrolling style
previous	integer	Previous page number
totalItemCount	integer	Total number of items

3. Configuration

`Zend_Paginator` has several configuration methods that can be called:

Table 112. Configuration methods for Zend_Paginator

Method	Description
setCurrentPageNumber	Sets the current page number (default 1).
setItemCountPerPage	Sets the maximum number of items to display on a page (default 10).
setPageRange	Sets the number of items to display in the pagination control (default 10). Note: Most of the time this number will be adhered to exactly, but scrolling styles do have the option of only using it as a guideline or starting value (e.g., Elastic).
setView	Sets the view instance, for rendering convenience.

4. Advanced usage

4.1. Custom data source adapters

At some point you may run across a data type that is not covered by the packaged adapters. In this case, you will need to write your own.

To do so, you must implement `Zend_Paginator_Adapter_Interface`. There are two methods required to do this:

- count()
- getItems($offset, $itemCountPerPage)

Additionally, you'll want to implement a constructor that takes your data source as a parameter and stores it as a protected or private property. How you wish to go about doing this specifically is up to you.

If you've ever used the SPL interface `Countable`, you're familiar with `count()`. As used with `Zend_Paginator`, this is the total number of items in the data collection. Additionally, the `Zend_Paginator` instance provides a method `countAllItems()` that proxies to the adapter `count()` method.

The `getItems()` method is only slightly more complicated. For this, your adapter is supplied with an offset and the number of items to display per page. You must return the appropriate slice of data. For an array, that would be:

```
return array_slice($this->_array, $offset, $itemCountPerPage);
```

Take a look at the packaged adapters (all of which implement the `Zend_Paginator_Adapter_Interface`) for ideas of how you might go about implementing your own.

4.2. Custom scrolling styles

Creating your own scrolling style requires that you implement `Zend_Paginator_ScrollingStyle_Interface`, which defines a single method, `getPages()`. Specifically,

```
public function getPages(Zend_Paginator $paginator, $pageRange = null);
```

This method should calculate a lower and upper bound for page numbers within the range of so-called "local" pages (that is, pages that are nearby the current page).

Unless it extends another scrolling style (see `Zend_Paginator_ScrollingStyle_Elastic` for an example), your custom scrolling style will inevitably end with something similar to the following line of code:

```
return $paginator->getPagesInRange($lowerBound, $upperBound);
```

There's nothing special about this call; it's merely a convenience method to check the validity of the lower and upper bound and return an array of the range to the paginator.

When you're ready to use your new scrolling style, you'll need to tell `Zend_Paginator` what directory to look in. To do that, do the following:

```
$prefix = 'My_Paginator_ScrollingStyle';
$path   = 'My/Paginator/ScrollingStyle/';
Zend_Paginator::addScrollingStylePrefixPath($prefix, $path);
```

4.3. Caching features

`Zend_Paginator` can be told to cache the data it has already passed on, preventing the adapter from fetching them each time they are used. To tell paginator to automatically cache the adapter's data, just pass to its `setCache()` method a `Zend_Cache_Core` instance.

```
$paginator = Zend_Paginator::factory($someData);
$fO = array('lifetime' => 3600, 'automatic_serialization' => true);
$bO = array('cache_dir'=>'/tmp');
$cache = Zend_cache::factory('Core', 'File', $fO, $bO);
Zend_Paginator::setCache($cache);
```

As far as `Zend_Paginator` has got a `Zend_Cache_Core` instance, data will be cached. Sometimes you would like not to cache data even if you already passed a cache instance. You should then use `setCacheEnable()` for that.

```
$paginator = Zend_Paginator::factory($someData);
// $cache is a Zend_Cache_Core instance
Zend_Paginator::setCache($cache);
// ... later on the script
$paginator->setCacheEnable(false);
// cache is now disabled
```

When a cache is set, data are automatically stored in it and pulled out from it. It then can be useful to empty the cache manually. You can get this done by calling `clearPageItemCache($pageNumber)`. If you don't pass any parameter, the whole cache will be empty. You can optionally pass a parameter representing the page number to empty in the cache:

```
$paginator = Zend_Paginator::factory($someData);
Zend_Paginator::setCache($cache);
$items = $paginator->getCurrentItems();
// page 1 is now in cache
$page3Items = $paginator->getItemsByPage(3);
// page 3 is now in cache

// clear the cache of the results for page 3
$paginator->clearPageItemCache(3);

// clear all the cache data
$paginator->clearPageItemCache();
```

Changing the item count per page will empty the whole cache as it would have become invalid:

```
$paginator = Zend_Paginator::factory($someData);
Zend_Paginator::setCache($cache);
// fetch some items
$items = $paginator->getCurrentItems();

// all the cache data will be flushed:
$paginator->setItemCountPerPage(2);
```

It is also possible to see the data in cache and ask for them directly. `getPageItemCache()` can be used for that:

```
$paginator = Zend_Paginator::factory($someData);
$paginator->setItemCountPerPage(3);
Zend_Paginator::setCache($cache);

// fetch some items
$items = $paginator->getCurrentItems();
$otherItems = $paginator->getItemsPerPage(4);

// see the cached items as a two-dimension array:
var_dump($paginator->getPageItemCache());
```

4.4. Zend_Paginator_AdapterAggregate Interface

Depending on your application you might want to paginate objects, whose internal data-structure is equal to existing adapters, but you don't want to break up your encapsulation to allow access to this data. In other cases an object might be in a "has-an adapter" relationship, rather than the "is-an adapter" relationsship that `Zend_Paginator_Adapter_Abstract` promotes. For this cases you can use the `Zend_Paginator_AdapterAggregate` interface that behaves much like the `IteratorAggregate` interface of the PHP SPL extension.

```
interface Zend_Paginator_AdapterAggregate
{
    /**
     * Return a fully configured Paginator Adapter from this method.
     *
     * @return Zend_Paginator_Adapter_Abstract
     */
    public function getPaginatorAdapter();
}
```

The interface is fairly small and only expects you to return an instance of `Zend_Paginator_Adapter_Abstract`. An Adapter Aggregate instance is then recognized by both `Zend_Paginator::factory` and the constructor of Zend_Paginator and handled accordingly.

Zend_Pdf

1. Introduction

The Zend_Pdf component is a PDF (Portable Document Format) manipulation engine. It can load, create, modify and save documents. Thus it can help any PHP application dynamically create PDF documents by modifying existing documents or generating new ones from scratch. Zend_Pdf offers the following features:

- Create a new document or load existing one. [1]
- Retrieve a specified revision of the document.
- Manipulate pages within a document. Change page order, add new pages, remove pages from a document.
- Different drawing primitives (lines, rectangles, polygons, circles, ellipses and sectors).
- Text drawing using any of the 14 standard (built-in) fonts or your own custom TrueType fonts.
- Rotations.
- Image drawing. [2]
- Incremental PDF file update.

2. Creating and Loading PDF Documents

The Zend_Pdf class represents PDF documents and provides document-level operations.

To create a new document, a new Zend_Pdf object should first be created.

Zend_Pdf class also provides two static methods to load an existing PDF document. These are the Zend_Pdf::load() and Zend_Pdf::parse() methods. Both of them return Zend_Pdf objects as a result or throw an exception if an error occurs.

Example 590. Create new or load existing PDF document

```
...
// Create a new PDF document
$pdf1 = new Zend_Pdf();

// Load a PDF document from a file
$pdf2 = Zend_Pdf::load($fileName);

// Load a PDF document from a string
$pdf3 = Zend_Pdf::parse($pdfString);
...
```

The PDF file format supports incremental document update. Thus each time a document is updated, then a new revision of the document is created. Zend_Pdf component supports the retrieval of a specified revision.

A revision can be specified as a second parameter to the `Zend_Pdf::load()` and `Zend_Pdf::parse()` methods or requested by calling the `Zend_Pdf::rollback()` method. [3] call.

Example 591. Requesting Specific Revisions of a PDF Document

```
...
// Load the previous revision of the PDF document
$pdf1 = Zend_Pdf::load($fileName, 1);

// Load the previous revision of the PDF document
$pdf2 = Zend_Pdf::parse($pdfString, 1);

// Load the first revision of the PDF document
$pdf3 = Zend_Pdf::load($fileName);
$revisions = $pdf3->revisions();
$pdf3->rollback($revisions - 1);
...
```

3. Save Changes to PDF Documents

There are two methods that save changes to PDF documents: the `Zend_Pdf::save()` and `Zend_Pdf::render()` methods.

`Zend_Pdf::save($filename, $updateOnly = false)` saves the PDF document to a file. If $updateOnly is TRUE, then only the new PDF file segment is appended to a file. Otherwise, the file is overwritten.

`Zend_Pdf::render($newSegmentOnly = false)` returns the PDF document as a string. If $newSegmentOnly is TRUE, then only the new PDF file segment is returned.

Example 592. Saving PDF Documents

```
...
// Load the PDF document
$pdf = Zend_Pdf::load($fileName);
...
// Update the PDF document
$pdf->save($fileName, true);
// Save document as a new file
$pdf->save($newFileName);

// Return the PDF document as a string
$pdfString = $pdf->render();

...
```

4. Working with Pages

4.1. Page Creation

The pages in a PDF document are represented as `Zend_Pdf_Page` instances in `Zend_Pdf`.

[3] `Zend_Pdf::rollback()` method must be invoked before any changes are applied to the document, otherwise the behavior is not defined.

PDF pages either are loaded from an existing PDF or created using the Zend_Pdf API.

New pages can be created by instantiating new Zend_Pdf_Page objects directly or by calling the Zend_Pdf::newPage() method, which returns a Zend_Pdf_Page object. Zend_Pdf::newPage() creates a page that is already attached to a document. Unattached pages can't be used with multiple PDF documents, but they are somewhat more performant. [4]

The Zend_Pdf::newPage() method and the Zend_Pdf_Page constructor take the same parameters specifying page size. They can take either the size of page ($x, $y) in points (1/72 inch) or a predefined constant representing a page type:

- Zend_Pdf_Page::SIZE_A4
- Zend_Pdf_Page::SIZE_A4_LANDSCAPE
- Zend_Pdf_Page::SIZE_LETTER
- Zend_Pdf_Page::SIZE_LETTER_LANDSCAPE

Document pages are stored in the $pages public attribute of the Zend_Pdf class. The attribute holds an array of Zend_Pdf_Page objects and completely defines the instances and order of pages. This array can be manipulated like any other PHP array:

Example 593. PDF document pages management

```
...
// Reverse page order
$pdf->pages = array_reverse($pdf->pages);
...
// Add new page
$pdf->pages[] = new Zend_Pdf_Page(Zend_Pdf_Page::SIZE_A4);
// Add new page
$pdf->pages[] = $pdf->newPage(Zend_Pdf_Page::SIZE_A4);

// Remove specified page.
unset($pdf->pages[$id]);

...
```

4.2. Page cloning

Existing PDF page can be cloned by creating new Zend_Pdf_Page object with existing page as a parameter:

[4] It's a limitation of current Zend Framework version. It will be eliminated in future versions. But unattached pages will always give better (more optimal) result for sharing pages between documents.

Example 594. Cloning existing page

```
...
// Store template page in a separate variable
$template = $pdf->pages[$templatePageIndex];
...
// Add new page
$page1 = new Zend_Pdf_Page($template);
$pdf->pages[] = $page1;
...

// Add another page
$page2 = new Zend_Pdf_Page($template);
$pdf->pages[] = $page2;
...

// Remove source template page from the documents.
unset($pdf->pages[$templatePageIndex]);

...
```

It's useful if you need several pages to be created using one template.

Important! Cloned page shares some PDF resources with a template page, so it can be used only within the same document as a template page. Modified document can be saved as new one.

5. Drawing

5.1. Geometry

PDF uses the same geometry as PostScript. It starts from bottom-left corner of page and by default is measured in points (1/72 of an inch).

Page size can be retrieved from a page object:

```
$width  = $pdfPage->getWidth();
$height = $pdfPage->getHeight();
```

5.2. Colors

PDF has a powerful capabilities for colors representation. `Zend_Pdf` module supports Gray Scale, RGB and CMYK color spaces. Any of them can be used in any place, where `Zend_Pdf_Color` object is required. `Zend_Pdf_Color_GrayScale`, `Zend_Pdf_Color_Rgb` and `Zend_Pdf_Color_Cmyk` classes provide this functionality:

```
// $grayLevel (float number). 0.0 (black) - 1.0 (white)
$color1 = new Zend_Pdf_Color_GrayScale($grayLevel);

// $r, $g, $b (float numbers). 0.0 (min intensity) - 1.0 (max intensity)
$color2 = new Zend_Pdf_Color_Rgb($r, $g, $b);

// $c, $m, $y, $k (float numbers). 0.0 (min intensity) - 1.0 (max intensity)
```

```
$color3 = new Zend_Pdf_Color_Cmyk($c, $m, $y, $k);
```

HTML style colors are also provided with `Zend_Pdf_Color_Html` class:

```
$color1 = new Zend_Pdf_Color_Html('#3366FF');
$color2 = new Zend_Pdf_Color_Html('silver');
$color3 = new Zend_Pdf_Color_Html('forestgreen');
```

5.3. Shape Drawing

All drawing operations can be done in a context of PDF page.

`Zend_Pdf_Page` class provides a set of drawing primitives:

```
/**
 * Draw a line from x1,y1 to x2,y2.
 *
 * @param float $x1
 * @param float $y1
 * @param float $x2
 * @param float $y2
 * @return Zend_Pdf_Page
 */
public function drawLine($x1, $y1, $x2, $y2);
```

```
/**
 * Draw a rectangle.
 *
 * Fill types:
 * Zend_Pdf_Page::SHAPE_DRAW_FILL_AND_STROKE - fill rectangle
 *                                             and stroke (default)
 * Zend_Pdf_Page::SHAPE_DRAW_STROKE          - stroke rectangle
 * Zend_Pdf_Page::SHAPE_DRAW_FILL            - fill rectangle
 *
 * @param float $x1
 * @param float $y1
 * @param float $x2
 * @param float $y2
 * @param integer $fillType
 * @return Zend_Pdf_Page
 */
public function drawRectangle($x1, $y1, $x2, $y2,
                    $fillType = Zend_Pdf_Page::SHAPE_DRAW_FILL_AND_STROKE);
```

```
/**
 * Draw a rounded rectangle.
 *
 * Fill types:
 * Zend_Pdf_Page::SHAPE_DRAW_FILL_AND_STROKE - fill rectangle and stroke (default)
 * Zend_Pdf_Page::SHAPE_DRAW_STROKE      - stroke rectangle
 * Zend_Pdf_Page::SHAPE_DRAW_FILL        - fill rectangle
 *
 * radius is an integer representing radius of the four corners, or an array
 * of four integers representing the radius starting at top left, going
 * clockwise
 *
 * @param float $x1
```

```
 * @param float $y1
 * @param float $x2
 * @param float $y2
 * @param integer|array $radius
 * @param integer $fillType
 * @return Zend_Pdf_Page
 */
public function drawRoundedRectangle($x1, $y1, $x2, $y2, $radius,
                        $fillType = Zend_Pdf_Page::SHAPE_DRAW_FILL_AND_STROKE);
```

```
/**
 * Draw a polygon.
 *
 * If $fillType is Zend_Pdf_Page::SHAPE_DRAW_FILL_AND_STROKE or
 * Zend_Pdf_Page::SHAPE_DRAW_FILL, then polygon is automatically closed.
 * See detailed description of these methods in a PDF documentation
 * (section 4.4.2 Path painting Operators, Filling)
 *
 * @param array $x  - array of float (the X co-ordinates of the vertices)
 * @param array $y  - array of float (the Y co-ordinates of the vertices)
 * @param integer $fillType
 * @param integer $fillMethod
 * @return Zend_Pdf_Page
 */
public function drawPolygon($x, $y,
                            $fillType =
                                Zend_Pdf_Page::SHAPE_DRAW_FILL_AND_STROKE,
                            $fillMethod =
                                Zend_Pdf_Page::FILL_METHOD_NON_ZERO_WINDING);
```

```
/**
 * Draw a circle centered on x, y with a radius of radius.
 *
 * Angles are specified in radians
 *
 * Method signatures:
 * drawCircle($x, $y, $radius);
 * drawCircle($x, $y, $radius, $fillType);
 * drawCircle($x, $y, $radius, $startAngle, $endAngle);
 * drawCircle($x, $y, $radius, $startAngle, $endAngle, $fillType);
 *
 *
 * It's not a really circle, because PDF supports only cubic Bezier
 * curves. But very good approximation.
 * It differs from a real circle on a maximum 0.00026 radiuses (at PI/8,
 * 3*PI/8, 5*PI/8, 7*PI/8, 9*PI/8, 11*PI/8, 13*PI/8 and 15*PI/8 angles).
 * At 0, PI/4, PI/2, 3*PI/4, PI, 5*PI/4, 3*PI/2 and 7*PI/4 it's exactly
 * a tangent to a circle.
 *
 * @param float $x
 * @param float $y
 * @param float $radius
 * @param mixed $param4
 * @param mixed $param5
 * @param mixed $param6
 * @return Zend_Pdf_Page
 */
public function  drawCircle($x,
                            $y,
```

```
                                $radius,
                                $param4 = null,
                                $param5 = null,
                                $param6 = null);
```

```
/**
 * Draw an ellipse inside the specified rectangle.
 *
 * Method signatures:
 * drawEllipse($x1, $y1, $x2, $y2);
 * drawEllipse($x1, $y1, $x2, $y2, $fillType);
 * drawEllipse($x1, $y1, $x2, $y2, $startAngle, $endAngle);
 * drawEllipse($x1, $y1, $x2, $y2, $startAngle, $endAngle, $fillType);
 *
 * Angles are specified in radians
 *
 * @param float $x1
 * @param float $y1
 * @param float $x2
 * @param float $y2
 * @param mixed $param5
 * @param mixed $param6
 * @param mixed $param7
 * @return Zend_Pdf_Page
 */
public function drawEllipse($x1,
                            $y1,
                            $x2,
                            $y2,
                            $param5 = null,
                            $param6 = null,
                            $param7 = null);
```

5.4. Text Drawing

Text drawing operations also exist in the context of a PDF page. You can draw a single line of text at any position on the page by supplying the x and y coordinates of the baseline. Current font and current font size are used for text drawing operations (see detailed description below).

```
/**
 * Draw a line of text at the specified position.
 *
 * @param string $text
 * @param float $x
 * @param float $y
 * @param string $charEncoding (optional) Character encoding of source
 *                 text.Defaults to current locale.
 * @throws Zend_Pdf_Exception
 * @return Zend_Pdf_Page
 */
public function drawText($text, $x, $y, $charEncoding = '');
```

Example 595. Draw a string on the page

```
...
$pdfPage->drawText('Hello world!', 72, 720);
...
```

By default, text strings are interpreted using the character encoding method of the current locale. If you have a string that uses a different encoding method (such as a UTF-8 string read from a file on disk, or a MacRoman string obtained from a legacy database), you can indicate the character encoding at draw time and `Zend_Pdf` will handle the conversion for you. You can supply source strings in any encoding method supported by PHP's `iconv()` function:

Example 596. Draw a UTF-8-encoded string on the page

```
...
// Read a UTF-8-encoded string from disk
$unicodeString = fread($fp, 1024);

// Draw the string on the page
$pdfPage->drawText($unicodeString, 72, 720, 'UTF-8');
...
```

5.5. Using fonts

`Zend_Pdf_Page::drawText()` uses the page's current font and font size, which is set with the `Zend_Pdf_Page::setFont()` method:

```
/**
 * Set current font.
 *
 * @param Zend_Pdf_Resource_Font $font
 * @param float $fontSize
 * @return Zend_Pdf_Page
 */
public function setFont(Zend_Pdf_Resource_Font $font, $fontSize);
```

PDF documents support PostScript Type 1 and TrueType fonts, as well as two specialized PDF types, Type 3 and composite fonts. There are also 14 standard Type 1 fonts built-in to every PDF viewer: Courier (4 styles), Helvetica (4 styles), Times (4 styles), Symbol, and Zapf Dingbats.

`Zend_Pdf` currently supports the standard 14 PDF fonts as well as your own custom TrueType fonts. Font objects are obtained via one of two factory methods: `Zend_Pdf_Font::fontWithName($fontName)` for the standard 14 PDF fonts or `Zend_Pdf_Font::fontWithPath($filePath)` for custom fonts.

Example 597. Create a standard font

```
...
// Create new font
$font = Zend_Pdf_Font::fontWithName(Zend_Pdf_Font::FONT_HELVETICA);

// Apply font
$pdfPage->setFont($font, 36);
...
```

Constants for the standard 14 PDF font names are defined in the `Zend_Pdf_Font` class:

- Zend_Pdf_Font::FONT_COURIER
- Zend_Pdf_Font::FONT_COURIER_BOLD
- Zend_Pdf_Font::FONT_COURIER_ITALIC

- Zend_Pdf_Font::FONT_COURIER_BOLD_ITALIC
- Zend_Pdf_Font::FONT_TIMES
- Zend_Pdf_Font::FONT_TIMES_BOLD
- Zend_Pdf_Font::FONT_TIMES_ITALIC
- Zend_Pdf_Font::FONT_TIMES_BOLD_ITALIC
- Zend_Pdf_Font::FONT_HELVETICA
- Zend_Pdf_Font::FONT_HELVETICA_BOLD
- Zend_Pdf_Font::FONT_HELVETICA_ITALIC
- Zend_Pdf_Font::FONT_HELVETICA_BOLD_ITALIC
- Zend_Pdf_Font::FONT_SYMBOL
- Zend_Pdf_Font::FONT_ZAPFDINGBATS

You can also use any individual TrueType font (which usually has a '.ttf' extension) or an OpenType font ('.otf' extension) if it contains TrueType outlines. Currently unsupported, but planned for a future release are Mac OS X .dfont files and Microsoft TrueType Collection ('.ttc' extension) files.

To use a TrueType font, you must provide the full file path to the font program. If the font cannot be read for some reason, or if it is not a TrueType font, the factory method will throw an exception:

Example 598. Create a TrueType font

```
...
// Create new font
$goodDogCoolFont = Zend_Pdf_Font::fontWithPath('/path/to/GOODDC__.TTF');

// Apply font
$pdfPage->setFont($goodDogCoolFont, 36);
...
```

By default, custom fonts will be embedded in the resulting PDF document. This allows recipients to view the page as intended, even if they don't have the proper fonts installed on their system. If you are concerned about file size, you can request that the font program not be embedded by passing a 'do not embed' option to the factory method:

Example 599. Create a TrueType font, but do not embed it in the PDF document

```
...
// Create new font
$goodDogCoolFont = Zend_Pdf_Font::fontWithPath('/path/to/GOODDC__.TTF',
                                               Zend_Pdf_Font::EMBED_DONT_EMBED);

// Apply font
$pdfPage->setFont($goodDogCoolFont, 36);
...
```

If the font program is not embedded but the recipient of the PDF file has the font installed on their system, they will see the document as intended. If they do not have the correct font installed, the PDF viewer application will do its best to synthesize a replacement.

Some fonts have very specific licensing rules which prevent them from being embedded in PDF documents. So you are not caught off-guard by this, if you try to use a font that cannot be embedded, the factory method will throw an exception.

You can still use these fonts, but you must either pass the do not embed flag as described above, or you can simply suppress the exception:

Example 600. Do not throw an exception for fonts that cannot be embedded

```
...
$font = Zend_Pdf_Font::fontWithPath(
           '/path/to/unEmbeddableFont.ttf',
           Zend_Pdf_Font::EMBED_SUPPRESS_EMBED_EXCEPTION
        );
...
```

This suppression technique is preferred if you allow an end-user to choose their own fonts. Fonts which can be embedded in the PDF document will be; those that cannot, won't.

Font programs can be rather large, some reaching into the tens of megabytes. By default, all embedded fonts are compressed using the Flate compression scheme, resulting in a space savings of 50% on average. If, for some reason, you do not want to compress the font program, you can disable it with an option:

Example 601. Do not compress an embedded font

```
...
$font = Zend_Pdf_Font::fontWithPath('/path/to/someReallyBigFont.ttf',
                                    Zend_Pdf_Font::EMBED_DONT_COMPRESS);
...
```

Finally, when necessary, you can combine the embedding options by using the bitwise OR operator:

Example 602. Combining font embedding options

```
...
$font = Zend_Pdf_Font::fontWithPath(
            $someUserSelectedFontPath,
            (Zend_Pdf_Font::EMBED_SUPPRESS_EMBED_EXCEPTION |
            Zend_Pdf_Font::EMBED_DONT_COMPRESS));
...
```

5.6. Standard PDF fonts limitations

Standard PDF fonts use several single byte encodings internally (see PDF Reference, Sixth Edition, version 1.7 Appendix D for details). They are generally equal to Latin1 character set (except Symbol and ZapfDingbats fonts).

`Zend_Pdf` uses CP1252 (WinLatin1) for drawing text with standard fonts.

Text still can be provided in any other encoding, which must be specified if it differs from a current locale. Only WinLatin1 characters will be actually drawn.

Example 603. Combining font embedding options

```
...
$font = Zend_Pdf_Font::fontWithName(Zend_Pdf_Font::FONT_COURIER);
$pdfPage->setFont($font, 36)
        ->drawText('Euro sign - €', 72, 720, 'UTF-8')
        ->drawText('Text with umlauts - à è ì', 72, 650, 'UTF-8');
...
```

5.7. Extracting fonts

`Zend_Pdf` module provides a possibility to extract fonts from loaded documents.

It may be useful for incremental document updates. Without this functionality you have to attach and possibly embed font into a document each time you want to update it.

`Zend_Pdf` and `Zend_Pdf_Page` objects provide special methods to extract all fonts mentioned within a document or a page:

Example 604. Extracting fonts from a loaded document

```
...
$pdf = Zend_Pdf::load($documentPath);
...
// Get all document fonts
$fontList = $pdf->extractFonts();
$pdf->pages[] = ($page = $pdf->newPage(Zend_Pdf_Page::SIZE_A4));
$yPosition = 700;
foreach ($fontList as $font) {
    $page->setFont($font, 15);
    $fontName = $font->getFontName(Zend_Pdf_Font::NAME_POSTSCRIPT,
                                   'en',
                                   'UTF-8');
    $page->drawText($fontName . ': The quick brown fox jumps over the lazy dog',
                    100,
                    $yPosition,
                    'UTF-8');
    $yPosition -= 30;
}
...
// Get fonts referenced within the first document page
$firstPage = reset($pdf->pages);
$firstPageFonts = $firstPage->extractFonts();
...
```

Example 605. Extracting font from a loaded document by specifying font name

```
...
$pdf = new Zend_Pdf();
...
$pdf->pages[] = ($page = $pdf->newPage(Zend_Pdf_Page::SIZE_A4));

$font = Zend_Pdf_Font::fontWithPath($fontPath);
$page->setFont($font, $fontSize);
$page->drawText($text, $x, $y);
...
// This font name should be stored somewhere...
$fontName = $font->getFontName(Zend_Pdf_Font::NAME_POSTSCRIPT,
                               'en',
                               'UTF-8');
...
$pdf->save($docPath);
...
```

```
...
$pdf = Zend_Pdf::load($docPath);
...
$pdf->pages[] = ($page = $pdf->newPage(Zend_Pdf_Page::SIZE_A4));

/* $srcPage->extractFont($fontName) can also be used here */
$font = $pdf->extractFont($fontName);

$page->setFont($font, $fontSize);
$page->drawText($text, $x, $y);
...
$pdf->save($docPath, true /* incremental update mode */);
...
```

Extracted fonts can be used in the place of any other font with the following limitations:

- Extracted font can be used only in the context of the document from which it was extracted.
- Possibly embedded font program is actually not extracted. So extracted font can't provide correct font metrics and original font has to be used for text width calculations:

```
...
$font = $pdf->extractFont($fontName);
$originalFont = Zend_Pdf_Font::fontWithPath($fontPath);

$page->setFont($font /* use extracted font for drawing */, $fontSize);
$xPosition = $x;
for ($charIndex = 0; $charIndex < strlen($text); $charIndex++) {
    $page->drawText($text[$charIndex], xPosition, $y);

    // Use original font for text width calculation
    $width = $originalFont->widthForGlyph(
                $originalFont->glyphNumberForCharacter($text[$charIndex])
             );
    $xPosition += $width/$originalFont->getUnitsPerEm()*$fontSize;
}
...
```

5.8. Image Drawing

Zend_Pdf_Page class provides drawImage() method to draw image:

```
/**
 * Draw an image at the specified position on the page.
 *
 * @param Zend_Pdf_Resource_Image $image
 * @param float $x1
 * @param float $y1
 * @param float $x2
 * @param float $y2
 * @return Zend_Pdf_Page
 */
public function drawImage(Zend_Pdf_Resource_Image $image, $x1, $y1, $x2, $y2);
```

Image objects should be created with Zend_Pdf_Image::imageWithPath($filePath) method (JPG, PNG and TIFF images are supported now):

Example 606. Image drawing

```
...
// load image
$image = Zend_Pdf_Image::imageWithPath('my_image.jpg');

$pdfPage->drawImage($image, 100, 100, 400, 300);
...
```

Important! JPEG support requires PHP GD extension to be configured. Important! PNG support requires ZLIB extension to be configured to work with Alpha channel images.

Refer to the PHP documentation for detailed information (http://www.php.net/manual/en/ref.image.php). (http://www.php.net/manual/en/ref.zlib.php).

5.9. Line drawing style

Line drawing style is defined by line width, line color and line dashing pattern. All of this parameters can be assigned by Zend_Pdf_Page class methods:

```
/** Set line color. */
public function setLineColor(Zend_Pdf_Color $color);

/** Set line width. */
public function setLineWidth(float $width);

/**
 * Set line dashing pattern.
 *
 * Pattern is an array of floats:
 *     array(on_length, off_length, on_length, off_length, ...)
 * Phase is shift from the beginning of line.
 *
 * @param array $pattern
 * @param array $phase
 * @return Zend_Pdf_Page
 */
```

```
public function setLineDashingPattern($pattern, $phase = 0);
```

5.10. Fill style

`Zend_Pdf_Page::drawRectangle()`, `Zend_Pdf_Page::drawPolygon()`, `Zend_Pdf_Page::drawCircle()` and `Zend_Pdf_Page::drawEllipse()` methods take `$fillType` argument as an optional parameter. It can be:

- Zend_Pdf_Page::SHAPE_DRAW_STROKE - stroke shape
- Zend_Pdf_Page::SHAPE_DRAW_FILL - only fill shape
- Zend_Pdf_Page::SHAPE_DRAW_FILL_AND_STROKE - fill and stroke (default behavior)

`Zend_Pdf_Page::drawPolygon()` methods also takes an additional parameter `$fillMethod`:

- Zend_Pdf_Page::FILL_METHOD_NON_ZERO_WINDING (default behavior)

  PDF reference describes this rule as follows:

  > The nonzero winding number rule determines whether a given point is inside a path by conceptually drawing a ray from that point to infinity in any direction and then examining the places where a segment of the path crosses the ray. Starting with a count of 0, the rule adds 1 each time a path segment crosses the ray from left to right and subtracts 1 each time a segment crosses from right to left. After counting all the crossings, if the result is 0 then the point is outside the path; otherwise it is inside. Note: The method just described does not specify what to do if a path segment coincides with or is tangent to the chosen ray. Since the direction of the ray is arbitrary, the rule simply chooses a ray that does not encounter such problem intersections. For simple convex paths, the nonzero winding number rule defines the inside and outside as one would intuitively expect. The more interesting cases are those involving complex or self-intersecting paths like the ones shown in Figure 4.10 (in a PDF Reference). For a path consisting of a five-pointed star, drawn with five connected straight line segments intersecting each other, the rule considers the inside to be the entire area enclosed by the star, including the pentagon in the center. For a path composed of two concentric circles, the areas enclosed by both circles are considered to be inside, provided that both are drawn in the same direction. If the circles are drawn in opposite directions, only the "doughnut" shape between them is inside, according to the rule; the "doughnut hole" is outside.

- Zend_Pdf_Page::FILL_METHOD_EVEN_ODD

  PDF reference describes this rule as follows:

  > An alternative to the nonzero winding number rule is the even-odd rule. This rule determines the "insideness" of a point by drawing a ray from that point in any direction and simply counting the number of path segments that cross the ray, regardless of direction. If this number is odd, the point is inside; if even, the point is outside. This yields the same results as the nonzero winding number rule for paths with simple shapes, but produces different results for more complex shapes. Figure 4.11 (in a PDF Reference) shows the effects of applying the even-odd rule to complex paths. For the five-pointed star, the rule

considers the triangular points to be inside the path, but not the pentagon in the center. For the two concentric circles, only the "doughnut" shape between the two circles is considered inside, regardless of the directions in which the circles are drawn.

5.11. Linear Transformations

5.11.1. Rotations

PDF page can be rotated before applying any draw operation. It can be done by `Zend_Pdf_Page::rotate()` method:

```
/**
 * Rotate the page.
 *
 * @param float $x  - the X co-ordinate of rotation point
 * @param float $y  - the Y co-ordinate of rotation point
 * @param float $angle - rotation angle
 * @return Zend_Pdf_Page
 */
public function rotate($x, $y, $angle);
```

5.11.2. Starting from ZF 1.8, scaling

Scaling transformation is provided by `Zend_Pdf_Page::scale()` method:

```
/**
 * Scale coordination system.
 *
 * @param float $xScale - X dimention scale factor
 * @param float $yScale - Y dimention scale factor
 * @return Zend_Pdf_Page
 */
public function scale($xScale, $yScale);
```

5.11.3. Starting from ZF 1.8, translating

Coordinate system shifting is performed by `Zend_Pdf_Page::translate()` method:

```
/**
 * Translate coordination system.
 *
 * @param float $xShift - X coordinate shift
 * @param float $yShift - Y coordinate shift
 * @return Zend_Pdf_Page
 */
public function translate($xShift, $yShift);
```

5.11.4. Starting from ZF 1.8, skewing

Page skewing can be done using `Zend_Pdf_Page::skew()` method:

```
/**
 * Translate coordination system.
 *
```

```
 * @param float $x  - the X co-ordinate of axis skew point
 * @param float $y  - the Y co-ordinate of axis skew point
 * @param float $xAngle - X axis skew angle
 * @param float $yAngle - Y axis skew angle
 * @return Zend_Pdf_Page
 */
public function skew($x, $y, $xAngle, $yAngle);
```

5.12. Save/restore graphics state

At any time page graphics state (current font, font size, line color, fill color, line style, page rotation, clip area) can be saved and then restored. Save operation puts data to a graphics state stack, restore operation retrieves it from there.

There are two methods in `Zend_Pdf_Page` class for these operations:

```
/**
 * Save the graphics state of this page.
 * This takes a snapshot of the currently applied style, position,
 * clipping area and any rotation/translation/scaling that has been
 * applied.
 *
 * @return Zend_Pdf_Page
 */
public function saveGS();

/**
 * Restore the graphics state that was saved with the last call to
 * saveGS().
 *
 * @return Zend_Pdf_Page
 */
public function restoreGS();
```

5.13. Clipping draw area

PDF and `Zend_Pdf` module support clipping of draw area. Current clip area limits the regions of the page affected by painting operators. It's a whole page initially.

`Zend_Pdf_Page` class provides a set of methods for clipping operations.

```
/**
 * Intersect current clipping area with a rectangle.
 *
 * @param float $x1
 * @param float $y1
 * @param float $x2
 * @param float $y2
 * @return Zend_Pdf_Page
 */
public function clipRectangle($x1, $y1, $x2, $y2);
```

```
/**
 * Intersect current clipping area with a polygon.
 *
 * @param array $x  - array of float (the X co-ordinates of the vertices)
```

```
     * @param array $y  - array of float (the Y co-ordinates of the vertices)
     * @param integer $fillMethod
     * @return Zend_Pdf_Page
     */
    public function clipPolygon($x,
                                $y,
                                $fillMethod =
                                    Zend_Pdf_Page::FILL_METHOD_NON_ZERO_WINDING);
```

```
    /**
     * Intersect current clipping area with a circle.
     *
     * @param float $x
     * @param float $y
     * @param float $radius
     * @param float $startAngle
     * @param float $endAngle
     * @return Zend_Pdf_Page
     */
    public function clipCircle($x,
                               $y,
                               $radius,
                               $startAngle = null,
                               $endAngle = null);
```

```
    /**
     * Intersect current clipping area with an ellipse.
     *
     * Method signatures:
     * drawEllipse($x1, $y1, $x2, $y2);
     * drawEllipse($x1, $y1, $x2, $y2, $startAngle, $endAngle);
     *
     * @todo process special cases with $x2-$x1 == 0 or $y2-$y1 == 0
     *
     * @param float $x1
     * @param float $y1
     * @param float $x2
     * @param float $y2
     * @param float $startAngle
     * @param float $endAngle
     * @return Zend_Pdf_Page
     */
    public function clipEllipse($x1,
                                $y1,
                                $x2,
                                $y2,
                                $startAngle = null,
                                $endAngle = null);
```

5.14. Styles

`Zend_Pdf_Style` class provides styles functionality.

Styles can be used to store a set of graphic state parameters and apply it to a PDF page by one operation:

```
    /**
     * Set the style to use for future drawing operations on this page
```

```
     *
     * @param Zend_Pdf_Style $style
     * @return Zend_Pdf_Page
     */
    public function setStyle(Zend_Pdf_Style $style);

    /**
     * Return the style, applied to the page.
     *
     * @return Zend_Pdf_Style|null
     */
    public function getStyle();
```

`Zend_Pdf_Style` class provides a set of methods to set or get different graphics state parameters:

```
    /**
     * Set line color.
     *
     * @param Zend_Pdf_Color $color
     * @return Zend_Pdf_Page
     */
    public function setLineColor(Zend_Pdf_Color $color);
```

```
    /**
     * Get line color.
     *
     * @return Zend_Pdf_Color|null
     */
    public function getLineColor();
```

```
    /**
     * Set line width.
     *
     * @param float $width
     * @return Zend_Pdf_Page
     */
    public function setLineWidth($width);
```

```
    /**
     * Get line width.
     *
     * @return float
     */
    public function getLineWidth();
```

```
    /**
     * Set line dashing pattern
     *
     * @param array $pattern
     * @param float $phase
     * @return Zend_Pdf_Page
     */
    public function setLineDashingPattern($pattern, $phase = 0);
```

```
    /**
     * Get line dashing pattern
```

```
     *
     * @return array
     */
    public function getLineDashingPattern();
```

```
    /**
     * Get line dashing phase
     *
     * @return float
     */
    public function getLineDashingPhase();
```

```
    /**
     * Set fill color.
     *
     * @param Zend_Pdf_Color $color
     * @return Zend_Pdf_Page
     */
    public function setFillColor(Zend_Pdf_Color $color);
```

```
    /**
     * Get fill color.
     *
     * @return Zend_Pdf_Color|null
     */
    public function getFillColor();
```

```
    /**
     * Set current font.
     *
     * @param Zend_Pdf_Resource_Font $font
     * @param float $fontSize
     * @return Zend_Pdf_Page
     */
    public function setFont(Zend_Pdf_Resource_Font $font, $fontSize);
```

```
    /**
     * Modify current font size
     *
     * @param float $fontSize
     * @return Zend_Pdf_Page
     */
    public function setFontSize($fontSize);
```

```
    /**
     * Get current font.
     *
     * @return Zend_Pdf_Resource_Font $font
     */
    public function getFont();
```

```
    /**
     * Get current font size
     *
     * @return float $fontSize
     */
```

```
public function getFontSize();
```

5.15. Transparency

`Zend_Pdf` module supports transparency handling.

Transparency may be set using `Zend_Pdf_Page::setAlpha()` method:

```
/**
 * Set the transparency
 *
 * $alpha == 0  - transparent
 * $alpha == 1  - opaque
 *
 * Transparency modes, supported by PDF:
 * Normal (default), Multiply, Screen, Overlay, Darken, Lighten,
 * ColorDodge, ColorBurn, HardLight, SoftLight, Difference, Exclusion
 *
 * @param float $alpha
 * @param string $mode
 * @throws Zend_Pdf_Exception
 * @return Zend_Pdf_Page
 */
public function setAlpha($alpha, $mode = 'Normal');
```

6. Interactive Features

6.1. Destinations

A destination defines a particular view of a document, consisting of the following items:

- The page of the document to be displayed.
- The location of the document window on that page.
- The magnification (zoom) factor to use when displaying the page.

Destinations may be associated with outline items (Document Outline (bookmarks)), annotations (Annotations), or actions (Actions). In each case, the destination specifies the view of the document to be presented when the outline item or annotation is opened or the action is performed. In addition, the optional document open action can be specified.

6.1.1. Supported Destination Types

The following types are supported by `Zend_Pdf` component.

6.1.1.1. Zend_Pdf_Destination_Zoom

Display the specified page, with the coordinates (left, top) positioned at the upper-left corner of the window and the contents of the page magnified by the factor zoom.

Destination object may be created using `Zend_Pdf_Destination_Zoom::create($page, $left = null, $top = null, $zoom = null)` method.

Where:

- $page is a destination page (a Zend_Pdf_Page object or a page number).
- $left is a left edge of the displayed page (float).
- $top is a top edge of the displayed page (float).
- $zoom is a zoom factor (float).

NULL, specified for $left, $top or $zoom parameter means "current viewer application value".

Zend_Pdf_Destination_Zoom class also provides the following methods:

- FloatgetLeftEdge();
- setLeftEdge(float $left);
- FloatgetTopEdge();
- setTopEdge(float $top);
- FloatgetZoomFactor();
- setZoomFactor(float $zoom);

6.1.1.2. Zend_Pdf_Destination_Fit

Display the specified page, with the coordinates (left, top) positioned at the upper-left corner of the window and the contents of the page magnified by the factor zoom. Display the specified page, with its contents magnified just enough to fit the entire page within the window both horizontally and vertically. If the required horizontal and vertical magnification factors are different, use the smaller of the two, centering the page within the window in the other dimension.

Destination object may be created using Zend_Pdf_Destination_Fit::create($page) method.

Where $page is a destination page (a Zend_Pdf_Page object or a page number).

6.1.1.3. Zend_Pdf_Destination_FitHorizontally

Display the specified page, with the vertical coordinate top positioned at the top edge of the window and the contents of the page magnified just enough to fit the entire width of the page within the window.

Destination object may be created using Zend_Pdf_Destination_FitHorizontally::create($page, $top) method.

Where:

- $page is a destination page (a Zend_Pdf_Page object or a page number).
- $top is a top edge of the displayed page (float).

Zend_Pdf_Destination_FitHorizontally class also provides the following methods:

- FloatgetTopEdge();
- setTopEdge(float $top);

6.1.1.4. Zend_Pdf_Destination_FitVertically

Display the specified page, with the horizontal coordinate left positioned at the left edge of the window and the contents of the page magnified just enough to fit the entire height of the page within the window.

Destination object may be created using `Zend_Pdf_Destination_FitVertically::create($page, $left)` method.

Where:

- `$page` is a destination page (a `Zend_Pdf_Page` object or a page number).
- `$left` is a left edge of the displayed page (float).

`Zend_Pdf_Destination_FitVertically` class also provides the following methods:

- Float`getLeftEdge()`;
- `setLeftEdge(float $left)`;

6.1.1.5. Zend_Pdf_Destination_FitRectangle

Display the specified page, with its contents magnified just enough to fit the rectangle specified by the coordinates left, bottom, right, and top entirely within the window both horizontally and vertically. If the required horizontal and vertical magnification factors are different, use the smaller of the two, centering the rectangle within the window in the other dimension.

Destination object may be created using `Zend_Pdf_Destination_FitRectangle::create($page, $left, $bottom, $right, $top)` method.

Where:

- `$page` is a destination page (a `Zend_Pdf_Page` object or a page number).
- `$left` is a left edge of the displayed page (float).
- `$bottom` is a bottom edge of the displayed page (float).
- `$right` is a right edge of the displayed page (float).
- `$top` is a top edge of the displayed page (float).

`Zend_Pdf_Destination_FitRectangle` class also provides the following methods:

- Float`getLeftEdge()`;
- `setLeftEdge(float $left)`;
- Float`getBottomEdge()`;
- `setBottomEdge(float $bottom)`;
- Float`getRightEdge()`;
- `setRightEdge(float $right)`;

- FloatgetTopEdge();
- setTopEdge(float $top);

6.1.1.6. Zend_Pdf_Destination_FitBoundingBox

Display the specified page, with its contents magnified just enough to fit its bounding box entirely within the window both horizontally and vertically. If the required horizontal and vertical magnification factors are different, use the smaller of the two, centering the bounding box within the window in the other dimension.

Destination object may be created using Zend_Pdf_Destination_FitBoundingBox::create($page, $left, $bottom, $right, $top) method.

Where $page is a destination page (a Zend_Pdf_Page object or a page number).

6.1.1.7. Zend_Pdf_Destination_FitBoundingBoxHorizontally

Display the specified page, with the vertical coordinate top positioned at the top edge of the window and the contents of the page magnified just enough to fit the entire width of its bounding box within the window.

Destination object may be created using Zend_Pdf_Destination_FitBoundingBoxHorizontally::create($page, $top) method.

Where

- $page is a destination page (a Zend_Pdf_Page object or a page number).
- $top is a top edge of the displayed page (float).

Zend_Pdf_Destination_FitBoundingBoxHorizontally class also provides the following methods:

- FloatgetTopEdge();
- setTopEdge(float $top);

6.1.1.8. Zend_Pdf_Destination_FitBoundingBoxVertically

Display the specified page, with the horizontal coordinate left positioned at the left edge of the window and the contents of the page magnified just enough to fit the entire height of its bounding box within the window.

Destination object may be created using Zend_Pdf_Destination_FitBoundingBoxVertically::create($page, $left) method.

Where

- $page is a destination page (a Zend_Pdf_Page object or a page number).
- $left is a left edge of the displayed page (float).

`Zend_Pdf_Destination_FitBoundingBoxVertically` class also provides the following methods:

- Float`getLeftEdge()`;
- `setLeftEdge(float $left)`;

6.1.1.9. Zend_Pdf_Destination_Named

All destinations listed above are "Explicit Destinations".

In addition to this, PDF document may contain a dictionary of such destinations which may be used to reference from outside the PDF (e.g. '`http://www.mycompany.com/document.pdf#chapter3`').

`Zend_Pdf_Destination_Named` objects allow to refer destinations from the document named destinations dictionary.

Named destination object may be created using `Zend_Pdf_Destination_Named::create(string $name)` method.

`Zend_Pdf_Destination_Named` class provides the only one additional method:

String`getName()`;

6.1.2. Document level destination processing

`Zend_Pdf` class provides a set of destinations processing methods.

Each destination object (including named destinations) can be resolved using the `resolveDestination($destination)` method. It returns corresponding `Zend_Pdf_Page` object, if destination target is found, or `NULL` otherwise.

`Zend_Pdf::resolveDestination()` method also takes an optional boolean parameter `$refreshPageCollectionHashes`, which is `TRUE` by default. It forces `Zend_Pdf` object to refresh internal page collection hashes since document pages list may be updated by user using `Zend_Pdf::$pages` property (Working with Pages). It may be turned off for performance reasons, if it's known that document pages list wasn't changed since last method request.

Complete list of named destinations can be retrieved using `Zend_Pdf::getNamedDestinations()` method. It returns an array of `Zend_Pdf_Target` objects, which are actually either an explicit destination or a GoTo action (Actions).

`Zend_Pdf::getNamedDestination(string $name)` method returns specified named destination (an explicit destination or a GoTo action).

PDF document named destinations dictionary may be updated with `Zend_Pdf::setNamedDestination(string $name, $destination)` method, where `$destination` is either an explicit destination (any destination except `Zend_Pdf_Destination_Named`) or a GoTo action.

If `NULL` is specified in place of `$destination`, then specified named destination is removed.

Unresolvable named destinations are automatically removed from a document while document saving.

Example 607. Destinations usage example

```
$pdf = new Zend_Pdf();
$page1 = $pdf->newPage(Zend_Pdf_Page::SIZE_A4);
$page2 = $pdf->newPage(Zend_Pdf_Page::SIZE_A4);
$page3 = $pdf->newPage(Zend_Pdf_Page::SIZE_A4);
// Page created, but not included into pages list

$pdf->pages[] = $page1;
$pdf->pages[] = $page2;

$destination1 = Zend_Pdf_Destination_Fit::create($page2);
$destination2 = Zend_Pdf_Destination_Fit::create($page3);

// Returns $page2 object
$page = $pdf->resolveDestination($destination1);

// Returns null, page 3 is not included into document yet
$page = $pdf->resolveDestination($destination2);

$pdf->setNamedDestination('Page2', $destination1);
$pdf->setNamedDestination('Page3', $destination2);

// Returns $destination2
$destination = $pdf->getNamedDestination('Page3');

// Returns $destination1
$pdf->resolveDestination(Zend_Pdf_Destination_Named::create('Page2'));

// Returns null, page 3 is not included into document yet
$pdf->resolveDestination(Zend_Pdf_Destination_Named::create('Page3'));
```

6.2. Actions

Instead of simply jumping to a destination in the document, an annotation or outline item can specify an action for the viewer application to perform, such as launching an application, playing a sound, or changing an annotation's appearance state.

6.2.1. Supported action types

The following action types are recognized while loading PDF document:

- `Zend_Pdf_Action_GoTo` - go to a destination in the current document.
- `Zend_Pdf_Action_GoToR` - go to a destination in another document.
- `Zend_Pdf_Action_GoToE` - go to a destination in an embedded file.
- `Zend_Pdf_Action_Launch` - launch an application or open or print a document.
- `Zend_Pdf_Action_Thread` - begin reading an article thread.
- `Zend_Pdf_Action_URI` - resolve a URI.
- `Zend_Pdf_Action_Sound` - play a sound.
- `Zend_Pdf_Action_Movie` - play a movie.
- `Zend_Pdf_Action_Hide` - hides or shows one or more annotations on the screen.

- `Zend_Pdf_Action_Named` - execute an action predefined by the viewer application:
  - *NextPage* - Go to the next page of the document.
  - *PrevPage* - Go to the previous page of the document.
  - *FirstPage* - Go to the first page of the document.
  - *LastPage* - Go to the last page of the document.
- `Zend_Pdf_Action_SubmitForm` - send data to a uniform resource locator.
- `Zend_Pdf_Action_ResetForm` - set fields to their default values.
- `Zend_Pdf_Action_ImportData` - import field values from a file.
- `Zend_Pdf_Action_JavaScript` - execute a JavaScript script.
- `Zend_Pdf_Action_SetOCGState` - set the state of one or more optional content groups.
- `Zend_Pdf_Action_Rendition` - control the playing of multimedia content (begin, stop, pause, or resume a playing rendition).
- `Zend_Pdf_Action_Trans` - update the display of a document, using a transition dictionary.
- `Zend_Pdf_Action_GoTo3DView` - set the current view of a 3D annotation.

Only `Zend_Pdf_Action_GoTo` and `Zend_Pdf_Action_URI` actions can be created by user now.

GoTo action object can be created using `Zend_Pdf_Action_GoTo::create($destination)` method, where `$destination` is a `Zend_Pdf_Destination` object or a string which can be used to identify named destination.

`Zend_Pdf_Action_URI::create($uri[, $isMap])` method has to be used to create a URI action (see API documentation for the details). Optional `$isMap` parameter is set to FALSE by default.

It also supports the following methods:

6.2.2. Actions chaining

Actions objects can be chained using `Zend_Pdf_Action::$next` public property.

It's an array of `Zend_Pdf_Action` objects, which also may have their sub-actions.

`Zend_Pdf_Action` class supports RecursiveIterator interface, so child actions may be iterated recursively:

```
$pdf = new Zend_Pdf();
$page1 = $pdf->newPage(Zend_Pdf_Page::SIZE_A4);
$page2 = $pdf->newPage(Zend_Pdf_Page::SIZE_A4);
// Page created, but not included into pages list
$page3 = $pdf->newPage(Zend_Pdf_Page::SIZE_A4);

$pdf->pages[] = $page1;
$pdf->pages[] = $page2;

$action1 = Zend_Pdf_Action_GoTo::create(
                            Zend_Pdf_Destination_Fit::create($page2));
$action2 = Zend_Pdf_Action_GoTo::create(
```

```
                                Zend_Pdf_Destination_Fit::create($page3));
$action3 = Zend_Pdf_Action_GoTo::create(
                                Zend_Pdf_Destination_Named::create('Chapter1'));
$action4 = Zend_Pdf_Action_GoTo::create(
                                Zend_Pdf_Destination_Named::create('Chapter5'));

$action2->next[] = $action3;
$action2->next[] = $action4;

$action1->next[] = $action2;

$actionsCount = 1; // Note! Iteration doesn't include top level action and
                   // walks through children only
$iterator = new RecursiveIteratorIterator(
                                        $action1,
                                        RecursiveIteratorIterator::SELF_FIRST);
foreach ($iterator as $chainedAction) {
    $actionsCount++;
}

// Prints 'Actions in a tree: 4'
printf("Actions in a tree: %d\n", $actionsCount++);
```

6.2.3. Document Open Action

Special open action may be specify a destination to be displayed or an action to be performed when the document is opened.

`Zend_Pdf_Target Zend_Pdf::getOpenAction()` method returns current document open action (or `NULL` if open action is not set).

`setOpenAction(Zend_Pdf_Target $openAction = null)` method sets document open action or clean it if `$openAction` is `NULL`.

6.3. Document Outline (bookmarks)

A PDF document may optionally display a document outline on the screen, allowing the user to navigate interactively from one part of the document to another. The outline consists of a tree-structured hierarchy of outline items (sometimes called bookmarks), which serve as a visual table of contents to display the document's structure to the user. The user can interactively open and close individual items by clicking them with the mouse. When an item is open, its immediate children in the hierarchy become visible on the screen; each child may in turn be open or closed, selectively revealing or hiding further parts of the hierarchy. When an item is closed, all of its descendants in the hierarchy are hidden. Clicking the text of any visible item activates the item, causing the viewer application to jump to a destination or trigger an action associated with the item.

`Zend_Pdf` class provides public property `$outlines` which is an array of `Zend_Pdf_Outline` objects.

```
$pdf = Zend_Pdf::load($path);

// Remove outline item
unset($pdf->outlines[0]->childOutlines[1]);

// Set Outline to be displayed in bold
$pdf->outlines[0]->childOutlines[3]->setIsBold(true);
```

```
// Add outline entry
$pdf->outlines[0]->childOutlines[5]->childOutlines[] =
    Zend_Pdf_Outline::create('Chapter 2', 'chapter_2');

$pdf->save($path, true);
```

Outline attributes may be retrieved or set using the following methods:

- `string getTitle()` - get outline item title.
- `setTitle(string $title)` - set outline item title.
- `boolean isOpen()` - TRUE if outline is open by default.
- `setIsOpen(boolean $isOpen)` - set isOpen state.
- `boolean isItalic()` - TRUE if outline item is displayed in italic.
- `setIsItalic(boolean $isItalic)` - set isItalic state.
- `boolean isBold()` - TRUE if outline item is displayed in bold.
- `setIsBold(boolean $isBold)` - set isBold state.
- `Zend_Pdf_Color_Rgb getColor()` - get outline text color (`NULL` means black).
- `setColor(Zend_Pdf_Color_Rgb $color)` - set outline text color (`NULL` means black).
- `Zend_Pdf_Target getTarget()` - get outline target (action or explicit or named destination object).
- `setTarget(Zend_Pdf_Target|string $target)` - set outline target (action or destination). String may be used to identify named destination. `NULL` means 'no target'.
- `array getOptions()` - get outline attributes as an array.
- `setOptions(array $options)` - set outline options. The following options are recognized: 'title', 'open', 'color', 'italic', 'bold', and 'target'.

New outline may be created in two ways:

- `Zend_Pdf_Outline::create(string $title[, Zend_Pdf_Target|string $target])`
- `Zend_Pdf_Outline::create(array $options)`

Each outline object may have child outline items listed in `Zend_Pdf_Outline::$childOutlines` public property. It's an array of `Zend_Pdf_Outline` objects, so outlines are organized in a tree.

`Zend_Pdf_Outline` class implements RecursiveArray interface, so child outlines may be recursively iterated using RecursiveIteratorIterator:

```
$pdf = Zend_Pdf::load($path);

foreach ($pdf->outlines as $documentRootOutlineEntry) {
    $iterator = new RecursiveIteratorIterator(
                    $documentRootOutlineEntry,
                    RecursiveIteratorIterator::SELF_FIRST
                );
    foreach ($iterator as $childOutlineItem) {
```

```
        $OutlineItemTarget = $childOutlineItem->getTarget();
        if ($OutlineItemTarget instanceof Zend_Pdf_Destination) {
            if ($pdf->resolveDestination($OutlineItemTarget) === null) {
                // Mark Outline item with unresolvable destination
                // using RED color
                $childOutlineItem->setColor(new Zend_Pdf_Color_Rgb(1, 0, 0));
            }
        } else if ($OutlineItemTarget instanceof Zend_Pdf_Action_GoTo) {
            $OutlineItemTarget->setDestination();
            if ($pdf->resolveDestination($OutlineItemTarget) === null) {
                // Mark Outline item with unresolvable destination
                // using RED color
                $childOutlineItem->setColor(new Zend_Pdf_Color_Rgb(1, 0, 0));
            }
        }
    }
}

$pdf->save($path, true);
```

All outline items with unresolved destinations (or destinations of GoTo actions) are updated while document saving by setting their targets to NULL. So document will not be corrupted by removing pages referenced by outlines.

6.4. Annotations

An annotation associates an object such as a note, sound, or movie with a location on a page of a PDF document, or provides a way to interact with the user by means of the mouse and keyboard.

All annotations are represented by `Zend_Pdf_Annotation` abstract class.

Annotation may be attached to a page using `Zend_Pdf_Page::attachAnnotation(Zend_Pdf_Annotation $annotation)` method.

Three types of annotations may be created by user now:

- `Zend_Pdf_Annotation_Link::create($x1, $y1, $x2, $y2, $target)` where `$target` is an action object or a destination or string (which may be used in place of named destination object).
- `Zend_Pdf_Annotation_Text::create($x1, $y1, $x2, $y2, $text)`
- `Zend_Pdf_Annotation_FileAttachment::create($x1, $y1, $x2, $y2, $fileSpecification)`

A link annotation represents either a hypertext link to a destination elsewhere in the document or an action to be performed.

A text annotation represents a "sticky note" attached to a point in the PDF document.

A file attachment annotation contains a reference to a file.

The following methods are shared between all annotation types:

- `setLeft(float $left)`
- `float getLeft()`
- `setRight(float $right)`

- `float getRight()`
- `setTop(float $top)`
- `float getTop()`
- `setBottom(float $bottom)`
- `float getBottom()`
- `setText(string $text)`
- `string getText()`

Text annotation property is a text to be displayed for the annotation or, if this type of annotation does not display text, an alternate description of the annotation's contents in human-readable form.

Link annotation objects also provide two additional methods:

- `setDestination(Zend_Pdf_Target|string $target)`
- `Zend_Pdf_Target getDestination()`

7. Document Info and Metadata

A PDF document may include general information such as the document's title, author, and creation and modification dates.

Historically this information is stored using special Info structure. This structure is available for read and writing as an associative array using `properties` public property of `Zend_Pdf` objects:

```
$pdf = Zend_Pdf::load($pdfPath);

echo $pdf->properties['Title'] . "\n";
echo $pdf->properties['Author'] . "\n";

$pdf->properties['Title'] = 'New Title.';
$pdf->save($pdfPath);
```

The following keys are defined by PDF v1.4 (Acrobat 5) standard:

- *Title* - string, optional, the document's title.
- *Author* - string, optional, the name of the person who created the document.
- *Subject* - string, optional, the subject of the document.
- *Keywords* - string, optional, keywords associated with the document.
- *Creator* - string, optional, if the document was converted to PDF from another format, the name of the application (for example, Adobe FrameMaker®) that created the original document from which it was converted.
- *Producer* - string, optional, if the document was converted to PDF from another format, the name of the application (for example, Acrobat Distiller) that converted it to PDF..
- *CreationDate* - string, optional, the date and time the document was created, in the following form: "D:YYYYMMDDHHmmSSOHH'mm'", where:

- *YYYY* is the year.
- *MM* is the month.
- *DD* is the day (01–31).
- *HH* is the hour (00–23).
- *mm*is the minute (00–59).
- *SS* is the second (00–59).
- *O* is the relationship of local time to Universal Time (UT), denoted by one of the characters +, #, or Z (see below).
- *HH* followed by ' is the absolute value of the offset from UT in hours (00–23).
- *mm* followed by ' is the absolute value of the offset from UT in minutes (00–59).

  The apostrophe character (') after HH and mm is part of the syntax. All fields after the year are optional. (The prefix D:, although also optional, is strongly recommended.) The default values for MM and DD are both 01; all other numerical fields default to zero values. A plus sign (+) as the value of the O field signifies that local time is later than UT, a minus sign (#) that local time is earlier than UT, and the letter Z that local time is equal to UT. If no UT information is specified, the relationship of the specified time to UT is considered to be unknown. Whether or not the time zone is known, the rest of the date should be specified in local time.

  For example, December 23, 1998, at 7:52 PM, U.S. Pacific Standard Time, is represented by the string "D:199812231952#08'00'".

- *ModDate* - string, optional, the date and time the document was most recently modified, in the same form as *CreationDate*.
- *Trapped* - boolean, optional, indicates whether the document has been modified to include trapping information.
  - *TRUE* - The document has been fully trapped; no further trapping is needed.
  - *FALSE* - The document has not yet been trapped; any desired trapping must still be done.
  - *NULL* - Either it is unknown whether the document has been trapped or it has been partly but not yet fully trapped; some additional trapping may still be needed.

Since PDF v 1.6 metadata can be stored in the special XML document attached to the PDF (XMP - Extensible Metadata Platform).

This XML document can be retrieved and attached to the PDF with `Zend_Pdf::getMetadata()` and `Zend_Pdf::setMetadata($metadata)` methods:

```
$pdf = Zend_Pdf::load($pdfPath);
$metadata = $pdf->getMetadata();
$metadataDOM = new DOMDocument();
$metadataDOM->loadXML($metadata);

$xpath = new DOMXPath($metadataDOM);
$pdfPreffixNamespaceURI = $xpath->query('/rdf:RDF/rdf:Description')
                                ->item(0)
```

```
                                    ->lookupNamespaceURI('pdf');
$xpath->registerNamespace('pdf', $pdfPreffixNamespaceURI);

$titleNode = $xpath->query('/rdf:RDF/rdf:Description/pdf:Title')->item(0);
$title = $titleNode->nodeValue;
...

$titleNode->nodeValue = 'New title';
$pdf->setMetadata($metadataDOM->saveXML());
$pdf->save($pdfPath);
```

Common document properties are duplicated in the Info structure and Metadata document (if presented). It's user application responsibility now to keep them synchronized.

8. Zend_Pdf module usage example

This section provides an example of module usage.

This example can be found in a `demos/Zend/Pdf/demo.php` file.

There are also `test.pdf` file, which can be used with this demo for test purposes.

```
                    Zend_Pdf_Page::SHAPE_DRAW_FILL_AND_STROKE,
                    Zend_Pdf_Page::FILL_METHOD_EVEN_ODD);

// Draw line
$page2->setLineWidth(0.5)
      ->drawLine(0, 25, 340, 25);

$page2->restoreGS();

// Coordination system movement, skewing and scaling
$page2->saveGS();
$page2->translate(60, 150)     // Shift coordination system
      ->skew(0, 0, 0, -M_PI/9) // Skew coordination system
      ->scale(0.9, 0.9);       // Scale coordination system

// Draw rectangle
$page2->setFillColor(new Zend_Pdf_Color_GrayScale(0.8))
      ->setLineColor(new Zend_Pdf_Color_GrayScale(0.2))
      ->setLineDashingPattern(array(3, 2, 3, 4), 1.6)
      ->drawRectangle(0, 50, 340, 0);

// Draw circle
$page2->setLineDashingPattern(Zend_Pdf_Page::LINE_DASHING_SOLID)
      ->setFillColor(new Zend_Pdf_Color_Rgb(1, 0, 0))
      ->drawCircle(25, 25, 25);

// Draw sectors
$page2->drawCircle(140, 25, 25, 2*M_PI/3, -M_PI/6)
      ->setFillColor(new Zend_Pdf_Color_Cmyk(1, 0, 0, 0))
      ->drawCircle(140, 25, 25, M_PI/6, 2*M_PI/3)
      ->setFillColor(new Zend_Pdf_Color_Rgb(1, 1, 0))
      ->drawCircle(140, 25, 25, -M_PI/6, M_PI/6);

// Draw ellipse
$page2->setFillColor(new Zend_Pdf_Color_Rgb(1, 0, 0))
      ->drawEllipse(190, 50, 340, 0)
      ->setFillColor(new Zend_Pdf_Color_Cmyk(1, 0, 0, 0))
      ->drawEllipse(190, 50, 340, 0, M_PI/6, 2*M_PI/3)
      ->setFillColor(new Zend_Pdf_Color_Rgb(1, 1, 0))
      ->drawEllipse(190, 50, 340, 0, -M_PI/6, M_PI/6);

// Draw and fill polygon
$page2->setFillColor(new Zend_Pdf_Color_Rgb(1, 0, 1));
$x = array();
$y = array();
for ($count = 0; $count < 8; $count++) {
    $x[] = 80 + 25*cos(3*M_PI_4*$count);
    $y[] = 25 + 25*sin(3*M_PI_4*$count);
}
$page2->drawPolygon($x, $y,
                    Zend_Pdf_Page::SHAPE_DRAW_FILL_AND_STROKE,
                    Zend_Pdf_Page::FILL_METHOD_EVEN_ODD);

// Draw line
$page2->setLineWidth(0.5)
      ->drawLine(0, 25, 340, 25);

$page2->restoreGS();

//------------------------------------------------------------------------

if (isset($argv[2])) {
    $pdf->save($argv[2]);
} else {
    $pdf->save($argv[1], true /* update */);
}
```

Zend_ProgressBar

1. Zend_ProgressBar

1.1. Introduction

`Zend_ProgressBar` is a component to create and update progressbars in different environments. It consists of a single backend, which outputs the progress through one of the multiple adapters. On every update, it takes an absolute value and optionally a status message, and then calls the adapter with some precalculated values like percentage and estimated time left.

1.2. Basic Usage of Zend_Progressbar

`Zend_ProgressBar` is quite easy in its usage. You simply create a new instance of `Zend_Progressbar`, defining a min- and a max-value, and choose an adapter to output the data. If you want to process a file, you would do something like:

```
$progressBar = new Zend_ProgressBar($adapter, 0, $fileSize);

while (!feof($fp)) {
    // Do something

    $progressBar->update($currentByteCount);
}

$progressBar->finish();
```

In the first step, an instance of `Zend_ProgressBar` is created, with a specific adapter, a min-value of 0 and a max-value of the total filesize. Then a file is processed and in every loop the progressbar is updated with the current byte count. At the end of the loop, the progressbar status is set to finished.

You can also call the `update()` method of `Zend_ProgressBar` without arguments, which just recalculates ETA and notifies the adapter. This is useful when there is no data update but you want the progressbar to be updated.

1.3. Persistent progress

If you want the progressbar to be persistent over multiple requests, you can give the name of a session namespace as fourth argument to the constructor. In that case, the progressbar will not notify the adapter within the constructor, but only when you call `update()` or `finish()`. Also the current value, the status text and the start time for ETA calculation will be fetched in the next request run again.

1.4. Standard adapters

`Zend_ProgressBar` comes with the following three adapters:

- Section 1.4.1, “Zend_ProgressBar_Adapter_Console”

- Section 1.4.2, “Zend_ProgressBar_Adapter_JsPush”
- Section 1.4.3, “Zend_ProgressBar_Adapter_JsPull”

1.4.1. Zend_ProgressBar_Adapter_Console

`Zend_ProgressBar_Adapter_Console` is a text-based adapter for terminals. It can automatically detect terminal widths but supports custom widths as well. You can define which elements are displayed with the progressbar and as well customize the order of them. You can also define the style of the progressbar itself.

Automatic console width recognition

`shell_exec` is required for this feature to work on *nix based systems. On windows, there is always a fixed terminal width of 80 character, so no recognition is required there.

You can set the adapter options either via the `set*` methods or give an array or a `Zend_Config` instance with options as first parameter to the constructor. The available options are:

- `outputStream`: A different output-stream, if you don't want to stream to STDOUT. Can be any other stream like `php://stderr` or a path to a file.
- `width`: Either an integer or the `AUTO` constant of `Zend_Console_ProgressBar`.
- `elements`: Either `NULL` for default or an array with at least one of the following constants of `Zend_Console_ProgressBar` as value:
  - `ELEMENT_PERCENT`: The current value in percent.
  - `ELEMENT_BAR`: The visual bar which display the percentage.
  - `ELEMENT_ETA`: The automatic calculated ETA. This element is firstly displayed after five seconds, because in this time, it is not able to calculate accurate results.
  - `ELEMENT_TEXT`: An optional status message about the current process.
- `textWidth`: Width in characters of the `ELEMENT_TEXT` element. Default is 20.
- `charset`: Charset of the `ELEMENT_TEXT` element. Default is utf-8.
- `barLeftChar`: A string which is used left-hand of the indicator in the progressbar.
- `barRightChar`: A string which is used right-hand of the indicator in the progressbar.
- `barIndicatorChar`: A string which is used for the indicator in the progressbar. This one can be empty.

1.4.2. Zend_ProgressBar_Adapter_JsPush

`Zend_ProgressBar_Adapter_JsPush` is an adapter which let's you update a progressbar in a browser via Javascript Push. This means that no second connection is required to gather the status about a running process, but that the process itself sends its status directly to the browser.

You can set the adapter options either via the `set*` methods or give an array or a `Zend_Config` instance with options as first parameter to the constructor. The available options are:

- `updateMethodName`: The javascript method which should be called on every update. Default value is `Zend_ProgressBar_Update`.

- `finishMethodName`: The javascript method which should be called after finish status was set. Default value is `NULL`, which means nothing is done.

The usage of this adapter is quite simple. First you create a progressbar in your browser, either with JavaScript or previously created with plain HTML. Then you define the update method and optionally the finish method in JavaScript, both taking a json object as single argument. Then you call a webpage with the long-running process in a hidden `iframe` or `object` tag. While the process is running, the adapter will call the update method on every update with a json object, containing the following parameters:

- `current`: The current absolute value

- `max`: The max absolute value

- `percent`: The calculated percentage

- `timeTaken`: The time how long the process ran yet

- `timeRemaining`: The expected time for the process to finish

- `text`: The optional status message, if given

Example 609. Basic example for the client-side stuff

This example illustrates a basic setup of HTML, CSS and JavaScript for the JsPush adapter

```
<div id="zend-progressbar-container">
    <div id="zend-progressbar-done"></div>
</div>

<iframe src="long-running-process.php" id="long-running-process"></iframe>
```

```
#long-running-process {
    position: absolute;
    left: -100px;
    top: -100px;

    width: 1px;
    height: 1px;
}

#zend-progressbar-container {
    width: 100px;
    height: 30px;

    border: 1px solid #000000;
    background-color: #ffffff;
}

#zend-progressbar-done {
    width: 0;
    height: 30px;

    background-color: #000000;
}
```

```
function Zend_ProgressBar_Update(data)
{
    document.getElementById('zend-progressbar-done').style.width = data.percent + '%';
}
```

This will create a simple container with a black border and a block which indicates the current process. You should not hide the `iframe` or `object` by `display: none;`, as some browsers like Safari 2 will not load the actual content then.

Instead of creating your custom progressbar, you may want to use one of the available JavaScript libraries like Dojo, jQuery etc. For example, there are:

- Dojo: http://dojotoolkit.org/book/dojo-book-0-9/part-2-dijit/user-assistance-and-feedback/progress-bar
- jQuery: http://t.wits.sg/2008/06/20/jquery-progress-bar-11/
- MooTools: http://davidwalsh.name/dw-content/progress-bar.php
- Prototype: http://livepipe.net/control/progressbar

Interval of updates

You should take care of not sending too many updates, as every update has a min-size of 1kb. This is a requirement for the Safari browser to actually render and execute the function call. Internet Explorer has a similar limitation of 256 bytes.

1.4.3. Zend_ProgressBar_Adapter_JsPull

`Zend_ProgressBar_Adapter_JsPull` is the opposite of jsPush, as it requires to pull for new updates, instead of pushing updates out to the browsers. Generally you should use the adapter with the persistence option of the `Zend_ProgressBar`. On notify, the adapter sends a JSON string to the browser, which looks exactly like the JSON string which is send by the jsPush adapter. The only difference is, that it contains an additional parameter, `finished`, which is either FALSE when `update()` is called or TRUE, when `finish()` is called.

You can set the adapter options either via the `set*` methods or give an array or a `Zend_Config` instance with options as first parameter to the constructor. The available options are:

- `exitAfterSend`: Exits the current request after the data were send to the browser. Default is TRUE.

Zend_Queue

1. Introduction

Zend_Queue provides a factory function to create specific queue client objects.

A message queue is a method for distributed processing. For example, a Job Broker application may accept multiple applications for jobs from a variety of sources.

You could create a queue "/queue/applications" that would have a sender and a receiver. The sender would be any available source that could connect to your message service or indirectly to an application (web) that could connect to the message service.

The sender sends a message to the queue:

```
<resume>
    <name>John Smith</name>
    <location>
        <city>San Francisco</city>
        <state>California</state>
        <zip>00001</zip>
    </location>
    <skills>
        <programming>PHP</programming>
        <programming>Perl</programming>
    </skills>
</resume>
```

The recipient or consumer of the queue would pick up the message and process the resume.

There are many messaging patterns that can be applied to queues to abstract the flow of control from the code and provide metrics, transformations, and monitoring of messages queues. A good book on messaging patterns is Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions (Addison-Wesley Signature Series) (ISBN-10 0321127420; ISBN-13 978-0321127426).

2. Example usage

The below example of Zend_Queue shows a variety of features, including queue creation, queue retrieval, message retrieval, message deletion, and sending messages.

```
// For configuration options
// @see Zend_Queue_Adapater::__construct()
$options = array(
    'name' => 'queue1',
);

// Create an array queue
$queue = new Zend_Queue('Array', $options);

// Get list of queues
foreach ($queue->getQueues() as $name) {
    echo $name, "\n";
}
```

```php
// Create a new queue
$queue2 = $queue->createQueue('queue2');

// Get number of messages in a queue (supports Countable interface from SPL)
echo count($queue);

// Get up to 5 messages from a queue
$messages = $queue->receive(5);

foreach ($messages as $i => $message) {
    echo $message->body, "\n";

    // We have processed the message; now we remove it from the queue.
    $queue->deleteMessage($message);
}

// Send a message to the currently active queue
$queue->send('My Test Message');

// Delete a queue we created and all of it's messages
$queue->deleteQueue('queue2');
```

3. Framework

The Zend_Queue is a proxy that hides the details of the queue services. The queue services are represented by Zend_Queue_Adapter_<service>. For example, Zend_Queue_Adapter_Db is a queue that will use database tables to store and retrieve messages.

Below is an example for using database tables for a queuing system:

```php
$options = array(
    'name'          => 'queue1',
    'driverOptions' => array(
        'host'      => '127.0.0.1',
        'port'      => '3306',
        'username'  => 'queue',
        'password'  => 'queue',
        'dbname'    => 'queue',
        'type'      => 'pdo_mysql'
    )
);

// Create a database queue.
// Zend_Queue will prepend Zend_Queue_Adapter_ to 'Db' for the class name.
$queue = new Zend_Queue('Db', $options);
```

The Zend_Queue constructor will create a Zend_Queue_Adapter_Db and initialize the adapter with the configuration settings.

The accepted configuration settings for each adapter are provided in the adapter notes.

Zend_Queue returns messages using the class Zend_Queue_Message_Iterator, which is an implementation of SPL Iterator and Countable. Zend_Queue_Message_Iterator contains an array of Zend_Queue_Message objects.

```php
$messages = $queue->receive(5);
```

```
foreach ($messages as $i => $message) {
    echo "$i) Message => ", $message->body, "\n";
}
```

Any exceptions thrown are of class `Zend_Queue_Exception`.

3.1. Introduction

`Zend_Queue` is a proxy class that represents an adapter.

The `send()`, `count($queue)`, and `receive()` methods are employed by each adapter to interact with queues.

The `createQueue()`, `deleteQueue()` methods are used to manage queues.

3.2. Commonality among adapters

The queue services supported by `Zend_Queue` do not all support the same functions. For example, `Zend_Queue_Adapter_Array`, `Zend_Queue_Adapter_Db`, support all functions, while `Zend_Queue_Adapter_Activemq` does not support queue listing, queue deletion, or counting of messages.

You can determine what functions are supported by using `Zend_Queue::isSupported()` or `Zend_Queue::getCapabilities()`.

- *`createQueue()`* - create a queue
- *`deleteQueue()`* - delete a queue
- *`send()`* - send a message

  `send()` is not available in all adapters; the `Zend_Queue_Adapter_Null` does not support `send()`.
- *`receive()`* - receive messages

  `receive()` is not available in all adapters; the `Zend_Queue_Adapter_Null` does not support `receive()`.
- *`deleteMessage()`* - delete a message
- *`count()`* - count the number of messages in a queue
- *`isExists()`* - checks the existence of a queue

`receive()` methods are employed by each adapter to interact with queues.

The `createQueue()` and `deleteQueue()` methods are used to manage queues.

4. Adapters

`Zend_Queue` supports all queues implementing the interface `Zend_Queue_Adapter_AdapterInterface`. The following Message Queue services are supported:

- Apache ActiveMQ.
- A database driven queue via `Zend_Db`.

- A MemcacheQ queue driven via `Memcache`.
- Zend Platform's Job Queue.
- A local array. Useful for unit testing.

Limitations

Message transaction handling is not supported.

4.1. Specific Adapters - Configuration settings

If a default setting is indicated then the parameter is optional. If a default setting is not specified then the parameter is required.

4.1.1. Apache ActiveMQ - Zend_Queue_Adapter_Activemq

Options listed here are known requirements. Not all messaging servers require username or password.

- *$options['name'] = '/temp/queue1';*

  This is the name of the queue that you wish to start using. (Required)

- *$options['driverOptions']['host'] = 'host.domain.tld';*

  $options['driverOptions']['host'] = '127.0.0.1';

  You may set host to an IP address or a hostname.

  Default setting for host is '127.0.0.1'.

- *$options['driverOptions']['port'] = 61613;*

  Default setting for port is 61613.

- *$options['driverOptions']['username'] = 'username';*

  Optional for some messaging servers. Read the manual for your messaging server.

- *$options['driverOptions']['password'] = 'password';*

  Optional for some messaging servers. Read the manual for your messaging server.

- *$options['driverOptions']['timeout_sec'] = 2;*

  $options['driverOptions']['timeout_usec'] = 0;

  This is the amount of time that `Zend_Queue_Adapter_Activemq` will wait for read activity on a socket before returning no messages.

4.1.2. Db - Zend_Queue_Adapter_Db

Driver options are checked for a few required options such as *type*, *host*, *username*, *password*, and *dbname*. You may pass along additional parameters for `Zend_DB::factory()` as parameters in `$options['driverOptions']`. An example of an additional option not listed here, but could be passed would be *port*.

```
$options = array(
    'driverOptions' => array(
        'host'      => 'db1.domain.tld',
        'username'  => 'my_username',
        'password'  => 'my_password',
        'dbname'    => 'messaging',
        'type'      => 'pdo_mysql',
        'port'      => 3306, // optional parameter.
    ),
    'options' => array(
        // use Zend_Db_Select for update, not all databases can support this
        // feature.
        Zend_Db_Select::FOR_UPDATE => true
    )
);

// Create a database queue.
$queue = new Zend_Queue('Db', $options);
```

- *$options['name'] = 'queue1';*

  This is the name of the queue that you wish to start using. (Required)

- *$options['driverOptions']['type'] = 'Pdo';*

  type is the adapter you wish to have `Zend_Db::factory()` use. This is the first parameter for the `Zend_Db::factory()` class method call.

- *$options['driverOptions']['host'] = 'host.domain.tld';*

  $options['driverOptions']['host'] = '127.0.0.1';

  You may set host to an IP address or a hostname.

  Default setting for host is '127.0.0.1'.

- *$options['driverOptions']['username'] = 'username';*

- *$options['driverOptions']['password'] = 'password';*

- *$options['driverOptions']['dbname'] = 'dbname';*

  The database name that you have created the required tables for. See the notes section below.

4.1.3. MemcacheQ - Zend_Queue_Adapter_Memcacheq

- *$options['name'] = 'queue1';*

  This is the name of the queue that you wish to start using. (Required)

- *$options['driverOptions']['host'] = 'host.domain.tld';*

  $options['driverOptions']['host'] = '127.0.0.1;'

  You may set host to an IP address or a hostname.

  Default setting for host is '127.0.0.1'.

- *$options['driverOptions']['port'] = 22201;*

The default setting for port is 22201.

4.1.4. Zend Platform Job Queue - Zend_Queue_Adapter_PlatformJobQueue

- *$options['daemonOptions']['host'] = '127.0.0.1:10003';*

  The hostname and port corresponding to the Zend Platform Job Queue daemon you will use. (Required)

- *$options['daemonOptions']['password'] = '1234';*

  The password required for accessing the Zend Platform Job Queue daemon. (Required)

4.1.5. Array - Zend_Queue_Adapter_Array

- *$options['name'] = 'queue1';*

  This is the name of the queue that you wish to start using. (Required)

4.2. Notes for Specific Adapters

The following adapters have notes:

4.2.1. Apache ActiveMQ

Visibility duration for `Zend_Queue_Adapter_Activemq` is not available.

While Apache's ActiveMQ will support multiple subscriptions, the `Zend_Queue` does not. You must create a new `Zend_Queue` object for each individual subscription.

ActiveMQ queue/topic names must begin with one of:

- `/queue/`
- `/topic/`
- `/temp-queue/`
- `/temp-topic/`

For example: `/queue/testing`

The following functions are not supported:

- `create()` - create queue. Calling this function will throw an exception.
- `delete()` - delete queue. Calling this function will throw an exception.
- `getQueues()` - list queues. Calling this function will throw an exception.

4.2.2. Zend_Db

The database *CREATE TABLE (...)* SQL statement can be found in `Zend/Queue/Adapter/Db/queue.sql`.

4.2.3. MemcacheQ

Memcache can be downloaded from http://www.danga.com/memcached/.

MemcacheQ can be downloaded from http://memcachedb.org/memcacheq/.

- `deleteMessage()` - Messages are deleted upon reception from the queue. Calling this function would have no effect. Calling this function will throw an error.
- `count()` or `count($adapter)` - MemcacheQ does not support a method for counting the number of items in a queue. Calling this function will throw an error.

4.2.4. Zend Platform Job Queue

Job Queue is a feature of Zend Platform's Enterprise Solution offering. It is not a traditional message queue, and instead allows you to queue a script to execute, along with the parameters you wish to pass to it. You can find out more about Job Queue on the zend.com website.

The following is a list of methods where this adapter's behavior diverges from the standard offerings:

- `create()` - Zend Platform does not have the concept of discrete queues; instead, it allows administrators to provide scripts for processing jobs. Since adding new scripts is restricted to the administration interface, this method simply throws an exception indicating the action is forbidden.
- `isExists()` - Just like `create()`, since Job Queue does not have a notion of named queues, this method throws an exception when invoked.
- `delete()` - similar to `create()`, deletion of JQ scripts is not possible except via the admin interface; this method raises an exception.
- `getQueues()` - Zend Platform does not allow introspection into the attached job handling scripts via the API. This method throws an exception.
- `count()` - returns the total number of jobs currently active in the Job Queue.
- `send()` - this method is perhaps the one method that diverges most from other adapters. The `$message` argument may be one of three possible types, and will operate differently based on the value passed:
  - string - the name of a script registered with Job Queue to invoke. If passed in this way, no arguments are provided to the script.
  - array - an array of values with which to configure a `ZendApi_Job` object. These may include the following:
    - `script` - the name of the Job Queue script to invoke. (Required)
    - `priority` - the job priority to use when registering with the queue.
    - `name` - a short string describing the job.
    - `predecessor` - the ID of a job on which this one depends, and which must be executed before this one may begin.
    - `preserved` - whether or not to retain the job within the Job Queue history. By default, off; pass a `TRUE` value to retain it.

- `user_variables` - an associative array of all variables you wish to have in scope during job execution (similar to named arguments).
- `interval` - how often, in seconds, the job should run. By default, this is set to 0, indicating it should run once, and once only.
- `end_time` - an expiry time, past which the job should not run. If the job was set to run only once, and `end_time` has passed, then the job will not be executed. If the job was set to run on an interval, it will not execute again once `end_time` has passed.
- `schedule_time` - a UNIX timestamp indicating when to run the job; by default, 0, indicating the job should run as soon as possible.
- `application_id` - the application identifier of the job. By default, this is `NULL`, indicating that one will be automatically assigned by the queue, if the queue was assigned an application ID.

As noted, only the `script` argument is required; all others are simply available to allow passing more fine-grained detail on how and when to run the job.

- `ZendApi_Job` - finally, you may simply pass a `ZendApi_Job` instance, and it will be passed along to Platform's Job Queue.

In all instances, `send()` returns a `Zend_Queue_Message_PlatformJob` object, which provides access to the `ZendApi_Job` object used to communicate with Job Queue.

- `receive()` - retrieves a list of active jobs from Job Queue. Each job in the returned set will be an instance of `Zend_Queue_Message_PlatformJob`.
- `deleteMessage()` - since this adapter only works with Job Queue, this method expects the provided `$message` to be a `Zend_Queue_Message_PlatformJob` instance, and will throw an exception otherwise.

4.2.5. Array (local)

The Array queue is a PHP `array()` in local memory. The `Zend_Queue_Adapter_Array` is good for unit testing.

5. Customizing Zend_Queue

5.1. Creating your own adapter

`Zend_Queue` will accept any adapter that implements `Zend_Queue_Adapter_AdapterAbstract`. You can create your own adapter by extending one of the existing adapters, or the abstract class `Zend_Queue_Adapter_AdapterAbstract`. I suggest reviewing `Zend_Queue_Adapter_Array` as this adapter is the easiest to conceptualize.

```
class Custom_DbForUpdate extends Zend_Queue_Adapter_Db
{
    /**
     * @see code in tests/Zend/Queue/Custom/DbForUpdate.php
     *
     * Custom_DbForUpdate uses the SELECT ... FOR UPDATE to find it's rows.
```

```
     * this is more likely to produce the wanted rows than the existing code.
     *
     * However, not all databases have SELECT ... FOR UPDATE as a feature.
     *
     * Note: this was later converted to be an option for Zend_Queue_Adapter_Db
     *
     * This code still serves as a good example.
     */
}

$options = array(
    'name'          => 'queue1',
    'driverOptions' => array(
        'host'      => '127.0.0.1',
        'port'      => '3306',
        'username'  => 'queue',
        'password'  => 'queue',
        'dbname'    => 'queue',
        'type'      => 'pdo_mysql'
    )
);

$adapter = new Custom_DbForUpdate($options);
$queue = new Zend_Queue($adapter, $options);
```

You can also change the adapter on the fly as well.

```
$adapter = new MyCustom_Adapter($options);
$queue   = new Zend_Queue($options);
$queue->setAdapter($adapter);
echo "Adapter: ", get_class($queue->getAdapter()), "\n";
```

or

```
$options = array(
    'name'           => 'queue1',
    'namespace'      => 'Custom',
    'driverOptions'  => array(
        'host'       => '127.0.0.1',
        'port'       => '3306',
        'username'   => 'queue',
        'password'   => 'queue',
        'dbname'     => 'queue',
        'type'       => 'pdo_mysql'
    )
);
$queue = new Zend_Queue('DbForUpdate', $config); // loads Custom_DbForUpdate
```

5.2. Creating your own message class

Zend_Queue will also accept your own message class. Our variables start with an underscore. For example:

```
class Zend_Queue_Message
{
    protected $_data = array();
}
```

You can extend the existing messaging class. See the example code in `tests/Zend/Queue/Custom/Message.php`.

5.3. Creating your own message iterator class

`Zend_Queue` will also accept your own message iterator class. The message iterator class is used to return messages from `Zend_Queue_Adapter_Abstract::recieve()`. `Zend_Queue_Abstract::receive()` should always return a container class like `Zend_Queue_Message_Iterator`, even if there is only one message.

See the example filename in `tests/Zend/Queue/Custom/Messages.php`.

5.4. Creating your own queue class

`Zend_Queue` can also be overloaded easily.

See the example filename in `tests/Zend/Queue/Custom/Queue.php`.

6. Stomp

`Zend_Queue_Stomp` provides a basic client to communicate with Stomp compatible servers. Some servers, such as Apache ActiveMQ and RabbitMQ, will allow you to communicate by other methods, such as HTTP, and XMPP.

The Stomp protocol provides StompConnect which supports any Java Message Service (JMS) provider. Stomp is supported by Apache ActiveMQ, RabbitMQ, stompserver, and Gozirra.

6.1. Stomp - Supporting classes

- `Zend_Queue_Stomp_Frame`. This class provides the basic functions for manipulating a Stomp Frame.
- `Zend_Queue_Stomp_Client`. This class provides the basic functions to `send()` and `receive()` `Zend_Queue_Stomp_Frames` to and from a Stomp compatible server.

Zend_Reflection

1. Introduction

`Zend_Reflection` is a drop-in extension to PHP's own Reflection API, providing several additional features:

- Ability to retrieve return values types.
- Ability to retrieve method and function parameter types.
- Ability to retrieve class property types.
- DocBlocks gain a Reflection class, allowing introspection of docblocks. This provides the ability to determine what annotation tags have been defined as well as to retrieve their values, and the ability to retrieve the short and long descriptions.
- Files gain a Reflection class, allowing introspection of PHP files. This provides the ability to determine what functions and classes are defined in a given file, as well as to introspect them.
- Ability to override any Reflection class with your own variant, for the entire reflection tree you create.

In general, `Zend_Reflection` works just like the standard Reflection API, but provides a few additional methods for retrieving artifacts not defined in the Reflection API.

2. Zend_Reflection Examples

Example 610. Performing reflection on a file

```
$r = new Zend_Reflection_File($filename);
printf(
    "===> The %s file\n".
    "     has %d lines\n",
    $r->getFileName(),
    $r->getEndLine()
);

$classes = $r->getClasses();
echo "     It has " . count($classes) . ":\n";
foreach ($classes as $class) {
    echo "         " . $class->getName() . "\n";
}

$functions = $r->getFunctions();
echo "     It has " . count($functions) . ":\n";
foreach ($functions as $function) {
    echo "         " . $function->getName() . "\n";
}
```

Example 611. Performing reflection on a class

```
$r = new Zend_Reflection_Class($class);

printf(
    "The class level docblock has the short description: %s\n".
    "The class level docblock has the long description:\n%s\n",
    $r->getDocblock()->getShortDescription(),
    $r->getDocblock()->getLongDescription(),
);

// Get the declaring file reflection
$file = $r->getDeclaringFile();
```

Example 612. Performing reflection on a method

```
$r = new Zend_Reflection_Method($class, $name);

printf(
"The method '%s' has a return type of %s",
    $r->getName(),
    $r->getReturn()
);

foreach ($r->getParameters() as $key => $param) {
    printf(
        "Param at position '%d' is of type '%s'\n",
        $key,
        $param->getType()
    );
}
```

Example 613. Performing reflection on a docblock

```
$r = new Zend_Reflection_Method($class, $name);
$docblock = $r->getDocblock();

printf(
    "The short description: %s\n".
    "The long description:\n%s\n",
    $r->getDocblock()->getShortDescription(),
    $r->getDocblock()->getLongDescription(),
);

foreach ($docblock->getTags() as $tag) {
    printf(
        "Annotation tag '%s' has the description '%s'\n",
        $tag->getName(),
        $tag->getDescription()
    );
}
```

3. Zend_Reflection Reference

The various classes in `Zend_Reflection` mimic the API of PHP's Reflection API - with one important difference. PHP's Reflection API does not provide introspection into docblock annotation tags, nor into parameter variable types or return types.

Zend_Reflection analyzes method docblock annotations to determine parameter variable types and the return type. Specifically, the @param and @return annotations are used. However, you can also check for any other annotation tags, as well as the standard "short" and "long" descriptions.

Each reflection object in Zend_Reflection overrides the getDocblock() method to return an instance of Zend_Reflection_Docblock. This class provides introspection into the docblocks and annotation tags.

Zend_Reflection_File is a new reflection class that allows introspection of PHP files. With it, you can retrieve the classes, functions, and global PHP code contained in the file.

Finally, the various methods that return other reflection objects allow a second parameter, the name of the reflection class to use for the returned reflection object.

3.1. Zend_Reflection_Docblock

Zend_Reflection_Docblock is the heart of Zend_Reflection's value-add over PHP's Reflection API. It provides the following methods:

- getContents(): returns the full contents of the docblock.
- getStartLine(): returns the starting position of the docblock within the defining file.
- getEndLine(): get last line of docblock within the defining file.
- getShortDescription(): get the short, one-line description (usually the first line of the docblock).
- getLongDescription(): get the long description from the docblock.
- hasTag($name): determine if the docblock has the given annotation tag.
- getTag($name): Retrieve the given annotation tag reflection object, or a boolean FALSE if it's not present.
- getTags($filter): Retrieve all tags, or all tags matching the given $filter string. The tags returned will be an array of Zend_Reflection_Docblock_Tag objects.

3.2. Zend_Reflection_Docblock_Tag

Zend_Reflection_Docblock_Tag provides reflection for individual annotation tags. Most tags consist of only a name and a description. In the case of some special tags, the class provides a factory method for retrieving an instance of the appropriate class.

The following methods are defined for Zend_Reflection_Docblock_Tag:

- factory($tagDocblockLine): instantiate the appropriate tag reflection class and return it.
- getName(): return the annotation tag name.
- getDescription(): return the annotation description.

3.3. Zend_Reflection_Docblock_Tag_Param

Zend_Reflection_Docblock_Tag_Param is a specialized version of Zend_Reflection_Docblock_Tag. The @param annotation tag description consists of the

parameter type, variable name, and variable description. It adds the following methods to `Zend_Reflection_Docblock_Tag`:

- `getType()`: return the parameter variable type.
- `getVariableName()`: return the parameter variable name.

3.4. Zend_Reflection_Docblock_Tag_Return

Like `Zend_Reflection_Docblock_Tag_Param`, `Zend_Reflection_Docblock_Tag_Return` is a specialized version of `Zend_Reflection_Docblock_Tag`. The `@return` annotation tag description consists of the return type and variable description. It adds the following method to `Zend_Reflection_Docblock_Tag`:

- `getType()`: return the return type.

3.5. Zend_Reflection_File

`Zend_Reflection_File` provides introspection into PHP files. With it, you can introspect the classes, functions, and bare PHP code defined in a file. It defines the following methods:

- `getFileName()`: retrieve the filename of the file being reflected.
- `getStartLine()`: retrieve the starting line of the file (always "1").
- `getEndLine()` retrieve the last line / number of lines in the file.
- `getDocComment($reflectionClass = 'Zend_Reflection_Docblock')`: retrive the file-level docblock reflection object.
- `getClasses($reflectionClass = 'Zend_Reflection_Class')`: retrieve an array of reflection objects, one for each class defined in the file.
- `getFunctions($reflectionClass = 'Zend_Reflection_Function')`: retrieve an array of reflection objects, one for each function defined in the file.
- `getClass($name = null, $reflectionClass = 'Zend_Reflection_Class')`: retrieve the reflection object for a single class.
- `getContents()`: retrieve the full contents of the file.

3.6. Zend_Reflection_Class

`Zend_Reflection_Class` extends `ReflectionClass`, and follows its API. It adds one additional method, `getDeclaringFile()`, which may be used to retrieve the `Zend_Reflection_File` reflection object for the defining file.

Additionally, the following methods add an additional argument for specifying the reflection class to use when fetching a reflection object:

- `getDeclaringFile($reflectionClass = 'Zend_Reflection_File')`
- `getDocblock($reflectionClass = 'Zend_Reflection_Docblock')`
- `getInterfaces($reflectionClass = 'Zend_Reflection_Class')`

- `getMethod($reflectionClass = 'Zend_Reflection_Method')`
- `getMethods($filter = -1, $reflectionClass = 'Zend_Reflection_Method')`
- `getParentClass($reflectionClass = 'Zend_Reflection_Class')`
- `getProperty($name, $reflectionClass = 'Zend_Reflection_Property')`
- `getProperties($filter = -1, $reflectionClass = 'Zend_Reflection_Property')`

3.7. Zend_Reflection_Extension

`Zend_Reflection_Extension` extends `ReflectionExtension`, and follows its API. It overrides the following methods to add an additional argument for specifying the reflection class to use when fetching a reflection object:

- `getFunctions($reflectionClass = 'Zend_Reflection_Function')`: retrieve an array of reflection objects representing the functions defined by the extension.
- `getClasses($reflectionClass = 'Zend_Reflection_Class')`: retrieve an array of reflection objects representing the classes defined by the extension.

3.8. Zend_Reflection_Function

`Zend_Reflection_Function` adds a method for retrieving the function return type, as well as overrides several methods to allow specifying the reflection class to use for returned reflection objects.

- `getDocblock($reflectionClass = 'Zend_Reflection_Docblock')`: retrieve the function docblock reflection object.
- `getParameters($reflectionClass = 'Zend_Reflection_Parameter')`: retrieve an array of all function parameter reflection objects.
- `getReturn()`: retrieve the return type reflection object.

3.9. Zend_Reflection_Method

`Zend_Reflection_Method` mirrors `Zend_Reflection_Function`, and only overrides one additional method:

- `getParentClass($reflectionClass = 'Zend_Reflection_Class')`: retrieve the parent class reflection object.

3.10. Zend_Reflection_Parameter

`Zend_Reflection_Parameter` adds a method for retrieving the parameter type, as well as overrides methods to allow specifying the reflection class to use on returned reflection objects.

- `getDeclaringClass($reflectionClass = 'Zend_Reflection_Class')`: get the declaring class of the parameter as a reflection object (if available).
- `getClass($reflectionClass = 'Zend_Reflection_Class')`: get the class of the parameter as a reflection object (if available).

- `getDeclaringFunction($reflectionClass = 'Zend_Reflection_Function')`: get the function of the parameter as a reflection object (if available).
- `getType()`: get the parameter type.

3.11. Zend_Reflection_Property

`Zend_Reflection_Property` overrides a single method in order to allow specifying the returned reflection object class:

- `getDeclaringClass($reflectionClass = 'Zend_Reflection_Class')`: retrieve the declaring class of the property as a reflection object.

Zend_Registry

1. Using the Registry

A registry is a container for storing objects and values in the application space. By storing the value in a registry, the same object is always available throughout your application. This mechanism is an alternative to using global storage.

The typical method to use registries with Zend Framework is through static methods in the `Zend_Registry` class. Alternatively, the registry can be used as an array object, so you can access elements stored within it with a convenient array-like interface.

1.1. Setting Values in the Registry

Use the static method `set()` to store an entry in the registry, .

Example 614. Example of set() Method Usage

```
Zend_Registry::set('index', $value);
```

The value returned can be an object, an array, or a scalar. You can change the value stored in a specific entry of the registry by calling the `set()` method to set the entry to a new value.

The index can be a scalar (`NULL`, string, or number), like an ordinary array.

1.2. Getting Values from the Registry

To retrieve an entry from the registry, use the static `get()` method.

Example 615. Example of get() Method Usage

```
$value = Zend_Registry::get('index');
```

The `getInstance()` method returns the singleton registry object. This registry object is iterable, making all values stored in the registry easily accessible.

Example 616. Example of Iterating over the Registry

```
$registry = Zend_Registry::getInstance();

foreach ($registry as $index => $value) {
    echo "Registry index $index contains:\n";
    var_dump($value);
}
```

1.3. Constructing a Registry Object

In addition to accessing the static registry via static methods, you can create an instance directly and use it as an object.

The registry instance you access through the static methods is simply one such instance. It is for convenience that it is stored statically, so that it is accessible from anywhere in an application.

Use the traditional `new` operator to instantiate `Zend_Registry`. Instantiating `Zend_Registry` using its constructor also makes initializing the entries in the registry simple by taking an associative array as an argument.

Example 617. Example of Constructing a Registry

```
$registry = new Zend_Registry(array('index' => $value));
```

Once such a `Zend_Registry` object is instantiated, you can use it by calling any array object method or by setting it as the singleton instance for `Zend_Registry` with the static method `setInstance()`.

Example 618. Example of Initializing the Singleton Registry

```
$registry = new Zend_Registry(array('index' => $value));

Zend_Registry::setInstance($registry);
```

The `setInstance()` method throws a `Zend_Exception` if the static registry has already been initialized.

1.4. Accessing the Registry as an Array

If you have several values to get or set, you may find it convenient to access the registry with array notation.

Example 619. Example of Array Access

```
$registry = Zend_Registry::getInstance();

$registry['index'] = $value;

var_dump( $registry['index'] );
```

1.5. Accessing the Registry as an Object

You may also find it convenient to access the registry in an object-oriented fashion by using index names as object properties. You must specifically construct the registry object using the `ArrayObject::ARRAY_AS_PROPS` option and initialize the static instance to enable this functionality.

You must set the `ArrayObject::ARRAY_AS_PROPS` option *before* the static registry has been accessed for the first time.

Known Issues with the ArrayObject::ARRAY_AS_PROPS Option

Some versions of PHP have proven very buggy when using the registry with the `ArrayObject::ARRAY_AS_PROPS` option.

Example 620. Example of Object Access

```
// in your application bootstrap:
$registry = new Zend_Registry(array(), ArrayObject::ARRAY_AS_PROPS)
Zend_Registry::setInstance($registry);
$registry->tree = 'apple';

.
.
.

// in a different function, elsewhere in your application:
$registry = Zend_Registry::getInstance();

echo $registry->tree; // echo's "apple"

$registry->index = $value;

var_dump($registry->index);
```

1.6. Querying if an Index Exists

To find out if a particular index in the registry has been set, use the static method `isRegistered()`.

Example 621. Example of isRegistered() Method Usage

```
if (Zend_Registry::isRegistered($index)) {
    $value = Zend_Registry::get($index);
}
```

To find out if a particular index in a registry array or object has a value, use the `isset()` function as you would with an ordinary array.

Example 622. Example of isset() Method Usage

```
$registry = Zend_Registry::getInstance();

// using array access syntax
if (isset($registry['index'])) {
    var_dump( $registry['index'] );
}

// using object access syntax
if (isset($registry->index)) {
    var_dump( $registry->index );
}
```

1.7. Extending the Registry

The static registry is an instance of the class `Zend_Registry`. If you want to add functionality to the registry, you should create a class that extends `Zend_Registry` and specify this class to instantiate for the singleton in the static registry. Use the static method `setClassName()` to specify the class.

The class must be a subclass of `Zend_Registry`.

Example 623. Example of Specifying the Singleton Registry's Class Name

```
Zend_Registry::setClassName('My_Registry');

Zend_Registry::set('index', $value);
```

The registry throws a `Zend_Exception` if you attempt to set the classname after the registry has been accessed for the first time. It is therefore recommended that you specify the class name for your static registry in your application bootstrap.

1.8. Unsetting the Static Registry

Although it is not normally necessary, you can unset the singleton instance of the registry, if desired. Use the static method `_unsetInstance()` to do so.

Data Loss Risk

When you use `_unsetInstance()`, all data in the static registry are discarded and cannot be recovered.

You might use this method, for example, if you want to use `setInstance()` or `setClassName()` after the singleton registry object has been initialized. Unsetting the singleton instance allows you to use these methods even after the singleton registry object has been set. Using `Zend_Registry` in this manner is not recommended for typical applications and environments.

Example 624. Example of _unsetInstance() Method Usage

```
Zend_Registry::set('index', $value);

Zend_Registry::_unsetInstance();

// change the class
Zend_Registry::setClassName('My_Registry');

Zend_Registry::set('index', $value);
```

Zend_Rest

1. Introduction

REST Web Services use service-specific XML formats. These ad-hoc standards mean that the manner for accessing a REST web service is different for each service. REST web services typically use URL parameters (GET data) or path information for requesting data and POST data for sending data.

Zend Framework provides both Client and Server capabilities, which, when used together allow for a much more "local" interface experience via virtual object property access. The Server component features automatic exposition of functions and classes using a meaningful and simple XML format. When accessing these services using the Client, it is possible to easily retrieve the return data from the remote call. Should you wish to use the client with a non-Zend_Rest_Server based service, it will still provide easier data access.

In addition to `Zend_Rest_Server` and `Zend_Rest_Client` components, `Zend_Rest_Route` and `Zend_Rest_Controller` classes are provided to aid routing REST requests to controllers.

2. Zend_Rest_Client

2.1. Introduction

Using the `Zend_Rest_Client` is very similar to using `SoapClient` objects (SOAP web service extension). You can simply call the REST service procedures as `Zend_Rest_Client` methods. Specify the service's full address in the `Zend_Rest_Client` constructor.

Example 625. A basic REST request

```
/**
 * Connect to framework.zend.com server and retrieve a greeting
 */
$client = new Zend_Rest_Client('http://framework.zend.com/rest');

echo $client->sayHello('Davey', 'Day')->get(); // "Hello Davey, Good Day"
```

Differences in calling

`Zend_Rest_Client` attempts to make remote methods look as much like native methods as possible, the only difference being that you must follow the method call with one of either `get()`, `post()`, `put()` or `delete()`. This call may be made via method chaining or in separate method calls:

```
$client->sayHello('Davey', 'Day');
echo $client->get();
```

2.2. Responses

All requests made using `Zend_Rest_Client` return a `Zend_Rest_Client_Response` object. This object has many properties that make it easier to access the results.

When the service is based on `Zend_Rest_Server`, `Zend_Rest_Client` can make several assumptions about the response, including response status (success or failure) and return type.

Example 626. Response Status

```
$result = $client->sayHello('Davey', 'Day')->get();

if ($result->isSuccess()) {
    echo $result; // "Hello Davey, Good Day"
}
```

In the example above, you can see that we use the request result as an object, to call `isSuccess()`, and then because of `__toString()`, we can simply `echo` the object to get the result. `Zend_Rest_Client_Response` will allow you to echo any scalar value. For complex types, you can use either array or object notation.

If however, you wish to query a service not using `Zend_Rest_Server` the `Zend_Rest_Client_Response` object will behave more like a `SimpleXMLElement`. However, to make things easier, it will automatically query the XML using XPath if the property is not a direct descendant of the document root element. Additionally, if you access a property as a method, you will receive the PHP value for the object, or an array of PHP value results.

Example 627. Using Technorati's Rest Service

```
$technorati = new Zend_Rest_Client('http://api.technorati.com/bloginfo');
$technorati->key($key);
$technorati->url('http://pixelated-dreams.com');
$result = $technorati->get();
echo $result->firstname() .' '. $result->lastname();
```

Example 628. Example Technorati Response

```
<?xml version="1.0" encoding="utf-8"?>
<!-- generator="Technorati API version 1.0 /bloginfo" -->
<!DOCTYPE tapi PUBLIC "-//Technorati, Inc.//DTD TAPI 0.02//EN"
                      "http://api.technorati.com/dtd/tapi-002.xml">
<tapi version="1.0">
    <document>
        <result>
            <url>http://pixelated-dreams.com</url>
            <weblog>
                <name>Pixelated Dreams</name>
                <url>http://pixelated-dreams.com</url>
                <author>
                    <username>DShafik</username>
                    <firstname>Davey</firstname>
                    <lastname>Shafik</lastname>
                </author>
                <rssurl>
                    http://pixelated-dreams.com/feeds/index.rss2
                </rssurl>
                <atomurl>
                    http://pixelated-dreams.com/feeds/atom.xml
                </atomurl>
                <inboundblogs>44</inboundblogs>
                <inboundlinks>218</inboundlinks>
                <lastupdate>2006-04-26 04:36:36 GMT</lastupdate>
                <rank>60635</rank>
            </weblog>
            <inboundblogs>44</inboundblogs>
            <inboundlinks>218</inboundlinks>
        </result>
    </document>
</tapi>
```

Here we are accessing the `firstname` and `lastname` properties. Even though these are not top-level elements, they are automatically returned when accessed by name.

Multiple items

If multiple items are found when accessing a value by name, an array of SimpleXMLElements will be returned; accessing via method notation will return an array of PHP values.

2.3. Request Arguments

Unless you are making a request to a `Zend_Rest_Server` based service, chances are you will need to send multiple arguments with your request. This is done by calling a method with the name of the argument, passing in the value as the first (and only) argument. Each of these method calls returns the object itself, allowing for chaining, or "fluent" usage. The first call, or the first argument if you pass in more than one argument, is always assumed to be the method when calling a `Zend_Rest_Server` service.

Example 629. Setting Request Arguments

```
$client = new Zend_Rest_Client('http://example.org/rest');

$client->arg('value1');
$client->arg2('value2');
$client->get();

// or

$client->arg('value1')->arg2('value2')->get();
```

Both of the methods in the example above, will result in the following get args: `?method=arg&arg1=value1&arg=value1&arg2=value2`

You will notice that the first call of `$client->arg('value1');` resulted in both `method=arg&arg1=value1` and `arg=value1`; this is so that `Zend_Rest_Server` can understand the request properly, rather than requiring pre-existing knowledge of the service.

Strictness of Zend_Rest_Client

Any REST service that is strict about the arguments it receives will likely fail using `Zend_Rest_Client`, because of the behavior described above. This is not a common practice and should not cause problems.

3. Zend_Rest_Server

3.1. Introduction

`Zend_Rest_Server` is intended as a fully-featured REST server.

3.2. REST Server Usage

Example 630. Basic Zend_Rest_Server Usage - Classes

```
$server = new Zend_Rest_Server();
$server->setClass('My_Service_Class');
$server->handle();
```

Example 631. Basic Zend_Rest_Server Usage - Functions

```
/**
 * Say Hello
 *
 * @param string $who
 * @param string $when
 * @return string
 */
function sayHello($who, $when)
{
    return "Hello $who, Good $when";
}

$server = new Zend_Rest_Server();
$server->addFunction('sayHello');
$server->handle();
```

3.3. Calling a Zend_Rest_Server Service

To call a `Zend_Rest_Server` service, you must supply a GET/POST `method` argument with a value that is the method you wish to call. You can then follow that up with any number of arguments using either the name of the argument (i.e. "who") or using `arg` following by the numeric position of the argument (i.e. "arg1").

Numeric index

Numeric arguments use a 1-based index.

To call `sayHello` from the example above, you can use either:

```
?method=sayHello&who=Davey&when=Day
```

or:

```
?method=sayHello&arg1=Davey&arg2=Day
```

3.4. Sending A Custom Status

When returning values, to return a custom status, you may return an array with a `status` key.

Example 632. Returning Custom Status

```php
/**
 * Say Hello
 *
 * @param string $who
 * @param string $when
 * @return array
 */
function sayHello($who, $when)
{
    return array('msg' => "An Error Occurred", 'status' => false);
}

$server = new Zend_Rest_Server();
$server->addFunction('sayHello');
$server->handle();
```

3.5. Returning Custom XML Responses

If you wish to return custom XML, simply return a `DOMDocument`, `DOMElement` or `SimpleXMLElement` object.

Example 633. Return Custom XML

```
/**
 * Say Hello
 *
 * @param string $who
 * @param string $when
 * @return SimpleXMLElement
 */
function sayHello($who, $when)
{
    $xml ='<?xml version="1.0" encoding="ISO-8859-1"?>
<mysite>
    <value>Hey $who! Hope you\'re having a good $when</value>
    <code>200</code>
</mysite>';

    $xml = simplexml_load_string($xml);
    return $xml;
}

$server = new Zend_Rest_Server();
$server->addFunction('sayHello');

$server->handle();
```

The response from the service will be returned without modification to the client.

Zend_Search_Lucene

1. Overview

1.1. Introduction

Zend_Search_Lucene is a general purpose text search engine written entirely in PHP 5. Since it stores its index on the filesystem and does not require a database server, it can add search capabilities to almost any PHP-driven website. Zend_Search_Lucene supports the following features:

- Ranked searching - best results returned first
- Many powerful query types: phrase queries, boolean queries, wildcard queries, proximity queries, range queries and many others.
- Search by specific field (e.g., title, author, contents)

Zend_Search_Lucene was derived from the Apache Lucene project. The currently (starting from ZF 1.6) supported Lucene index format versions are 1.4 - 2.3. For more information on Lucene, visit http://lucene.apache.org/java/docs/.

Previous Zend_Search_Lucene implementations support the Lucene 1.4 (1.9) - 2.1 index formats.

Starting from Zend Framework 1.5 any index created using pre-2.1 index format is automatically upgraded to Lucene 2.1 format after the Zend_Search_Lucene update and will not be compatible with Zend_Search_Lucene implementations included into Zend Framework 1.0.x.

1.2. Document and Field Objects

Zend_Search_Lucene operates with documents as atomic objects for indexing. A document is divided into named fields, and fields have content that can be searched.

A document is represented by the Zend_Search_Lucene_Document class, and this objects of this class contain instances of Zend_Search_Lucene_Field that represent the fields on the document.

It is important to note that any information can be added to the index. Application-specific information or metadata can be stored in the document fields, and later retrieved with the document during search.

It is the responsibility of your application to control the indexer. This means that data can be indexed from any source that is accessible by your application. For example, this could be the filesystem, a database, an HTML form, etc.

Zend_Search_Lucene_Field class provides several static methods to create fields with different characteristics:

```
$doc = new Zend_Search_Lucene_Document();
```

```
// Field is not tokenized, but is indexed and stored within the index.
// Stored fields can be retrived from the index.
$doc->addField(Zend_Search_Lucene_Field::Keyword('doctype',
                                                'autogenerated'));

// Field is not tokenized nor indexed, but is stored in the index.
$doc->addField(Zend_Search_Lucene_Field::UnIndexed('created',
                                                   time()));

// Binary String valued Field that is not tokenized nor indexed,
// but is stored in the index.
$doc->addField(Zend_Search_Lucene_Field::Binary('icon',
                                                $iconData));

// Field is tokenized and indexed, and is stored in the index.
$doc->addField(Zend_Search_Lucene_Field::Text('annotation',
                                              'Document annotation text'));

// Field is tokenized and indexed, but is not stored in the index.
$doc->addField(Zend_Search_Lucene_Field::UnStored('contents',
                                                  'My document content'));
```

Each of these methods (excluding the `Zend_Search_Lucene_Field::Binary()` method) has an optional `$encoding` parameter for specifying input data encoding.

Encoding may differ for different documents as well as for different fields within one document:

```
$doc = new Zend_Search_Lucene_Document();
$doc->addField(Zend_Search_Lucene_Field::Text('title',
                                              $title,
                                              'iso-8859-1'));
$doc->addField(Zend_Search_Lucene_Field::UnStored('contents',
                                                  $contents,
                                                  'utf-8'));
```

If encoding parameter is omitted, then the current locale is used at processing time. For example:

```
setlocale(LC_ALL, 'de_DE.iso-8859-1');
...
$doc->addField(Zend_Search_Lucene_Field::UnStored('contents', $contents));
```

Fields are always stored and returned from the index in UTF-8 encoding. Any required conversion to UTF-8 happens automatically.

Text analyzers (see below) may also convert text to some other encodings. Actually, the default analyzer converts text to 'ASCII//TRANSLIT' encoding. Be careful, however; this translation may depend on current locale.

Fields' names are defined at your discretion in the `addField()` method.

Java Lucene uses the 'contents' field as a default field to search. `Zend_Search_Lucene` searches through all fields by default, but the behavior is configurable. See the "Default search field" chapter for details.

1.3. Understanding Field Types

- `Keyword` fields are stored and indexed, meaning that they can be searched as well as displayed in search results. They are not split up into separate words by

tokenization. Enumerated database fields usually translate well to Keyword fields in Zend_Search_Lucene.

- UnIndexed fields are not searchable, but they are returned with search hits. Database timestamps, primary keys, file system paths, and other external identifiers are good candidates for UnIndexed fields.
- Binary fields are not tokenized or indexed, but are stored for retrieval with search hits. They can be used to store any data encoded as a binary string, such as an image icon.
- Text fields are stored, indexed, and tokenized. Text fields are appropriate for storing information like subjects and titles that need to be searchable as well as returned with search results.
- UnStored fields are tokenized and indexed, but not stored in the index. Large amounts of text are best indexed using this type of field. Storing data creates a larger index on disk, so if you need to search but not redisplay the data, use an UnStored field. UnStored fields are practical when using a Zend_Search_Lucene index in combination with a relational database. You can index large data fields with UnStored fields for searching, and retrieve them from your relational database by using a separate field as an identifier.

Table 113. Zend_Search_Lucene Field Types

Field Type	Stored	Indexed	Tokenized	Binary
Keyword	Yes	Yes	No	No
UnIndexed	Yes	No	No	No
Binary	Yes	No	No	Yes
Text	Yes	Yes	Yes	No
UnStored	No	Yes	Yes	No

1.4. HTML documents

Zend_Search_Lucene offers a HTML parsing feature. Documents can be created directly from a HTML file or string:

```
$doc = Zend_Search_Lucene_Document_Html::loadHTMLFile($filename);
$index->addDocument($doc);
...
$doc = Zend_Search_Lucene_Document_Html::loadHTML($htmlString);
$index->addDocument($doc);
```

Zend_Search_Lucene_Document_Html class uses the DOMDocument::loadHTML() and DOMDocument::loadHTMLFile() methods to parse the source HTML, so it doesn't need HTML to be well formed or to be XHTML. On the other hand, it's sensitive to the encoding specified by the "meta http-equiv" header tag.

Zend_Search_Lucene_Document_Html class recognizes document title, body and document header meta tags.

The 'title' field is actually the /html/head/title value. It's stored within the index, tokenized and available for search.

The 'body' field is the actual body content of the HTML file or string. It doesn't include scripts, comments or attributes.

The `loadHTML()` and `loadHTMLFile()` methods of `Zend_Search_Lucene_Document_Html` class also have second optional argument. If it's set to `TRUE`, then body content is also stored within index and can be retrieved from the index. By default, the body is tokenized and indexed, but not stored.

The third parameter of `loadHTML()` and `loadHTMLFile()` methods optionally specifies source HTML document encoding. It's used if encoding is not specified using Content-type HTTP-EQUIV meta tag.

Other document header meta tags produce additional document fields. The field 'name' is taken from 'name' attribute, and the 'content' attribute populates the field 'value'. Both are tokenized, indexed and stored, so documents may be searched by their meta tags (for example, by keywords).

Parsed documents may be augmented by the programmer with any other field:

```
$doc = Zend_Search_Lucene_Document_Html::loadHTML($htmlString);
$doc->addField(Zend_Search_Lucene_Field::UnIndexed('created',
                                                   time()));
$doc->addField(Zend_Search_Lucene_Field::UnIndexed('updated',
                                                   time()));
$doc->addField(Zend_Search_Lucene_Field::Text('annotation',
                                              'Document annotation text'));
$index->addDocument($doc);
```

Document links are not included in the generated document, but may be retrieved with the `Zend_Search_Lucene_Document_Html::getLinks()` and `Zend_Search_Lucene_Document_Html::getHeaderLinks()` methods:

```
$doc = Zend_Search_Lucene_Document_Html::loadHTML($htmlString);
$linksArray = $doc->getLinks();
$headerLinksArray = $doc->getHeaderLinks();
```

Starting from Zend Framework 1.6 it's also possible to exclude links with `rel` attribute set to `'nofollow'`. Use `Zend_Search_Lucene_Document_Html::setExcludeNoFollowLinks($true)` to turn on this option.

`Zend_Search_Lucene_Document_Html::getExcludeNoFollowLinks()` method returns current state of "Exclude nofollow links" flag.

1.5. Word 2007 documents

`Zend_Search_Lucene` offers a Word 2007 parsing feature. Documents can be created directly from a Word 2007 file:

```
$doc = Zend_Search_Lucene_Document_Docx::loadDocxFile($filename);
$index->addDocument($doc);
```

`Zend_Search_Lucene_Document_Docx` class uses the `ZipArchive` class and `simplexml` methods to parse the source document. If the `ZipArchive` class (from module php_zip) is not available, the `Zend_Search_Lucene_Document_Docx` will also not be available for use with Zend Framework.

`Zend_Search_Lucene_Document_Docx` class recognizes document meta data and document text. Meta data consists, depending on document contents, of filename, title, subject, creator, keywords, description, lastModifiedBy, revision, modified, created.

The 'filename' field is the actual Word 2007 file name.

The 'title' field is the actual document title.

The 'subject' field is the actual document subject.

The 'creator' field is the actual document creator.

The 'keywords' field contains the actual document keywords.

The 'description' field is the actual document description.

The 'lastModifiedBy' field is the username who has last modified the actual document.

The 'revision' field is the actual document revision number.

The 'modified' field is the actual document last modified date / time.

The 'created' field is the actual document creation date / time.

The 'body' field is the actual body content of the Word 2007 document. It only includes normal text, comments and revisions are not included.

The `loadDocxFile()` methods of `Zend_Search_Lucene_Document_Docx` class also have second optional argument. If it's set to `TRUE`, then body content is also stored within index and can be retrieved from the index. By default, the body is tokenized and indexed, but not stored.

Parsed documents may be augmented by the programmer with any other field:

```
$doc = Zend_Search_Lucene_Document_Docx::loadDocxFile($filename);
$doc->addField(Zend_Search_Lucene_Field::UnIndexed(
    'indexTime',
    time())
);
$doc->addField(Zend_Search_Lucene_Field::Text(
    'annotation',
    'Document annotation text')
);
$index->addDocument($doc);
```

1.6. Powerpoint 2007 documents

`Zend_Search_Lucene` offers a Powerpoint 2007 parsing feature. Documents can be created directly from a Powerpoint 2007 file:

```
$doc = Zend_Search_Lucene_Document_Pptx::loadPptxFile($filename);
$index->addDocument($doc);
```

`Zend_Search_Lucene_Document_Pptx` class uses the `ZipArchive` class and `simplexml` methods to parse the source document. If the `ZipArchive` class (from module php_zip) is not available, the `Zend_Search_Lucene_Document_Pptx` will also not be available for use with Zend Framework.

`Zend_Search_Lucene_Document_Pptx` class recognizes document meta data and document text. Meta data consists, depending on document contents, of filename, title, subject, creator, keywords, description, lastModifiedBy, revision, modified, created.

The 'filename' field is the actual Powerpoint 2007 file name.

The 'title' field is the actual document title.

The 'subject' field is the actual document subject.

The 'creator' field is the actual document creator.

The 'keywords' field contains the actual document keywords.

The 'description' field is the actual document description.

The 'lastModifiedBy' field is the username who has last modified the actual document.

The 'revision' field is the actual document revision number.

The 'modified' field is the actual document last modified date / time.

The 'created' field is the actual document creation date / time.

The 'body' field is the actual content of all slides and slide notes in the Powerpoint 2007 document.

The `loadPptxFile()` methods of `Zend_Search_Lucene_Document_Pptx` class also have second optional argument. If it's set to `TRUE`, then body content is also stored within index and can be retrieved from the index. By default, the body is tokenized and indexed, but not stored.

Parsed documents may be augmented by the programmer with any other field:

```
$doc = Zend_Search_Lucene_Document_Pptx::loadPptxFile($filename);
$doc->addField(Zend_Search_Lucene_Field::UnIndexed(
    'indexTime',
    time()));
$doc->addField(Zend_Search_Lucene_Field::Text(
    'annotation',
    'Document annotation text'));
$index->addDocument($doc);
```

1.7. Excel 2007 documents

`Zend_Search_Lucene` offers a Excel 2007 parsing feature. Documents can be created directly from a Excel 2007 file:

```
$doc = Zend_Search_Lucene_Document_Xlsx::loadXlsxFile($filename);
$index->addDocument($doc);
```

`Zend_Search_Lucene_Document_Xlsx` class uses the `ZipArchive` class and `simplexml` methods to parse the source document. If the `ZipArchive` class (from module php_zip) is not available, the `Zend_Search_Lucene_Document_Xlsx` will also not be available for use with Zend Framework.

`Zend_Search_Lucene_Document_Xlsx` class recognizes document meta data and document text. Meta data consists, depending on document contents, of filename, title, subject, creator, keywords, description, lastModifiedBy, revision, modified, created.

The 'filename' field is the actual Excel 2007 file name.

The 'title' field is the actual document title.

The 'subject' field is the actual document subject.

The 'creator' field is the actual document creator.

The 'keywords' field contains the actual document keywords.

The 'description' field is the actual document description.

The 'lastModifiedBy' field is the username who has last modified the actual document.

The 'revision' field is the actual document revision number.

The 'modified' field is the actual document last modified date / time.

The 'created' field is the actual document creation date / time.

The 'body' field is the actual content of all cells in all worksheets of the Excel 2007 document.

The `loadXlsxFile()` methods of `Zend_Search_Lucene_Document_Xlsx` class also have second optional argument. If it's set to `TRUE`, then body content is also stored within index and can be retrieved from the index. By default, the body is tokenized and indexed, but not stored.

Parsed documents may be augmented by the programmer with any other field:

```
$doc = Zend_Search_Lucene_Document_Xlsx::loadXlsxFile($filename);
$doc->addField(Zend_Search_Lucene_Field::UnIndexed(
    'indexTime',
    time()));
$doc->addField(Zend_Search_Lucene_Field::Text(
    'annotation',
    'Document annotation text'));
$index->addDocument($doc);
```

2. Building Indexes

2.1. Creating a New Index

Index creation and updating capabilities are implemented within the `Zend_Search_Lucene` component, as well as the Java Lucene project. You can use either of these options to create indexes that `Zend_Search_Lucene` can search.

The PHP code listing below provides an example of how to index a file using `Zend_Search_Lucene` indexing API:

```
// Create index
$index = Zend_Search_Lucene::create('/data/my-index');

$doc = new Zend_Search_Lucene_Document();

// Store document URL to identify it in the search results
$doc->addField(Zend_Search_Lucene_Field::Text('url', $docUrl));

// Index document contents
$doc->addField(Zend_Search_Lucene_Field::UnStored('contents', $docContent));

// Add document to the index
$index->addDocument($doc);
```

Newly added documents are immediately searchable in the index.

2.2. Updating Index

The same procedure is used to update an existing index. The only difference is that the open() method is called instead of the create() method:

```
// Open existing index
$index = Zend_Search_Lucene::open('/data/my-index');

$doc = new Zend_Search_Lucene_Document();
// Store document URL to identify it in search result.
$doc->addField(Zend_Search_Lucene_Field::Text('url', $docUrl));
// Index document content
$doc->addField(Zend_Search_Lucene_Field::UnStored('contents',
                                                  $docContent));

// Add document to the index.
$index->addDocument($doc);
```

2.3. Updating Documents

The Lucene index file format doesn't support document updating. Documents should be removed and re-added to the index to effectively update them.

`Zend_Search_Lucene::delete()` method operates with an internal index document id. It can be retrieved from a query hit by 'id' property:

```
$removePath = ...;
$hits = $index->find('path:' . $removePath);
foreach ($hits as $hit) {
    $index->delete($hit->id);
}
```

2.4. Retrieving Index Size

There are two methods to retrieve the size of an index in `Zend_Search_Lucene`.

`Zend_Search_Lucene::maxDoc()` returns one greater than the largest possible document number. It's actually the overall number of the documents in the index including deleted documents, so it has a synonym: `Zend_Search_Lucene::count()`.

`Zend_Search_Lucene::numDocs()` returns the total number of non-deleted documents.

```
$indexSize = $index->count();
$documents = $index->numDocs();
```

`Zend_Search_Lucene::isDeleted($id)` method may be used to check if a document is deleted.

```
for ($count = 0; $count < $index->maxDoc(); $count++) {
    if ($index->isDeleted($count)) {
        echo "Document #$id is deleted.\n";
    }
}
```

Index optimization removes deleted documents and squeezes documents' IDs in to a smaller range. A document's internal id may therefore change during index optimization.

2.5. Index optimization

A Lucene index consists of many segments. Each segment is a completely independent set of data.

Lucene index segment files can't be updated by design. A segment update needs full segment reorganization. See Lucene index file formats for details (http://lucene.apache.org/java/2_3_0/fileformats.html) [1]. New documents are added to the index by creating new segment.

Increasing number of segments reduces quality of the index, but index optimization restores it. Optimization essentially merges several segments into a new one. This process also doesn't update segments. It generates one new large segment and updates segment list ('segments' file).

Full index optimization can be trigger by calling the `Zend_Search_Lucene::optimize()` method. It merges all index segments into one new segment:

```
// Open existing index
$index = Zend_Search_Lucene::open('/data/my-index');

// Optimize index.
$index->optimize();
```

Automatic index optimization is performed to keep indexes in a consistent state.

Automatic optimization is an iterative process managed by several index options. It merges very small segments into larger ones, then merges these larger segments into even larger segments and so on.

2.5.1. MaxBufferedDocs auto-optimization option

MaxBufferedDocs is a minimal number of documents required before the buffered in-memory documents are written into a new segment.

MaxBufferedDocs can be retrieved or set by `$index->getMaxBufferedDocs()` or `$index->setMaxBufferedDocs($maxBufferedDocs)` calls.

Default value is 10.

2.5.2. MaxMergeDocs auto-optimization option

MaxMergeDocs is a largest number of documents ever merged by addDocument(). Small values (e.g., less than 10.000) are best for interactive indexing, as this limits the length of pauses while indexing to a few seconds. Larger values are best for batched indexing and speedier searches.

MaxMergeDocs can be retrieved or set by `$index->getMaxMergeDocs()` or `$index->setMaxMergeDocs($maxMergeDocs)` calls.

Default value is PHP_INT_MAX.

2.5.3. MergeFactor auto-optimization option

MergeFactor determines how often segment indices are merged by addDocument(). With smaller values, less RAM is used while indexing, and searches on unoptimized indices are faster, but indexing speed is slower. With larger values, more RAM is used during indexing, and while

[1] The currently supported Lucene index file format is version 2.3 (starting from Zend Framework 1.6).

searches on unoptimized indices are slower, indexing is faster. Thus larger values (> 10) are best for batch index creation, and smaller values (< 10) for indices that are interactively maintained.

MergeFactor is a good estimation for average number of segments merged by one auto-optimization pass. Too large values produce large number of segments while they are not merged into new one. It may be a cause of "failed to open stream: Too many open files" error message. This limitation is system dependent.

MergeFactor can be retrieved or set by `$index->getMergeFactor()` or `$index->setMergeFactor($mergeFactor)` calls.

Default value is 10.

Lucene Java and Luke (Lucene Index Toolbox - http://www.getopt.org/luke/) can also be used to optimize an index. Latest Luke release (v0.8) is based on Lucene v2.3 and compatible with current implementation of `Zend_Search_Lucene` component (Zend Framework 1.6). Earlier versions of `Zend_Search_Lucene` implementations need another versions of Java Lucene tools to be compatible:

- Zend Framework 1.5 - Java Lucene 2.1 (Luke tool v0.7.1 - http://www.getopt.org/luke/luke-0.7.1/)
- Zend Framework 1.0 - Java Lucene 1.4 - 2.1 (Luke tool v0.6 - http://www.getopt.org/luke/luke-0.6/)

2.6. Permissions

By default, index files are available for reading and writing by everyone.

It's possible to override this with the `Zend_Search_Lucene_Storage_Directory_Filesystem::setDefaultFilePermissions()` method:

```
// Get current default file permissions
$currentPermissions =
    Zend_Search_Lucene_Storage_Directory_Filesystem::getDefaultFilePermissions();

// Give read-writing permissions only for current user and group
Zend_Search_Lucene_Storage_Directory_Filesystem::setDefaultFilePermissions(0660);
```

2.7. Limitations

2.7.1. Index size

Index size is limited by 2GB for 32-bit platforms.

Use 64-bit platforms for larger indices.

2.7.2. Supported Filesystems

`Zend_Search_Lucene` uses `flock()` to provide concurrent searching, index updating and optimization.

According to the PHP documentation, "`flock()` will not work on NFS and many other networked file systems".

Do not use networked file systems with Zend_Search_Lucene.

3. Searching an Index

3.1. Building Queries

There are two ways to search the index. The first method uses query parser to construct a query from a string. The second is to programmatically create your own queries through the Zend_Search_Lucene API.

Before choosing to use the provided query parser, please consider the following:

1. If you are programmatically creating a query string and then parsing it with the query parser then you should consider building your queries directly with the query API. Generally speaking, the query parser is designed for human-entered text, not for program-generated text.

2. Untokenized fields are best added directly to queries and not through the query parser. If a field's values are generated programmatically by the application, then the query clauses for this field should also be constructed programmatically. An analyzer, which the query parser uses, is designed to convert human-entered text to terms. Program-generated values, like dates, keywords, etc., should be added with the query API.

3. In a query form, fields that are general text should use the query parser. All others, such as date ranges, keywords, etc., are better added directly through the query API. A field with a limited set of values that can be specified with a pull-down menu should not be added to a query string that is subsequently parsed but instead should be added as a TermQuery clause.

4. Boolean queries allow the programmer to logically combine two or more queries into new one. Thus it's the best way to add additional criteria to a search defined by a query string.

Both ways use the same API method to search through the index:

```
$index = Zend_Search_Lucene::open('/data/my_index');

$index->find($query);
```

The Zend_Search_Lucene::find() method determines the input type automatically and uses the query parser to construct an appropriate Zend_Search_Lucene_Search_Query object from an input of type string.

It is important to note that the query parser uses the standard analyzer to tokenize separate parts of query string. Thus all transformations which are applied to indexed text are also applied to query strings.

The standard analyzer may transform the query string to lower case for case-insensitivity, remove stop-words, and stem among other transformations.

The API method doesn't transform or filter input terms in any way. It's therefore more suitable for computer generated or untokenized fields.

3.1.1. Query Parsing

Zend_Search_Lucene_Search_QueryParser::parse() method may be used to parse query strings into query objects.

This query object may be used in query construction API methods to combine user entered queries with programmatically generated queries.

Actually, in some cases it's the only way to search for values within untokenized fields:

```
$userQuery = Zend_Search_Lucene_Search_QueryParser::parse($queryStr);

$pathTerm  = new Zend_Search_Lucene_Index_Term(
                    '/data/doc_dir/' . $filename, 'path'
                );
$pathQuery = new Zend_Search_Lucene_Search_Query_Term($pathTerm);

$query = new Zend_Search_Lucene_Search_Query_Boolean();
$query->addSubquery($userQuery, true /* required */);
$query->addSubquery($pathQuery, true /* required */);

$hits = $index->find($query);
```

`Zend_Search_Lucene_Search_QueryParser::parse()` method also takes an optional encoding parameter, which can specify query string encoding:

```
$userQuery = Zend_Search_Lucene_Search_QueryParser::parse($queryStr,
                                                          'iso-8859-5');
```

If the encoding parameter is omitted, then current locale is used.

It's also possible to specify the default query string encoding with `Zend_Search_Lucene_Search_QueryParser::setDefaultEncoding()` method:

```
Zend_Search_Lucene_Search_QueryParser::setDefaultEncoding('iso-8859-5');
...
$userQuery = Zend_Search_Lucene_Search_QueryParser::parse($queryStr);
```

`Zend_Search_Lucene_Search_QueryParser::getDefaultEncoding()` returns the current default query string encoding (the empty string means "current locale").

3.2. Search Results

The search result is an array of `Zend_Search_Lucene_Search_QueryHit` objects. Each of these has two properties: `$hit->id` is a document number within the index and `$hit->score` is a score of the hit in a search result. The results are ordered by score (descending from highest score).

The `Zend_Search_Lucene_Search_QueryHit` object also exposes each field of the `Zend_Search_Lucene_Document` found in the search as a property of the hit. In the following example, a hit is returned with two fields from the corresponding document: title and author.

```
$index = Zend_Search_Lucene::open('/data/my_index');

$hits = $index->find($query);

foreach ($hits as $hit) {
    echo $hit->score;
    echo $hit->title;
    echo $hit->author;
```

```
}
```

Stored fields are always returned in UTF-8 encoding.

Optionally, the original `Zend_Search_Lucene_Document` object can be returned from the `Zend_Search_Lucene_Search_QueryHit`. You can retrieve stored parts of the document by using the `getDocument()` method of the index object and then get them by `getFieldValue()` method:

```
$index = Zend_Search_Lucene::open('/data/my_index');

$hits = $index->find($query);
foreach ($hits as $hit) {
    // return Zend_Search_Lucene_Document object for this hit
    echo $document = $hit->getDocument();

    // return a Zend_Search_Lucene_Field object
    // from the Zend_Search_Lucene_Document
    echo $document->getField('title');

    // return the string value of the Zend_Search_Lucene_Field object
    echo $document->getFieldValue('title');

    // same as getFieldValue()
    echo $document->title;
}
```

The fields available from the `Zend_Search_Lucene_Document` object are determined at the time of indexing. The document fields are either indexed, or index and stored, in the document by the indexing application (e.g. LuceneIndexCreation.jar).

Note that the document identity ('path' in our example) is also stored in the index and must be retrieved from it.

3.3. Limiting the Result Set

The most computationally expensive part of searching is score calculation. It may take several seconds for large result sets (tens of thousands of hits).

`Zend_Search_Lucene` gives the possibility to limit result set size with `getResultSetLimit()` and `setResultSetLimit()` methods:

```
$currentResultSetLimit = Zend_Search_Lucene::getResultSetLimit();

Zend_Search_Lucene::setResultSetLimit($newLimit);
```

The default value of 0 means 'no limit'.

It doesn't give the 'best N' results, but only the 'first N' [2].

3.4. Results Scoring

`Zend_Search_Lucene` uses the same scoring algorithms as Java Lucene. All hits in the search result are ordered by score by default. Hits with greater score come first, and documents having higher scores should match the query more precisely than documents having lower scores.

[2] Returned hits are still ordered by score or by the specified order, if given.

Roughly speaking, search hits that contain the searched term or phrase more frequently will have a higher score.

A hit's score can be retrieved by accessing the `score` property of the hit:

```
$hits = $index->find($query);

foreach ($hits as $hit) {
    echo $hit->id;
    echo $hit->score;
}
```

The `Zend_Search_Lucene_Search_Similarity` class is used to calculate the score for each hit. See Extensibility. Scoring Algorithms section for details.

3.5. Search Result Sorting

By default, the search results are ordered by score. The programmer can change this behavior by setting a sort field (or a list of fields), sort type and sort order parameters.

`$index->find()` call may take several optional parameters:

```
$index->find($query [, $sortField [, $sortType [, $sortOrder]]]
                    [, $sortField2 [, $sortType [, $sortOrder]]]
             ...);
```

A name of stored field by which to sort result should be passed as the `$sortField` parameter.

`$sortType` may be omitted or take the following enumerated values: `SORT_REGULAR` (compare items normally- default value), `SORT_NUMERIC` (compare items numerically), `SORT_STRING` (compare items as strings).

`$sortOrder` may be omitted or take the following enumerated values: `SORT_ASC` (sort in ascending order- default value), `SORT_DESC` (sort in descending order).

Examples:

```
$index->find($query, 'quantity', SORT_NUMERIC, SORT_DESC);
```

```
$index->find($query, 'fname', SORT_STRING, 'lname', SORT_STRING);
```

```
$index->find($query, 'name', SORT_STRING, 'quantity', SORT_NUMERIC, SORT_DESC);
```

Please use caution when using a non-default search order; the query needs to retrieve documents completely from an index, which may dramatically reduce search performance.

3.6. Search Results Highlighting

`Zend_Search_Lucene` provides two options for search results highlighting.

The first one is utilizing `Zend_Search_Lucene_Document_Html` class (see HTML documents section for details) using the following methods:

```
/**
```

```
     * Highlight text with specified color
     *
     * @param string|array $words
     * @param string $colour
     * @return string
     */
    public function highlight($words, $colour = '#66ffff');
```

```
    /**
     * Highlight text using specified View helper or callback function.
     *
     * @param string|array $words  Words to highlight. Words could be organized
                                   using the array or string.
     * @param callback $callback   Callback method, used to transform
                                   (highlighting) text.
     * @param array    $params     Array of additionall callback parameters passed
                                   through into it (first non-optional parameter
                                   is an HTML fragment for highlighting)
     * @return string
     * @throws Zend_Search_Lucene_Exception
     */
    public function highlightExtended($words, $callback, $params = array())
```

To customize highlighting behavior use `highlightExtended()` method with specified callback, which takes one or more parameters[3], or extend `Zend_Search_Lucene_Document_Html` class and redefine `applyColour($stringToHighlight, $colour)` method used as a default highlighting callback. [4]

View helpers also can be used as callbacks in context of view script:

```
$doc->highlightExtended('word1 word2 word3...', array($this, 'myViewHelper'));
```

The result of highlighting operation is retrieved by `Zend_Search_Lucene_Document_Html->getHTML()` method.

Highlighting is performed in terms of current analyzer. So all forms of the word(s) recognized by analyzer are highlighted.

E.g. if current analyzer is case insensitive and we request to highlight 'text' word, then 'text', 'Text', 'TEXT' and other case combinations will be highlighted.

In the same way, if current analyzer supports stemming and we request to highlight 'indexed', then 'index', 'indexing', 'indices' and other word forms will be highlighted.

On the other hand, if word is skipped by current analyzer (e.g. if short words filter is applied to the analyzer), then nothing will be highlighted.

The second option is to use `Zend_Search_Lucene_Search_Query->highlightMatches(string $inputHTML[, $defaultEncoding = 'UTF-8'[, Zend_Search_Lucene_Search_Highlighter_Interface $highlighter]])` method:

[3] The first is an HTML fragment for highlighting and others are callback behavior dependent. Returned value is a highlighted HTML fragment.
[4] In both cases returned HTML is automatically transformed into valid XHTML.

```
$query = Zend_Search_Lucene_Search_QueryParser::parse($queryStr);
$highlightedHTML = $query->highlightMatches($sourceHTML);
```

Optional second parameter is a default HTML document encoding. It's used if encoding is not specified using Content-type HTTP-EQUIV meta tag.

Optional third parameter is a highlighter object which has to implement `Zend_Search_Lucene_Search_Highlighter_Interface` interface:

```
interface Zend_Search_Lucene_Search_Highlighter_Interface
{
    /**
     * Set document for highlighting.
     *
     * @param Zend_Search_Lucene_Document_Html $document
     */
    public function setDocument(Zend_Search_Lucene_Document_Html $document);

    /**
     * Get document for highlighting.
     *
     * @return Zend_Search_Lucene_Document_Html $document
     */
    public function getDocument();

    /**
     * Highlight specified words (method is invoked once per subquery)
     *
     * @param string|array $words  Words to highlight. They could be
                                   organized using the array or string.
     */
    public function highlight($words);
}
```

Where `Zend_Search_Lucene_Document_Html` object is an object constructed from the source HTML provided to the `Zend_Search_Lucene_Search_Query->highlightMatches()` method.

If `$highlighter` parameter is omitted, then `Zend_Search_Lucene_Search_Highlighter_Default` object is instantiated and used.

Highlighter `highlight()` method is invoked once per subquery, so it has an ability to differentiate highlighting for them.

Actually, default highlighter does this walking through predefined color table. So you can implement your own highlighter or just extend the default and redefine color table.

`Zend_Search_Lucene_Search_Query->htmlFragmentHighlightMatches()` has similar behavior. The only difference is that it takes as an input and returns HTML fragment without <>HTML>, <HEAD>, <BODY> tags. Nevertheless, fragment is automatically transformed to valid XHTML.

4. Query Language

Java Lucene and `Zend_Search_Lucene` provide quite powerful query languages.

These languages are mostly the same with some minor differences, which are mentioned below.

Full Java Lucene query language syntax documentation can be found here.

4.1. Terms

A query is broken up into terms and operators. There are three types of terms: Single Terms, Phrases, and Subqueries.

A Single Term is a single word such as "test" or "hello".

A Phrase is a group of words surrounded by double quotes such as "hello dolly".

A Subquery is a query surrounded by parentheses such as "(hello dolly)".

Multiple terms can be combined together with boolean operators to form complex queries (see below).

4.2. Fields

Lucene supports fields of data. When performing a search you can either specify a field, or use the default field. The field names depend on indexed data and default field is defined by current settings.

The first and most significant difference from Java Lucene is that terms are searched through *all fields* by default.

There are two static methods in the `Zend_Search_Lucene` class which allow the developer to configure these settings:

```
$defaultSearchField = Zend_Search_Lucene::getDefaultSearchField();
...
Zend_Search_Lucene::setDefaultSearchField('contents');
```

The `NULL` value indicated that the search is performed across all fields. It's the default setting.

You can search specific fields by typing the field name followed by a colon ":" followed by the term you are looking for.

As an example, let's assume a Lucene index contains two fields- title and text- with text as the default field. If you want to find the document entitled "The Right Way" which contains the text "don't go this way", you can enter:

```
title:"The Right Way" AND text:go
```

or

```
title:"Do it right" AND go
```

Because "text" is the default field, the field indicator is not required.

Note: The field is only valid for the term, phrase or subquery that it directly precedes, so the query

```
title:Do it right
```

Will only find "Do" in the title field. It will find "it" and "right" in the default field (if the default field is set) or in all indexed fields (if the default field is set to `NULL`).

4.3. Wildcards

Lucene supports single and multiple character wildcard searches within single terms (but not within phrase queries).

To perform a single character wildcard search use the "?" symbol.

To perform a multiple character wildcard search use the "*" symbol.

The single character wildcard search looks for string that match the term with the "?" replaced by any single character. For example, to search for "text" or "test" you can use the search:

```
te?t
```

Multiple character wildcard searches look for 0 or more characters when matching strings against terms. For example, to search for test, tests or tester, you can use the search:

```
test*
```

You can use "?", "*" or both at any place of the term:

```
*wr?t*
```

It searches for "write", "wrote", "written", "rewrite", "rewrote" and so on.

Starting from ZF 1.7.7 wildcard patterns need some non-wildcard prefix. Default prefix length is 3 (like in Java Lucene). So "*", "te?t", "*wr?t*" terms will cause an exception[5].

It can be altered using `Zend_Search_Lucene_Search_Query_Wildcard::getMinPrefixLength()` and `Zend_Search_Lucene_Search_Query_Wildcard::setMinPrefixLength()` methods.

4.4. Term Modifiers

Lucene supports modifying query terms to provide a wide range of searching options.

"~" modifier can be used to specify proximity search for phrases or fuzzy search for individual terms.

4.5. Range Searches

Range queries allow the developer or user to match documents whose field(s) values are between the lower and upper bound specified by the range query. Range Queries can be inclusive or exclusive of the upper and lower bounds. Sorting is performed lexicographically.

```
mod_date:[20020101 TO 20030101]
```

This will find documents whose mod_date fields have values between 20020101 and 20030101, inclusive. Note that Range Queries are not reserved for date fields. You could also use range queries with non-date fields:

```
title:{Aida TO Carmen}
```

[5] Please note, that it's not a `Zend_Search_Lucene_Search_QueryParserException`, but a `Zend_Search_Lucene_Exception`. It's thrown during query rewrite (execution) operation.

This will find all documents whose titles would be sorted between Aida and Carmen, but not including Aida and Carmen.

Inclusive range queries are denoted by square brackets. Exclusive range queries are denoted by curly brackets.

If field is not specified then `Zend_Search_Lucene` searches for specified interval through all fields by default.

```
{Aida TO Carmen}
```

4.6. Fuzzy Searches

`Zend_Search_Lucene` as well as Java Lucene supports fuzzy searches based on the Levenshtein Distance, or Edit Distance algorithm. To do a fuzzy search use the tilde, "~", symbol at the end of a Single word Term. For example to search for a term similar in spelling to "roam" use the fuzzy search:

```
roam~
```

This search will find terms like foam and roams. Additional (optional) parameter can specify the required similarity. The value is between 0 and 1, with a value closer to 1 only terms with a higher similarity will be matched. For example:

```
roam~0.8
```

The default that is used if the parameter is not given is 0.5.

4.7. Matched terms limitation

Wildcard, range and fuzzy search queries may match too many terms. It may cause incredible search performance downgrade.

So Zend_Search_Lucene sets a limit of matching terms per query (subquery). This limit can be retrieved and set using `Zend_Search_Lucene::getTermsPerQueryLimit()`/ `Zend_Search_Lucene::setTermsPerQueryLimit($limit)` methods.

Default matched terms per query limit is 1024.

4.8. Proximity Searches

Lucene supports finding words from a phrase that are within a specified word distance in a string. To do a proximity search use the tilde, "~", symbol at the end of the phrase. For example to search for a "Zend" and "Framework" within 10 words of each other in a document use the search:

```
"Zend Framework"~10
```

4.9. Boosting a Term

Java Lucene and `Zend_Search_Lucene` provide the relevance level of matching documents based on the terms found. To boost the relevance of a term use the caret, "^", symbol with a boost factor (a number) at the end of the term you are searching. The higher the boost factor, the more relevant the term will be.

Boosting allows you to control the relevance of a document by boosting individual terms. For example, if you are searching for

```
PHP framework
```

and you want the term "PHP" to be more relevant boost it using the ^ symbol along with the boost factor next to the term. You would type:

```
PHP^4 framework
```

This will make documents with the term PHP appear more relevant. You can also boost phrase terms and subqueries as in the example:

```
"PHP framework"^4 "Zend Framework"
```

By default, the boost factor is 1. Although the boost factor must be positive, it may be less than 1 (e.g. 0.2).

4.10. Boolean Operators

Boolean operators allow terms to be combined through logic operators. Lucene supports AND, "+", OR, NOT and "-" as Boolean operators. Java Lucene requires boolean operators to be ALL CAPS. `Zend_Search_Lucene` does not.

AND, OR, and NOT operators and "+", "-" defines two different styles to construct boolean queries. Unlike Java Lucene, `Zend_Search_Lucene` doesn't allow these two styles to be mixed.

If the AND/OR/NOT style is used, then an AND or OR operator must be present between all query terms. Each term may also be preceded by NOT operator. The AND operator has higher precedence than the OR operator. This differs from Java Lucene behavior.

4.10.1. AND

The AND operator means that all terms in the "AND group" must match some part of the searched field(s).

To search for documents that contain "PHP framework" and "Zend Framework" use the query:

```
"PHP framework" AND "Zend Framework"
```

4.10.2. OR

The OR operator divides the query into several optional terms.

To search for documents that contain "PHP framework" or "Zend Framework" use the query:

```
"PHP framework" OR "Zend Framework"
```

4.10.3. NOT

The NOT operator excludes documents that contain the term after NOT. But an "AND group" which contains only terms with the NOT operator gives an empty result set instead of a full set of indexed documents.

To search for documents that contain "PHP framework" but not "Zend Framework" use the query:

```
"PHP framework" AND NOT "Zend Framework"
```

4.10.4. &&, ||, and ! operators

&&, ||, and ! may be used instead of AND, OR, and NOT notation.

4.10.5. +

The "+" or required operator stipulates that the term after the "+" symbol must match the document.

To search for documents that must contain "Zend" and may contain "Framework" use the query:

```
+Zend Framework
```

4.10.6. -

The "-" or prohibit operator excludes documents that match the term after the "-" symbol.

To search for documents that contain "PHP framework" but not "Zend Framework" use the query:

```
"PHP framework" -"Zend Framework"
```

4.10.7. No Operator

If no operator is used, then the search behavior is defined by the "default boolean operator".

This is set to `OR` by default.

That implies each term is optional by default. It may or may not be present within document, but documents with this term will receive a higher score.

To search for documents that requires "PHP framework" and may contain "Zend Framework" use the query:

```
+"PHP framework" "Zend Framework"
```

The default boolean operator may be set or retrieved with the `Zend_Search_Lucene_Search_QueryParser::setDefaultOperator($operator)` and `Zend_Search_Lucene_Search_QueryParser::getDefaultOperator()` methods, respectively.

These methods operate with the `Zend_Search_Lucene_Search_QueryParser::B_AND` and `Zend_Search_Lucene_Search_QueryParser::B_OR` constants.

4.11. Grouping

Java Lucene and `Zend_Search_Lucene` support using parentheses to group clauses to form sub queries. This can be useful if you want to control the precedence of boolean logic operators for a query or mix different boolean query styles:

```
+(framework OR library) +php
```

`Zend_Search_Lucene` supports subqueries nested to any level.

4.12. Field Grouping

Lucene also supports using parentheses to group multiple clauses to a single field.

To search for a title that contains both the word "return" and the phrase "pink panther" use the query:

```
title:(+return +"pink panther")
```

4.13. Escaping Special Characters

Lucene supports escaping special characters that are used in query syntax. The current list of special characters is:

+ - && || ! ( ) { } [ ] ^ " ~ * ? : \

+ and - inside single terms are automatically treated as common characters.

For other instances of these characters use the \ before each special character you'd like to escape. For example to search for (1+1):2 use the query:

```
\(1\+1\)\:2
```

5. Query Construction API

In addition to parsing a string query automatically it's also possible to construct them with the query API.

User queries can be combined with queries created through the query API. Simply use the query parser to construct a query from a string:

```
$query = Zend_Search_Lucene_Search_QueryParser::parse($queryString);
```

5.1. Query Parser Exceptions

The query parser may generate two types of exceptions:

- `Zend_Search_Lucene_Exception` is thrown if something goes wrong in the query parser itself.
- `Zend_Search_Lucene_Search_QueryParserException` is thrown when there is an error in the query syntax.

It's a good idea to catch `Zend_Search_Lucene_Search_QueryParserExceptions` and handle them appropriately:

```
try {
    $query = Zend_Search_Lucene_Search_QueryParser::parse($queryString);
} catch (Zend_Search_Lucene_Search_QueryParserException $e) {
    echo "Query syntax error: " . $e->getMessage() . "\n";
}
```

The same technique should be used for the find() method of a `Zend_Search_Lucene` object.

Starting in 1.5, query parsing exceptions are suppressed by default. If query doesn't conform query language, then it's tokenized using current default analyzer and all tokenized terms are used for searching. Use `Zend_Search_Lucene_Search_QueryParser::dontSuppressQueryParsingExceptions()` method to turn exceptions on. `Zend_Search_Lucene_Search_QueryParser::suppressQueryParsingExceptions()` and `Zend_Search_Lucene_Search_QueryParser::queryParsingExceptionsSuppressed()` methods are also intended to manage exceptions handling behavior.

5.2. Term Query

Term queries can be used for searching with a single term.

Query string:

```
word1
```

or

Query construction by API:

```
$term  = new Zend_Search_Lucene_Index_Term('word1', 'field1');
$query = new Zend_Search_Lucene_Search_Query_Term($term);
$hits  = $index->find($query);
```

The term field is optional. `Zend_Search_Lucene` searches through all indexed fields in each document if the field is not specified:

```
// Search for 'word1' in all indexed fields
$term  = new Zend_Search_Lucene_Index_Term('word1');
$query = new Zend_Search_Lucene_Search_Query_Term($term);
$hits  = $index->find($query);
```

5.3. Multi-Term Query

Multi-term queries can be used for searching with a set of terms.

Each term in a set can be defined as *required*, *prohibited*, or *neither*.

- *required* means that documents not matching this term will not match the query;
- *prohibited* means that documents matching this term will not match the query;
- *neither*, in which case matched documents are neither prohibited from, nor required to, match the term. A document must match at least 1 term, however, to match the query.

If optional terms are added to a query with required terms, both queries will have the same result set but the optional terms may affect the score of the matched documents.

Both search methods can be used for multi-term queries.

Query string:

```
+word1 author:word2 -word3
```

- '+' is used to define a required term.
- '-' is used to define a prohibited term.
- 'field:' prefix is used to indicate a document field for a search. If it's omitted, then all fields are searched.

or

Query construction by API:

```
$query = new Zend_Search_Lucene_Search_Query_MultiTerm();

$query->addTerm(new Zend_Search_Lucene_Index_Term('word1'), true);
$query->addTerm(new Zend_Search_Lucene_Index_Term('word2', 'author'),
                null);
$query->addTerm(new Zend_Search_Lucene_Index_Term('word3'), false);

$hits  = $index->find($query);
```

It's also possible to specify terms list within MultiTerm query constructor:

```
$terms = array(new Zend_Search_Lucene_Index_Term('word1'),
               new Zend_Search_Lucene_Index_Term('word2', 'author'),
               new Zend_Search_Lucene_Index_Term('word3'));
$signs = array(true, null, false);

$query = new Zend_Search_Lucene_Search_Query_MultiTerm($terms, $signs);

$hits  = $index->find($query);
```

The `$signs` array contains information about the term type:

- `TRUE` is used to define required term.
- `FALSE` is used to define prohibited term.
- `NULL` is used to define a term that is neither required nor prohibited.

5.4. Boolean Query

Boolean queries allow to construct query using other queries and boolean operators.

Each subquery in a set can be defined as *required*, *prohibited*, or *optional*.

- *required* means that documents not matching this subquery will not match the query;
- *prohibited* means that documents matching this subquery will not match the query;
- *optional*, in which case matched documents are neither prohibited from, nor required to, match the subquery. A document must match at least 1 subquery, however, to match the query.

If optional subqueries are added to a query with required subqueries, both queries will have the same result set but the optional subqueries may affect the score of the matched documents.

Both search methods can be used for boolean queries.

Query string:

```
+(word1 word2 word3) (author:word4 author:word5) -(word6)
```

- '+' is used to define a required subquery.
- '-' is used to define a prohibited subquery.
- 'field:' prefix is used to indicate a document field for a search. If it's omitted, then all fields are searched.

or

Query construction by API:

```
$query = new Zend_Search_Lucene_Search_Query_Boolean();

$subquery1 = new Zend_Search_Lucene_Search_Query_MultiTerm();
$subquery1->addTerm(new Zend_Search_Lucene_Index_Term('word1'));
$subquery1->addTerm(new Zend_Search_Lucene_Index_Term('word2'));
$subquery1->addTerm(new Zend_Search_Lucene_Index_Term('word3'));

$subquery2 = new Zend_Search_Lucene_Search_Query_MultiTerm();
$subquery2->addTerm(new Zend_Search_Lucene_Index_Term('word4', 'author'));
$subquery2->addTerm(new Zend_Search_Lucene_Index_Term('word5', 'author'));

$term6 = new Zend_Search_Lucene_Index_Term('word6');
$subquery3 = new Zend_Search_Lucene_Search_Query_Term($term6);

$query->addSubquery($subquery1, true  /* required */);
$query->addSubquery($subquery2, null  /* optional */);
$query->addSubquery($subquery3, false /* prohibited */);

$hits  = $index->find($query);
```

It's also possible to specify subqueries list within Boolean query constructor:

```
...
$subqueries = array($subquery1, $subquery2, $subquery3);
$signs = array(true, null, false);

$query = new Zend_Search_Lucene_Search_Query_Boolean($subqueries, $signs);

$hits  = $index->find($query);
```

The `$signs` array contains information about the subquery type:

- `TRUE` is used to define required subquery.
- `FALSE` is used to define prohibited subquery.
- `NULL` is used to define a subquery that is neither required nor prohibited.

Each query which uses boolean operators can be rewritten using signs notation and constructed using API. For example:

```
word1 AND (word2 AND word3 AND NOT word4) OR word5
```

is equivalent to

```
(+(word1) +(+word2 +word3 -word4)) (word5)
```

5.5. Wildcard Query

Wildcard queries can be used to search for documents containing strings matching specified patterns.

The '?' symbol is used as a single character wildcard.

The '*' symbol is used as a multiple character wildcard.

Query string:

```
field1:test*
```

or

Query construction by API:

```
$pattern = new Zend_Search_Lucene_Index_Term('test*', 'field1');
$query = new Zend_Search_Lucene_Search_Query_Wildcard($pattern);
$hits  = $index->find($query);
```

The term field is optional. Zend_Search_Lucene searches through all fields on each document if a field is not specified:

```
$pattern = new Zend_Search_Lucene_Index_Term('test*');
$query = new Zend_Search_Lucene_Search_Query_Wildcard($pattern);
$hits  = $index->find($query);
```

5.6. Fuzzy Query

Fuzzy queries can be used to search for documents containing strings matching terms similar to specified term.

Query string:

```
field1:test~
```

This query matches documents containing 'test' 'text' 'best' words and others.

or

Query construction by API:

```
$term = new Zend_Search_Lucene_Index_Term('test', 'field1');
$query = new Zend_Search_Lucene_Search_Query_Fuzzy($term);
$hits  = $index->find($query);
```

Optional similarity can be specified after "~" sign.

Query string:

```
field1:test~0.4
```

or

Query construction by API:

```
$term = new Zend_Search_Lucene_Index_Term('test', 'field1');
$query = new Zend_Search_Lucene_Search_Query_Fuzzy($term, 0.4);
$hits  = $index->find($query);
```

The term field is optional. `Zend_Search_Lucene` searches through all fields on each document if a field is not specified:

```
$term = new Zend_Search_Lucene_Index_Term('test');
$query = new Zend_Search_Lucene_Search_Query_Fuzzy($term);
$hits  = $index->find($query);
```

5.7. Phrase Query

Phrase Queries can be used to search for a phrase within documents.

Phrase Queries are very flexible and allow the user or developer to search for exact phrases as well as 'sloppy' phrases.

Phrases can also contain gaps or terms in the same places; they can be generated by the analyzer for different purposes. For example, a term can be duplicated to increase the term its weight, or several synonyms can be placed into a single position.

```
$query1 = new Zend_Search_Lucene_Search_Query_Phrase();

// Add 'word1' at 0 relative position.
$query1->addTerm(new Zend_Search_Lucene_Index_Term('word1'));

// Add 'word2' at 1 relative position.
$query1->addTerm(new Zend_Search_Lucene_Index_Term('word2'));

// Add 'word3' at 3 relative position.
$query1->addTerm(new Zend_Search_Lucene_Index_Term('word3'), 3);

...

$query2 = new Zend_Search_Lucene_Search_Query_Phrase(
                array('word1', 'word2', 'word3'), array(0,1,3));

...

// Query without a gap.
$query3 = new Zend_Search_Lucene_Search_Query_Phrase(
                array('word1', 'word2', 'word3'));

...

$query4 = new Zend_Search_Lucene_Search_Query_Phrase(
                array('word1', 'word2'), array(0,1), 'annotation');
```

A phrase query can be constructed in one step with a class constructor or step by step with `Zend_Search_Lucene_Search_Query_Phrase::addTerm()` method calls.

`Zend_Search_Lucene_Search_Query_Phrase` class constructor takes three optional arguments:

```
Zend_Search_Lucene_Search_Query_Phrase(
    [array $terms[, array $offsets[, string $field]]]
);
```

The `$terms` parameter is an array of strings that contains a set of phrase terms. If it's omitted or equal to `NULL`, then an empty query is constructed.

The `$offsets` parameter is an array of integers that contains offsets of terms in a phrase. If it's omitted or equal to `NULL`, then the terms' positions are assumed to be sequential with no gaps.

The `$field` parameter is a string that indicates the document field to search. If it's omitted or equal to `NULL`, then the default field is searched.

Thus:

```
$query =
    new Zend_Search_Lucene_Search_Query_Phrase(array('zend', 'framework'));
```

will search for the phrase 'zend framework' in all fields.

```
$query = new Zend_Search_Lucene_Search_Query_Phrase(
                array('zend', 'download'), array(0, 2)
            );
```

will search for the phrase 'zend ????? download' and match 'zend platform download', 'zend studio download', 'zend core download', 'zend framework download', and so on.

```
$query = new Zend_Search_Lucene_Search_Query_Phrase(
                array('zend', 'framework'), null, 'title'
            );
```

will search for the phrase 'zend framework' in the 'title' field.

`Zend_Search_Lucene_Search_Query_Phrase::addTerm()` takes two arguments, a required `Zend_Search_Lucene_Index_Term` object and an optional position:

```
Zend_Search_Lucene_Search_Query_Phrase::addTerm(
    Zend_Search_Lucene_Index_Term $term[, integer $position]
);
```

The `$term` parameter describes the next term in the phrase. It must indicate the same field as previous terms, or an exception will be thrown.

The `$position` parameter indicates the term position in the phrase.

Thus:

```
$query = new Zend_Search_Lucene_Search_Query_Phrase();
$query->addTerm(new Zend_Search_Lucene_Index_Term('zend'));
$query->addTerm(new Zend_Search_Lucene_Index_Term('framework'));
```

will search for the phrase 'zend framework'.

```
$query = new Zend_Search_Lucene_Search_Query_Phrase();
$query->addTerm(new Zend_Search_Lucene_Index_Term('zend'), 0);
$query->addTerm(new Zend_Search_Lucene_Index_Term('framework'), 2);
```

will search for the phrase 'zend ????? download' and match 'zend platform download', 'zend studio download', 'zend core download', 'zend framework download', and so on.

```
$query = new Zend_Search_Lucene_Search_Query_Phrase();
$query->addTerm(new Zend_Search_Lucene_Index_Term('zend', 'title'));
$query->addTerm(new Zend_Search_Lucene_Index_Term('framework', 'title'));
```

will search for the phrase 'zend framework' in the 'title' field.

The slop factor sets the number of other words permitted between specified words in the query phrase. If set to zero, then the corresponding query is an exact phrase search. For larger values this works like the WITHIN or NEAR operators.

The slop factor is in fact an edit distance, where the edits correspond to moving terms in the query phrase. For example, to switch the order of two words requires two moves (the first move places the words atop one another), so to permit re-orderings of phrases, the slop factor must be at least two.

More exact matches are scored higher than sloppier matches; thus, search results are sorted by exactness. The slop is zero by default, requiring exact matches.

The slop factor can be assigned after query creation:

```
// Query without a gap.
$query =
    new Zend_Search_Lucene_Search_Query_Phrase(array('word1', 'word2'));

// Search for 'word1 word2', 'word1 ... word2'
$query->setSlop(1);
$hits1 = $index->find($query);

// Search for 'word1 word2', 'word1 ... word2',
// 'word1 ... ... word2', 'word2 word1'
$query->setSlop(2);
$hits2 = $index->find($query);
```

5.8. Range Query

Range queries are intended for searching terms within specified interval.

Query string:

```
mod_date:[20020101 TO 20030101]
title:{Aida TO Carmen}
```

or

Query construction by API:

```
$from = new Zend_Search_Lucene_Index_Term('20020101', 'mod_date');
$to   = new Zend_Search_Lucene_Index_Term('20030101', 'mod_date');
$query = new Zend_Search_Lucene_Search_Query_Range(
```

```
                $from, $to, true // inclusive
            );
$hits  = $index->find($query);
```

Term fields are optional. Zend_Search_Lucene searches through all fields if the field is not specified:

```
$from = new Zend_Search_Lucene_Index_Term('Aida');
$to   = new Zend_Search_Lucene_Index_Term('Carmen');
$query = new Zend_Search_Lucene_Search_Query_Range(
                $from, $to, false // non-inclusive
            );
$hits  = $index->find($query);
```

Either (but not both) of the boundary terms may be set to NULL. Zend_Search_Lucene searches from the beginning or up to the end of the dictionary for the specified field(s) in this case:

```
// searches for ['20020101' TO ...]
$from = new Zend_Search_Lucene_Index_Term('20020101', 'mod_date');
$query = new Zend_Search_Lucene_Search_Query_Range(
                $from, null, true // inclusive
            );
$hits  = $index->find($query);
```

6. Character Set

6.1. UTF-8 and single-byte character set support

Zend_Search_Lucene works with the UTF-8 charset internally. Index files store unicode data in Java's "modified UTF-8 encoding". Zend_Search_Lucene core completely supports this encoding with one exception. [6]

Actual input data encoding may be specified through Zend_Search_Lucene API. Data will be automatically converted into UTF-8 encoding.

6.2. Default text analyzer

However, the default text analyzer (which is also used within query parser) uses ctype_alpha() for tokenizing text and queries.

ctype_alpha() is not UTF-8 compatible, so the analyzer converts text to 'ASCII//TRANSLIT' encoding before indexing. The same processing is transparently performed during query parsing. [7]

Default analyzer doesn't treats numbers as parts of terms. Use corresponding 'Num' analyzer if you don't want words to be broken by numbers.

[6] Zend_Search_Lucene supports only Basic Multilingual Plane (BMP) characters (from 0x0000 to 0xFFFF) and doesn't support "supplementary characters" (characters whose code points are greater than 0xFFFF)

Java 2 represents these characters as a pair of char (16-bit) values, the first from the high-surrogates range (0xD800-0xDBFF), the second from the low-surrogates range (0xDC00-0xDFFF). Then they are encoded as usual UTF-8 characters in six bytes. Standard UTF-8 representation uses four bytes for supplementary characters.

[7] Conversion to 'ASCII//TRANSLIT' may depend on current locale and OS.

6.3. UTF-8 compatible text analyzers

`Zend_Search_Lucene` also contains a set of UTF-8 compatible analyzers: `Zend_Search_Lucene_Analysis_Analyzer_Common_Utf8`, `Zend_Search_Lucene_Analysis_Analyzer_Common_Utf8Num`, `Zend_Search_Lucene_Analysis_Analyzer_Common_Utf8_CaseInsensitive`, `Zend_Search_Lucene_Analysis_Analyzer_Common_Utf8Num_CaseInsensitive`.

Any of this analyzers can be enabled with the code like this:

```
Zend_Search_Lucene_Analysis_Analyzer::setDefault(
    new Zend_Search_Lucene_Analysis_Analyzer_Common_Utf8());
```

UTF-8 compatible analyzers were improved in Zend Framework 1.5. Early versions of analyzers assumed all non-ascii characters are letters. New analyzers implementation has more accurate behavior.

This may need you to re-build index to have data and search queries tokenized in the same way, otherwise search engine may return wrong result sets.

All of these analyzers need PCRE (Perl-compatible regular expressions) library to be compiled with UTF-8 support turned on. PCRE UTF-8 support is turned on for the PCRE library sources bundled with PHP source code distribution, but if shared library is used instead of bundled with PHP sources, then UTF-8 support state may depend on you operating system.

Use the following code to check, if PCRE UTF-8 support is enabled:

```
if (@preg_match('/\pL/u', 'a') == 1) {
    echo "PCRE unicode support is turned on.\n";
} else {
    echo "PCRE unicode support is turned off.\n";
}
```

Case insensitive versions of UTF-8 compatible analyzers also need mbstring extension to be enabled.

If you don't want mbstring extension to be turned on, but need case insensitive search, you may use the following approach: normalize source data before indexing and query string before searching by converting them to lowercase:

```
// Indexing
setlocale(LC_CTYPE, 'de_DE.iso-8859-1');

...

Zend_Search_Lucene_Analysis_Analyzer::setDefault(
    new Zend_Search_Lucene_Analysis_Analyzer_Common_Utf8());

...

$doc = new Zend_Search_Lucene_Document();

$doc->addField(Zend_Search_Lucene_Field::UnStored('contents',
                                                  strtolower($contents)));
```

```
// Title field for search through (indexed, unstored)
$doc->addField(Zend_Search_Lucene_Field::UnStored('title',
                                                  strtolower($title)));

// Title field for retrieving (unindexed, stored)
$doc->addField(Zend_Search_Lucene_Field::UnIndexed('_title', $title));
```

```
// Searching
setlocale(LC_CTYPE, 'de_DE.iso-8859-1');

...

Zend_Search_Lucene_Analysis_Analyzer::setDefault(
    new Zend_Search_Lucene_Analysis_Analyzer_Common_Utf8());

...

$hits = $index->find(strtolower($query));
```

7. Extensibility

7.1. Text Analysis

The `Zend_Search_Lucene_Analysis_Analyzer` class is used by the indexer to tokenize document text fields.

The `Zend_Search_Lucene_Analysis_Analyzer::getDefault()` and `Zend_Search_Lucene_Analysis_Analyzer::setDefault()` methods are used to get and set the default analyzer.

You can assign your own text analyzer or choose it from the set of predefined analyzers: `Zend_Search_Lucene_Analysis_Analyzer_Common_Text` and `Zend_Search_Lucene_Analysis_Analyzer_Common_Text_CaseInsensitive` (default). Both of them interpret tokens as sequences of letters. `Zend_Search_Lucene_Analysis_Analyzer_Common_Text_CaseInsensitive` converts all tokens to lower case.

To switch between analyzers:

```
Zend_Search_Lucene_Analysis_Analyzer::setDefault(
    new Zend_Search_Lucene_Analysis_Analyzer_Common_Text());
...
$index->addDocument($doc);
```

The `Zend_Search_Lucene_Analysis_Analyzer_Common` class is designed to be an ancestor of all user defined analyzers. User should only define the `reset()` and `nextToken()` methods, which takes its string from the $_input member and returns tokens one by one (a `NULL` value indicates the end of the stream).

The `nextToken()` method should call the `normalize()` method on each token. This will allow you to use token filters with your analyzer.

Here is an example of a custom analyzer, which accepts words with digits as terms:

```
class My_Analyzer extends Zend_Search_Lucene_Analysis_Analyzer_Common
{
    private $_position;

    /**
     * Reset token stream
     */
    public function reset()
    {
        $this->_position = 0;
    }

    /**
     * Tokenization stream API
     * Get next token
     * Returns null at the end of stream
     *
     * @return Zend_Search_Lucene_Analysis_Token|null
     */
    public function nextToken()
    {
        if ($this->_input === null) {
            return null;
        }

        while ($this->_position < strlen($this->_input)) {
            // skip white space
            while ($this->_position < strlen($this->_input) &&
                   !ctype_alnum( $this->_input[$this->_position] )) {
                $this->_position++;
            }

            $termStartPosition = $this->_position;

            // read token
            while ($this->_position < strlen($this->_input) &&
                   ctype_alnum( $this->_input[$this->_position] )) {
                $this->_position++;
            }

            // Empty token, end of stream.
            if ($this->_position == $termStartPosition) {
                return null;
            }

            $token = new Zend_Search_Lucene_Analysis_Token(
                                    substr($this->_input,
                                           $termStartPosition,
                                           $this->_position -
                                           $termStartPosition),
                                    $termStartPosition,
                                    $this->_position);
            $token = $this->normalize($token);
            if ($token !== null) {
                return $token;
            }
            // Continue if token is skipped
        }

        return null;
    }
}

Zend_Search_Lucene_Analysis_Analyzer::setDefault(
    new My_Analyzer());
```

7.2. Tokens Filtering

The Zend_Search_Lucene_Analysis_Analyzer_Common analyzer also offers a token filtering mechanism.

The Zend_Search_Lucene_Analysis_TokenFilter class provides an abstract interface for such filters. Your own filters should extend this class either directly or indirectly.

Any custom filter must implement the normalize() method which may transform input token or signal that the current token should be skipped.

There are three filters already defined in the analysis subpackage:

- Zend_Search_Lucene_Analysis_TokenFilter_LowerCase
- Zend_Search_Lucene_Analysis_TokenFilter_ShortWords
- Zend_Search_Lucene_Analysis_TokenFilter_StopWords

The LowerCase filter is already used for Zend_Search_Lucene_Analysis_Analyzer_Common_Text_CaseInsensitive analyzer by default.

The ShortWords and StopWords filters may be used with pre-defined or custom analyzers like this:

```
$stopWords = array('a', 'an', 'at', 'the', 'and', 'or', 'is', 'am');
$stopWordsFilter =
    new Zend_Search_Lucene_Analysis_TokenFilter_StopWords($stopWords);

$analyzer =
    new Zend_Search_Lucene_Analysis_Analyzer_Common_TextNum_CaseInsensitive();
$analyzer->addFilter($stopWordsFilter);

Zend_Search_Lucene_Analysis_Analyzer::setDefault($analyzer);
```

```
$shortWordsFilter = new Zend_Search_Lucene_Analysis_TokenFilter_ShortWords();

$analyzer =
    new Zend_Search_Lucene_Analysis_Analyzer_Common_TextNum_CaseInsensitive();
$analyzer->addFilter($shortWordsFilter);

Zend_Search_Lucene_Analysis_Analyzer::setDefault($analyzer);
```

The Zend_Search_Lucene_Analysis_TokenFilter_StopWords constructor takes an array of stop-words as an input. But stop-words may be also loaded from a file:

```
$stopWordsFilter = new Zend_Search_Lucene_Analysis_TokenFilter_StopWords();
$stopWordsFilter->loadFromFile($my_stopwords_file);

$analyzer =
   new Zend_Search_Lucene_Analysis_Analyzer_Common_TextNum_CaseInsensitive();
$analyzer->addFilter($stopWordsFilter);

Zend_Search_Lucene_Analysis_Analyzer::setDefault($analyzer);
```

This file should be a common text file with one word in each line. The '#' character marks a line as a comment.

The `Zend_Search_Lucene_Analysis_TokenFilter_ShortWords` constructor has one optional argument. This is the word length limit, set by default to 2.

7.3. Scoring Algorithms

The score of a document d for a query q is defined as follows:

```
score(q,d) = sum( tf(t in d) * idf(t) * getBoost(t.field in d) *
lengthNorm(t.field in d) ) * coord(q,d) * queryNorm(q)
```

tf(t in d) - `Zend_Search_Lucene_Search_Similarity::tf($freq)` - a score factor based on the frequency of a term or phrase in a document.

idf(t) - `Zend_Search_Lucene_Search_Similarity::idf($input, $reader)` - a score factor for a simple term with the specified index.

getBoost(t.field in d) - the boost factor for the term field.

lengthNorm($term) - the normalization value for a field given the total number of terms contained in a field. This value is stored within the index. These values, together with field boosts, are stored in an index and multiplied into scores for hits on each field by the search code.

Matches in longer fields are less precise, so implementations of this method usually return smaller values when numTokens is large, and larger values when numTokens is small.

coord(q,d) - `Zend_Search_Lucene_Search_Similarity::coord($overlap, $maxOverlap)` - a score factor based on the fraction of all query terms that a document contains.

The presence of a large portion of the query terms indicates a better match with the query, so implementations of this method usually return larger values when the ratio between these parameters is large and smaller values when the ratio between them is small.

queryNorm(q) - the normalization value for a query given the sum of the squared weights of each of the query terms. This value is then multiplied into the weight of each query term.

This does not affect ranking, but rather just attempts to make scores from different queries comparable.

The scoring algorithm can be customized by defining your own Similarity class. To do this extend the `Zend_Search_Lucene_Search_Similarity` class as defined below, then use the `Zend_Search_Lucene_Search_Similarity::setDefault($similarity);` method to set it as default.

```php
class MySimilarity extends Zend_Search_Lucene_Search_Similarity {
    public function lengthNorm($fieldName, $numTerms) {
        return 1.0/sqrt($numTerms);
    }

    public function queryNorm($sumOfSquaredWeights) {
        return 1.0/sqrt($sumOfSquaredWeights);
    }
```

```
    public function tf($freq) {
        return sqrt($freq);
    }

    /**
     * It's not used now. Computes the amount of a sloppy phrase match,
     * based on an edit distance.
     */
    public function sloppyFreq($distance) {
        return 1.0;
    }

    public function idfFreq($docFreq, $numDocs) {
        return log($numDocs/(float)($docFreq+1)) + 1.0;
    }

    public function coord($overlap, $maxOverlap) {
        return $overlap/(float)$maxOverlap;
    }
}

$mySimilarity = new MySimilarity();
Zend_Search_Lucene_Search_Similarity::setDefault($mySimilarity);
```

7.4. Storage Containers

The abstract class `Zend_Search_Lucene_Storage_Directory` defines directory functionality.

The `Zend_Search_Lucene` constructor uses either a string or a `Zend_Search_Lucene_Storage_Directory` object as an input.

The `Zend_Search_Lucene_Storage_Directory_Filesystem` class implements directory functionality for a file system.

If a string is used as an input for the `Zend_Search_Lucene` constructor, then the index reader (`Zend_Search_Lucene` object) treats it as a file system path and instantiates the `Zend_Search_Lucene_Storage_Directory_Filesystem` object.

You can define your own directory implementation by extending the `Zend_Search_Lucene_Storage_Directory` class.

`Zend_Search_Lucene_Storage_Directory` methods:

```
abstract class Zend_Search_Lucene_Storage_Directory {
/**
 * Closes the store.
 *
 * @return void
 */
abstract function close();

/**
 * Creates a new, empty file in the directory with the given $filename.
 *
 * @param string $name
 * @return void
 */
```

```
abstract function createFile($filename);

/**
 * Removes an existing $filename in the directory.
 *
 * @param string $filename
 * @return void
 */
abstract function deleteFile($filename);

/**
 * Returns true if a file with the given $filename exists.
 *
 * @param string $filename
 * @return boolean
 */
abstract function fileExists($filename);

/**
 * Returns the length of a $filename in the directory.
 *
 * @param string $filename
 * @return integer
 */
abstract function fileLength($filename);

/**
 * Returns the UNIX timestamp $filename was last modified.
 *
 * @param string $filename
 * @return integer
 */
abstract function fileModified($filename);

/**
 * Renames an existing file in the directory.
 *
 * @param string $from
 * @param string $to
 * @return void
 */
abstract function renameFile($from, $to);

/**
 * Sets the modified time of $filename to now.
 *
 * @param string $filename
 * @return void
 */
abstract function touchFile($filename);

/**
 * Returns a Zend_Search_Lucene_Storage_File object for a given
 * $filename in the directory.
 *
 * @param string $filename
 * @return Zend_Search_Lucene_Storage_File
 */
abstract function getFileObject($filename);
```

```
}
```

The `getFileObject($filename)` method of a `Zend_Search_Lucene_Storage_Directory` instance returns a `Zend_Search_Lucene_Storage_File` object.

The `Zend_Search_Lucene_Storage_File` abstract class implements file abstraction and index file reading primitives.

You must also extend `Zend_Search_Lucene_Storage_File` for your directory implementation.

Only two methods of `Zend_Search_Lucene_Storage_File` must be overridden in your implementation:

```
class MyFile extends Zend_Search_Lucene_Storage_File {
    /**
     * Sets the file position indicator and advances the file pointer.
     * The new position, measured in bytes from the beginning of the file,
     * is obtained by adding offset to the position specified by whence,
     * whose values are defined as follows:
     * SEEK_SET - Set position equal to offset bytes.
     * SEEK_CUR - Set position to current location plus offset.
     * SEEK_END - Set position to end-of-file plus offset. (To move to
     * a position before the end-of-file, you need to pass a negative value
     * in offset.)
     * Upon success, returns 0; otherwise, returns -1
     *
     * @param integer $offset
     * @param integer $whence
     * @return integer
     */
    public function seek($offset, $whence=SEEK_SET) {
        ...
    }

    /**
     * Read a $length bytes from the file and advance the file pointer.
     *
     * @param integer $length
     * @return string
     */
    protected function _fread($length=1) {
        ...
    }
}
```

8. Interoperating with Java Lucene

8.1. File Formats

`Zend_Search_Lucene` index file formats are binary compatible with Java Lucene version 1.4 and greater.

A detailed description of this format is available here: http://lucene.apache.org/java/2_3_0/fileformats.html [8].

[8] The currently supported Lucene index file format version is 2.3 (starting from Zend Framework 1.6).

8.2. Index Directory

After index creation, the index directory will contain several files:

- The `segments` file is a list of index segments.
- The `*.cfs` files contain index segments. Note! An optimized index always has only one segment.
- The `deletable` file is a list of files that are no longer used by the index, but which could not be deleted.

8.3. Java Source Code

The Java program listing below provides an example of how to index a file using Java Lucene:

```
/**
* Index creation:
*/
import org.apache.lucene.index.IndexWriter;
import org.apache.lucene.document.*;

import java.io.*

...

IndexWriter indexWriter = new IndexWriter("/data/my_index",
                                          new SimpleAnalyzer(), true);

...

String filename = "/path/to/file-to-index.txt"
File f = new File(filename);

Document doc = new Document();
doc.add(Field.Text("path", filename));
doc.add(Field.Keyword("modified",DateField.timeToString(f.lastModified())));
doc.add(Field.Text("author", "unknown"));
FileInputStream is = new FileInputStream(f);
Reader reader = new BufferedReader(new InputStreamReader(is));
doc.add(Field.Text("contents", reader));

indexWriter.addDocument(doc);
```

9. Advanced

9.1. Starting from 1.6, handling index format transformations

`Zend_Search_Lucene` component works with Java Lucene 1.4-1.9, 2.1 and 2.3 index formats.

Current index format may be requested using `$index->getFormatVersion()` call. It returns one of the following values:

- `Zend_Search_Lucene::FORMAT_PRE_2_1` for Java Lucene 1.4-1.9 index format.
- `Zend_Search_Lucene::FORMAT_2_1` for Java Lucene 2.1 index format (also used for Lucene 2.2).

- `Zend_Search_Lucene::FORMAT_2_3` for Java Lucene 2.3 index format.

Index modifications are performed *only* if any index update is done. That happens if a new document is added to an index or index optimization is started manually by `$index->optimize()` call.

In a such case `Zend_Search_Lucene` may convert index to the higher format version. That *always* happens for the indices in `Zend_Search_Lucene::FORMAT_PRE_2_1` format, which are automatically converted to 2.1 format.

You may manage conversion process and assign target index format by `$index->setFormatVersion()` which takes `Zend_Search_Lucene::FORMAT_2_1` or `Zend_Search_Lucene::FORMAT_2_3` constant as a parameter:

- `Zend_Search_Lucene::FORMAT_2_1` actually does nothing since pre-2.1 indices are automatically converted to 2.1 format.
- `Zend_Search_Lucene::FORMAT_2_3` forces conversion to the 2.3 format.

Backward conversions are not supported.

Important!

Once index is converted to upper version it can't be converted back. So make a backup of your index when you plan migration to upper version, but want to have possibility to go back.

9.2. Using the index as static property

The `Zend_Search_Lucene` object uses the destructor method to commit changes and clean up resources.

It stores added documents in memory and dumps new index segment to disk depending on `MaxBufferedDocs` parameter.

If `MaxBufferedDocs` limit is not reached then there are some "unsaved" documents which are saved as a new segment in the object's destructor method. The index auto-optimization procedure is invoked if necessary depending on the values of the `MaxBufferedDocs`, `MaxMergeDocs` and `MergeFactor` parameters.

Static object properties (see below) are destroyed *after* the last line of the executed script.

```
class Searcher {
    private static $_index;

    public static function initIndex() {
        self::$_index = Zend_Search_Lucene::open('path/to/index');
    }
}

Searcher::initIndex();
```

All the same, the destructor for static properties is correctly invoked at this point in the program's execution.

One potential problem is exception handling. Exceptions thrown by destructors of static objects don't have context, because the destructor is executed after the script has already completed.

You might see a "Fatal error: Exception thrown without a stack frame in Unknown on line 0" error message instead of exception description in such cases.

`Zend_Search_Lucene` provides a workaround to this problem with the `commit()` method. It saves all unsaved changes and frees memory used for storing new segments. You are free to use the commit operation any time- or even several times- during script execution. You can still use the `Zend_Search_Lucene` object for searching, adding or deleting document after the commit operation. But the `commit()` call guarantees that if there are no document added or deleted after the call to `commit()`, then the `Zend_Search_Lucene` destructor has nothing to do and will not throw exception:

```
class Searcher {
    private static $_index;

    public static function initIndex() {
        self::$_index = Zend_Search_Lucene::open('path/to/index');
    }

    ...

    public static function commit() {
        self::$_index->commit();
    }
}

Searcher::initIndex();

...

// Script shutdown routine
...
Searcher::commit();
...
```

10. Best Practices

10.1. Field names

There are no limitations for field names in `Zend_Search_Lucene`.

Nevertheless it's a good idea not to use '*id*' and '*score*' names to avoid ambiguity in `QueryHit` properties names.

The `Zend_Search_Lucene_Search_QueryHit` `id` and `score` properties always refer to internal Lucene document id and hit score. If the indexed document has the same stored fields, you have to use the `getDocument()` method to access them:

```
$hits = $index->find($query);

foreach ($hits as $hit) {
    // Get 'title' document field
    $title = $hit->title;

    // Get 'contents' document field
    $contents = $hit->contents;

    // Get internal Lucene document id
```

```
    $id = $hit->id;

    // Get query hit score
    $score = $hit->score;

    // Get 'id' document field
    $docId = $hit->getDocument()->id;

    // Get 'score' document field
    $docId = $hit->getDocument()->score;

    // Another way to get 'title' document field
    $title = $hit->getDocument()->title;
}
```

10.2. Indexing performance

Indexing performance is a compromise between used resources, indexing time and index quality.

Index quality is completely determined by number of index segments.

Each index segment is entirely independent portion of data. So indexes containing more segments need more memory and time for searching.

Index optimization is a process of merging several segments into a new one. A fully optimized index contains only one segment.

Full index optimization may be performed with the `optimize()` method:

```
$index = Zend_Search_Lucene::open($indexPath);

$index->optimize();
```

Index optimization works with data streams and doesn't take a lot of memory but does require processor resources and time.

Lucene index segments are not updatable by their nature (the update operation requires the segment file to be completely rewritten). So adding new document(s) to an index always generates a new segment. This, in turn, decreases index quality.

An index auto-optimization process is performed after each segment generation and consists of merging partial segments.

There are three options to control the behavior of auto-optimization (see Index optimization section):

- *MaxBufferedDocs* is the number of documents that can be buffered in memory before a new segment is generated and written to the hard drive.
- *MaxMergeDocs* is the maximum number of documents merged by auto-optimization process into a new segment.
- *MergeFactor* determines how often auto-optimization is performed.

All these options are `Zend_Search_Lucene` object properties- not index properties. They affect only current `Zend_Search_Lucene` object behavior and may vary for different scripts.

MaxBufferedDocs doesn't have any effect if you index only one document per script execution. On the other hand, it's very important for batch indexing. Greater values increase indexing performance, but also require more memory.

There is simply no way to calculate the best value for the *MaxBufferedDocs* parameter because it depends on average document size, the analyzer in use and allowed memory.

A good way to find the right value is to perform several tests with the largest document you expect to be added to the index [9]. It's a best practice not to use more than a half of the allowed memory.

MaxMergeDocs limits the segment size (in terms of documents). It therefore also limits auto-optimization time by guaranteeing that the `addDocument()` method is not executed more than a certain number of times. This is very important for interactive applications.

Lowering the *MaxMergeDocs* parameter also may improve batch indexing performance. Index auto-optimization is an iterative process and is performed from bottom up. Small segments are merged into larger segment, which are in turn merged into even larger segments and so on. Full index optimization is achieved when only one large segment file remains.

Small segments generally decrease index quality. Many small segments may also trigger the "Too many open files" error determined by OS limitations [10].

in general, background index optimization should be performed for interactive indexing mode and *MaxMergeDocs* shouldn't be too low for batch indexing.

MergeFactor affects auto-optimization frequency. Lower values increase the quality of unoptimized indexes. Larger values increase indexing performance, but also increase the number of merged segments. This again may trigger the "Too many open files" error.

MergeFactor groups index segments by their size:

1. Not greater than *MaxBufferedDocs*.
2. Greater than *MaxBufferedDocs*, but not greater than *MaxBufferedDocs***MergeFactor*.
3. Greater than *MaxBufferedDocs***MergeFactor*, but not greater than *MaxBufferedDocs***MergeFactor***MergeFactor*.
4. ...

`Zend_Search_Lucene` checks during each `addDocument()` call to see if merging any segments may move the newly created segment into the next group. If yes, then merging is performed.

So an index with N groups may contain *MaxBufferedDocs* + (N-1)**MergeFactor* segments and contains at least *MaxBufferedDocs***MergeFactor*$^{(N-1)}$ documents.

This gives good approximation for the number of segments in the index:

NumberOfSegments <= *MaxBufferedDocs* + *MergeFactor**$\log_{MergeFactor}$ (*NumberOfDocuments*/*MaxBufferedDocs*)

MaxBufferedDocs is determined by allowed memory. This allows for the appropriate merge factor to get a reasonable number of segments.

[9] `memory_get_usage()` and `memory_get_peak_usage()` may be used to control memory usage.
[10] `Zend_Search_Lucene` keeps each segment file opened to improve search performance.

Tuning the *MergeFactor* parameter is more effective for batch indexing performance than *MaxMergeDocs*. But it's also more course-grained. So use the estimation above for tuning *MergeFactor*, then play with *MaxMergeDocs* to get best batch indexing performance.

10.3. Index during Shut Down

The `Zend_Search_Lucene` instance performs some work at exit time if any documents were added to the index but not written to a new segment.

It also may trigger an auto-optimization process.

The index object is automatically closed when it, and all returned QueryHit objects, go out of scope.

If index object is stored in global variable than it's closed only at the end of script execution[11].

PHP exception processing is also shut down at this moment.

It doesn't prevent normal index shutdown process, but may prevent accurate error diagnostic if any error occurs during shutdown.

There are two ways with which you may avoid this problem.

The first is to force going out of scope:

```
$index = Zend_Search_Lucene::open($indexPath);

...

unset($index);
```

And the second is to perform a commit operation before the end of script execution:

```
$index = Zend_Search_Lucene::open($indexPath);

$index->commit();
```

This possibility is also described in the "Advanced. Using index as static property" section.

10.4. Retrieving documents by unique id

It's a common practice to store some unique document id in the index. Examples include url, path, or database id.

`Zend_Search_Lucene` provides a `termDocs()` method for retrieving documents containing specified terms.

This is more efficient than using the `find()` method:

```
// Retrieving documents with find() method using a query string
$query = $idFieldName . ':' . $docId;
$hits  = $index->find($query);
foreach ($hits as $hit) {
    $title    = $hit->title;
    $contents = $hit->contents;
```

[11] This also may occur if the index or QueryHit instances are referred to in some cyclical data structures, because PHP garbage collects objects with cyclic references only at the end of script execution.

```
    ...
}
...

// Retrieving documents with find() method using the query API
$term = new Zend_Search_Lucene_Index_Term($docId, $idFieldName);
$query = new Zend_Search_Lucene_Search_Query_Term($term);
$hits  = $index->find($query);
foreach ($hits as $hit) {
    $title    = $hit->title;
    $contents = $hit->contents;
    ...
}

...

// Retrieving documents with termDocs() method
$term = new Zend_Search_Lucene_Index_Term($docId, $idFieldName);
$docIds  = $index->termDocs($term);
foreach ($docIds as $id) {
    $doc = $index->getDocument($id);
    $title    = $doc->title;
    $contents = $doc->contents;
    ...
}
```

10.5. Memory Usage

`Zend_Search_Lucene` is a relatively memory-intensive module.

It uses memory to cache some information and optimize searching and indexing performance.

The memory required differs for different modes.

The terms dictionary index is loaded during the search. It's actually each 128th [12] term of the full dictionary.

Thus memory usage is increased if you have a high number of unique terms. This may happen if you use untokenized phrases as a field values or index a large volume of non-text information.

An unoptimized index consists of several segments. It also increases memory usage. Segments are independent, so each segment contains its own terms dictionary and terms dictionary index. If an index consists of *N* segments it may increase memory usage by *N* times in worst case. Perform index optimization to merge all segments into one to avoid such memory consumption.

Indexing uses the same memory as searching plus memory for buffering documents. The amount of memory used may be managed with *MaxBufferedDocs* parameter.

Index optimization (full or partial) uses stream-style data processing and doesn't require a lot of memory.

10.6. Encoding

`Zend_Search_Lucene` works with UTF-8 strings internally. So all strings returned by `Zend_Search_Lucene` are UTF-8 encoded.

[12] The Lucene file format allows you to configure this number, but `Zend_Search_Lucene` doesn't expose this in its API. Nevertheless you still have the ability to configure this value if the index is prepared with another Lucene implementation.

You shouldn't be concerned with encoding if you work with pure ASCII data, but you should be careful if this is not the case.

Wrong encoding may cause error notices at the encoding conversion time or loss of data.

Zend_Search_Lucene offers a wide range of encoding possibilities for indexed documents and parsed queries.

Encoding may be explicitly specified as an optional parameter of field creation methods:

```
$doc = new Zend_Search_Lucene_Document();
$doc->addField(Zend_Search_Lucene_Field::Text('title',
                                              $title,
                                              'iso-8859-1'));
$doc->addField(Zend_Search_Lucene_Field::UnStored('contents',
                                                  $contents,
                                                  'utf-8'));
```

This is the best way to avoid ambiguity in the encoding used.

If optional encoding parameter is omitted, then the current locale is used. The current locale may contain character encoding data in addition to the language specification:

```
setlocale(LC_ALL, 'fr_FR');
...

setlocale(LC_ALL, 'de_DE.iso-8859-1');
...

setlocale(LC_ALL, 'ru_RU.UTF-8');
...
```

The same approach is used to set query string encoding.

If encoding is not specified, then the current locale is used to determine the encoding.

Encoding may be passed as an optional parameter, if the query is parsed explicitly before search:

```
$query =
    Zend_Search_Lucene_Search_QueryParser::parse($queryStr, 'iso-8859-5');
$hits = $index->find($query);
...
```

The default encoding may also be specified with `setDefaultEncoding()` method:

```
Zend_Search_Lucene_Search_QueryParser::setDefaultEncoding('iso-8859-1');
$hits = $index->find($queryStr);
...
```

The empty string implies 'current locale'.

If the correct encoding is specified it can be correctly processed by analyzer. The actual behavior depends on which analyzer is used. See the Character Set documentation section for details.

10.7. Index maintenance

It should be clear that Zend_Search_Lucene as well as any other Lucene implementation does not comprise a "database".

Indexes should not be used for data storage. They do not provide partial backup/restore functionality, journaling, logging, transactions and many other features associated with database management systems.

Nevertheless, `Zend_Search_Lucene` attempts to keep indexes in a consistent state at all times.

Index backup and restoration should be performed by copying the contents of the index folder.

If index corruption occurs for any reason, the corrupted index should be restored or completely rebuilt.

So it's a good idea to backup large indexes and store changelogs to perform manual restoration and roll-forward operations if necessary. This practice dramatically reduces index restoration time.

Zend_Serializer

1. Introduction

`Zend_Serializer` provides an adapter based interface to simply generate storable representation of php types by different facilities, and recover.

Example 635. Using `Zend_Serializer` dynamic interface

To instantiate a serializer you should use the factory method with the name of the adapter:

```
$serializer = Zend_Serializer::factory('PhpSerialize');
// Now $serializer is an instance of Zend_Serializer_Adapter_AdapterInterface,
// specifically Zend_Serializer_Adapter_PhpSerialize

try {
    $serialized = $serializer->serialize($data);
    // now $serialized is a string

    $unserialized = $serializer->unserialize($serialized);
    // now $data == $unserialized
} catch (Zend_Serializer_Exception $e) {
    echo $e;
}
```

The method `serialize` generates a storable string. To regenerate this serialized data you can simply call the method `unserialize`.

Any time an error is encountered serializing or unserializing, `Zend_Serializer` will throw a `Zend_Serializer_Exception`.

To configure a given serializer adapter, you can optionally add an array or an instance of `Zend_Config` to the `factory` or to the `un-/serialize` methods:

```
$serializer = Zend_Serializer::factory('Wddx', array(
    'comment' => 'serialized by Zend_Serializer',
));

try {
    $serialized = $serializer->serialize($data, array('comment' => 'change comment'));
    $unserialized = $serializer->unserialize($serialized, array(/* options for unserialize
} catch (Zend_Serializer_Exception $e) {
    echo $e;
}
```

Options passed to the `factory` are valid for the instantiated object. You can change these options using the `setOption(s)` method. To change one or more options only for a single call, pass them as the second argument to either the `serialize` or `unserialize` method.

1.1. Using the Zend_Serializer static interface

You can register a specific serializer adapter as a default serialization adapter for use with `Zend_Serializer`. By default, the `PhpSerialize` adapter will be registered, but you can change this option using the `setDefaultAdapter()` static method.

```
Zend_Serializer::setDefaultAdapter('PhpSerialize', $options);
// or
$serializer = Zend_Serializer::factory('PhpSerialize', $options);
Zend_Serializer::setDefaultAdapter($serializer);

try {
    $serialized   = Zend_Serializer::serialize($data, $options);
    $unserialized = Zend_Serializer::unserialize($serialized, $options);
} catch (Zend_Serializer_Exception $e) {
    echo $e;
}
```

2. Zend_Serializer_Adapter

`Zend_Serializer` adapters create a bridge for different methods of serializing with very little effort.

Every adapter has different pros and cons. In some cases, not every PHP datatype (e.g., objects) can be converted to a string representation. In most such cases, the type will be converted to a similar type that is serializable -- as an example, PHP objects will often be cast to arrays. If this fails, a `Zend_Serializer_Exception` will be thrown.

Below is a list of available adapters.

2.1. Zend_Serializer_Adapter_PhpSerialize

This adapter uses the built-in `un/serialize` PHP functions, and is a good default adapter choice.

There are no configurable options for this adapter.

2.2. Zend_Serializer_Adapter_Igbinary

Igbinary is Open Source Software released by Sulake Dynamoid Oy. It's a drop-in replacement for the standard PHP serializer. Instead of time and space consuming textual representation, igbinary stores PHP data structures in a compact binary form. Savings are significant when using memcached or similar memory based storages for serialized data.

You need the igbinary PHP extension installed on your system in order to use this adapter.

There adapter takes no configuration options.

2.3. Zend_Serializer_Adapter_Wddx

WDDX (Web Distributed Data eXchange) is a programming-language-, platform-, and transport-neutral data interchange mechanism for passing data between different environments and different computers.

The adapter simply uses the wddx_*() PHP functions. Please read the php manual to determine how you may enable them in your PHP installation.

Additionally, the SimpleXML PHP extension is used to check if a returned `NULL` value from `wddx_unserialize()` is based on a serialized `NULL` or on invalid data.

Available options include:

Table 114. Zend_Serializer_Adapter_Wddx Options

Option	Data Type	Default Value	Description
comment	string		An optional comment that appears in the packet header.

2.4. Zend_Serializer_Adapter_Json

The JSON adapter provides a bridge to the `Zend_Json` component and/or ext/json. Please read the Zend_Json documentation for further information.

Available options include:

Table 115. Zend_Serializer_Adapter_Json Options

Option	Data Type	Default Value	Description
cycleCheck	boolean	false	See Section 3.1, "JSON Objects"
objectDecodeType	Zend_Json::TYPE_*	Zend_Json::TYPE_ARRAY	See Section 3.1, "JSON Objects"
enableJsonExprFinder	boolean	false	See Section 3.4, "JSON Expressions"

2.5. Zend_Serializer_Adapter_Amf 0 and 3

The AMF adapters, `Zend_Serializer_Adapter_Amf0` and `Zend_Serializer_Adapter_Amf3`, provide a bridge to the serializer of the `Zend_Amf` component. Please read the Zend_Amf documentation for further information.

There are no options for these adapters.

2.6. Zend_Serializer_Adapter_PythonPickle

This adapter converts PHP types to a Python Pickle string representation. With it, you can read the serialized data with Python and read Pickled data of Python with PHP.

Available options include:

Table 116. Zend_Serializer_Adapter_PythonPickle Options

Option	Data Type	Default Value	Description
protocol	integer (0 | 1 | 2 | 3)	0	The Pickle protocol version used on `serialize`

Datatype merging (PHP to Python) occurs as follows:

Table 117. Datatype merging (PHP to Python)

PHP Type	Python Type
NULL	None
boolean	boolean

PHP Type	Python Type
integer	integer
float	float
string	string
array	list
associative array	dictionary
object	dictionary

Datatype merging (Python to PHP) occurs per the following:

Table 118. Datatype merging (Python to PHP)

Python-Type	PHP-Type
None	NULL
boolean	boolean
integer	integer
long	integer | float | string | `Zend_Serializer_Exception`
float	float
string	string
bytes	string
Unicode string	UTF-8 string
list	array
tuple	array
dictionary	associative array
All other types	`Zend_Serializer_Exception`

2.7. Zend_Serializer_Adapter_PhpCode

This adapter generates a parsable PHP code representation using var_export(). On restoring, the data will be executed using eval.

There are no configuration options for this adapter.

Unserializing objects

Objects will be serialized using the __set_state magic method. If the class doesn't implement this method, a fatal error will occur during execution.

Uses eval()

The `PhpCode` adapter utilizes `eval()` to unserialize. This introduces both a performance and potential security issue as a new process will be executed. Typically, you should use the `PhpSerialize` adapter unless you require human-readability of the serialized data.

Zend_Server

1. Introduction

The `Zend_Server` family of classes provides functionality for the various server classes, including `Zend_XmlRpc_Server`, `Zend_Rest_Server`, `Zend_Json_Server` and `Zend_Soap_Wsdl`. `Zend_Server_Interface` provides an interface that mimics PHP 5's `SoapServer` class; all server classes should implement this interface in order to provide a standard server API.

The `Zend_Server_Reflection` tree provides a standard mechanism for performing function and class introspection for use as callbacks with the server classes, and provides data suitable for use with `Zend_Server_Interface`'s `getFunctions()` and `loadFunctions()` methods.

2. Zend_Server_Reflection

2.1. Introduction

`Zend_Server_Reflection` provides a standard mechanism for performing function and class introspection for use with server classes. It is based on PHP 5's Reflection API, augmenting it with methods for retrieving parameter and return value types and descriptions, a full list of function and method prototypes (i.e., all possible valid calling combinations), and function/method descriptions.

Typically, this functionality will only be used by developers of server classes for the framework.

2.2. Usage

Basic usage is simple:

```
$class    = Zend_Server_Reflection::reflectClass('My_Class');
$function = Zend_Server_Reflection::reflectFunction('my_function');

// Get prototypes
$prototypes = $reflection->getPrototypes();

// Loop through each prototype for the function
foreach ($prototypes as $prototype) {

    // Get prototype return type
    echo "Return type: ", $prototype->getReturnType(), "\n";

    // Get prototype parameters
    $parameters = $prototype->getParameters();

    echo "Parameters: \n";
    foreach ($parameters as $parameter) {
        // Get parameter type
        echo "    ", $parameter->getType(), "\n";
    }
}

// Get namespace for a class, function, or method.
```

```
// Namespaces may be set at instantiation time (second argument), or using
// setNamespace()
$reflection->getNamespace();
```

`reflectFunction()` returns a `Zend_Server_Reflection_Function` object; `reflectClass` returns a `Zend_Server_Reflection_Class` object. Please refer to the API documentation to see what methods are available to each.

Zend_Service

1. Introduction

`Zend_Service` is an abstract class which serves as a foundation for web service implementations, such as SOAP or REST.

If you need support for generic, XML-based REST services, you may want to look at `Zend_Rest_Client`.

In addition to being able to extend the `Zend_Service` and use `Zend_Rest_Client` for REST-based web services, Zend also provides support for popular web services. See the following sections for specific information on each supported web service.

- Akismet
- Amazon
- Audioscrobbler
- Del.icio.us
- Flickr
- Simpy
- SlideShare
- StrikeIron
- Yahoo!

Additional services are coming in the future.

2. Zend_Service_Akismet

2.1. Introduction

`Zend_Service_Akismet` provides a client for the Akismet API. The Akismet service is used to determine if incoming data is potentially spam. It also exposes methods for submitting data as known spam or as false positives (ham). It was originally intended to help categorize and identify spam for Wordpress, but it can be used for any type of data.

Akismet requires an API key for usage. You can get one by signing up for a WordPress.com account. You do not need to activate a blog. Simply acquiring the account will provide you with the API key.

Akismet requires that all requests contain a URL to the resource for which data is being filtered. Because of Akismet's origins in WordPress, this resource is called the blog URL. This value should be passed as the second argument to the constructor, but may be reset at any time using the `setBlogUrl()` method, or overridden by specifying a 'blog' key in the various method calls.

2.2. Verify an API key

`Zend_Service_Akismet::verifyKey($key)` is used to verify that an Akismet API key is valid. In most cases, you will not need to check, but if you need a sanity check, or to determine if a newly acquired key is active, you may do so with this method.

```
// Instantiate with the API key and a URL to the application or
// resource being used
$akismet = new Zend_Service_Akismet($apiKey,
                                    'http://framework.zend.com/wiki/');
if ($akismet->verifyKey($apiKey) {
    echo "Key is valid.\n";
} else {
    echo "Key is not valid\n";
}
```

If called with no arguments, `verifyKey()` uses the API key provided to the constructor.

`verifyKey()` implements Akismet's `verify-key` REST method.

2.3. Check for spam

`Zend_Service_Akismet::isSpam($data)` is used to determine if the data provided is considered spam by Akismet. It accepts an associative array as the sole argument. That array requires the following keys be set:

- `user_ip`, the IP address of the user submitting the data (not your IP address, but that of a user on your site).
- `user_agent`, the reported UserAgent string (browser and version) of the user submitting the data.

The following keys are also recognized specifically by the API:

- `blog`, the fully qualified URL to the resource or application. If not specified, the URL provided to the constructor will be used.
- `referrer`, the content of the HTTP_REFERER header at the time of submission. (Note spelling; it does not follow the header name.)
- `permalink`, the permalink location, if any, of the entry the data was submitted to.
- `comment_type`, the type of data provided. Values specified in the API include 'comment', 'trackback', 'pingback', and an empty string (''), but it may be any value.
- `comment_author`, the name of the person submitting the data.
- `comment_author_email`, the email of the person submitting the data.
- `comment_author_url`, the URL or home page of the person submitting the data.
- `comment_content`, the actual data content submitted.

You may also submit any other environmental variables you feel might be a factor in determining if data is spam. Akismet suggests the contents of the entire $_SERVER array.

The `isSpam()` method will return either `TRUE` or `FALSE`, or throw an exception if the API key is invalid.

Example 636. isSpam() Usage

```
$data = array(
    'user_ip'             => '111.222.111.222',
    'user_agent'          => 'Mozilla/5.0 ' . (Windows; U; Windows NT ' .
                             '5.2; en-GB; rv:1.8.1) Gecko/20061010 ' .
                             'Firefox/2.0',
    'comment_type'        => 'contact',
    'comment_author'      => 'John Doe',
    'comment_author_email' => 'nospam@myhaus.net',
    'comment_content'     => "I'm not a spammer, honest!"
);
if ($akismet->isSpam($data)) {
    echo "Sorry, but we think you're a spammer.";
} else {
    echo "Welcome to our site!";
}
```

`isSpam()` implements the `comment-check` Akismet API method.

2.4. Submitting known spam

Spam data will occasionally get through the filter. If you discover spam that you feel should have been caught, you can submit it to Akismet to help improve their filter.

`Zend_Service_Akismet::submitSpam()` takes the same data array as passed to `isSpam()`, but does not return a value. An exception will be raised if the API key used is invalid.

Example 637. submitSpam() Usage

```
$data = array(
    'user_ip'             => '111.222.111.222',
    'user_agent'          => 'Mozilla/5.0 (Windows; U; Windows NT 5.2;' .
                             'en-GB; rv:1.8.1) Gecko/20061010 Firefox/2.0',
    'comment_type'        => 'contact',
    'comment_author'      => 'John Doe',
    'comment_author_email' => 'nospam@myhaus.net',
    'comment_content'     => "I'm not a spammer, honest!"
);
$akismet->submitSpam($data));
```

`submitSpam()` implements the `submit-spam` Akismet API method.

2.5. Submitting false positives (ham)

Data will occasionally be trapped erroneously as spam by Akismet. For this reason, you should probably keep a log of all data trapped as spam by Akismet and review it periodically. If you find such occurrences, you can submit the data to Akismet as "ham", or a false positive (ham is good, spam is not).

`Zend_Service_Akismet::submitHam()` takes the same data array as passed to `isSpam()` or `submitSpam()`, and, like `submitSpam()`, does not return a value. An exception will be raised if the API key used is invalid.

Example 638. submitHam() Usage

```
$data = array(
    'user_ip'              => '111.222.111.222',
    'user_agent'           => 'Mozilla/5.0 (Windows; U; Windows NT 5.2;' .
                              'en-GB; rv:1.8.1) Gecko/20061010 Firefox/2.0',
    'comment_type'         => 'contact',
    'comment_author'       => 'John Doe',
    'comment_author_email' => 'nospam@myhaus.net',
    'comment_content'      => "I'm not a spammer, honest!"
);
$akismet->submitHam($data));
```

`submitHam()` implements the `submit-ham` Akismet API method.

2.6. Zend-specific Methods

While the Akismet API only specifies four methods, `Zend_Service_Akismet` has several additional methods that may be used for retrieving and modifying internal properties.

- `getBlogUrl()` and `setBlogUrl()` allow you to retrieve and modify the blog URL used in requests.
- `getApiKey()` and `setApiKey()` allow you to retrieve and modify the API key used in requests.
- `getCharset()` and `setCharset()` allow you to retrieve and modify the character set used to make the request.
- `getPort()` and `setPort()` allow you to retrieve and modify the TCP port used to make the request.
- `getUserAgent()` and `setUserAgent()` allow you to retrieve and modify the HTTP user agent used to make the request. Note: this is not the user_agent used in data submitted to the service, but rather the value provided in the HTTP User-Agent header when making a request to the service.

  The value used to set the user agent should be of the form `some user agent/version | Akismet/version`. The default is `Zend Framework/ZF-VERSION | Akismet/1.11`, where `ZF-VERSION` is the current Zend Framework version as stored in the `Zend_Framework::VERSION` constant.

3. Zend_Service_Amazon

3.1. Introduction

`Zend_Service_Amazon` is a simple API for using Amazon web services. `Zend_Service_Amazon` has two APIs: a more traditional one that follows Amazon's own API, and a simpler "Query API" for constructing even complex search queries easily.

`Zend_Service_Amazon` enables developers to retrieve information appearing throughout Amazon.com web sites directly through the Amazon Web Services API. Examples include:

- Store item information, such as images, descriptions, pricing, and more

- Customer and editorial reviews
- Similar products and accessories
- Amazon.com offers
- ListMania lists

In order to use `Zend_Service_Amazon`, you should already have an Amazon developer API key aswell as a secret key. To get a key and for more information, please visit the Amazon Web Services web site. As of August 15th, 2009 you can only use the Amazon Product Advertising API through `Zend_Service_Amazon`, when specifying the additional secret key.

Attention

Your Amazon developer API and secret keys are linked to your Amazon identity, so take appropriate measures to keep them private.

Example 639. Search Amazon Using the Traditional API

In this example, we search for PHP books at Amazon and loop through the results, printing them.

```
$amazon = new Zend_Service_Amazon('AMAZON_API_KEY', 'US', 'AMAZON_SECRET_KEY');
$results = $amazon->itemSearch(array('SearchIndex' => 'Books',
                                     'Keywords' => 'php'));
foreach ($results as $result) {
    echo $result->Title . '<br />';
}
```

Example 640. Search Amazon Using the Query API

Here, we also search for PHP books at Amazon, but we instead use the Query API, which resembles the Fluent Interface design pattern.

```
$query = new Zend_Service_Amazon_Query('AMAZON_API_KEY',
                                       'US',
                                       'AMAZON_SECRET_KEY');
$query->category('Books')->Keywords('PHP');
$results = $query->search();
foreach ($results as $result) {
    echo $result->Title . '<br />';
}
```

3.2. Country Codes

By default, `Zend_Service_Amazon` connects to the United States ("US") Amazon web service. To connect from a different country, simply specify the appropriate country code string as the second parameter to the constructor:

Example 641. Choosing an Amazon Web Service Country

```
// Connect to Amazon in Japan
$amazon = new Zend_Service_Amazon('AMAZON_API_KEY', 'JP', 'AMAZON_SECRET_KEY');
```

Country codes

Valid country codes are: `CA`, `DE`, `FR`, `JP`, `UK`, and `US`.

3.3. Looking up a Specific Amazon Item by ASIN

The `itemLookup()` method provides the ability to fetch a particular Amazon item when the ASIN is known.

Example 642. Looking up a Specific Amazon Item by ASIN

```
$amazon = new Zend_Service_Amazon('AMAZON_API_KEY', 'US', 'AMAZON_SECRET_KEY');
$item = $amazon->itemLookup('B0000A432X');
```

The `itemLookup()` method also accepts an optional second parameter for handling search options. For full details, including a list of available options, please see the relevant Amazon documentation.

Image information

To retrieve images information for your search results, you must set `ResponseGroup` option to `Medium` or `Large`.

3.4. Performing Amazon Item Searches

Searching for items based on any of various available criteria are made simple using the `itemSearch()` method, as in the following example:

Example 643. Performing Amazon Item Searches

```
$amazon = new Zend_Service_Amazon('AMAZON_API_KEY', 'US', 'AMAZON_SECRET_KEY');
$results = $amazon->itemSearch(array('SearchIndex' => 'Books',
                                     'Keywords' => 'php'));
foreach ($results as $result) {
    echo $result->Title . '<br />';
}
```

Example 644. Using the ResponseGroup Option

The `ResponseGroup` option is used to control the specific information that will be returned in the response.

```
$amazon = new Zend_Service_Amazon('AMAZON_API_KEY', 'US', 'AMAZON_SECRET_KEY');
$results = $amazon->itemSearch(array(
    'SearchIndex'   => 'Books',
    'Keywords'      => 'php',
    'ResponseGroup' => 'Small,ItemAttributes,Images,SalesRank,Reviews,' .
                       'EditorialReview,Similarities,ListmaniaLists'
    ));
foreach ($results as $result) {
    echo $result->Title . '<br />';
}
```

The `itemSearch()` method accepts a single array parameter for handling search options. For full details, including a list of available options, please see the relevant Amazon documentation

The Zend_Service_Amazon_Query class is an easy to use wrapper around this method.

3.5. Using the Alternative Query API

3.5.1. Introduction

Zend_Service_Amazon_Query provides an alternative API for using the Amazon Web Service. The alternative API uses the Fluent Interface pattern. That is, all calls can be made using chained method calls. (e.g., $obj->method()->method2($arg))

The Zend_Service_Amazon_Query API uses overloading to easily set up an item search and then allows you to search based upon the criteria specified. Each of the options is provided as a method call, and each method's argument corresponds to the named option's value:

Example 645. Search Amazon Using the Alternative Query API

In this example, the alternative query API is used as a fluent interface to specify options and their respective values:

```
$query = new Zend_Service_Amazon_Query('MY_API_KEY', 'US', 'AMAZON_SECRET_KEY');
$query->Category('Books')->Keywords('PHP');
$results = $query->search();
foreach ($results as $result) {
    echo $result->Title . '<br />';
}
```

This sets the option Category to "Books" and Keywords to "PHP".

For more information on the available options, please refer to the relevant Amazon documentation.

3.6. Zend_Service_Amazon Classes

The following classes are all returned by Zend_Service_Amazon::itemLookup() and Zend_Service_Amazon::itemSearch():

- Zend_Service_Amazon_Item
- Zend_Service_Amazon_Image
- Zend_Service_Amazon_ResultSet
- Zend_Service_Amazon_OfferSet
- Zend_Service_Amazon_Offer
- Zend_Service_Amazon_SimilarProduct
- Zend_Service_Amazon_Accessories
- Zend_Service_Amazon_CustomerReview
- Zend_Service_Amazon_EditorialReview
- Zend_Service_Amazon_ListMania

3.6.1. Zend_Service_Amazon_Item

`Zend_Service_Amazon_Item` is the class type used to represent an Amazon item returned by the web service. It encompasses all of the items attributes, including title, description, reviews, etc.

3.6.1.1. Zend_Service_Amazon_Item::asXML()

```
string asXML();
```

Return the original XML for the item

3.6.1.2. Properties

`Zend_Service_Amazon_Item` has a number of properties directly related to their standard Amazon API counterparts.

Table 119. Zend_Service_Amazon_Item Properties

Name	Type	Description
ASIN	string	Amazon Item ID
DetailPageURL	string	URL to the Items Details Page
SalesRank	int	Sales Rank for the Item
SmallImage	Zend_Service_Amazon_Image	Small Image of the Item
MediumImage	Zend_Service_Amazon_Image	Medium Image of the Item
LargeImage	Zend_Service_Amazon_Image	Large Image of the Item
Subjects	array	Item Subjects
Offers	`Zend_Service_Amazon_OfferSet`	Offer Summary and Offers for the Item
CustomerReviews	array	Customer reviews represented as an array of `Zend_Service_Amazon_CustomerReview` objects
EditorialReviews	array	Editorial reviews represented as an array of `Zend_Service_Amazon_EditorialReview` objects
SimilarProducts	array	Similar Products represented as an array of `Zend_Service_Amazon_SimilarProduct` objects
Accessories	array	Accessories for the item represented as an array of `Zend_Service_Amazon_Accessories` objects
Tracks	array	An array of track numbers and names for Music CDs and DVDs
ListmaniaLists	array	Item related Listmania Lists as an array of

Name	Type	Description
		Zend_Service_Amazon_ListmainList objects
PromotionalTag	string	Item Promotional Tag

Back to Class List

3.6.2. Zend_Service_Amazon_Image

`Zend_Service_Amazon_Image` represents a remote Image for a product.

3.6.2.1. Properties

Table 120. Zend_Service_Amazon_Image Properties

Name	Type	Description
Url	Zend_Uri	Remote URL for the Image
Height	int	The Height of the image in pixels
Width	int	The Width of the image in pixels

Back to Class List

3.6.3. Zend_Service_Amazon_ResultSet

`Zend_Service_Amazon_ResultSet` objects are returned by Zend_Service_Amazon::itemSearch() and allow you to easily handle the multiple results returned.

SeekableIterator

Implements the `SeekableIterator` for easy iteration (e.g. using `foreach`), as well as direct access to a specific result using `seek()`.

3.6.3.1. Zend_Service_Amazon_ResultSet::totalResults()

```
int totalResults();
```

Returns the total number of results returned by the search

Back to Class List

3.6.4. Zend_Service_Amazon_OfferSet

Each result returned by Zend_Service_Amazon::itemSearch() and Zend_Service_Amazon::itemLookup() contains a `Zend_Service_Amazon_OfferSet` object through which pricing information for the item can be retrieved.

3.6.4.1. Properties

Table 121. Zend_Service_Amazon_OfferSet Properties

Name	Type	Description
LowestNewPrice	int	Lowest Price for the item in "New" condition

Name	Type	Description
LowestNewPriceCurrency	string	The currency for the `LowestNewPrice`
LowestOldPrice	int	Lowest Price for the item in "Used" condition
LowestOldPriceCurrency	string	The currency for the `LowestOldPrice`
TotalNew	int	Total number of "new" condition available for the item
TotalUsed	int	Total number of "used" condition available for the item
TotalCollectible	int	Total number of "collectible" condition available for the item
TotalRefurbished	int	Total number of "refurbished" condition available for the item
Offers	array	An array of `Zend_Service_Amazon_Offer` objects.

Back to Class List

3.6.5. Zend_Service_Amazon_Offer

Each offer for an item is returned as an `Zend_Service_Amazon_Offer` object.

3.6.5.1. Zend_Service_Amazon_Offer Properties

Table 122. Properties

Name	Type	Description
MerchantId	string	Merchants Amazon ID
GlancePage	string	URL for a page with a summary of the Merchant
Condition	string	Condition of the item
OfferListingId	string	ID of the Offer Listing
Price	int	Price for the item
CurrencyCode	string	Currency Code for the price of the item
Availability	string	Availability of the item
IsEligibleForSuperSaverShipping	boolean	Whether the item is eligible for Super Saver Shipping or not

Back to Class List

3.6.6. Zend_Service_Amazon_SimilarProduct

When searching for items, Amazon also returns a list of similar products that the searcher may find to their liking. Each of these is returned as a `Zend_Service_Amazon_SimilarProduct` object.

Each object contains the information to allow you to make sub-sequent requests to get the full information on the item.

3.6.6.1. Properties

Table 123. Zend_Service_Amazon_SimilarProduct Properties

Name	Type	Description
ASIN	string	Products Amazon Unique ID (ASIN)
Title	string	Products Title

Back to Class List

3.6.7. Zend_Service_Amazon_Accessories

Accessories for the returned item are represented as `Zend_Service_Amazon_Accessories` objects

3.6.7.1. Properties

Table 124. Zend_Service_Amazon_Accessories Properties

Name	Type	Description
ASIN	string	Products Amazon Unique ID (ASIN)
Title	string	Products Title

Back to Class List

3.6.8. Zend_Service_Amazon_CustomerReview

Each Customer Review is returned as a `Zend_Service_Amazon_CustomerReview` object.

3.6.8.1. Properties

Table 125. Zend_Service_Amazon_CustomerReview Properties

Name	Type	Description
Rating	string	Item Rating
HelpfulVotes	string	Votes on how helpful the review is
CustomerId	string	Customer ID
TotalVotes	string	Total Votes
Date	string	Date of the Review
Summary	string	Review Summary
Content	string	Review Content

Back to Class List

3.6.9. Zend_Service_Amazon_EditorialReview

Each items Editorial Reviews are returned as a `Zend_Service_Amazon_EditorialReview` object

3.6.9.1. Properties

Table 126. Zend_Service_Amazon_EditorialReview Properties

Name	Type	Description
Source	string	Source of the Editorial Review
Content	string	Review Content

Back to Class List

3.6.10. Zend_Service_Amazon_Listmania

Each results List Mania List items are returned as `Zend_Service_Amazon_Listmania` objects.

3.6.10.1. Properties

Table 127. Zend_Service_Amazon_Listmania Properties

Name	Type	Description
ListId	string	List ID
ListName	string	List Name

Back to Class List

4. Zend_Service_Amazon_Ec2

4.1. Introduction

`Zend_Service_Amazon_Ec2` provides an interface to Amazon Elastic Clound Computing (EC2).

4.2. What is Amazon Ec2?

Amazon EC2 is a web service that enables you to launch and manage server instances in Amazon's data centers using APIs or available tools and utilities. You can use Amazon EC2 server instances at any time, for as long as you need, and for any legal purpose.

4.3. Static Methods

To make using the Ec2 class easier to use there are two static methods that can be invoked from any of the Ec2 Elements. The first static method is `setKeys` which will defind you AWS Access Keys as default keys. When you then create any new object you don't need to pass in any keys to the constructor.

Example 646. setKeys() Example

```
Zend_Service_Amazon_Ec2_Ebs::setKeys('aws_key','aws_secret_key');
```

To set the region that you are working in you can call the `setRegion` to set which Amazon Ec2 Region you are working in. Currently there is only two region available us-east-1 and eu-west-1. If an invalid value is passed it will throw an exception stating that.

Example 647. setRegion() Example

```
Zend_Service_Amazon_Ec2_Ebs::setRegion('us-east-1');
```

Set Amazon Ec2 Region

Alternativly you can set the region when you create each class as the third parameter in the constructor method.

5. Zend_Service_Amazon_Ec2: Instances

5.1. Instance Types

Amazon EC2 instances are grouped into two families: standard and High-CPU. Standard instances have memory to CPU ratios suitable for most general purpose applications; High-CPU instances have proportionally more CPU resources than memory (RAM) and are well suited for compute-intensive applications. When selecting instance types, you might want to use less powerful instance types for your web server instances and more powerful instance types for your database instances. Additionally, you might want to run CPU instance types for CPU-intensive data processing tasks.

One of the advantages of EC2 is that you pay by the instance hour, which makes it convenient and inexpensive to test the performance of your application on different instance families and types. One good way to determine the most appropriate instance family and instance type is to launch test instances and benchmark your application.

Instance Types

The instance types are defined as constants in the code. Column eight in the table is the defined constant name

Table 128. Available Instance Types

Type	CPU	Memory	Storage	Platform	I/O	Name	Constant Name
`Small`	1 EC2 Compute Unit (1 virtual core with 1 EC2 Compute Unit)	1.7 GB	160 GB instance storage (150 GB plus 10 GB root partition)	32-bit	Moderate	m1.small	`Zend_Service_Amazon_E`
`Large`	4 EC2 Compute Units (2 virtual cores with 2 EC2 Compute Units each)	7.5 GB	850 GB instance storage (2 x 420 GB plus 10 GB root partition)	64-bit	High	m1.large	`Zend_Service_Amazon_E`
`Extra Large`	8 EC2 Compute Units (4 virtual cores with 2 EC2	15 GB	1,690 GB instance storage (4 x 420 GB plus 10	64-bit	High	m1.xlarge	`Zend_Service_Amazon_E`

Type	CPU	Memory	Storage	Platform	I/O	Name	Constant Name
	Compute Units each)		GB root partition)				
`High-CPU Medium`	5 EC2 Compute Units (2 virtual cores with 2.5 EC2 Compute Units each)	1.7 GB	350 GB instance storage (340 GB plus 10 GB root partition)	32-bit	Moderate	c1.medium	`Zend_Service_Amazon_E`
`High-CPU Extra Large`	20 EC2 Compute Units (8 virtual cores with 2.5 EC2 Compute Units each)	7 GB	1,690 GB instance storage (4 x 420 GB plus 10 GB root partition)	64-bit	High	c1.xlarge	`Zend_Service_Amazon_E`

5.2. Running Amazon EC2 Instances

This section describes the operation methods for maintaining Amazon EC2 Instances.

Example 648. Starting New Ec2 Instances

`run` will launch a specified number of EC2 Instances. `run` takes an array of parameters to start, below is a table containing the valid values.

Table 129. Valid Run Options

Name	Description	Required
`imageId`	ID of the AMI with which to launch instances.	Yes
`minCount`	Minimum number of instances to launch. Default: 1	No
`maxCount`	Maximum number of instances to launch. Default: 1	No
`keyName`	Name of the key pair with which to launch instances. If you do not provide a key, all instances will be inaccessible.	No
`securityGroup`	Names of the security groups with which to associate the instances.	No

Name	Description	Required
`userData`	The user data available to the launched instances. This should not be Base64 encoded.	No
`instanceType`	Specifies the instance type. Default: m1.small	No
`placement`	Specifies the availability zone in which to launch the instance(s). By default, Amazon EC2 selects an availability zone for you.	No
`kernelId`	The ID of the kernel with which to launch the instance.	No
`ramdiskId`	The ID of the RAM disk with which to launch the instance.	No
`blockDeviceVirtualName`	Specifies the virtual name to map to the corresponding device name. For example: instancestore0	No
`blockDeviceName`	Specifies the device to which you are mapping a virtual name. For example: sdb	No
`monitor`	Turn on AWS CloudWatch Instance Monitoring	No

`run` will return information about each instance that is starting up.

```
$ec2_instance = new Zend_Service_Amazon_Ec2_Instance('aws_key',
                                                     'aws_secret_key');
$return = $ec2_instance->run(array('imageId' => 'ami-509320',
                                   'keyName' => 'myKey',
                                   'securityGroup' => array('web',
                                                            'default')));
```

Example 649. Rebooting an Ec2 Instances

`reboot` will reboot one or more instances.

This operation is asynchronous; it only queues a request to reboot the specified instance(s). The operation will succeed if the instances are valid and belong to the user. Requests to reboot terminated instances are ignored.

`reboot` returns boolean TRUE or FALSE

```
$ec2_instance = new Zend_Service_Amazon_Ec2_Instance('aws_key',
                                                     'aws_secret_key');
$return = $ec2_instance->reboot('instanceId');
```

Example 650. Terminating an Ec2 Instances

`terminate` shuts down one or more instances. This operation is idempotent; if you terminate an instance more than once, each call will succeed.

`terminate` returns boolean TRUE or FALSE

```
$ec2_instance = new Zend_Service_Amazon_Ec2_Instance('aws_key',
                                                     'aws_secret_key');
$return = $ec2_instance->terminate('instanceId');
```

Terminated Instances

Terminated instances will remain visible after termination (approximately one hour).

5.3. Amazon Instance Utilities

In this section you will find out how to retreive information, the console output and see if an instance contains a product code.

Example 651. Describing Instances

`describe` returns information about instances that you own.

If you specify one or more instance IDs, Amazon EC2 returns information for those instances. If you do not specify instance IDs, Amazon EC2 returns information for all relevant instances. If you specify an invalid instance ID, a fault is returned. If you specify an instance that you do not own, it will not be included in the returned results.

`describe` will return an array containing information on the instance.

```
$ec2_instance = new Zend_Service_Amazon_Ec2_Instance('aws_key',
                                                     'aws_secret_key');
$return = $ec2_instance->describe('instanceId');
```

Terminated Instances

Recently terminated instances might appear in the returned results. This interval is usually less than one hour. If you do not want terminated instances to be returned, pass in a second variable of boolean `TRUE` to `describe` and the terminated instances will be ignored.

Example 652. Describing Instances By Image Id

`describeByImageId` is functionally the same as `describe` but it will only return the instances that are using the provided imageId.

`describeByImageId` will return an array containing information on the instances thare were started by the passed in imageId

```
$ec2_instance = new Zend_Service_Amazon_Ec2_Instance('aws_key',
                                                     'aws_secret_key');
$return = $ec2_instance->describeByImageId('imageId');
```

Terminated Instances

Recently terminated instances might appear in the returned results. This interval is usually less than one hour. If you do not want terminated instances to be returned, pass in a second variable of boolean `TRUE` to `describe` and the terminated instances will be ignored.

Example 653. Retreiving Console Output

`consoleOutput` retrieves console output for the specified instance.

Instance console output is buffered and posted shortly after instance boot, reboot, and termination. Amazon EC2 preserves the most recent 64 KB output which will be available for at least one hour after the most recent post.

`consoleOutput` returns an array containing the `instanceId`, `timestamp` from the last output and the `output` from the console.

```
$ec2_instance = new Zend_Service_Amazon_Ec2_Instance('aws_key',
                                                     'aws_secret_key');
$return = $ec2_instance->consoleOutput('instanceId');
```

Example 654. Confirm Product Code on an Instance

`confirmProduct` returns `TRUE` if the specified product code is attached to the specified instance. The operation returns FALSE if the product code is not attached to the instance.

The `confirmProduct` operation can only be executed by the owner of the AMI. This feature is useful when an AMI owner is providing support and wants to verify whether a user's instance is eligible.

```
$ec2_instance = new Zend_Service_Amazon_Ec2_Instance('aws_key',
                                                     'aws_secret_key');
$return = $ec2_instance->confirmProduct('productCode', 'instanceId');
```

Example 655. Turn on CloudWatch Monitoring on an Instance(s)

`monitor` returns the list of instances and their current state of the CloudWatch Monitoring. If the instance does not currently have Monitoring enabled it will be turned on.

```
$ec2_instance = new Zend_Service_Amazon_Ec2_Instance('aws_key',
                                                     'aws_secret_key');
$return = $ec2_instance->monitor('instanceId');
```

Example 656. Turn off CloudWatch Monitoring on an Instance(s)

`monitor` returns the list of instances and their current state of the CloudWatch Monitoring. If the instance currently has Monitoring enabled it will be turned off.

```
$ec2_instance = new Zend_Service_Amazon_Ec2_Instance('aws_key',
                                                     'aws_secret_key');
$return = $ec2_instance->unmonitor('instanceId');
```

6. Zend_Service_Amazon_Ec2: Windows Instances

Using Amazon EC2 instances running Windows is similar to using instances running Linux and UNIX. The following are the major differences between instances that use Linux or UNIX and Windows:

- Remote Desktop—To access Windows instances, you use Remote Desktop instead of SSH.
- Administrative Password—To access Windows instances the first time, you must obtain the administrative password using the ec2-get-password command.
- Simplified Bundling—To bundle a Windows instance, you use a single command that shuts down the instance, saves it as an AMI, and restarts it.

As part of this service, Amazon EC2 instances can now run Microsoft Windows Server 2003. Our base Windows image provides you with most of the common functionality associated with Windows. However, if you require more than two concurrent Windows users or need to leverage applications that require LDAP, Kerberos, RADIUS, or other credential services, you must use Windows with Authentication Services. For example, Microsoft Exchange Server and Microsoft SharePoint Server require Windows with Authentication Services.

To get started using Windows instances, we recommend using the AWS Management Console. There are differences in pricing between Windows and Windows with Authentication Services instances. For information on pricing, go to the Amazon EC2 Product Page.

Amazon EC2 currently provides the following Windows AMIs:

- Windows Authenticated (32-bit)
- Windows Authenticated (64-bit)
- Windows Anonymous (32-bit)
- Windows Anonymous (64-bit)

The Windows public AMIs that Amazon provides are unmodified versions of Windows with the following two exceptions: we added drivers to improve the networking and disk I/O performance and we created the Amazon EC2 configuration service. The Amazon EC2 configuration service performs the following functions:

- Randomly sets the Administrator password on initial launch, encrypts the password with the user's SSH key, and reports it to the console. This operation happens upon initial AMI launch. If you change the password, AMIs that are created from this instance use the new password.
- Configures the computer name to the internal DNS name. To determine the internal DNS name, see Using Instance Addressing.
- Sends the last three system and application errors from the event log to the console. This helps developers to identify problems that caused an instance to crash or network connectivity to be lost.

6.1. Windows Instances Usage

Example 657. Bundles an Amazon EC2 instance running Windows

`bundle()` has three require paramters and one optional

- *instanceId* The instance you want to bundle
- *s3Bucket* Where you want the ami to live on S3
- *s3Prefix* The prefix you want to assign to the AMI on S3
- *uploadExpiration* The expiration of the upload policy. Amazon recommends 12 hours or longer. This is based in nubmer of minutes. Default is 1440 minutes (24 hours)

`bundle()` returns a multi-demential array that contains instanceId, bundleId, state, startTime, updateTime, progress s3Bucket and s3Prefix.

```
$ec2_instance = new Zend_Service_Amazon_Ec2_Instance_Windows('aws_key',
                                                             'aws_secret_key');
$return = $ec2_instance->bundle('instanceId', 's3Bucket', 's3Prefix');
```

Example 658. Describes current bundling tasks

`describeBundle()` Describes current bundling tasks

`describeBundle()` returns a multi-demential array that contains instanceId, bundleId, state, startTime, updateTime, progress s3Bucket and s3Prefix.

```
$ec2_instance = new Zend_Service_Amazon_Ec2_Instance_Windows('aws_key',
                                                             'aws_secret_key');
$return = $ec2_instance->describeBundle('bundleId');
```

Example 659. Cancels an Amazon EC2 bundling operation

`cancelBundle()` Cancels an Amazon EC2 bundling operation

`cancelBundle()` returns a multi-demential array that contains instanceId, bundleId, state, startTime, updateTime, progress s3Bucket and s3Prefix.

```
$ec2_instance = new Zend_Service_Amazon_Ec2_Instance_Windows('aws_key',
                                                             'aws_secret_key');
$return = $ec2_instance->cancelBundle('bundleId');
```

7. Zend_Service_Amazon_Ec2: Reserved Instances

With Amazon EC2 Reserved Instances, you can make a low one-time payment for each instance to reserve and receive a significant discount on the hourly usage charge for that instance.

Amazon EC2 Reserved Instances are based on instance type and location (region and Availability Zone) for a specified period of time (e.g., 1 year or 3 years) and are only available for Linux or UNIX instances.

7.1. How Reserved Instances are Applied

Reserved Instances are applied to instances that meet the type/location criteria during the specified period. In this example, a user is running the following instances:

- (4) m1.small instances in Availability Zone us-east-1a

- (4) c1.medium instances in Availability Zone us-east-1b
- (2) c1.xlarge instances in Availability Zone us-east-1b

The user then purchases the following Reserved Instances.

- (2) m1.small instances in Availability Zone us-east-1a
- (2) c1.medium instances in Availability Zone us-east-1a
- (2) m1.xlarge instances in Availability Zone us-east-1a

Amazon EC2 applies the two m1.small Reserved Instances to two of the instances in Availability Zone us-east-1a. Amazon EC2 doesn't apply the two c1.medium Reserved Instances because the c1.medium instances are in a different Availability Zone and does not apply the m1.xlarge Reserved Instances because there are no running m1.xlarge instances.

7.2. Reserved Instances Usage

Example 660. Describes Reserved Instances that you purchased

`describeInstances()` will return information about a reserved instance or instances that you purchased.

`describeInstances()` returns a multi-demential array that contains reservedInstancesId, instanceType, availabilityZone, duration, fixedPrice, usagePrice, productDescription, instanceCount and state.

```
$ec2_instance = new Zend_Service_Amazon_Ec2_Instance_Reserved('aws_key',
                                                              'aws_secret_key');
$return = $ec2_instance->describeInstances('instanceId');
```

Example 661. Describe current Reserved Instance Offerings available

`describeOfferings()` Describes Reserved Instance offerings that are available for purchase. With Amazon EC2 Reserved Instances, you purchase the right to launch Amazon EC2 instances for a period of time (without getting insufficient capacity errors) and pay a lower usage rate for the actual time used.

`describeOfferings()` returns a multi-demential array that contains reservedInstancesId, instanceType, availabilityZone, duration, fixedPrice, usagePrice and productDescription.

```
$ec2_instance = new Zend_Service_Amazon_Ec2_Instance_Reserved('aws_key',
                                                              'aws_secret_key');
$return = $ec2_instance->describeOfferings();
```

Example 662. Turn off CloudWatch Monitoring on an Instance(s)

`purchaseOffering()` Purchases a Reserved Instance for use with your account. With Amazon EC2 Reserved Instances, you purchase the right to launch Amazon EC2 instances for a period of time (without getting insufficient capacity errors) and pay a lower usage rate for the actual time used.

`purchaseOffering()` returns the reservedInstanceId.

```
$ec2_instance = new Zend_Service_Amazon_Ec2_Instance_Reserved('aws_key',
                                                              'aws_secret_key');
$return = $ec2_instance->purchaseOffering('offeringId', 'instanceCount');
```

8. Zend_Service_Amazon_Ec2: CloudWatch Monitoring

Amazon CloudWatch is an easy-to-use web service that provides comprehensive monitoring for Amazon Elastic Compute Cloud (Amazon EC2) and Elastic Load Balancing. For more details information check cout the Amazon CloudWatch Developers Guide

8.1. CloudWatch Usage

Example 663. Listing Aviable Metrics

`listMetrics()` returns a list of up to 500 valid metrics for which there is recorded data available to a you and a NextToken string that can be used to query for the next set of results.

```
$ec2_ebs = new Zend_Service_Amazon_Ec2_CloudWatch('aws_key','aws_secret_key');
$return = $ec2_ebs->listMetrics();
```

Example 664. Return Statistics for a given metric

`getMetricStatistics()` Returns data for one or more statistics of given a metric.

The maximum number of datapoints that the Amazon CloudWatch service will return in a single GetMetricStatistics request is 1,440. If a request is made that would generate more datapoints than this amount, Amazon CloudWatch will return an error. You can alter your request by narrowing the time range (StartTime, EndTime) or increasing the Period in your single request. You may also get all of the data at the granularity you originally asked for by making multiple requests with adjacent time ranges.

`getMetricStatistics()` only requires two parameters but it also has four additional parameters that are optional.

- *Required:*
- *MeasureName* The measure name that corresponds to the measure for the gathered metric. Valid EC2 Values are CPUUtilization, NetworkIn, NetworkOut, DiskWriteOps DiskReadBytes, DiskReadOps, DiskWriteBytes. Valid Elastic Load Balancing Metrics are Latency, RequestCount, HealthyHostCount UnHealthyHostCount. For more information click here
- *Statistics* The statistics to be returned for the given metric. Valid values are Average, Maximum, Minimum, Samples, Sum. You can specify this as a string or as an array of values. If you don't specify one it will default to Average instead of failing out. If you specify an incorrect option it will just skip it. For more information click here
- *Optional:*
- *Dimensions* Amazon CloudWatch allows you to specify one Dimension to further filter metric data on. If you don't specify a dimension, the service returns the aggregate of all the measures with the given measure name and time range.
- *Unit* The standard unit of Measurement for a given Measure. Valid Values: Seconds, Percent, Bytes, Bits, Count, Bytes/Second, Bits/Second, Count/Second, and None. Constraints: When using count/second as the unit, you should use Sum as the statistic instead of Average. Otherwise, the sample returns as equal to the number of requests instead of the number of 60-second intervals. This will cause the Average to always equals one when the unit is count/second.
- *StartTime* The timestamp of the first datapoint to return, inclusive. For example, 2008-02-26T19:00:00+00:00. We round your value down to the nearest minute. You can set your start time for more than two weeks in the past. However, you will only get data for the past two weeks. (in ISO 8601 format). Constraints: Must be before EndTime.
- *EndTime* The timestamp to use for determining the last datapoint to return. This is the last datapoint to fetch, exclusive. For example, 2008-02-26T20:00:00+00:00 (in ISO 8601 format).

```
$ec2_ebs = new Zend_Service_Amazon_Ec2_CloudWatch('aws_key','aws_secret_key');
$return = $ec2_ebs->getMetricStatistics(
                                        array('MeasureName' => 'NetworkIn',
                                              'Statistics' => array('Average')));
```

9. Zend_Service_Amazon_Ec2: Amazon Machine Images (AMI)

Amazon Machine Images (AMIs) are preconfigured with an ever-growing list of operating systems.

9.1. AMI Information Utilities

Example 665. Register an AMI with EC2

`register` Each AMI is associated with an unique ID which is provided by the Amazon EC2 service through the RegisterImage operation. During registration, Amazon EC2 retrieves the specified image manifest from Amazon S3 and verifies that the image is owned by the user registering the image.

`register` returns the imageId for the registered Image.

```
$ec2_img = new Zend_Service_Amazon_Ec2_Image('aws_key','aws_secret_key');
$ip = $ec2_img->register('imageLocation');
```

Example 666. Deregister an AMI with EC2

`deregister`, Deregisters an AMI. Once deregistered, instances of the AMI can no longer be launched.

`deregister` returns boolean TRUE or FALSE.

```
$ec2_img = new Zend_Service_Amazon_Ec2_Image('aws_key','aws_secret_key');
$ip = $ec2_img->deregister('imageId');
```

Example 667. Describe an AMI

`describe` Returns information about AMIs, AKIs, and ARIs available to the user. Information returned includes image type, product codes, architecture, and kernel and RAM disk IDs. Images available to the user include public images available for any user to launch, private images owned by the user making the request, and private images owned by other users for which the user has explicit launch permissions.

Table 130. Launch permissions fall into three categories

Name	Description
`public`	The owner of the AMI granted launch permissions for the AMI to the all group. All users have launch permissions for these `AMIs`.
`explicit`	The owner of the AMI granted launch permissions to a specific user.

Name	Description
`implicit`	A user has implicit launch permissions for all `AMIs` he or she owns.

The list of AMIs returned can be modified by specifying AMI IDs, AMI owners, or users with launch permissions. If no options are specified, Amazon EC2 returns all AMIs for which the user has launch permissions.

If you specify one or more AMI IDs, only AMIs that have the specified IDs are returned. If you specify an invalid AMI ID, a fault is returned. If you specify an AMI ID for which you do not have access, it will not be included in the returned results.

If you specify one or more AMI owners, only AMIs from the specified owners and for which you have access are returned. The results can include the account IDs of the specified owners, amazon for AMIs owned by Amazon or self for AMIs that you own.

If you specify a list of executable users, only users that have launch permissions for the AMIs are returned. You can specify account IDs (if you own the AMI(s)), self for AMIs for which you own or have explicit permissions, or all for public AMIs.

`describe` returns an array for all the images that match the critera that was passed in. The array contains the imageId, imageLocation, imageState, imageOwnerId, isPublic, architecture, imageType, kernelId, ramdiskId and platform.

```
$ec2_img = new Zend_Service_Amazon_Ec2_Image('aws_key','aws_secret_key');
$ip = $ec2_img->describe();
```

9.2. AMI Attribute Utilities

Example 668. Modify Image Attributes

Modifies an attribute of an AMI

Table 131. Valid Attributes

Name	Description
`launchPermission`	Controls who has permission to launch the AMI. Launch permissions can be granted to specific users by adding userIds. To make the AMI public, add the all group.

Name	Description
productCodes	Associates a product code with AMIs. This allows developers to charge users for using AMIs. The user must be signed up for the product before they can launch the AMI. *This is a write once attribute; after it is set, it cannot be changed or removed.*

modifyAttribute returns boolean TRUE or FALSE.

```
$ec2_img = new Zend_Service_Amazon_Ec2_Image('aws_key','aws_secret_key');
// modify the launchPermission of an AMI
$return = $ec2_img->modifyAttribute('imageId',
                                    'launchPermission',
                                    'add',
                                    'userId',
                                    'userGroup');

// set the product code of the AMI.
$return = $ec2_img->modifyAttribute('imageId',
                                    'productCodes',
                                    'add',
                                    null,
                                    null,
                                    'productCode');
```

Example 669. Reset an AMI Attribute

resetAttribute will reset the attribute of an AMI to its default value. *The productCodes attribute cannot be reset.*

```
$ec2_img = new Zend_Service_Amazon_Ec2_Image('aws_key','aws_secret_key');
$return = $ec2_img->resetAttribute('imageId', 'launchPermission');
```

Example 670. Describe AMI Attribute

describeAttribute returns information about an attribute of an AMI. Only one attribute can be specified per call. Currently only launchPermission and productCodes are supported.

describeAttribute returns an array with the value of the attribute that was requested.

```
$ec2_img = new Zend_Service_Amazon_Ec2_Image('aws_key','aws_secret_key');
$return = $ec2_img->describeAttribute('imageId', 'launchPermission');
```

10. Zend_Service_Amazon_Ec2: Elastic Block Stroage (EBS)

Amazon Elastic Block Store (Amazon EBS) is a new type of storage designed specifically for Amazon EC2 instances. Amazon EBS allows you to create volumes that can be mounted as devices by Amazon EC2 instances. Amazon EBS volumes behave like raw unformatted external block devices. They have user supplied device names and provide a block device interface. You can load a file system on top of Amazon EBS volumes, or use them just as you would use a block device.

You can create up to twenty Amazon EBS volumes of any size (from one GiB up to one TiB). Each Amazon EBS volume can be attached to any Amazon EC2 instance in the same Availability Zone or can be left unattached.

Amazon EBS provides the ability to create snapshots of your Amazon EBS volumes to Amazon S3. You can use these snapshots as the starting point for new Amazon EBS volumes and can protect your data for long term durability.

10.1. Create EBS Volumes and Snapshots

Example 671. Create a new EBS Volume

Creating a brand new EBS Volume requires the size and which zone you want the EBS Volume to be in.

`createNewVolume` will return an array containing information about the new Volume which includes the volumeId, size, zone, status and createTime.

```
$ec2_ebs = new Zend_Service_Amazon_Ec2_Ebs('aws_key','aws_secret_key');
$return = $ec2_ebs->createNewVolume(40, 'us-east-1a');
```

Example 672. Create an EBS Volume from a Snapshot

Creating an EBS Volume from a snapshot requires the snapshot_id and which zone you want the EBS Volume to be in.

`createVolumeFromSnapshot` will return an array containing information about the new Volume which includes the volumeId, size, zone, status, createTime and snapshotId.

```
$ec2_ebs = new Zend_Service_Amazon_Ec2_Ebs('aws_key','aws_secret_key');
$return = $ec2_ebs->createVolumeFromSnapshot('snap-78a54011', 'us-east-1a');
```

Example 673. Create a Snapshot of an EBS Volume

Creating a Snapshot of an EBS Volume requires the volumeId of the EBS Volume.

`createSnapshot` will return an array containing information about the new Volume Snapshot which includes the snapshotId, volumeId, status, startTime and progress.

```
$ec2_ebs = new Zend_Service_Amazon_Ec2_Ebs('aws_key','aws_secret_key');
$return = $ec2_ebs->createSnapshot('volumeId');
```

10.2. Describing EBS Volumes and Snapshots

Example 674. Describing an EBS Volume

`describeVolume` allows you to get information on an EBS Volume or a set of EBS Volumes. If nothing is passed in then it will return all EBS Volumes. If only one EBS Volume needs to be described a string can be passed in while an array of EBS Volume Id's can be passed in to describe them.

`describeVolume` will return an array with information about each Volume which includes the volumeId, size, status and createTime. If the volume is attached to an instance, an addition value of attachmentSet will be returned. The attachment set contains information about the instance that the EBS Volume is attached to, which includes volumeId, instanceId, device, status and attachTime.

```
$ec2_ebs = new Zend_Service_Amazon_Ec2_Ebs('aws_key','aws_secret_key');
$return = $ec2_ebs->describeVolume('volumeId');
```

Example 675. Describe Attached Volumes

To return a list of EBS Volumes currently attached to a running instance you can call this method. It will only return EBS Volumes attached to the instance with the passed in instanceId.

`describeAttachedVolumes` returns the same information as the `describeVolume` but only for the EBS Volumes that are currently attached to the specified instanceId.

```
$ec2_ebs = new Zend_Service_Amazon_Ec2_Ebs('aws_key','aws_secret_key');
$return = $ec2_ebs->describeAttachedVolumes('instanceId');
```

Example 676. Describe an EBS Volume Snapshot

`describeSnapshot` allows you to get information on an EBS Volume Snapshot or a set of EBS Volume Snapshots. If nothing is passed in then it will return information about all EBS Volume Snapshots. If only one EBS Volume Snapshot needs to be described its snapshotId can be passed in while an array of EBS Volume Snapshot Id's can be passed in to describe them.

`describeSnapshot` will return an array containing information about each EBS Volume Snapshot which includes the snapshotId, volumeId, status, startTime and progress.

```
$ec2_ebs = new Zend_Service_Amazon_Ec2_Ebs('aws_key','aws_secret_key');
$return = $ec2_ebs->describeSnapshot('volumeId');
```

10.3. Attach and Detaching Volumes from Instances

Example 677. Attaching an EBS Volume

`attachVolume` will attach an EBS Volume to a running Instance. To attach a volume you need to specify the volumeId, the instanceId and the device *(ex: /dev/sdh)*.

`attachVolume` will return an array with information about the attach status which contains volumeId, instanceId, device, status and attachTime

```
$ec2_ebs = new Zend_Service_Amazon_Ec2_Ebs('aws_key','aws_secret_key');
$return = $ec2_ebs->attachVolume('volumeId', 'instanceid', '/dev/sdh');
```

Example 678. Detaching an EBS Volume

`detachVolume` will detach an EBS Volume from a running Instance. `detachVolume` requires that you specify the volumeId with the optional instanceId and device name that was passed when attaching the volume. If you need to force the detachment you can set the forth parameter to be TRUE and it will force the volume to detach.

`detachVolume` returns an array containing status information about the EBS Volume which includes volumeId, instanceId, device, status and attachTime.

```
$ec2_ebs = new Zend_Service_Amazon_Ec2_Ebs('aws_key','aws_secret_key');
$return = $ec2_ebs->detachVolume('volumeId');
```

Forced Detach

You should only force a detach if the previous detachment attempt did not occur cleanly (logging into an instance, unmounting the volume, and detaching

normally). This option can lead to data loss or a corrupted file system. Use this option only as a last resort to detach a volume from a failed instance. The instance will not have an opportunity to flush file system caches or file system meta data. If you use this option, you must perform file system check and repair procedures.

10.4. Deleting EBS Volumes and Snapshots

Example 679. Deleting an EBS Volume

`deleteVolume` will delete an unattached EBS Volume.

`deleteVolume` will return boolean TRUE or FALSE.

```
$ec2_ebs = new Zend_Service_Amazon_Ec2_Ebs('aws_key','aws_secret_key');
$return = $ec2_ebs->deleteVolume('volumeId');
```

Example 680. Deleting an EBS Volume Snapshot

`deleteSnapshot` will delete an EBS Volume Snapshot.

`deleteSnapshot` returns booleen TRUE or FALSE.

```
$ec2_ebs = new Zend_Service_Amazon_Ec2_Ebs('aws_key','aws_secret_key');
$return = $ec2_ebs->deleteSnapshot('snapshotId');
```

11. Zend_Service_Amazon_Ec2: Elastic IP Addresses

By default, all Amazon EC2 instances are assigned two IP addresses at launch: a private (RFC 1918) address and a public address that is mapped to the private IP address through Network Address Translation (NAT).

If you use dynamic DNS to map an existing DNS name to a new instance's public IP address, it might take up to 24 hours for the IP address to propagate through the Internet. As a result, new instances might not receive traffic while terminated instances continue to receive requests.

To solve this problem, Amazon EC2 provides elastic IP addresses. Elastic IP addresses are static IP addresses designed for dynamic cloud computing. Elastic IP addresses are associated with your account, not specific instances. Any elastic IP addresses that you associate with your account remain associated with your account until you explicitly release them. Unlike traditional static IP addresses, however, elastic IP addresses allow you to mask instance or Availability Zone failures by rapidly remapping your public IP addresses to any instance in your account.

Example 681. Allocating a new Elastic IP

`allocate` will assign your account a new Elastic IP Address.

`allocate` returns the newly allocated ip.

```
$ec2_eip = new Zend_Service_Amazon_Ec2_Elasticip('aws_key','aws_secret_key');
$ip = $ec2_eip->allocate();

// print out your newly allocated elastic ip address;
print $ip;
```

Example 682. Describing Allocated Elastic IP Addresses

`describe` has an optional paramater to describe all of your allocated Elastic IP addresses or just some of your allocated addresses.

`describe` returns an array that contains information on each Elastic IP Address which contains the publicIp and the instanceId if it is assocated.

```
$ec2_eip = new Zend_Service_Amazon_Ec2_Elasticip('aws_key','aws_secret_key');
// describe all
$ips = $ec2_eip->describe();

// describe a subset
$ips = $ec2_eip->describe(array('ip1', 'ip2', 'ip3'));

// describe a single ip address
$ip = $ec2_eip->describe('ip1');
```

Example 683. Releasing Elastic IP

`release` will release an Elastic IP to Amazon.

Returns a boolean TRUE or FALSE.

```
$ec2_eip = new Zend_Service_Amazon_Ec2_Elasticip('aws_key','aws_secret_key');
$ec2_eip->release('ipaddress');
```

Example 684. Associates an Elastic IP to an Instance

`associate` will assign an Elastic IP to an already running instance.

Returns a boolean TRUE or FALSE.

```
$ec2_eip = new Zend_Service_Amazon_Ec2_Elasticip('aws_key','aws_secret_key');
$ec2_eip->associate('instance_id', 'ipaddress');
```

Example 685. Disassociate an Elastic IP from an instance

`disassociate` will disassociate an Elastic IP from an instance. If you terminate an Instance it will automaticly disassociate the Elastic IP address for you.

Returns a boolean TRUE or FALSE.

```
$ec2_eip = new Zend_Service_Amazon_Ec2_Elasticip('aws_key','aws_secret_key');
$ec2_eip->disassociate('ipaddress');
```

12. Zend_Service_Amazon_Ec2: Keypairs

Keypairs are used to access instances.

Example 686. Creating a new Amazon Keypair

`create`, creates a new 2048 bit RSA key pair and returns a unique ID that can be used to reference this key pair when launching new instances.

`create` returns an array which contains the keyName, keyFingerprint and keyMaterial.

```
$ec2_kp = new Zend_Service_Amazon_Ec2_Keypair('aws_key','aws_secret_key');
$return = $ec2_kp->create('my-new-key');
```

Example 687. Deleting an Amazon Keypair

`delete`, will delete the key pair. This will only prevent it from being used with new instances. Instances currently running with the keypair will still allow you to access them.

`delete` returns boolean TRUE or FALSE

```
$ec2_kp = new Zend_Service_Amazon_Ec2_Keypair('aws_key','aws_secret_key');
$return = $ec2_kp->delete('my-new-key');
```

Example 688. Describe an Amazon Keypair

`describe` returns information about key pairs available to you. If you specify key pairs, information about those key pairs is returned. Otherwise, information for all registered key pairs is returned.

`describe` returns an array which contains keyName and keyFingerprint

```
$ec2_kp = new Zend_Service_Amazon_Ec2_Keypair('aws_key','aws_secret_key');
$return = $ec2_kp->describe('my-new-key');
```

13. Zend_Service_Amazon_Ec2: Regions and Availability Zones

Amazon EC2 provides the ability to place instances in different regions and Availability Zones. Regions are dispersed in separate geographic areas or countries. Availability Zones are located within regions and are engineered to be insulated from failures in other Availability Zones and provide inexpensive low latency network connectivity to other Availability Zones in the same region. By launching instances in separate Availability Zones, you can protect your applications from the failure of a single Availability Zone.

13.1. Amazon EC2 Regions

Amazon EC2 provides multiple regions so you can launch Amazon EC2 instances in locations that meet your requirements. For example, you might want to launch instances in Europe to be closer to your European customers or to meet legal requirements.

Each Amazon EC2 region is designed to be completely isolated from the other Amazon EC2 regions. This achieves the greatest possible failure independence and stability, and it makes the locality of each EC2 resource unambiguous.

Example 689. Viewing the available regions

`describe` is used to find out which regions your account has access to.

`describe` will return an array containing information about which regions are available. Each array will contain regionName and regionUrl.

```
$ec2_region = new Zend_Service_Amazon_Ec2_Region('aws_key','aws_secret_key');
$regions = $ec2_region->describe();

foreach($regions as $region) {
    print $region['regionName'] . ' -- ' . $region['regionUrl'] . '<br />';
}
```

13.2. Amazon EC2 Availability Zones

When you launch an instance, you can optionally specify an Availability Zone. If you do not specify an Availability Zone, Amazon EC2 selects one for you in the region that you are using. When launching your initial instances, we recommend accepting the default Availability Zone, which allows Amazon EC2 to select the best Availability Zone for you based on system health and available capacity. Even if you have other instances running, you might consider not specifying an Availability Zone if your new instances do not need to be close to, or separated from, your existing instances.

Example 690. Viewing the available zones

`describe` is used to find out which what the status is of each availability zone.

`describe` will return an array containing information about which zones are available. Each array will contain zoneName and zoneState.

```
$ec2_zones = new Zend_Service_Amazon_Ec2_Availabilityzones('aws_key',
                                                           'aws_secret_key');
$zones = $ec2_zones->describe();

foreach($zones as $zone) {
    print $zone['zoneName'] . ' -- ' . $zone['zoneState'] . '<br />';
}
```

14. Zend_Service_Amazon_Ec2: Security Groups

A security group is a named collection of access rules. These access rules specify which ingress (i.e., incoming) network traffic should be delivered to your instance. All other ingress traffic will be discarded.

You can modify rules for a group at any time. The new rules are automatically enforced for all running instances and instances launched in the future.

Maximum Security Groups

You can create up to 100 security groups.

14.1. Security Group Maintenance

Example 691. Create a new Security Group

`create` a new security group. Every instance is launched in a security group. If no security group is specified during launch, the instances are launched in the default security group. Instances within the same security group have unrestricted network access to each other. Instances will reject network access attempts from other instances in a different security group.

`create` returns boolean TRUE or FALSE

```
$ec2_sg = new Zend_Service_Amazon_Ec2_Securitygroups('aws_key',
                                                     'aws_secret_key');
$return = $ec2_sg->create('mygroup', 'my group description');
```

Example 692. Describe a Security Group

`describe` returns information about security groups that you own.

If you specify security group names, information about those security groups is returned. Otherwise, information for all security groups is returned. If you specify a group that does not exist, a fault is returned.

`describe` will return an array containing information about security groups which includes the ownerId, groupName, groupDescription and an array containing all the rules for that security group.

```
$ec2_sg = new Zend_Service_Amazon_Ec2_Securitygroups('aws_key',
                                                     'aws_secret_key');
$return = $ec2_sg->describe('mygroup');
```

Example 693. Delete a Security Group

`delete` will remove the security group. If you attempt to delete a security group that contains instances, a fault is returned. If you attempt to delete a security group that is referenced by another security group, a fault is returned. For example, if security group B has a rule that allows access from security group A, security group A cannot be deleted until the allow rule is removed.

`delete` returns boolean TRUE or FALSE.

```
$ec2_sg = new Zend_Service_Amazon_Ec2_Securitygroups('aws_key',
                                                     'aws_secret_key');
$return = $ec2_sg->delete('mygroup');
```

14.2. Authorizing Access

Example 694. Authorizing by IP

`authorizeIp` Adds permissions to a security group based on an IP address, protocol type and port range.

Permissions are specified by the IP protocol (TCP, UDP or ICMP), the source of the request (by IP range or an Amazon EC2 user-group pair), the source and destination port ranges (for TCP and UDP), and the ICMP codes and types (for ICMP). When authorizing ICMP, -1 can be used as a wildcard in the type and code fields.

Permission changes are propagated to instances within the security group as quickly as possible. However, depending on the number of instances, a small delay might occur.

`authorizeIp` returns boolean TRUE or FALSE

```
$ec2_sg = new Zend_Service_Amazon_Ec2_Securitygroups('aws_key',
                                                     'aws_secret_key');
$return = $ec2_sg->authorizeIp('mygroup',
                               'protocol',
                               'fromPort',
                               'toPort',
                               'ipRange');
```

Example 695. Authorize By Group

`authorizeGroup` Adds permissions to a security group.

Permission changes are propagated to instances within the security group as quickly as possible. However, depending on the number of instances, a small delay might occur.

`authorizeGroup` returns boolean TRUE or FALSE.

```
$ec2_sg = new Zend_Service_Amazon_Ec2_Securitygroups('aws_key',
                                                     'aws_secret_key');
$return = $ec2_sg->authorizeGroup('mygroup', 'securityGroupName', 'ownerId');
```

14.3. Revoking Access

Example 696. Revoke by IP

`revokeIp` Revokes permissions to a security group based on an IP address, protocol type and port range. The permissions used to revoke must be specified using the same values used to grant the permissions.

Permissions are specified by the IP protocol (TCP, UDP or ICMP), the source of the request (by IP range or an Amazon EC2 user-group pair), the source and destination port ranges (for TCP and UDP), and the ICMP codes and types (for ICMP). When authorizing ICMP, -1 can be used as a wildcard in the type and code fields.

Permission changes are propagated to instances within the security group as quickly as possible. However, depending on the number of instances, a small delay might occur.

`revokeIp` returns boolean TRUE or FALSE

```
$ec2_sg = new Zend_Service_Amazon_Ec2_Securitygroups('aws_key',
                                                     'aws_secret_key');
$return = $ec2_sg->revokeIp('mygroup',
                            'protocol',
                            'fromPort',
                            'toPort',
                            'ipRange');
```

Example 697. Revoke By Group

`revokeGroup` Adds permissions to a security group. The permissions to revoke must be specified using the same values used to grant the permissions.

Permission changes are propagated to instances within the security group as quickly as possible. However, depending on the number of instances, a small delay might occur.

`revokeGroup` returns boolean TRUE or FALSE.

```
$ec2_sg = new Zend_Service_Amazon_Ec2_Securitygroups('aws_key',
                                                     'aws_secret_key');
$return = $ec2_sg->revokeGroup('mygroup', 'securityGroupName', 'ownerId');
```

15. Zend_Service_Amazon_S3

15.1. Introduction

Amazon S3 provides a simple web services interface that can be used to store and retrieve any amount of data, at any time, from anywhere on the web. It gives any developer access to the same highly scalable, reliable, fast, inexpensive data storage infrastructure that Amazon uses to run its own global network of web sites. The service aims to maximize benefits of scale and to pass those benefits on to developers.

15.2. Registering with Amazon S3

Before you can get started with `Zend_Service_Amazon_S3`, you must first register for an account. Please see the S3 FAQ page on the Amazon website for more information.

After registering, you will receive an application key and a secret key. You will need both to access the S3 service.

15.3. API Documentation

The `Zend_Service_Amazon_S3` class provides the PHP wrapper to the Amazon S3 REST interface. Please consult the Amazon S3 documentation for detailed description of the service. You will need to be familiar with basic concepts in order to use this service.

15.4. Features

`Zend_Service_Amazon_S3` provides the following functionality:

- A single point for configuring your amazon.s3 authentication credentials that can be used across the amazon.s3 namespaces.
- A proxy object that is more convenient to use than an HTTP client alone, mostly removing the need to manually construct HTTP POST requests to access the REST service.
- A response wrapper that parses each response body and throws an exception if an error occurred, alleviating the need to repeatedly check the success of many commands.
- Additional convenience methods for some of the more common operations.

15.5. Getting Started

Once you have registered with Amazon S3, you're ready to store your first data object on the S3. The objects on S3 are stored in containers, called "buckets". Bucket names are unique on S3, and each user can have no more than 100 buckets simultaneously. Each bucket can contain unlimited amount of objects, identified by name.

The following example demonstrates creating a bucket, storing and retrieving the data.

Example 698. Zend_Service_Amazon_S3 Usage Example

```
require_once 'Zend/Service/Amazon/S3.php';

$s3 = new Zend_Service_Amazon_S3($my_aws_key, $my_aws_secret_key);

$s3->createBucket("my-own-bucket");

$s3->putObject("my-own-bucket/myobject", "somedata");

echo $s3->getObject("my-own-bucket/myobject");
```

Since `Zend_Service_Amazon_S3` service requires authentication, you should pass your credentials (AWS key and secret key) to the constructor. If you only use one account, you can set default credentials for the service:

```
require_once 'Zend/Service/Amazon/S3.php';

Zend_Service_Amazon_S3::setKeys($my_aws_key, $my_aws_secret_key);
$s3 = new Zend_Service_Amazon_S3();
```

15.6. Bucket operations

All objects in S3 system are stored in buckets. Bucket has to be created before any storage operation. Bucket name is unique in the system, so you can not have bucket named the same as someone else's bucket.

Bucket name can contain lowercase letters, digits, periods (.), underscores (_), and dashes (-). No other symbols allowed. Bucket name should start with letter or digit, and be 3 to 255 characters long. Names looking like an IP address (e.g. "192.168.16.255") are not allowed.

- `createBucket()` creates a new bucket.
- `cleanBucket()` removes all objects that are contained in a bucket.
- `removeBucket()` removes the bucket from the system. The bucket should be empty to be removed.

Example 699. Zend_Service_Amazon_S3 Bucket Removal Example

```
require_once 'Zend/Service/Amazon/S3.php';

$s3 = new Zend_Service_Amazon_S3($my_aws_key, $my_aws_secret_key);

$s3->cleanBucket("my-own-bucket");
$s3->removeBucket("my-own-bucket");
```

- `getBuckets()` returns the list of the names of all buckets belonging to the user.

Example 700. Zend_Service_Amazon_S3 Bucket Listing Example

```
require_once 'Zend/Service/Amazon/S3.php';

$s3 = new Zend_Service_Amazon_S3($my_aws_key, $my_aws_secret_key);

$list = $s3->getBuckets();
foreach($list as $bucket) {
  echo "I have bucket $bucket\n";
}
```

- `isBucketAvailable()` check if the bucket exists and returns TRUE if it does.

15.7. Object operations

The object is the basic storage unit in S3. Object stores unstructured data, which can be any size up to 4 gigabytes. There's no limit on how many objects can be stored on the system.

The object are contained in buckets. Object is identified by name, which can be any utf-8 string. It is common to use hierarchical names (such as `Pictures/Myself/CodingInPHP.jpg`) to organise object names. Object name is prefixed with bucket name when using object functions, so for object "mydata" in bucket "my-own-bucket" the name would be `my-own-bucket/mydata`.

Objects can be replaced (by rewriting new data with the same key) or deleted, but not modified, appended, etc. Object is always stored whole.

By default, all objects are private and can be accessed only by their owner. However, it is possible to specify object with public access, in which case it will be available through the URL: `http://s3.amazonaws.com/[bucket-name]/[object-name]`.

- `putObject($object, $data, $meta)` created an object with name `$object` (should contain the bucket name as prefix!) having `$data` as its content.

  Optional `$meta` parameter is the array of metadata, which currently supports the following parameters as keys:

  `S3_CONTENT_TYPE_HEADER` — MIME content type of the data. If not specified, the type will be guessed according to the file extension of the object name.

  `S3_ACL_HEADER` — The access to the item. Following access constants can be used:

  - `S3_ACL_PRIVATE` — Only the owner has access to the item.
  - `S3_ACL_PUBLIC_READ` — Anybody can read the object, but only owner can write. This is setting may be used to store publicly accessible content.
  - `S3_ACL_PUBLIC_WRITE` — Anybody can read or write the object. This policy is rarely useful.

S3_ACL_AUTH_READ — Only the owner has write access to the item, and other authenticated S3 users have read access. This is useful for sharing data between S3 accounts without exposing them to the public.

By default, all the items are private.

Example 701. Zend_Service_Amazon_S3 Public Object Example

```
require_once 'Zend/Service/Amazon/S3.php';

$s3 = new Zend_Service_Amazon_S3($my_aws_key, $my_aws_secre

$s3->putObject("my-own-bucket/Pictures/Me.png", file_get_co
    array(Zend_Service_Amazon_S3::S3_ACL_HEADER =>
          Zend_Service_Amazon_S3::S3_ACL_PUBLIC_READ));
// or:
$s3->putFile("me.png", "my-own-bucket/Pictures/Me.png",
    array(Zend_Service_Amazon_S3::S3_ACL_HEADER =>
          Zend_Service_Amazon_S3::S3_ACL_PUBLIC_READ));
echo "Go to http://s3.amazonaws.com/my-own-bucket/Pictures/
```

- `getObject($object)` retrieves object data from the storage by name.
- `removeObject($object)` removes the object from the storage.
- `getInfo($object)` retrieves the metadata information about the object. The function will return array with metadata information. Some of the useful keys are:

  `type` The MIME type of the item.

  `size` The size of the object data.

  `mtime` UNIX-type timestamp of the last modification for the object.

  `etag` The ETag of the data, which is the MD5 hash of the data, surrounded by quotes ('). The function will return FALSE if the key does not correspond to any existing object.

- `getObjectsByBucket($bucket)` returns the list of the object keys, contained in the bucket.

Example 702. Zend_Service_Amazon_S3 Object Listing Example

```
require_once 'Zend/Service/Amazon/S3.php';

$s3 = new Zend_Service_Amazon_S3($my_aws_key, $my_aws_secret_key);

$list = $s3->getObjectsByBucket("my-own-bucket");
foreach($list as $name) {
  echo "I have $name key:\n";
  $data = $s3->getObject("my-own-bucket/$name");
  echo "with data: $data\n";
}
```

- `isObjectAvailable($object)` checks if the object with given name exists.

- `putFile($path, $object, $meta)` puts the content of the file in `$path` into the object named `$object`.

  The optional `$meta` argument is the same as for `putObject`. If the content type is omitted, it will be guessed basing on the source file name.

15.8. Data Streaming

It is possible to get and put objects using not stream data held in memory but files or PHP streams. This is especially useful when file sizes are large in order not to overcome memory limits.

To receive object using streaming, use method `getObjectStream($object, $filename)`. This method will return `Zend_Http_Response_Stream`, which can be used as described in HTTP Client Data Streaming section.

Example 703. Zend_Service_Amazon_S3 Data Streaming Example

```
$response = $amazon->getObjectStream("mybycket/zftest");
// copy file
copy($response->getStreamName(), "my/downloads/file");
// use stream
$fp = fopen("my/downloads/file2", "w");
stream_copy_to_stream($response->getStream(), $fp);
```

Second parameter for `getObjectStream()` is optional and specifies target file to write the data. If not specified, temporary file is used, which will be deleted after the respons eobject is destroyed.

To send object using streaming, use `putFileStream()` which has the same signature as `putFile()` but will use streaming and not read the file into memory.

Also, you can pass stream resource to `putObject()` method data parameter, in which case the data will be read from the stream when sending the request to the server.

15.9. Stream wrapper

In addition to the interfaces described above, `Zend_Service_Amazon_S3` also supports operating as a stream wrapper. For this, you need to register the client object as the stream wrapper:

Example 704. Zend_Service_Amazon_S3 Streams Example

```
require_once 'Zend/Service/Amazon/S3.php';

$s3 = new Zend_Service_Amazon_S3($my_aws_key, $my_aws_secret_key);

$s3->registerStreamWrapper("s3");

mkdir("s3://my-own-bucket");
file_put_contents("s3://my-own-bucket/testdata", "mydata");

echo file_get_contents("s3://my-own-bucket/testdata");
```

Directory operations (`mkdir`, `rmdir`, `opendir`, etc.) will operate on buckets and thus their arguments should be of the form of `s3://bucketname`. File operations operate on objects. Object creation, reading, writing, deletion, stat and directory listing is supported.

16. Zend_Service_Amazon_Sqs

16.1. Introduction

Amazon Simple Queue Service (Amazon SQS) offers a reliable, highly scalable, hosted queue for storing messages as they travel between computers. By using Amazon SQS, developers can simply move data between distributed components of their applications that perform different tasks, without losing messages or requiring each component to be always available. Amazon SQS makes it easy to build an automated workflow, working in close conjunction with the Amazon Elastic Compute Cloud (Amazon EC2) and the other AWS infrastructure web services.

Amazon SQS works by exposing Amazon's web-scale messaging infrastructure as a web service. Any computer on the Internet can add or read messages without any installed software or special firewall configurations. Components of applications using Amazon SQS can run independently, and do not need to be on the same network, developed with the same technologies, or running at the same time.

16.2. Registering with Amazon SQS

Before you can get started with `Zend_Service_Amazon_Sqs`, you must first register for an account. Please see the SQS FAQ page on the Amazon website for more information.

After registering, you will receive an application key and a secret key. You will need both to access the SQS service.

16.3. API Documentation

The `Zend_Service_Amazon_Sqs` class provides the PHP wrapper to the Amazon SQS REST interface. Please consult the Amazon SQS documentation for detailed description of the service. You will need to be familiar with basic concepts in order to use this service.

16.4. Features

`Zend_Service_Amazon_Sqs` provides the following functionality:

- A single point for configuring your amazon.sqs authentication credentials that can be used across the amazon.sqs namespaces.
- A proxy object that is more convenient to use than an HTTP client alone, mostly removing the need to manually construct HTTP POST requests to access the REST service.
- A response wrapper that parses each response body and throws an exception if an error occurred, alleviating the need to repeatedly check the success of many commands.
- Additional convenience methods for some of the more common operations.

16.5. Getting Started

Once you have registered with Amazon SQS, you're ready to create your queue and store some messages on SQS. Each queue can contain unlimited amount of messages, identified by name.

The following example demonstrates creating a queue, storing and retrieving messages.

Example 705. Zend_Service_Amazon_Sqs Usage Example

```
$sqs = new Zend_Service_Amazon_Sqs($my_aws_key, $my_aws_secret_key);

$queue_url = $sqs->create('test');

$message = 'this is a test';
$message_id = $sqs->send($queue_url, $message);

foreach ($sqs->receive($queue_url) as $message) {
    echo $message['body'].'<br/>';
}
```

Since the `Zend_Service_Amazon_Sqs` service requires authentication, you should pass your credentials (AWS key and secret key) to the constructor. If you only use one account, you can set default credentials for the service:

```
Zend_Service_Amazon_Sqs::setKeys($my_aws_key, $my_aws_secret_key);
$sqs = new Zend_Service_Amazon_Sqs();
```

16.6. Queue operations

All messages SQS are stored in queues. A queue has to be created before any message operations. Queue names must be unique under your access key and secret key.

Queue names can contain lowercase letters, digits, periods (.), underscores (_), and dashes (-). No other symbols allowed. Queue names can be a maximum of 80 characters.

- `create()` creates a new queue.
- `delete()` removes all messages in the queue.

Example 706. Zend_Service_Amazon_Sqs Queue Removal Example

```
$sqs = new Zend_Service_Amazon_Sqs($my_aws_key, $my_aws_secret_key);
$queue_url = $sqs->create('test_1');
$sqs->delete($queue_url);
```

- `count()` gets the approximate number of messages in the queue.

Example 707. Zend_Service_Amazon_Sqs Queue Count Example

```
$sqs = new Zend_Service_Amazon_Sqs($my_aws_key, $my_aws_secret_key);
$queue_url = $sqs->create('test_1');
$sqs->send($queue_url, 'this is a test');
$count = $sqs->count($queue_url); // Returns '1'
```

- `getQueues()` returns the list of the names of all queues belonging to the user.

Example 708. Zend_Service_Amazon_Sqs Queue Listing Example

```
$sqs = new Zend_Service_Amazon_Sqs($my_aws_key, $my_aws_secret_key);
$list = $sqs->getQueues();
foreach($list as $queue) {
   echo "I have queue $queue\n";
}
```

16.7. Message operations

After a queue is created, simple messages can be sent into the queue then received at a later point in time. Messages can be up to 8KB in length. If longer messages are needed please see S3. There is no limit to the number of messages a queue can contain.

- `sent($queue_url, $message)` send the `$message` to the `$queue_url` SQS queue URL.

Example 709. Zend_Service_Amazon_Sqs Message Send Example

```
$sqs = new Zend_Service_Amazon_Sqs($my_aws_key, $my_aws_secret_key);
$queue_url = $sqs->create('test_queue');
$sqs->send($queue_url, 'this is a test message');
```

- `receive($queue_url)` retrieves messages from the queue.

Example 710. Zend_Service_Amazon_Sqs Message Receive Example

```
$sqs = new Zend_Service_Amazon_Sqs($my_aws_key, $my_aws_secret_key);
$queue_url = $sqs->create('test_queue');
$sqs->send($queue_url, 'this is a test message');
foreach ($sqs->receive($queue_url) as $message) {
    echo "got message ".$message['body'].'<br/>';
}
```

- `deleteMessage($queue_url, $handle)` deletes a message from a queue. A message must first be received using the `receive()` method before it can be deleted.

Example 711. Zend_Service_Amazon_Sqs Message Delete Example

```
$sqs = new Zend_Service_Amazon_Sqs($my_aws_key, $my_aws_secret_key);
$queue_url = $sqs->create('test_queue');
$sqs->send($queue_url, 'this is a test message');
foreach ($sqs->receive($queue_url) as $message) {
    echo "got message ".$message['body'].'<br/>';

    if ($sqs->deleteMessage($queue_url, $message['handle'])) {
        echo "Message deleted";
    }
    else {
        echo "Message not deleted";
    }
}
```

17. Zend_Service_Audioscrobbler

17.1. Introduction

`Zend_Service_Audioscrobbler` is a simple API for using the Audioscrobbler REST Web Service. The Audioscrobbler Web Service provides access to its database of Users, Artists, Albums, Tracks, Tags, Groups, and Forums. The methods of the `Zend_Service_Audioscrobbler` class begin with one of these terms. The syntax and namespaces of the Audioscrobbler Web Service are mirrored in `Zend_Service_Audioscrobbler`. For more information about the Audioscrobbler REST Web Service, please visit the Audioscrobbler Web Service site.

17.2. Users

In order to retrieve information for a specific user, the `setUser()` method is first used to select the user for which data are to be retrieved. `Zend_Service_Audioscrobbler` provides several methods for retrieving data specific to a single user:

- `userGetProfileInformation()`: Returns a SimpleXML object containing the current user's profile information.
- `userGetTopArtists()`: Returns a SimpleXML object containing a list of the current user's most listened to artists.
- `userGetTopAlbums()`: Returns a SimpleXML object containing a list of the current user's most listened to albums.
- `userGetTopTracks()`: Returns a SimpleXML object containing a list of the current user's most listened to tracks.
- `userGetTopTags()`: Returns a SimpleXML object containing a list of tags most applied by the current user.
- `userGetTopTagsForArtist()`: Requires that an artist be set via `setArtist()`. Returns a SimpleXML object containing the tags most applied to the current artist by the current user.
- `userGetTopTagsForAlbum()`: Requires that an album be set via `setAlbum()`. Returns a SimpleXML object containing the tags most applied to the current album by the current user.
- `userGetTopTagsForTrack()`: Requires that a track be set via `setTrack()`. Returns a SimpleXML object containing the tags most applied to the current track by the current user.
- `userGetFriends()`: Returns a SimpleXML object containing the user names of the current user's friends.
- `userGetNeighbours()`: Returns a SimpleXML object containing the user names of people with similar listening habits to the current user.
- `userGetRecentTracks()`: Returns a SimpleXML object containing the 10 tracks most recently played by the current user.
- `userGetRecentBannedTracks()`: Returns a SimpleXML object containing a list of the 10 tracks most recently banned by the current user.
- `userGetRecentLovedTracks()`: Returns a SimpleXML object containing a list of the 10 tracks most recently loved by the current user.
- `userGetRecentJournals()`: Returns a SimpleXML object containing a list of the current user's most recent journal entries.
- `userGetWeeklyChartList()`: Returns a SimpleXML object containing a list of weeks for which there exist Weekly Charts for the current user.
- `userGetRecentWeeklyArtistChart()`: Returns a SimpleXML object containing the most recent Weekly Artist Chart for the current user.
- `userGetRecentWeeklyAlbumChart()`: Returns a SimpleXML object containing the most recent Weekly Album Chart for the current user.
- `userGetRecentWeeklyTrackChart()`: Returns a SimpleXML object containing the most recent Weekly Track Chart for the current user.

- `userGetPreviousWeeklyArtistChart($fromDate, $toDate)`: Returns a SimpleXML object containing the Weekly Artist Chart from `$fromDate` to `$toDate` for the current user.
- `userGetPreviousWeeklyAlbumChart($fromDate, $toDate)`: Returns a SimpleXML object containing the Weekly Album Chart from `$fromDate` to `$toDate` for the current user.
- `userGetPreviousWeeklyTrackChart($fromDate, $toDate)`: Returns a SimpleXML object containing the Weekly Track Chart from `$fromDate` to `$toDate` for the current user.

Example 712. Retrieving User Profile Information

In this example, we use the `setUser()` and `userGetProfileInformation()` methods to retrieve a specific user's profile information:

```
$as = new Zend_Service_Audioscrobbler();
// Set the user whose profile information we want to retrieve
$as->setUser('BigDaddy71');
// Retrieve BigDaddy71's profile information
$profileInfo = $as->userGetProfileInformation();
// Display some of it
print "Information for $profileInfo->realname "
    . "can be found at $profileInfo->url";
```

Example 713. Retrieving a User's Weekly Artist Chart

```
$as = new Zend_Service_Audioscrobbler();
// Set the user whose profile weekly artist chart we want to retrieve
$as->setUser('lo_fye');
// Retrieves a list of previous weeks for which there are chart data
$weeks = $as->userGetWeeklyChartList();
if (count($weeks) < 1) {
    echo 'No data available';
}
sort($weeks); // Order the list of weeks

$as->setFromDate($weeks[0]); // Set the starting date
$as->setToDate($weeks[0]); // Set the ending date

$previousWeeklyArtists = $as->userGetPreviousWeeklyArtistChart();

echo 'Artist Chart For Week Of '
   . date('Y-m-d h:i:s', $as->from_date)
   . '<br />';

foreach ($previousWeeklyArtists as $artist) {
    // Display the artists' names with links to their profiles
    print '<a href="' . $artist->url . '">' . $artist->name . '</a><br />';
}
```

17.3. Artists

`Zend_Service_Audioscrobbler` provides several methods for retrieving data about a specific artist, specified via the `setArtist()` method:

- `artistGetRelatedArtists()`: Returns a SimpleXML object containing a list of Artists similar to the current Artist.

- `artistGetTopFans()`: Returns a SimpleXML object containing a list of Users who listen most to the current Artist.
- `artistGetTopTracks()`: Returns a SimpleXML object containing a list of the current Artist's top-rated Tracks.
- `artistGetTopAlbums()`: Returns a SimpleXML object containing a list of the current Artist's top-rated Albums.
- `artistGetTopTags()`: Returns a SimpleXML object containing a list of the Tags most frequently applied to current Artist.

Example 714. Retrieving Related Artists

```
$as = new Zend_Service_Audioscrobbler();
// Set the artist for whom you would like to retrieve related artists
$as->setArtist('LCD Soundsystem');
// Retrieve the related artists
$relatedArtists = $as->artistGetRelatedArtists();
foreach ($relatedArtists as $artist) {
    // Display the related artists
    print '<a href="' . $artist->url . '">' . $artist->name . '</a><br />';
}
```

17.4. Tracks

`Zend_Service_Audioscrobbler` provides two methods for retrieving data specific to a single track, specified via the `setTrack()` method:

- `trackGetTopFans()`: Returns a SimpleXML object containing a list of Users who listen most to the current Track.
- `trackGetTopTags()`: Returns a SimpleXML object containing a list of the Tags most frequently applied to the current Track.

17.5. Tags

`Zend_Service_Audioscrobbler` provides several methods for retrieving data specific to a single tag, specified via the `setTag()` method:

- `tagGetOverallTopTags()`: Returns a SimpleXML object containing a list of Tags most frequently used on Audioscrobbler.
- `tagGetTopArtists()`: Returns a SimpleXML object containing a list of Artists to whom the current Tag was most frequently applied.
- `tagGetTopAlbums()`: Returns a SimpleXML object containing a list of Albums to which the current Tag was most frequently applied.
- `tagGetTopTracks()`: Returns a SimpleXML object containing a list of Tracks to which the current Tag was most frequently applied.

17.6. Groups

`Zend_Service_Audioscrobbler` provides several methods for retrieving data specific to a single group, specified via the `setGroup()` method:

- `groupGetRecentJournals()`: Returns a SimpleXML object containing a list of recent journal posts by Users in the current Group.

- `groupGetWeeklyChart()`: Returns a SimpleXML object containing a list of weeks for which there exist Weekly Charts for the current Group.
- `groupGetRecentWeeklyArtistChart()`: Returns a SimpleXML object containing the most recent Weekly Artist Chart for the current Group.
- `groupGetRecentWeeklyAlbumChart()`: Returns a SimpleXML object containing the most recent Weekly Album Chart for the current Group.
- `groupGetRecentWeeklyTrackChart()`: Returns a SimpleXML object containing the most recent Weekly Track Chart for the current Group.
- `groupGetPreviousWeeklyArtistChart($fromDate, $toDate)`: Requires `setFromDate()` and `setToDate()`. Returns a SimpleXML object containing the Weekly Artist Chart from the current fromDate to the current toDate for the current Group.
- `groupGetPreviousWeeklyAlbumChart($fromDate, $toDate)`: Requires `setFromDate()` and `setToDate()`. Returns a SimpleXML object containing the Weekly Album Chart from the current fromDate to the current toDate for the current Group.
- `groupGetPreviousWeeklyTrackChart($fromDate, $toDate)`: Returns a SimpleXML object containing the Weekly Track Chart from the current fromDate to the current toDate for the current Group.

17.7. Forums

`Zend_Service_Audioscrobbler` provides a method for retrieving data specific to a single forum, specified via the `setForum()` method:

- `forumGetRecentPosts()`: Returns a SimpleXML object containing a list of recent posts in the current forum.

18. Zend_Service_Delicious

18.1. Introduction

`Zend_Service_Delicious` is simple API for using del.icio.us XML and JSON web services. This component gives you read-write access to posts at del.icio.us if you provide credentials. It also allows read-only access to public data of all users.

Example 715. Get all posts

```
$delicious = new Zend_Service_Delicious('username', 'password');
$posts = $delicious->getAllPosts();

foreach ($posts as $post) {
    echo "--\n";
    echo "Title: {$post->getTitle()}\n";
    echo "Url: {$post->getUrl()}\n";
}
```

18.2. Retrieving posts

`Zend_Service_Delicious` provides three methods for retrieving posts: `getPosts()`, `getRecentPosts()` and `getAllPosts()`. All of these methods return an instance of `Zend_Service_Delicious_PostList`, which holds all retrieved posts.

```
/**
 * Get posts matching the arguments. If no date or url is given,
 * most recent date will be used.
 *
 * @param string $tag Optional filtering by tag
 * @param Zend_Date $dt Optional filtering by date
 * @param string $url Optional filtering by url
 * @return Zend_Service_Delicious_PostList
 */
public function getPosts($tag = null, $dt = null, $url = null);

/**
 * Get recent posts
 *
 * @param string $tag   Optional filtering by tag
 * @param string $count Maximal number of posts to be returned
 *                      (default 15)
 * @return Zend_Service_Delicious_PostList
 */
public function getRecentPosts($tag = null, $count = 15);

/**
 * Get all posts
 *
 * @param string $tag Optional filtering by tag
 * @return Zend_Service_Delicious_PostList
 */
public function getAllPosts($tag = null);
```

18.3. Zend_Service_Delicious_PostList

Instances of this class are returned by the `getPosts()`, `getAllPosts()`, `getRecentPosts()`, and `getUserPosts()` methods of `Zend_Service_Delicious`.

For easier data access this class implements the `Countable`, `Iterator`, and `ArrayAccess` interfaces.

Example 716. Accessing post lists

```
$delicious = new Zend_Service_Delicious('username', 'password');
$posts = $delicious->getAllPosts();

// count posts
echo count($posts);

// iterate over posts
foreach ($posts as $post) {
    echo "--\n";
    echo "Title: {$post->getTitle()}\n";
    echo "Url: {$post->getUrl()}\n";
}

// get post using array access
echo $posts[0]->getTitle();
```

The `ArrayAccess::offsetSet()` and `ArrayAccess::offsetUnset()` methods throw exceptions in this implementation. Thus, code like

`unset($posts[0]);` and `$posts[0] = 'A';` will throw exceptions because these properties are read-only.

Post list objects have two built-in filtering capabilities. Post lists may be filtered by tags and by URL.

Example 717. Filtering a Post List with Specific Tags

Posts may be filtered by specific tags using `withTags()`. As a convenience, `withTag()` is also provided for when only a single tag needs to be specified.

```
$delicious = new Zend_Service_Delicious('username', 'password');
$posts = $delicious->getAllPosts();

// Print posts having "php" and "zend" tags
foreach ($posts->withTags(array('php', 'zend')) as $post) {
    echo "Title: {$post->getTitle()}\n";
    echo "Url: {$post->getUrl()}\n";
}
```

Example 718. Filtering a Post List by URL

Posts may be filtered by URL matching a specified regular expression using the `withUrl()` method:

```
$delicious = new Zend_Service_Delicious('username', 'password');
$posts = $delicious->getAllPosts();

// Print posts having "help" in the URL
foreach ($posts->withUrl('/help/') as $post) {
    echo "Title: {$post->getTitle()}\n";
    echo "Url: {$post->getUrl()}\n";
}
```

18.4. Editing posts

Example 719. Post editing

```
$delicious = new Zend_Service_Delicious('username', 'password');
$posts = $delicious->getPosts();

// set title
$posts[0]->setTitle('New title');
// save changes
$posts[0]->save();
```

Example 720. Method call chaining

Every setter method returns the post object so that you can chain method calls using a fluent interface.

```
$delicious = new Zend_Service_Delicious('username', 'password');
$posts = $delicious->getPosts();

$posts[0]->setTitle('New title')
         ->setNotes('New notes')
         ->save();
```

18.5. Deleting posts

There are two ways to delete a post, by specifying the post URL or by calling the `delete()` method upon a post object.

Example 721. Deleting posts

```
$delicious = new Zend_Service_Delicious('username', 'password');

// by specifying URL
$delicious->deletePost('http://framework.zend.com');

// or by calling the method upon a post object
$posts = $delicious->getPosts();
$posts[0]->delete();

// another way of using deletePost()
$delicious->deletePost($posts[0]->getUrl());
```

18.6. Adding new posts

To add a post you first need to call the `createNewPost()` method, which returns a `Zend_Service_Delicious_Post` object. When you edit the post, you need to save it to the del.icio.us database by calling the `save()` method.

Example 722. Adding a post

```
$delicious = new Zend_Service_Delicious('username', 'password');

// create a new post and save it (with method call chaining)
$delicious->createNewPost('Zend Framework', 'http://framework.zend.com')
          ->setNotes('Zend Framework Homepage')
          ->save();

// create a new post and save it  (without method call chaining)
$newPost = $delicious->createNewPost('Zend Framework',
                                     'http://framework.zend.com');
$newPost->setNotes('Zend Framework Homepage');
$newPost->save();
```

18.7. Tags

Example 723. Tags

```
$delicious = new Zend_Service_Delicious('username', 'password');

// get all tags
print_r($delicious->getTags());

// rename tag ZF to zendFramework
$delicious->renameTag('ZF', 'zendFramework');
```

18.8. Bundles

Example 724. Bundles

```
$delicious = new Zend_Service_Delicious('username', 'password');

// get all bundles
print_r($delicious->getBundles());

// delete bundle someBundle
$delicious->deleteBundle('someBundle');

// add bundle
$delicious->addBundle('newBundle', array('tag1', 'tag2'));
```

18.9. Public data

The del.icio.us web API allows access to the public data of all users.

Table 132. Methods for retrieving public data

Name	Description	Return type
`getUserFans()`	Retrieves fans of a user	Array
`getUserNetwork()`	Retrieves network of a user	Array
`getUserPosts()`	Retrieves posts of a user	Zend_Service_Delicious_PostList
`getUserTags()`	Retrieves tags of a user	Array

When using only these methods, a username and password combination is not required when constructing a new `Zend_Service_Delicious` object.

Example 725. Retrieving public data

```
// username and password are not required
$delicious = new Zend_Service_Delicious();

// get fans of user someUser
print_r($delicious->getUserFans('someUser'));

// get network of user someUser
print_r($delicious->getUserNetwork('someUser'));

// get tags of user someUser
print_r($delicious->getUserTags('someUser'));
```

18.9.1. Public posts

When retrieving public posts with the `getUserPosts()` method, a `Zend_Service_Delicious_PostList` object is returned, and it contains `Zend_Service_Delicious_SimplePost` objects, which contain basic information about the posts, including URL, title, notes, and tags.

Table 133. Methods of the Zend_Service_Delicious_SimplePost class

Name	Description	Return type
`getNotes()`	Returns notes of a post	String

Name	Description	Return type
`getTags()`	Returns tags of a post	Array
`getTitle()`	Returns title of a post	String
`getUrl()`	Returns URL of a post	String

18.10. HTTP client

`Zend_Service_Delicious` uses `Zend_Rest_Client` for making HTTP requests to the del.icio.us web service. To change which HTTP client `Zend_Service_Delicious` uses, you need to change the HTTP client of `Zend_Rest_Client`.

Example 726. Changing the HTTP client of Zend_Rest_Client

```
$myHttpClient = new My_Http_Client();
Zend_Rest_Client::setHttpClient($myHttpClient);
```

When you are making more than one request with `Zend_Service_Delicious` to speed your requests, it's better to configure your HTTP client to keep connections alive.

Example 727. Configuring your HTTP client to keep connections alive

```
Zend_Rest_Client::getHttpClient()->setConfig(array(
        'keepalive' => true
));
```

When a `Zend_Service_Delicious` object is constructed, the SSL transport of `Zend_Rest_Client` is set to `'ssl'` rather than the default of `'ssl2'`. This is because del.icio.us has some problems with `'ssl2'`, such as requests taking a long time to complete (around 2 seconds).

19. Zend_Service_DeveloperGarden

19.1. Introduction to DeveloperGarden

DeveloperGarden is the name for the "Open Development services" of the German Telekom. The "Open Development services" are a set of SOAP API Services.

The family of `Zend_Service_DeveloperGarden` components provides a clean and simple interface to the DeveloperGarden API and additionally offers functionality to improve handling and performance.

- BaseUserService: Class to manage API quota and user accounting details.
- IpLocation: Locale the given IP and returns geo coordinates. Works only with IPs allocated in the network of the german telekom.
- LocalSearch: Allows you to search with options nearby or around a given geo coordinate or city.
- SendSms: Send a Sms or Flash Sms to a given number.
- SmsValidation: You can validate a number to use it with SendSms for also supply a back channel.

- VoiceCall: Initiates a call between two numbers.
- ConferenceCall: You can configure a whole conference room with participants for an adhoc conference or you can also schedule your conference.

The backend SOAP API is documented here.

19.1.1. Sign Up for an Account

Before you can start using the DeveloperGarden API, you must first sign up for an account.

19.1.2. The Environment

With the DeveloperGarden API you have the possibility to choose 3 different environments to work on.

- *production*: In Production environement you have to pay for calls, sms and other payable services.
- *sandbox*: In the Sandbox mode you can use the same features with some limitations like in the production just without to pay for them. Normally during development you can test your application.
- *mock*: The Mock environment allows you to build your application and have results but you don not initiate any action on the API side.

For every environment and service are some special features (options) available for testing. Please look here for details.

19.1.3. Your configuration

You can pass to all classes an array of configuration values. Possible values are:

- *username*: Your DeveloperGarden API username.
- *password*: Your DeveloperGarden API password.
- *environment*: The environment that you selected.

Example 728. Configuration Example

```
require_once 'Zend/Service/DeveloperGarden/SendSms.php';
$config = array(
    'username'    => 'yourUsername',
    'password'    => 'yourPassword',
    'environment' => Zend_Service_DeveloperGarden_SendSms::ENV_PRODUCTION,
);
$service = new Zend_Service_DeveloperGarden_SendSms($config);
```

19.2. BaseUserService

The class can be used to set and get quota values for the services and to fetch account details.

The `getAccountBalance()` method is there to fetch an array of account id's with the current balance status (credits).

Example 729. Get account balance example

```
$service = new Zend_Service_DeveloperGarden_BaseUserService($config);
print_r($service->getAccountBalance());
```

19.2.1. Get quota information

You can fetch quota informations for a specific service module with the provided methods.

Example 730. Get quota information example

```
$service = new Zend_Service_DeveloperGarden_BaseUserService($config);
$result = $service->getSmsQuotaInformation(
    Zend_Service_DeveloperGarden_BaseUserService::ENV_PRODUCTION
);
echo 'Sms Quota:<br />';
echo 'Max Quota: ', $result->getMaxQuota(), '<br />';
echo 'Max User Quota: ', $result->getMaxUserQuota(), '<br />';
echo 'Quota Level: ', $result->getQuotaLevel(), '<br />';
```

You get a `Result` object that contains all information that you need, optional you can pass to the `QuotaInformation` method the environment constant to fetch the quota for the specific environment.

Here a list of all `getQuotaInformation` methods:

- `getConfernceCallQuotaInformation()`
- `getIPLocationQuotaInformation()`
- `getLocalSearchQuotaInformation()`
- `getSmsQuotaInformation()`
- `getVoiceCallQuotaInformation()`

19.2.2. Change quota information

To change the current quota use one of the `changeQuotaPool` methods. First parameter is the new pool value and the second one is the environment.

Example 731. Change quota information example

```
$service = new Zend_Service_DeveloperGarden_BaseUserService($config);
$result = $service->changeSmsQuotaPool(
    1000,
    Zend_Service_DeveloperGarden_BaseUserService::ENV_PRODUCTION
);
if (!$result->hasError()) {
    echo 'updated Quota Pool';
}
```

Here a list of all `changeQuotaPool` methods:

- `changeConferenceCallQuotaPool()`
- `changeIPLocationQuotaPool()`
- `changeLocalSearchQuotaPool()`

- `changeSmsQuotaPool()`
- `changeVoiceCallQuotaPool()`

19.3. IP Location

This service allows you to retrieve location information for a given IP address.

There are some limitations:

- The IP address must be in the T-Home network
- Just the next big city will be resolved
- IPv6 is not supported yet

Example 732. Locate a given IP

```
$service = new Zend_Service_DeveloperGarden_IpLocation($config);
$service->setEnvironment(
    Zend_Service_DeveloperGarden_IpLocation::ENV_MOCK
);
$ip = new Zend_Service_DeveloperGarden_IpLocation_IpAddress('127.0.0.1');
print_r($service->locateIp($ip));
```

19.4. Local Search

The Local Search service provides the local search machine suchen.de via a the web service interface. For more details, refer to the documentation.

Example 733. Locate a Restaurant

```
$service = new Zend_Service_DeveloperGarden_LocalSearch($config);
$search  = new Zend_Service_DeveloperGarden_LocalSearch_SearchParameters();
/**
 * @see http://www.developergarden.com/static/docu/en/ch04s02s06s04.html
 */
$search->setWhat('pizza')
       ->setWhere('jena');
print_r($service->localSearch($search));
```

19.5. Send SMS

The Send SMS service is used to send normal and Flash SMS to any number.

The following restrictions apply to the use of the SMS service:

- An SMS or Flash SMS in the production environment must not be longer than 765 characters and must not be sent to more than 10 recipients.
- An SMS or Flash SMS in the sandbox environment is shortened and enhanced by a note in the DeveloperGarden. The maximum length of the message sent is 160 characters.
- In the sandbox environment, a maximum of 10 SMS can be sent per day.
- The following characters are counted twice: `|` `^` `€` `{` `}` `[` `]` `~` `\` LF (line break)
- If an SMS or Flash SMS is longer than 160 characters, one message is charged for each 153 characters (quota and credit).

- Delivery cannot be guaranteed for SMS or Flash SMS to landline numbers.
- The sender can exist of a maximum of 11 characters. Permitted characters are letters and numbers.
- The specification of a phone number as the sender is only permitted if the phone number has been validated. (See: SMS Validation)

Example 734. Sending an SMS

```
$service = new Zend_Service_DeveloperGarden_SendSms($config);
$sms = $service->createSms(
    '+49-172-123456; +49-177-789012',
    'your test message',
    'yourname'
);
print_r($service->send($sms));
```

19.6. SMS Validation

The SMS Validation service allows the validation of physical phone number to be used as the sender of an SMS.

First, call `setValidationKeyword()` to receive an SMS with a keyword.

After you get your keyword, you have to use the `validate()` to validate your number with the keyword against the service.

With the method `getValidatedNumbers()`, you will get a list of all already validated numbers and the status of each.

Example 735. Request validation keyword

```
$service = new Zend_Service_DeveloperGarden_SmsValidation($config);
print_r($service->sendValidationKeyword('+49-172-123456'));
```

Example 736. Validate a number with a keyword

```
$service = new Zend_Service_DeveloperGarden_SmsValidation($config);
print_r($service->validate('TheKeyWord', '+49-172-123456'));
```

To invalidate a validated number, call the method `inValidate()`.

19.7. Voice Call

The Voice Call service is used for setting up a voice connection between two telephone connections. For specific details please read the API Documentation.

Normally the Service works as followed:

- Call the first participant.
- If the connection is successful, call the second participant.
- If second participant connects succesfully, both participants are connected.
- The call is open until one of the participants hangs up or the expire mechanism intercepts.

Example 737. Call two numbers

```
$service = new Zend_Service_DeveloperGarden_VoiceCall($config);
$aNumber = '+49-30-000001';
$bNumber = '+49-30-000002';
$expiration  = 30;  // seconds
$maxDuration = 300; // 5 mins
$newCall = $service->newCall($aNumber, $bNumber, $expiration, $maxDuration);
echo $newCall->getSessionId();
```

If the call is initiated, you can ask the result object for the session ID and use this session ID for an additional call to the `callStatus` or `tearDownCall()` methods. The second parameter on the `callStatus()` method call extends the expiration for this call.

Example 738. Call two numbers, ask for status, and cancel

```
$service = new Zend_Service_DeveloperGarden_VoiceCall($config);
$aNumber = '+49-30-000001';
$bNumber = '+49-30-000002';
$expiration  = 30; // seconds
$maxDuration = 300; // 5 mins

$newCall = $service->newCall($aNumber, $bNumber, $expiration, $maxDuration);

$sessionId = $newCall->getSessionId();

$service->callStatus($sessionId, true); // extend the call

sleep(10); // sleep 10s and then tearDown

$service->tearDownCall($sessionId);
```

19.8. ConferenceCall

Conference Call allows you to setup and start a phone conference.

The following features are available:

- Conferences with an immediate start
- Conferences with a defined start date
- Recurring conference series
- Adding, removing, and muting of participants from a conference
- Templates for conferences

Here is a list of currently implemented API methods:

- `createConference()` creates a new conference
- `updateConference()` updates an existing conference
- `commitConference()` saves the conference, and, if no date is configured, immediately starts the conference
- `removeConference()` removes a conference
- `getConferenceList()` returns a list of all configured conferences

- `getConferenceStatus()` displays information for an existing conference
- `getParticipantStatus()` displays status information about a conference participant
- `newParticipant()` creates a new participant
- `addParticipant()` adds a participant to a conference
- `updateParticipant()` updates a participant, usually to mute or redial the participant
- `removeParticipant()` removes a participant from a conference
- `getRunningConference()` requests the running instance of a planned conference
- `createConferenceTemplate()` creates a new conference template
- `getConferenceTemplate()` requests an existing conference template
- `updateConferenceTemplate()` updates existing conference template details
- `removeConferenceTemplate()` removes a conference template
- `getConferenceTemplateList()` requests all conference templates of an owner
- `addConferenceTemplateParticipant()` adds a conference participant to conference template
- `getConferenceTemplateParticipant()` displays details of a participant of a conference template
- `updateConferenceTemplateParticipant()` updates participant details within a conference template
- `removeConferenceTemplateParticipant()` removes a participant from a conference template

Example 739. Ad-Hoc conference

```
$client = new Zend_Service_DeveloperGarden_ConferenceCall($config);

$conferenceDetails = new Zend_Service_DeveloperGarden_ConferenceCall_ConferenceDetail(
    'Zend-Conference',                    // name for the conference
    'this is my private zend conference', // description
    60                                    // duration in seconds
);

$conference = $client->createConference('MyName', $conferenceDetails);

$part1 = new Zend_Service_DeveloperGarden_ConferenceCall_ParticipantDetail(
    'Jon',
    'Doe',
    '+49-123-4321',
    'your.name@example.com',
    true
);

$client->newParticipant($conference->getConferenceId(), $part1);
// add a second, third ... participant

$client->commitConference($conference->getConferenceId());
```

19.9. Performance and Caching

You can setup various caching options to improve the performance for resolving WSDL and authentication tokens.

First of all, you can setup the internal SoapClient (PHP) caching values.

Example 740. WSDL cache options

```
Zend_Service_DeveloperGarden_SecurityTokenServer_Cache::setWsdlCache([PHP CONSTANT]);
```

The `[PHP CONSTANT]` can be one of the following values:

- `WSDL_CACHE_DISC`: enabled disc caching
- `WSDL_CACHE_MEMORY`: enabled memory caching
- `WSDL_CACHE_BOTH`: enabled disc and memory caching
- `WSDL_CACHE_NONE`: disabled both caching

If you want also to cache the result for calls to the SecuritTokenServer you can setup a `Zend_Cache` instance and pass it to the `setCache()`.

Example 741. SecurityTokenServer cache option

```
$cache = Zend_Cache::factory('Core', ...);
Zend_Service_DeveloperGarden_SecurityTokenServer_Cache::setCache($cache);
```

20. Zend_Service_Flickr

20.1. Introduction

`Zend_Service_Flickr` is a simple API for using the Flickr REST Web Service. In order to use the Flickr web services, you must have an API key. To obtain a key and for more information about the Flickr REST Web Service, please visit the Flickr API Documentation.

In the following example, we use the `tagSearch()` method to search for photos having "php" in the tags.

Example 742. Simple Flickr Photo Search

```
$flickr = new Zend_Service_Flickr('MY_API_KEY');

$results = $flickr->tagSearch("php");

foreach ($results as $result) {
    echo $result->title . '<br />';
}
```

Optional parameter

`tagSearch()` accepts an optional second parameter as an array of options.

20.2. Finding Flickr Users' Photos and Information

`Zend_Service_Flickr` provides several ways to get information about Flickr users:

- `userSearch()`: Accepts a string query of space-delimited tags and an optional second parameter as an array of search options, and returns a set of photos as a `Zend_Service_Flickr_ResultSet` object.
- `getIdByUsername()`: Returns a string user ID associated with the given username string.
- `getIdByEmail()`: Returns a string user ID associated with the given email address string.

Example 743. Finding a Flickr User's Public Photos by E-Mail Address

In this example, we have a Flickr user's e-mail address, and we search for the user's public photos by using the `userSearch()` method:

```
$flickr = new Zend_Service_Flickr('MY_API_KEY');

$results = $flickr->userSearch($userEmail);

foreach ($results as $result) {
    echo $result->title . '<br />';
}
```

20.3. Finding photos From a Group Pool

`Zend_Service_Flickr` allows to retrieve a group's pool photos based on the group ID. Use the `groupPoolGetPhotos()` method:

Example 744. Retrieving a Group's Pool Photos by Group ID

```
$flickr = new Zend_Service_Flickr('MY_API_KEY');

    $results = $flickr->groupPoolGetPhotos($groupId);

    foreach ($results as $result) {
        echo $result->title . '<br />';
    }
```

Optional parameter

`groupPoolGetPhotos()` accepts an optional second parameter as an array of options.

20.4. Retrieving Flickr Image Details

`Zend_Service_Flickr` makes it quick and easy to get an image's details based on a given image ID. Just use the `getImageDetails()` method, as in the following example:

Example 745. Retrieving Flickr Image Details

Once you have a Flickr image ID, it is a simple matter to fetch information about the image:

```
$flickr = new Zend_Service_Flickr('MY_API_KEY');

$image = $flickr->getImageDetails($imageId);

echo "Image ID $imageId is $image->width x $image->height pixels.<br />\n";
echo "<a href=\"$image->clickUri\">Click for Image</a>\n";
```

20.5. Zend_Service_Flickr Result Classes

The following classes are all returned by `tagSearch()` and `userSearch()`:

- `Zend_Service_Flickr_ResultSet`
- `Zend_Service_Flickr_Result`
- `Zend_Service_Flickr_Image`

20.5.1. Zend_Service_Flickr_ResultSet

Represents a set of Results from a Flickr search.

Implements the `SeekableIterator` interface for easy iteration (e.g., using `foreach()`), as well as direct access to a specific result using `seek()`.

20.5.1.1. Properties

Table 134. Zend_Service_Flickr_ResultSet Properties

Name	Type	Description
totalResultsAvailable	int	Total Number of Results available
totalResultsReturned	int	Total Number of Results returned
firstResultPosition	int	The offset in the total result set of this result set

20.5.1.2. Zend_Service_Flickr_ResultSet::totalResults()

```
int totalResults();
```

Returns the total number of results in this result set.

Back to Class List

20.5.2. Zend_Service_Flickr_Result

A single Image result from a Flickr query

20.5.2.1. Properties

Table 135. Zend_Service_Flickr_Result Properties

Name	Type	Description
id	string	Image ID
owner	string	The photo owner's NSID.
secret	string	A key used in url construction.
server	string	The servername to use for URL construction.

Name	Type	Description
title	string	The photo's title.
ispublic	string	The photo is public.
isfriend	string	The photo is visible to you because you are a friend of the owner.
isfamily	string	The photo is visible to you because you are family of the owner.
license	string	The license the photo is available under.
dateupload	string	The date the photo was uploaded.
datetaken	string	The date the photo was taken.
ownername	string	The screenname of the owner.
iconserver	string	The server used in assembling icon URLs.
Square	Zend_Service_Flickr_Image	A 75x75 thumbnail of the image.
Thumbnail	Zend_Service_Flickr_Image	A 100 pixel thumbnail of the image.
Small	Zend_Service_Flickr_Image	A 240 pixel version of the image.
Medium	Zend_Service_Flickr_Image	A 500 pixel version of the image.
Large	Zend_Service_Flickr_Image	A 640 pixel version of the image.
Original	Zend_Service_Flickr_Image	The original image.

Back to Class List

20.5.3. Zend_Service_Flickr_Image

Represents an Image returned by a Flickr search.

20.5.3.1. Properties

Table 136. Zend_Service_Flickr_Image Properties

Name	Type	Description
uri	string	URI for the original image
clickUri	string	Clickable URI (i.e. the Flickr page) for the image
width	int	Width of the Image
height	int	Height of the Image

Back to Class List

21. Zend_Service_LiveDocx

21.1. Introduction to LiveDocx

LiveDocx is a SOAP service that allows developers to generate word processing documents by combining structured data from PHP with a template, created in a word processor. The resulting document can be saved as a PDF, DOCX, DOC, HTML or RTF file. LiveDocx implements mail-merge in PHP.

The family of `Zend_Service_LiveDocx` components provides a clean and simple interface to the LiveDocx API and additionally offers functionality to improve network performance.

In addition to this section of the manual, if you are interested in learning more about `Zend_Service_LiveDocx` and the backend SOAP service LiveDocx, please take a look at the following resources:

- *Shipped demonstration applications*. There are a large number of demonstration applications in the directory */demos/Zend/Service/LiveDocx* of the Zend Framework distribution file or trunk version, checked out of the standard SVN repository. These are designed to get you up to speed with `Zend_Service_LiveDocx` within a matter of minutes.
- `Zend_Service_LiveDocx` blog and web site.
- LiveDocx SOAP API documentation.
- LiveDocx WSDL.
- LiveDocx blog and web site.

21.1.1. Sign Up for an Account

Before you can start using LiveDocx, you must first sign up for an account. The account is completely free of charge and you only need to specify a *username*, *password* and *e-mail address*. Your login credentials will be dispatched to the e-mail address you supply, so please type carefully.

21.1.2. Templates and Documents

LiveDocx differentiates between the following terms: 1) *template* and 2) *document*. In order to fully understand the documentation and indeed the actual API, it is important that any programmer deploying LiveDocx understands the difference.

The term *template* is used to refer to the input file, created in a word processor, containing formatting and text fields. You can download an example template, stored as a DOCX file. The term *document* is used to refer to the output file that contains the template file, populated with data - i.e. the finished document. You can download an example document, stored as a PDF file.

21.1.3. Supported File Formats

LiveDocx supports the following file formats:

21.1.3.1. Template File Formats (input)

Templates can be saved in any of the following file formats:

- DOCX - Office Open XML format
- DOC - Microsoft Word DOC format
- RTF - Rich text file format
- TXD - TX Text Control format

21.1.3.2. Document File Formats (output):

The resulting document can be saved in any of the following file formats:

- DOCX - Office Open XML format
- DOC - Microsoft Word DOC format
- HTML - XHTML 1.0 transitional format
- RTF - Rich text file format
- PDF - Acrobat Portable Document Format
- TXD - TX Text Control format
- TXT - ANSI plain text

21.1.3.3. Image File Formats (output):

The resulting document can be saved in any of the following graphical file formats:

- BMP - Bitmap image format
- GIF - Graphics Interchange Format
- JPG - Joint Photographic Experts Group format
- PNG - Portable Network Graphics format
- TIFF - Tagged Image File Format
- WMF - Windows Meta File format

21.2. Zend_Service_LiveDocx_MailMerge

`Zend_Service_LiveDocx_MailMerge` is the mail-merge object in the `Zend_Service_LiveDocx` family.

21.2.1. Document Generation Process

The document generation process can be simplified with the following equation:

Template + *Data* = *Document*

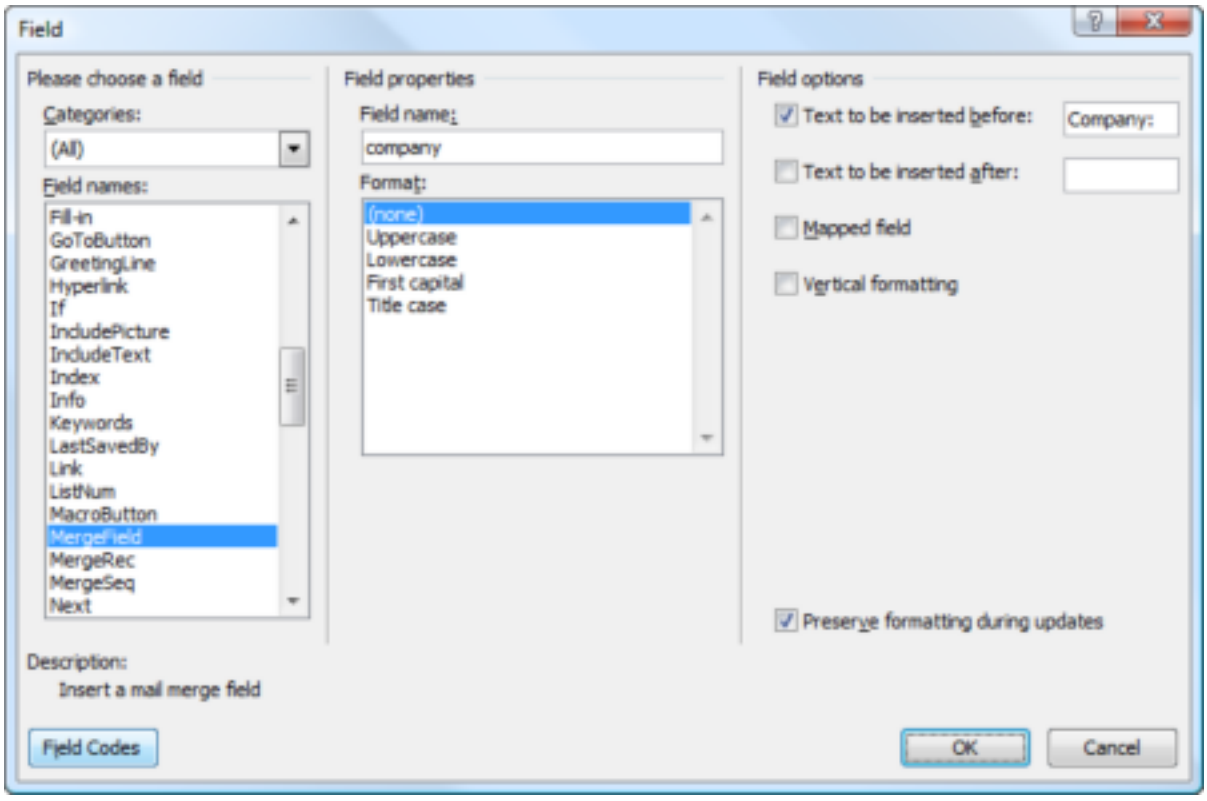
Or expressed by the following diagram:

Data is inserted into template to create a document (view larger).

A template, created in a word processing application, such as Microsoft Word, is loaded into LiveDocx. Data is then inserted into the template and the resulting document is saved to any supported format.

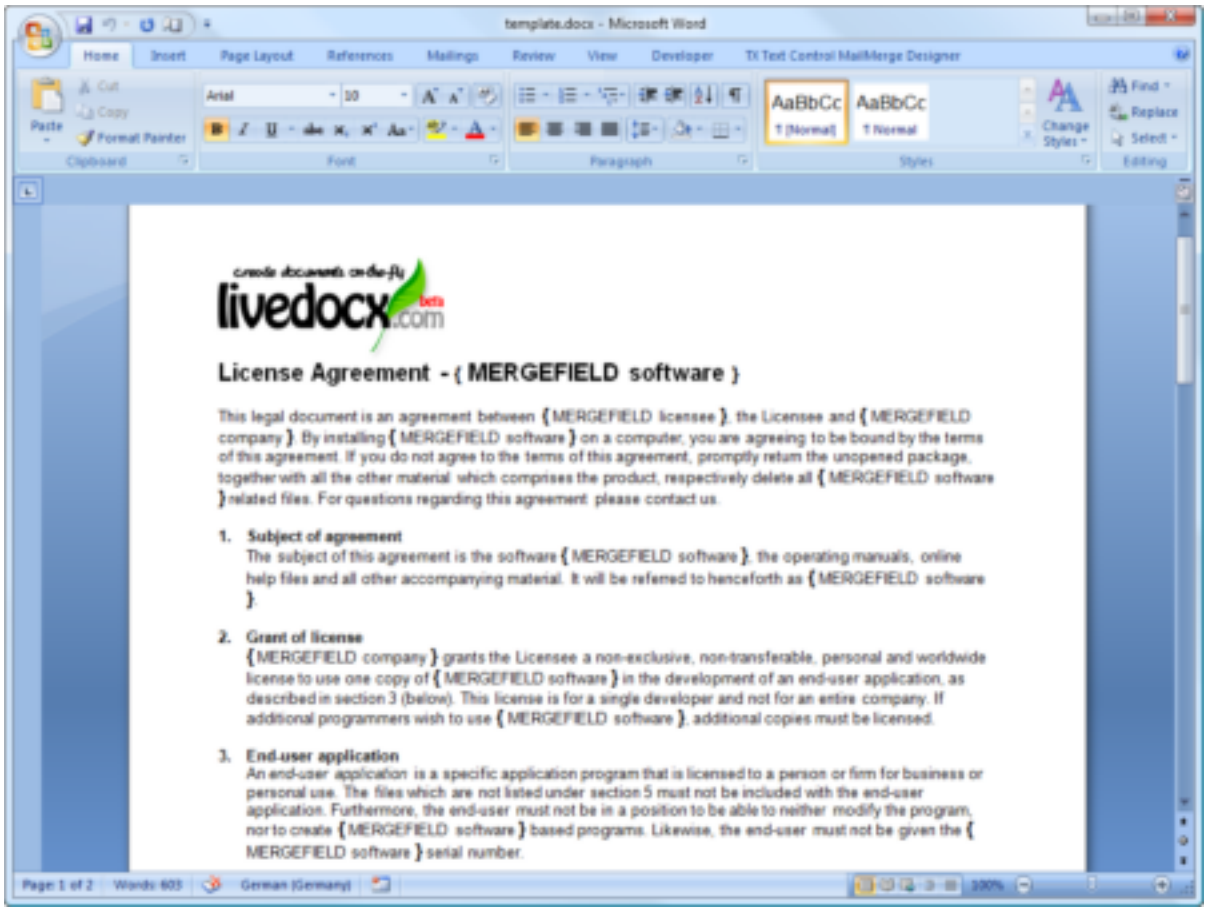
21.2.2. Creating Templates in Microsoft Word 2007

Start off by launching Microsoft Word and creating a new document. Next, open up the *Field* dialog box. This looks as follows:



Microsoft Word 2007 Field dialog box (view larger).

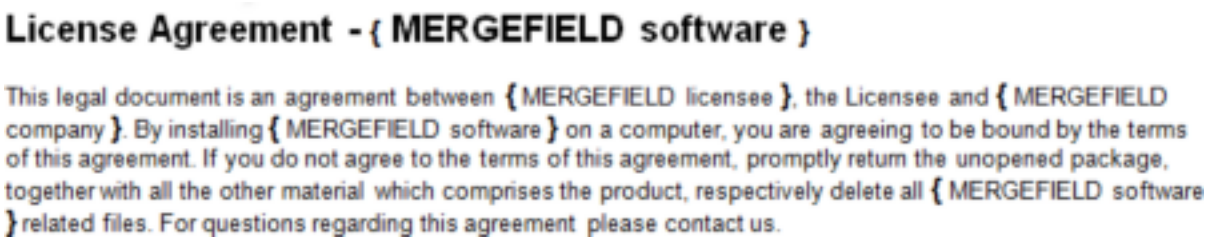
Using this dialog, you can insert the required merge fields into your document. Below is a screenshot of a license agreement in Microsoft Word 2007. The merge fields are marked as `{ MERGEFIELD FieldName }`:



Template in Microsoft Word 2007 (view larger).

Now, save the template as *template.docx*.

In the next step, we are going to populate the merge fields with textual data from PHP.



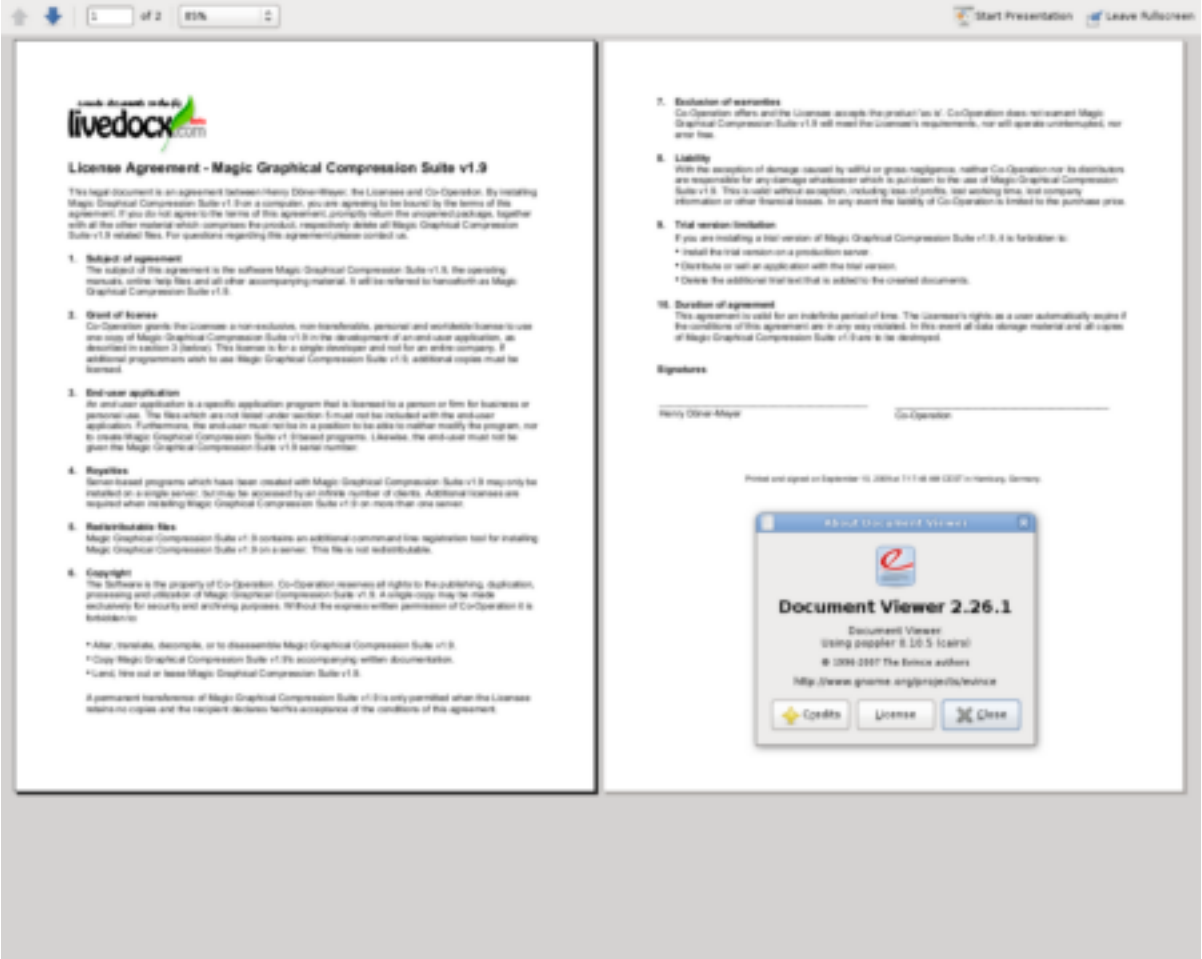
Cropped template in Microsoft Word 2007 (view larger).

To populate the merge fields in the above cropped screenshot of the template in Microsoft Word, all we have to code is as follows:

```
$phpLiveDocx = new Zend_Service_LiveDocx_MailMerge();

$phpLiveDocx->setUsername('myUsername')
            ->setPassword('myPassword');

$phpLiveDocx->setLocalTemplate('template.docx');
```

```
$phpLiveDocx->assign('software', 'Magic Graphical Compression Suite v1.9')
            ->assign('licensee', 'Henry Döner-Meyer')
            ->assign('company',  'Co-Operation');

$phpLiveDocx->createDocument();

$document = $phpLiveDocx->retrieveDocument('pdf');

file_put_contents('document.pdf', $document);
```

The resulting document is written to disk in the file *document.pdf*. This file can now be post-processed, sent via e-mail or simply displayed, as is illustrated below in *Document Viewer 2.26.1* on *Ubuntu 9.04*:



Resulting document as PDF in Document Viewer 2.26.1 (view larger).

21.2.3. Advanced Mail-Merge

`Zend_Service_LiveDocx_MailMerge` allows designers to insert any number of text fields into a template. These text fields are populated with data when *createDocument()* is called.

In addition to text fields, it is also possible specify regions of a document, which should be repeated.

For example, in a telephone bill it is necessary to print out a list of all connections, including the destination number, duration and cost of each call. This repeating row functionality can be achieved with so called blocks.

Blocks are simply regions of a document, which are repeated when `createDocument()` is called. In a block any number of *block fields* can be specified.

Blocks consist of two consecutive document targets with a unique name. The following screenshot illustrates these targets and their names in red:

(view larger).

The format of a block is as follows:

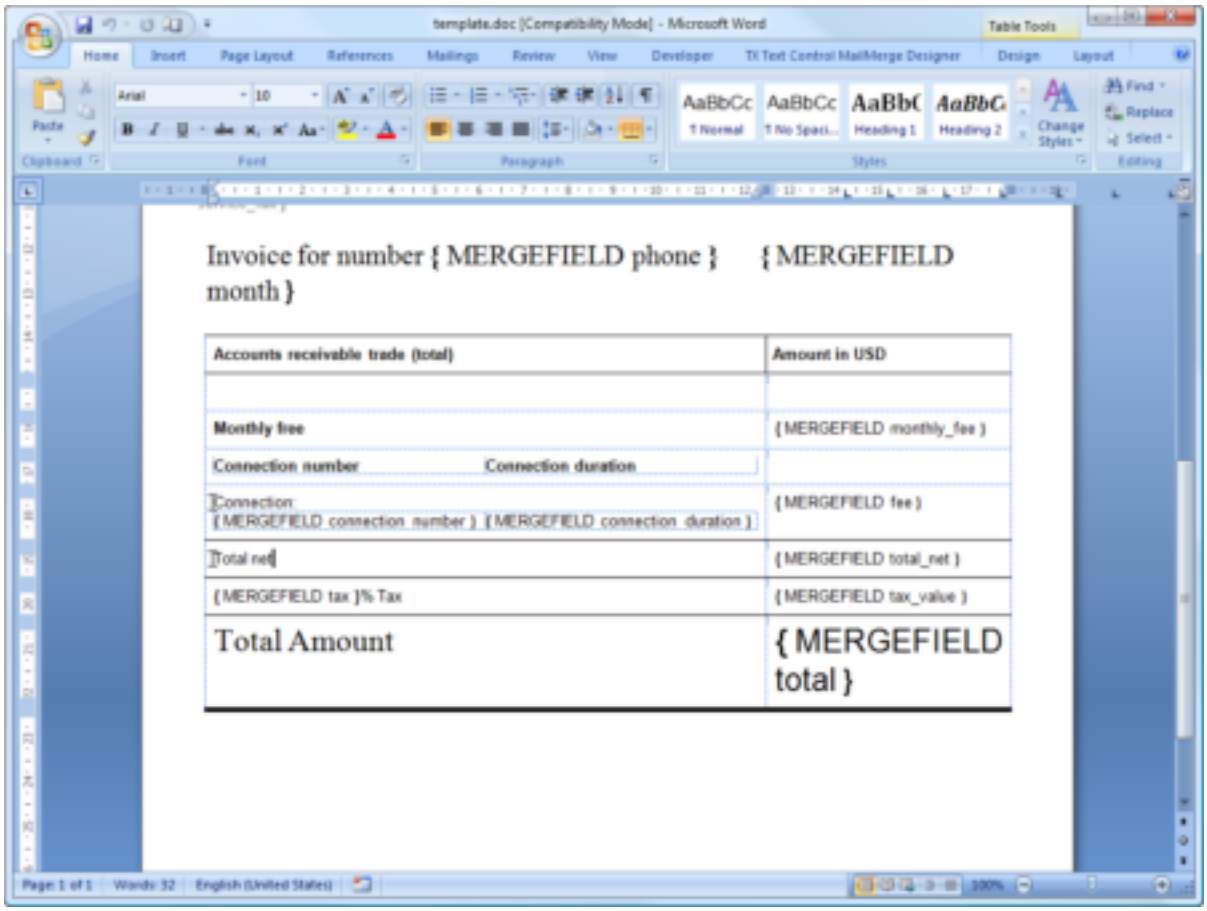
```
blockStart_ + unique name
blockEnd_ + unique name
```

For example:

```
blockStart_block1
blockEnd_block1
```

The content of a block is repeated, until all data assigned in the block fields has been injected into the template. The data for block fields is specified in PHP as a multi-assoc array.

The following screenshot of a template in Microsoft Word 2007 shows how block fields are used:



Template, illustrating blocks in Microsoft Word 2007 (view larger).

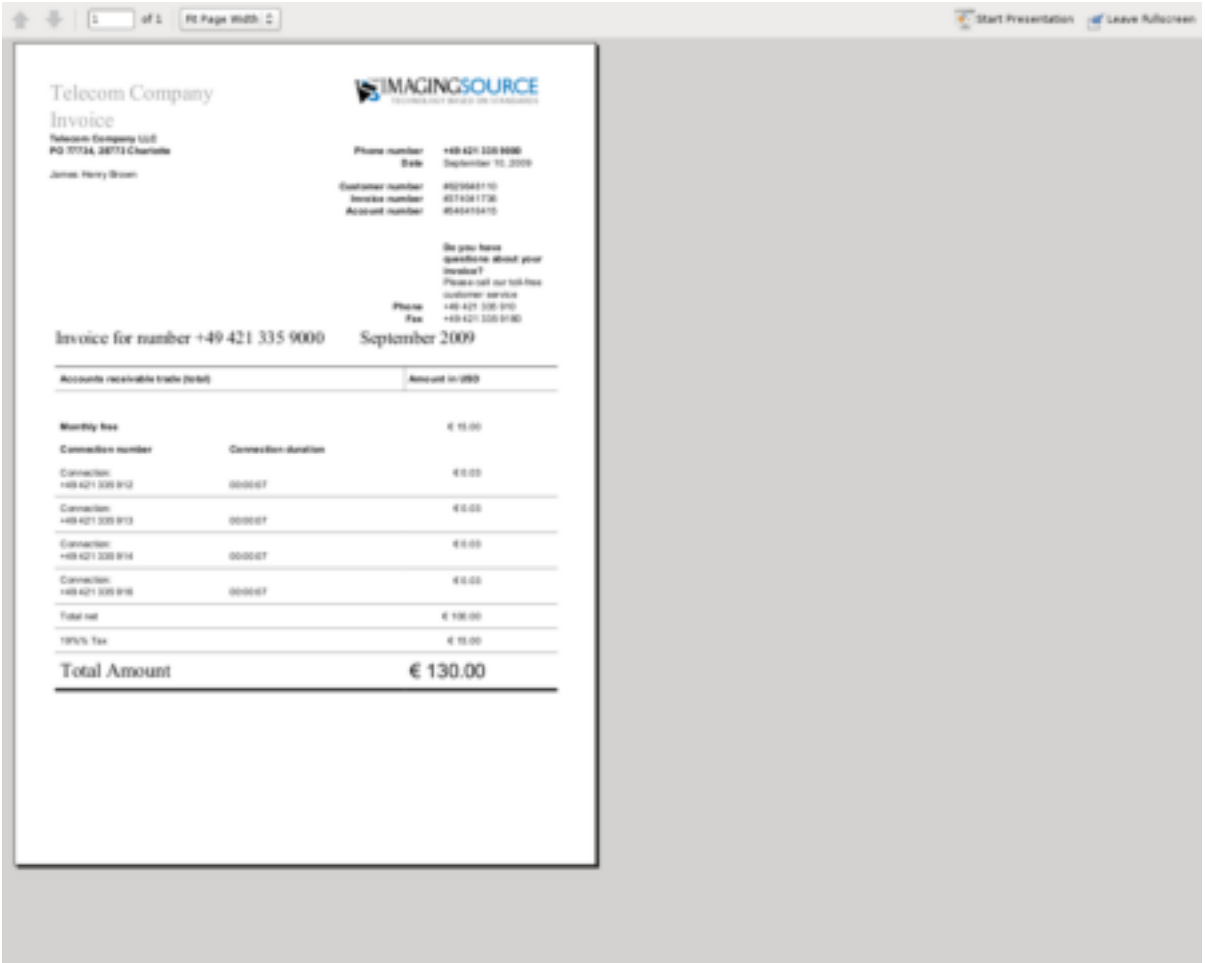
The following code populates the above template with data.

```
$phpLiveDocx = new Zend_Service_LiveDocx_MailMerge();

$phpLiveDocx->setUsername('myUsername')
            ->setPassword('myPassword');

$phpLiveDocx->setLocalTemplate('template.doc');

$billConnections = array(
    array(
        'connection_number'   => '+49 421 335 912',
        'connection_duration' => '00:00:07',
        'fee'                 => '€ 0.03',
    ),
    array(
        'connection_number'   => '+49 421 335 913',
        'connection_duration' => '00:00:07',
        'fee'                 => '€ 0.03',
    ),
    array(
        'connection_number'   => '+49 421 335 914',
        'connection_duration' => '00:00:07',
        'fee'                 => '€ 0.03',
    ),
```

```
    array(
        'connection_number'   => '+49 421 335 916',
        'connection_duration' => '00:00:07',
        'fee'                 => '€ 0.03',
    ),
);

$phpLiveDocx->assign('connection', $billConnections);

// ... assign other data here ...

$phpLiveDocx->createDocument();
$document = $phpLiveDocx->retrieveDocument('pdf');
file_put_contents('document.pdf', $document);
```

The data, which is specified in the array `$billConnections` is repeated in the template in the block connection. The keys of the array (`connection_number`, `connection_duration` and `fee`) are the block field names - their data is inserted, one row per iteration.

The resulting document is written to disk in the file *document.pdf*. This file can now be post-processed, sent via e-mail or simply displayed, as is illustrated below in *Document Viewer 2.26.1* on *Ubuntu 9.04*:



Resulting document as PDF in Document Viewer 2.26.1 (view larger).

You can download the DOC template file and the resulting PDF document.

NOTE: blocks may not be nested.

21.2.4. Generating bitmaps image files

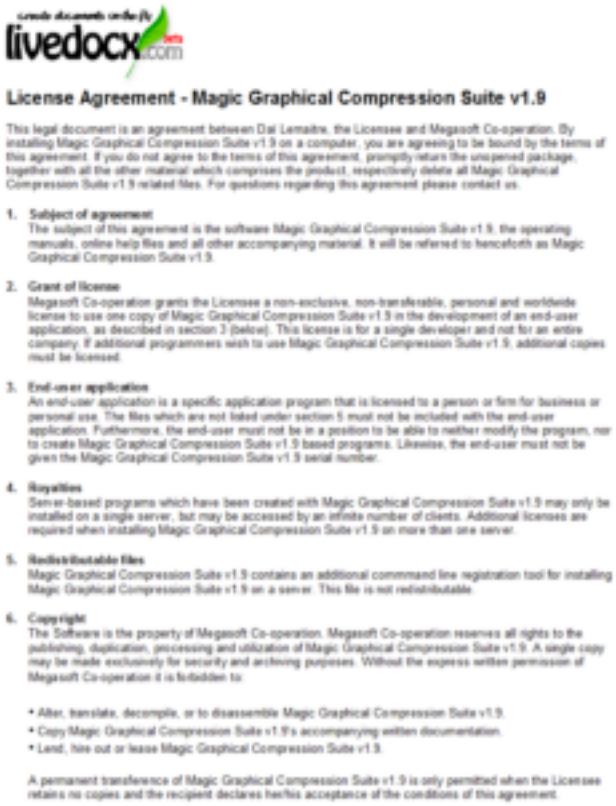
In addition to document file formats, `Zend_Service_LiveDocx_MailMerge` also allows documents to be saved to a number of image file formats (BMP, GIF, JPG, PNG and TIFF). Each page of the document is saved to one file.

The following sample illustrates the use of `getBitmaps($fromPage, $toPage, $zoomFactor, $format)` and `getAllBitmaps($zoomFactor, $format)`.

`$fromPage` is the lower-bound page number of the page range that should be returned as an image and `$toPage` the upper-bound page number. `$zoomFactor` is the size of the images, as a percent, relative to the original page size. The range of this parameter is 10 to 400. `$format` is the format of the images returned by this method. The supported formats can be obtained by calling `getImageFormats()`.

```
$date = new Zend_Date();
$date->setLocale('en_US');

$phpLiveDocx = new Zend_Service_LiveDocx_MailMerge();

$phpLiveDocx->setUsername('myUsername')
            ->setPassword('myPassword');

$phpLiveDocx->setLocalTemplate('template.docx');

$phpLiveDocx->assign('software', 'Magic Graphical Compression Suite v1.9')
            ->assign('licensee', 'Daï Lemaitre')
            ->assign('company',  'Megasoft Co-operation')
            ->assign('date',     $date->get(Zend_Date::DATE_LONG))
            ->assign('time',     $date->get(Zend_Date::TIME_LONG))
            ->assign('city',     'Lyon')
            ->assign('country',  'France');

$phpLiveDocx->createDocument();

// Get all bitmaps
// (zoomFactor, format)
$bitmaps = $phpLiveDocx->getAllBitmaps(100, 'png');

// Get just bitmaps in specified range
// (fromPage, toPage, zoomFactor, format)
// $bitmaps = $phpLiveDocx->getBitmaps(2, 2, 100, 'png');

foreach ($bitmaps as $pageNumber => $bitmapData) {
    $filename = sprintf('documentPage%d.png', $pageNumber);
    file_put_contents($filename, $bitmapData);
}
```

This produces two files (`documentPage1.png` and `documentPage2.png`) and writes them to disk in the same directory as the executable PHP file.



livedocx.com

License Agreement - Magic Graphical Compression Suite v1.9

This legal document is an agreement between Dal Lemaitre, the Licensee and Megasoft Co-operation. By installing Magic Graphical Compression Suite v1.9 on a computer, you are agreeing to be bound by the terms of this agreement. If you do not agree to the terms of this agreement, promptly return the unopened package, together with all the other material which comprises the product, respectively delete all Magic Graphical Compression Suite v1.9 related files. For questions regarding this agreement please contact us.

1. Subject of agreement
The subject of this agreement is the software Magic Graphical Compression Suite v1.9, the operating manuals, online help files and all other accompanying material. It will be referred to henceforth as Magic Graphical Compression Suite v1.9.

2. Grant of license
Megasoft Co-operation grants the Licensee a non-exclusive, non-transferable, personal and worldwide license to use one copy of Magic Graphical Compression Suite v1.9 in the development of an end-user application, as described in section 3 (below). This license is for a single developer and not for an entire company. If additional programmers wish to use Magic Graphical Compression Suite v1.9, additional copies must be licensed.

3. End-user application
An end-user application is a specific application program that is licensed to a person or firm for business or personal use. The files which are not listed under section 5 must not be included with the end-user application. Furthermore, the end-user must not be in a position to be able to neither modify the program, nor to create Magic Graphical Compression Suite v1.9 based programs. Likewise, the end-user must not be given the Magic Graphical Compression Suite v1.9 serial number.

4. Royalties
Server-based programs which have been created with Magic Graphical Compression Suite v1.9 may only be installed on a single server, but may be accessed by an infinite number of clients. Additional licenses are required when installing Magic Graphical Compression Suite v1.9 on more than one server.

5. Redistributable files
Magic Graphical Compression Suite v1.9 contains an additional command line registration tool for installing Magic Graphical Compression Suite v1.9 on a server. This file is not redistributable.

6. Copyright
The Software is the property of Megasoft Co-operation. Megasoft Co-operation reserves all rights to the publishing, duplication, processing and utilization of Magic Graphical Compression Suite v1.9. A single copy may be made exclusively for security and archiving purposes. Without the express written permission of Megasoft Co-operation it is forbidden to:

- Alter, translate, decompile, or to disassemble Magic Graphical Compression Suite v1.9.
- Copy Magic Graphical Compression Suite v1.9's accompanying written documentation.
- Lend, hire out or lease Magic Graphical Compression Suite v1.9.

A permanent transference of Magic Graphical Compression Suite v1.9 is only permitted when the Licensee retains no copies and the recipient declares her/his acceptance of the conditions of this agreement.

documentPage1.png (view larger).

7. **Exclusion of warranties**
Megasoft Co-operation offers and the Licensee accepts the product 'as is'. Megasoft Co-operation does not warrant Magic Graphical Compression Suite v1.9 will meet the Licensee's requirements, nor will operate uninterrupted, nor error free.

8. **Liability**
With the exception of damage caused by willful or gross negligence, neither Megasoft Co-operation nor its distributors are responsible for any damage whatsoever which is put down to the use of Magic Graphical Compression Suite v1.9. This is valid without exception, including loss of profits, lost working time, lost company information or other financial losses. In any event the liability of Megasoft Co-operation is limited to the purchase price.

9. **Trial version limitation**
If you are installing a trial version of Magic Graphical Compression Suite v1.9, it is forbidden to:
- Install the trial version on a production server.
- Distribute or sell an application with the trial version.
- Delete the additional trial text that is added to the created documents.

10. **Duration of agreement**
This agreement is valid for an indefinite period of time. The Licensee's rights as a user automatically expire if the conditions of this agreement are in any way violated. In this event all data storage material and all copies of Magic Graphical Compression Suite v1.9 are to be destroyed.

Signatures

Dat Lemaitre

Megasoft Co-operation

Printed and signed on December 2, 2009 at 6:34:57 AM CET in Lyon, France.

documentPage2.png (view larger).

21.2.5. Local vs. Remote Templates

Templates can be stored *locally*, on the client machine, or *remotely*, on the server. There are advantages and disadvantages to each approach.

In the case that a template is stored locally, it must be transfered from the client to the server on every request. If the content of the template rarely changes, this approach is inefficient. Similarly, if the template is several megabytes in size, it may take considerable time to transfer it to the server. Local template are useful in situations in which the content of the template is constantly changing.

The following code illustrates how to use a local template.

```
$phpLiveDocx = new Zend_Service_LiveDocx_MailMerge();

$phpLiveDocx->setUsername('myUsername')
            ->setPassword('myPassword');

$phpLiveDocx->setLocalTemplate('./template.docx');

// assign data and create document
```

In the case that a template is stored remotely, it is uploaded once to the server and then simply referenced on all subsequent requests. Obviously, this is much quicker than using a local template, as the template does not have to be transfered on every request. For speed critical applications, it is recommended to use the remote template method.

The following code illustrates how to upload a template to the server:

```
$phpLiveDocx = new Zend_Service_LiveDocx_MailMerge();

$phpLiveDocx->setUsername('myUsername')
            ->setPassword('myPassword');

$phpLiveDocx->uploadTemplate('template.docx');
```

The following code illustrates how to reference the remotely stored template on all subsequent requests:

```
$phpLiveDocx = new Zend_Service_LiveDocx_MailMerge();

$phpLiveDocx->setUsername('myUsername')
            ->setPassword('myPassword');

$phpLiveDocx->setRemoteTemplate('template.docx');

// assign data and create document
```

21.2.6. Getting Information

`Zend_Service_LiveDocx_MailMerge` provides a number of methods to get information on field names, available fonts and supported formats.

Example 746. Get Array of Field Names in Template

The following code returns and displays an array of all field names in the specified template. This functionality is useful, in the case that you create an application, in which an end-user can update a template.

```
$phpLiveDocx = new Zend_Service_LiveDocx_MailMerge();

$phpLiveDocx->setUsername('myUsername')
            ->setPassword('myPassword');

$templateName = 'template-1-text-field.docx';
$phpLiveDocx->setLocalTemplate($templateName);

$fieldNames = $phpLiveDocx->getFieldNames();
foreach ($fieldNames as $fieldName) {
    printf('- %s%s', $fieldName, PHP_EOL);
}
```

Example 747. Get Array of Block Field Names in Template

The following code returns and displays an array of all block field names in the specified template. This functionality is useful, in the case that you create an application, in which an end-user can update a template. Before such templates can be populated, it is necessary to find out the names of the contained block fields.

```
$phpLiveDocx = new Zend_Service_LiveDocx_MailMerge();

$phpLiveDocx->setUsername('myUsername')
            ->setPassword('myPassword');

$templateName = 'template-block-fields.doc';
$phpLiveDocx->setLocalTemplate($templateName);

$blockNames = $phpLiveDocx->getBlockNames();
foreach ($blockNames as $blockName) {
    $blockFieldNames = $phpLiveDocx->getBlockFieldNames($blockName);
    foreach ($blockFieldNames as $blockFieldName) {
        printf('- %s::%s%s', $blockName, $blockFieldName, PHP_EOL);
    }
}
```

Example 748. Get Array of Fonts Installed on Server

The following code returns and displays an array of all fonts installed on the server. You can use this method to present a list of fonts which may be used in a template. It is important to inform the end-user about the fonts installed on the server, as only these fonts may be used in a template. In the case that a template contains fonts, which are not available on the server, font-substitution will take place. This may lead to undesirable results.

```
$phpLiveDocx = new Zend_Service_LiveDocx_MailMerge();

$phpLiveDocx->setUsername('myUsername')
            ->setPassword('myPassword');

Zend_Debug::dump($phpLiveDocx->getFontNames());
```

NOTE: As the return value of this method changes very infrequently, it is highly recommended to use a cache, such as `Zend_Cache` - this will considerably speed up your application.

Example 749. Get Array of Supported Template File Formats

The following code returns and displays an array of all supported template file formats. This method is particularly useful in the case that a combo list should be displayed that allows the end-user to select the input format of the documentation generation process.

```
$phpLiveDocx = new Zend_Service_LiveDocx_MailMerge()

$phpLiveDocx->setUsername('myUsername')
            ->setPassword('myPassword');

Zend_Debug::dump($phpLiveDocx->getTemplateFormats());
```

NOTE: As the return value of this method changes very infrequently, it is highly recommended to use a cache, such as `Zend_Cache` - this will considerably speed up your application.

Example 750. Get Array of Supported Document File Formats

The following code returns and displays an array of all supported document file formats. This method is particularly useful in the case that a combo list should be displayed that allows the end-user to select the output format of the documentation generation process.

```
$phpLiveDocx = new Zend_Service_LiveDocx_MailMerge();

$phpLiveDocx->setUsername('myUsername')
            ->setPassword('myPassword');

Zend_Debug::dump($phpLiveDocx->getDocumentFormats());
```

Example 751. Get Array of Supported Image File Formats

The following code returns and displays an array of all supported image file formats. This method is particularly useful in the case that a combo list should be displayed that allows the end-user to select the output format of the documentation generation process.

```
$phpLiveDocx = new Zend_Service_LiveDocx_MailMerge();

$phpLiveDocx->setUsername('myUsername')
            ->setPassword('myPassword');

Zend_Debug::dump($phpLiveDocx->getImageFormats());
```

NOTE: As the return value of this method changes very infrequently, it is highly recommended to use a cache, such as `Zend_Cache` - this will considerably speed up your application.

22. Zend_Service_Nirvanix

22.1. Introduction

Nirvanix provides an Internet Media File System (IMFS), an Internet storage service that allows applications to upload, store and organize files and subsequently access them using a standard Web Services interface. An IMFS is distributed clustered file system, accessed over the Internet, and optimized for dealing with media files (audio, video, etc). The goal of an IMFS is to provide massive scalability to deal with the challenges of media storage growth, with guaranteed access and availability regardless of time and location. Finally, an IMFS gives applications the ability to access data securely, without the large fixed costs associated with acquiring and maintaining physical storage assets.

22.2. Registering with Nirvanix

Before you can get started with `Zend_Service_Nirvanix`, you must first register for an account. Please see the Getting Started page on the Nirvanix website for more information.

After registering, you will receive a Username, Password, and Application Key. All three are required to use `Zend_Service_Nirvanix`.

22.3. API Documentation

Access to the Nirvanix IMFS is available through both SOAP and a faster REST service. `Zend_Service_Nirvanix` provides a relatively thin PHP 5 wrapper around the REST service.

`Zend_Service_Nirvanix` aims to make using the Nirvanix REST service easier but understanding the service itself is still essential to be successful with Nirvanix.

The Nirvanix API Documentation provides an overview as well as detailed information using the service. Please familiarize yourself with this document and refer back to it as you use `Zend_Service_Nirvanix`.

22.4. Features

Nirvanix's REST service can be used effectively with PHP using the SimpleXML extension and `Zend_Http_Client` alone. However, using it this way is somewhat inconvenient due to repetitive operations like passing the session token on every request and repeatedly checking the response body for error codes.

`Zend_Service_Nirvanix` provides the following functionality:

- A single point for configuring your Nirvanix authentication credentials that can be used across the Nirvanix namespaces.
- A proxy object that is more convenient to use than an HTTP client alone, mostly removing the need to manually construct HTTP POST requests to access the REST service.
- A response wrapper that parses each response body and throws an exception if an error occurred, alleviating the need to repeatedly check the success of many commands.
- Additional convenience methods for some of the more common operations.

22.5. Getting Started

Once you have registered with Nirvanix, you're ready to store your first file on the IMFS. The most common operations that you will need to do on the IMFS are creating a new file, downloading an existing file, and deleting a file. `Zend_Service_Nirvanix` provides convenience methods for these three operations.

```
$auth = array('username' => 'your-username',
              'password' => 'your-password',
              'appKey'   => 'your-app-key');

$nirvanix = new Zend_Service_Nirvanix($auth);
$imfs = $nirvanix->getService('IMFS');

$imfs->putContents('/foo.txt', 'contents to store');

echo $imfs->getContents('/foo.txt');

$imfs->unlink('/foo.txt');
```

The first step to using `Zend_Service_Nirvanix` is always to authenticate against the service. This is done by passing your credentials to the `Zend_Service_Nirvanix` constructor above. The associative array is passed directly to Nirvanix as POST parameters.

Nirvanix divides its web services into namespaces. Each namespace encapsulates a group of related operations. After getting an instance of `Zend_Service_Nirvanix`, call the `getService()` method to create a proxy for the namespace you want to use. Above, a proxy for the `IMFS` namespace is created.

After you have a proxy for the namespace you want to use, call methods on it. The proxy will allow you to use any command available on the REST API. The proxy may also make convenience methods available, which wrap web service commands. The example above shows using the IMFS convenience methods to create a new file, retrieve and display that file, and finally delete the file.

22.6. Understanding the Proxy

In the previous example, we used the `getService()` method to return a proxy object to the `IMFS` namespace. The proxy object allows you to use the Nirvanix REST service in a way that's closer to making a normal PHP method call, as opposed to constructing your own HTTP request objects.

A proxy object may provide convenience methods. These are methods that the `Zend_Service_Nirvanix` provides to simplify the use of the Nirvanix web services. In the previous example, the methods `putContents()`, `getContents()`, and `unlink()` do not have direct equivalents in the REST API. They are convenience methods provided by `Zend_Service_Nirvanix` that abstract more complicated operations on the REST API.

For all other method calls to the proxy object, the proxy will dynamically convert the method call to the equivalent HTTP POST request to the REST API. It does this by using the method name as the API command, and an associative array in the first argument as the POST parameters.

Let's say you want to call the REST API method RenameFile, which does not have a convenience method in `Zend_Service_Nirvanix`:

```
$auth = array('username' => 'your-username',
              'password' => 'your-password',
              'appKey'   => 'your-app-key');

$nirvanix = new Zend_Service_Nirvanix($auth);
$imfs = $nirvanix->getService('IMFS');

$result = $imfs->renameFile(array('filePath' => '/path/to/foo.txt',
                                  'newFileName' => 'bar.txt'));
```

Above, a proxy for the `IMFS` namespace is created. A method, `renameFile()`, is then called on the proxy. This method does not exist as a convenience method in the PHP code, so it is trapped by `__call()` and converted into a POST request to the REST API where the associative array is used as the POST parameters.

Notice in the Nirvanix API documentation that `sessionToken` is required for this method but we did not give it to the proxy object. It is added automatically for your convenience.

The result of this operation will either be a `Zend_Service_Nirvanix_Response` object wrapping the XML returned by Nirvanix, or a `Zend_Service_Nirvanix_Exception` if an error occurred.

22.7. Examining Results

The Nirvanix REST API always returns its results in XML. `Zend_Service_Nirvanix` parses this XML with the `SimpleXML` extension and then decorates the resulting `SimpleXMLElement` with a `Zend_Service_Nirvanix_Response` object.

The simplest way to examine a result from the service is to use the built-in PHP functions like `print_r()`:

```
<?php
$auth = array('username' => 'your-username',
              'password' => 'your-password',
              'appKey'   => 'your-app-key');

$nirvanix = new Zend_Service_Nirvanix($auth);
$imfs = $nirvanix->getService('IMFS');

$result = $imfs->putContents('/foo.txt', 'fourteen bytes');
print_r($result);
?>
Zend_Service_Nirvanix_Response Object
(
    [_sxml:protected] => SimpleXMLElement Object
        (
            [ResponseCode] => 0
            [FilesUploaded] => 1
            [BytesUploaded] => 14
        )
)
```

You can access any property or method of the decorated `SimpleXMLElement`. In the above example, `$result->BytesUploaded` could be used to see the number of bytes received. Should you want to access the `SimpleXMLElement` directly, just use `$result->getSxml()`.

The most common response from Nirvanix is success (`ResponseCode` of zero). It is not normally necessary to check `ResponseCode` because any non-zero result will throw a `Zend_Service_Nirvanix_Exception`. See the next section on handling errors.

22.8. Handling Errors

When using Nirvanix, it's important to anticipate errors that can be returned by the service and handle them appropriately.

All operations against the REST service result in an XML return payload that contains a `ResponseCode` element, such as the following example:

```
<Response>
   <ResponseCode>0</ResponseCode>
</Response>
```

When the `ResponseCode` is zero such as in the example above, the operation was successful. When the operation is not successful, the `ResponseCode` is non-zero and an `ErrorMessage` element should be present.

To alleviate the need to repeatedly check if the `ResponseCode` is non-zero, `Zend_Service_Nirvanix` automatically checks each response returned by Nirvanix. If the `ResponseCode` indicates an error, a `Zend_Service_Nirvanix_Exception` will be thrown.

```
$auth = array('username' => 'your-username',
              'password' => 'your-password',
              'appKey'   => 'your-app-key');
$nirvanix = new Zend_Service_Nirvanix($auth);

try {

  $imfs = $nirvanix->getService('IMFS');
```

```
  $imfs->unlink('/a-nonexistant-path');

} catch (Zend_Service_Nirvanix_Exception $e) {
  echo $e->getMessage() . "\n";
  echo $e->getCode();
}
```

In the example above, `unlink()` is a convenience method that wraps the `DeleteFiles` command on the REST API. The `filePath` parameter required by the DeleteFiles command contains a path that does not exist. This will result in a `Zend_Service_Nirvanix` exception being thrown with the message "Invalid path" and code 70005.

The Nirvanix API Documentation describes the errors associated with each command. Depending on your needs, you may wrap each command in a `try` block or wrap many commands in the same `try` block for convenience.

23. Zend_Service_ReCaptcha

23.1. Introduction

`Zend_Service_ReCaptcha` provides a client for the reCAPTCHA Web Service. Per the reCAPTCHA site, "reCAPTCHA is a free CAPTCHA service that helps to digitize books." Each reCAPTCHA requires the user to input two words, the first of which is the actual CAPTCHA, and the second of which is a word from some scanned text that Optical Character Recognition (OCR) software has been unable to identify. The assumption is that if a user correctly provides the first word, the second is likely correctly entered as well, and can be used to improve OCR software for digitizing books.

In order to use the reCAPTCHA service, you will need to sign up for an account and register one or more domains with the service in order to generate public and private keys.

23.2. Simplest use

Instantiate a `Zend_Service_ReCaptcha` object, passing it your public and private keys:

Example 752. Creating an instance of the reCAPTCHA service

```
$recaptcha = new Zend_Service_ReCaptcha($pubKey, $privKey);
```

To render the reCAPTCHA, simply call the `getHTML()` method:

Example 753. Displaying the reCAPTCHA

```
echo $recaptcha->getHTML();
```

When the form is submitted, you should receive two fields, 'recaptcha_challenge_field' and 'recaptcha_response_field'. Pass these to the reCAPTCHA object's `verify()` method:

Example 754. Verifying the form fields

```
$result = $recaptcha->verify(
    $_POST['recaptcha_challenge_field'],
    $_POST['recaptcha_response_field']
);
```

Once you have the result, test against it to see if it is valid. The result is a `Zend_Service_ReCaptcha_Response` object, which provides an `isValid()` method.

Example 755. Validating the reCAPTCHA

```
if (!$result->isValid()) {
    // Failed validation
}
```

It is even simpler to use the reCAPTCHA `Zend_Captcha` adapter, or to use that adapter as a backend for the CAPTCHA form element. In each case, the details of rendering and validating the reCAPTCHA are automated for you.

23.3. Hiding email addresses

`Zend_Service_ReCaptcha_MailHide` can be used to hide email addresses. It will replace a part of an email address with a link that opens a popup window with a reCAPTCHA challenge. Solving the challenge will reveal the complete email address.

In order to use this component you will need an account to generate public and private keys for the mailhide API.

Example 756. Using the mail hide component

```
// The mail address we want to hide
$mail = 'mail@example.com';

// Create an instance of the mailhide component, passing it your public
// and private keys, as well as the mail address you want to hide
$mailHide = new Zend_Service_ReCaptcha_Mailhide();
$mailHide->setPublicKey($pubKey);
$mailHide->setPrivateKey($privKey);
$mailHide->setEmail($mail);

// Display it
print($mailHide);
```

The example above will display "m...@example.com" where "..." has a link that opens up a popup window with a reCAPTCHA challenge.

The public key, private key, and the email address can also be specified in the constructor of the class. A fourth argument also exists that enables you to set some options for the component. The available options are listed in the following table:

Table 137. Zend_Service_ReCaptcha_MailHide options

Option	Description	Expected Values	Default Value
linkTitle	The title attribute of the link	string	'Reveal this e-mail address'
linkHiddenText	The text that includes the popup link	string	'...'
popupWidth	The width of the popup window	int	500

Option	Description	Expected Values	Default Value
popupHeight	The height of the popup window	int	300

The configuration options can be set by sending them as the fourth argument to the constructor or by calling `setOptions($options)`, which takes an associative array or an instance of Zend_Config.

Example 757. Generating many hidden email addresses

```
// Create an instance of the mailhide component, passing it your public
// and private keys, as well as some configuration options
$mailHide = new Zend_Service_ReCaptcha_Mailhide();
$mailHide->setPublicKey($pubKey);
$mailHide->setPrivateKey($privKey);
$mailHide->setOptions(array(
    'linkTitle' => 'Click me',
    'linkHiddenText' => '+++++',
));

// The mail addresses we want to hide
$mailAddresses = array(
    'mail@example.com',
    'johndoe@example.com',
    'janedoe@example.com',
);

foreach ($mailAddresses as $mail) {
    $mailHide->setEmail($mail);
    print($mailHide);
}
```

24. Zend_Service_Simpy

24.1. Introduction

`Zend_Service_Simpy` is a lightweight wrapper for the free REST API available for the Simpy social bookmarking service.

In order to use `Zend_Service_Simpy`, you should already have a Simpy account. To get an account, visit the Simpy web site. For more information on the Simpy REST API, refer to the Simpy REST API documentation.

The Simpy REST API allows developers to interact with specific aspects of the service that the Simpy web site offers. The sections following will outline the use of `Zend_Service_Simpy` for each of these areas.

- Links: Create, Retrieve, Update, Delete
- Tags: Retrieve, Delete, Rename, Merge, Split
- Notes: Create, Retrieve, Update, Delete
- Watchlists: Get, Get All

24.2. Links

When querying links, results are returned in descending order by date added. Links can be searched by title, nickname, tags, note, or even the content of the web page associated with the link. Simpy offers searching by any or all of these fields with phrases, boolean operators, and wildcards. See the search syntax and search fields sections of the Simpy FAQ for more information.

Example 758. Querying Links

```
$simpy = new Zend_Service_Simpy('yourusername', 'yourpassword');

/* Search for the 10 links added most recently */
$linkQuery = new Zend_Service_Simpy_LinkQuery();
$linkQuery->setLimit(10);

/* Get and display the links */
$linkSet = $simpy->getLinks($linkQuery);
foreach ($linkSet as $link) {
    echo '<a href="';
    echo $link->getUrl();
    echo '">';
    echo $link->getTitle();
    echo '</a><br />';
}

/* Search for the 5 links added most recently with 'PHP' in
the title */
$linkQuery->setQueryString('title:PHP');
$linkQuery->setLimit(5);

/* Search for all links with 'French' in the title and
'language' in the tags */
$linkQuery->setQueryString('+title:French +tags:language');

/* Search for all links with 'French' in the title and without
'travel' in the tags */
$linkQuery->setQueryString('+title:French -tags:travel');

/* Search for all links added on 12/9/06 */
$linkQuery->setDate('2006-12-09');

/* Search for all links added after 12/9/06 (excluding that
date) */
$linkQuery->setAfterDate('2006-12-09');

/* Search for all links added before 12/9/06 (excluding that
date) */
$linkQuery->setBeforeDate('2006-12-09');

/* Search for all links added between 12/1/06 and 12/9/06
(excluding those two dates) */
$linkQuery->setBeforeDate('2006-12-01');
$linkQuery->setAfterDate('2006-12-09');
```

Links are represented uniquely by their URLs. In other words, if an attempt is made to save a link that has the same URL as an existing link, data for the existing link will be overwritten with the data specified in the save attempt.

Example 759. Modifying Links

```
$simpy = new Zend_Service_Simpy('yourusername', 'yourpassword');

/* Save a link */
$simpy->saveLink(
    'Zend Framework' // Title
    'http://framework.zend.com', // URL
    Zend_Service_Simpy_Link::ACCESSTYPE_PUBLIC, // Access Type
    'zend, framework, php' // Tags
    'Zend Framework home page' // Alternative title
    'This site rocks!' // Note
);

/* Overwrite the existing link with new data */
$simpy->saveLink(
    'Zend Framework'
    'http://framework.zend.com',
    Zend_Service_Simpy_Link::ACCESSTYPE_PRIVATE, // Access Type has changed
    'php, zend, framework' // Tags have changed order
    'Zend Framework' // Alternative title has changed
    'This site REALLY rocks!' // Note has changed
);

/* Delete the link */
$simpy->deleteLink('http://framework.zend.com');

/* A really easy way to do spring cleaning on your links ;) */
$linkSet = $this->_simpy->getLinks();
foreach ($linkSet as $link) {
    $this->_simpy->deleteLink($link->getUrl());
}
```

24.3. Tags

When retrieved, tags are sorted in decreasing order (i.e. highest first) by the number of links that use the tag.

Example 760. Working With Tags

```
$simpy = new Zend_Service_Simpy('yourusername', 'yourpassword');

/* Save a link with tags */
$simpy->saveLink(
    'Zend Framework' // Title
    'http://framework.zend.com', // URL
    Zend_Service_Simpy_Link::ACCESSTYPE_PUBLIC, // Access Type
    'zend, framework, php' // Tags
);

/* Get a list of all tags in use by links and notes */
$tagSet = $simpy->getTags();

/* Display each tag with the number of links using it */
foreach ($tagSet as $tag) {
    echo $tag->getTag();
    echo ' - ';
    echo $tag->getCount();
    echo '<br />';
}

/* Remove the 'zend' tag from all links using it */
$simpy->removeTag('zend');

/* Rename the 'framework' tag to 'frameworks' */
$simpy->renameTag('framework', 'frameworks');

/* Split the 'frameworks' tag into 'framework' and
'development', which will remove the 'frameworks' tag for
all links that use it and add the tags 'framework' and
'development' to all of those links */
$simpy->splitTag('frameworks', 'framework', 'development');

/* Merge the 'framework' and 'development' tags back into
'frameworks', basically doing the opposite of splitting them */
$simpy->mergeTags('framework', 'development', 'frameworks');
```

24.4. Notes

Notes can be saved, retrieved, and deleted. They are uniquely identified by a numeric ID value.

Example 761. Working With Notes

```
$simpy = new Zend_Service_Simpy('yourusername', 'yourpassword');

/* Save a note */
$simpy->saveNote(
    'Test Note', // Title
    'test,note', // Tags
    'This is a test note.' // Description
);

/* Overwrite an existing note */
$simpy->saveNote(
    'Updated Test Note', // Title
    'test,note,updated', // Tags
    'This is an updated test note.', // Description
    $note->getId() // Unique identifier
);

/* Search for the 10 most recently added notes */
$noteSet = $simpy->getNotes(null, 10);

/* Display the notes */
foreach ($noteSet as $note) {
    echo '<p>';
    echo $note->getTitle();
    echo '<br />';
    echo $note->getDescription();
    echo '<br >';
    echo $note->getTags();
    echo '</p>';
}

/* Search for all notes with 'PHP' in the title */
$noteSet = $simpy->getNotes('title:PHP');

/* Search for all notes with 'PHP' in the title and
without 'framework' in the description */
$noteSet = $simpy->getNotes('+title:PHP -description:framework');

/* Delete a note */
$simpy->deleteNote($note->getId());
```

24.5. Watchlists

Watchlists cannot be created or removed using the API, only retrieved. Thus, you must set up a watchlist via the Simpy web site prior to attempting to access it using the API.

Example 762. Retrieving Watchlists

```
$simpy = new Zend_Service_Simpy('yourusername', 'yourpassword');

/* Get a list of all watchlists */
$watchlistSet = $simpy->getWatchlists();

/* Display data for each watchlist */
foreach ($watchlistSet as $watchlist) {
    echo $watchlist->getId();
    echo '<br />';
    echo $watchlist->getName();
    echo '<br />';
    echo $watchlist->getDescription();
    echo '<br />';
    echo $watchlist->getAddDate();
    echo '<br />';
    echo $watchlist->getNewLinks();
    echo '<br />';

    foreach ($watchlist->getUsers() as $user) {
        echo $user;
        echo '<br />';
    }

    foreach ($watchlist->getFilters() as $filter) {
        echo $filter->getName();
        echo '<br />';
        echo $filter->getQuery();
        echo '<br />';
    }
}

/* Get an individual watchlist by its identifier */
$watchlist = $simpy->getWatchlist($watchlist->getId());
$watchlist = $simpy->getWatchlist(1);
```

25. Introduction

The `Zend_Service_SlideShare` component is used to interact with the slideshare.net web services for hosting slide shows online. With this component, you can embed slide shows which are hosted on this web site within a web site and even upload new slide shows to your account.

25.1. Getting Started with Zend_Service_SlideShare

In order to use the `Zend_Service_SlideShare` component you must first create an account on the slideshare.net servers (more information can be found here) in order to receive an API key, username, password and shared secret value -- all of which are needed in order to use the `Zend_Service_SlideShare` component.

Once you have setup an account, you can begin using the `Zend_Service_SlideShare` component by creating a new instance of the `Zend_Service_SlideShare` object and providing these values as shown below:

```
// Create a new instance of the component
$ss = new Zend_Service_SlideShare('APIKEY',
                                  'SHAREDSECRET',
                                  'USERNAME',
```

```
                                        'PASSWORD');
```

25.2. The SlideShow object

All slide shows in the `Zend_Service_SlideShare` component are represented using the `Zend_Service_SlideShare_SlideShow` object (both when retrieving and uploading new slide shows). For your reference a pseudo-code version of this class is provided below.

```
class Zend_Service_SlideShare_SlideShow {

    /**
     * Retrieves the location of the slide show
     */
    public function getLocation() {
        return $this->_location;
    }

    /**
     * Gets the transcript for this slide show
     */
    public function getTranscript() {
        return $this->_transcript;
    }

    /**
     * Adds a tag to the slide show
     */
    public function addTag($tag) {
        $this->_tags[] = (string)$tag;
        return $this;
    }

    /**
     * Sets the tags for the slide show
     */
    public function setTags(Array $tags) {
        $this->_tags = $tags;
        return $this;
    }

    /**
     * Gets all of the tags associated with the slide show
     */
    public function getTags() {
        return $this->_tags;
    }

    /**
     * Sets the filename on the local filesystem of the slide show
     * (for uploading a new slide show)
     */
    public function setFilename($file) {
        $this->_slideShowFilename = (string)$file;
        return $this;
    }

    /**
     * Retrieves the filename on the local filesystem of the slide show
     * which will be uploaded
```

```
     */
    public function getFilename() {
        return $this->_slideShowFilename;
    }

    /**
     * Gets the ID for the slide show
     */
    public function getId() {
        return $this->_slideShowId;
    }

    /**
     * Retrieves the HTML embed code for the slide show
     */
    public function getEmbedCode() {
        return $this->_embedCode;
    }

    /**
     * Retrieves the Thumbnail URi for the slide show
     */
    public function getThumbnailUrl() {
        return $this->_thumbnailUrl;
    }

    /**
     * Sets the title for the Slide show
     */
    public function setTitle($title) {
        $this->_title = (string)$title;
        return $this;
    }

    /**
     * Retrieves the Slide show title
     */
    public function getTitle() {
        return $this->_title;
    }

    /**
     * Sets the description for the Slide show
     */
    public function setDescription($desc) {
        $this->_description = (string)$desc;
        return $this;
    }

    /**
     * Gets the description of the slide show
     */
    public function getDescription() {
        return $this->_description;
    }

    /**
     * Gets the numeric status of the slide show on the server
     */
    public function getStatus() {
```

```
        return $this->_status;
    }

    /**
     * Gets the textual description of the status of the slide show on
     * the server
     */
    public function getStatusDescription() {
        return $this->_statusDescription;
    }

    /**
     * Gets the permanent link of the slide show
     */
    public function getPermaLink() {
        return $this->_permalink;
    }

    /**
     * Gets the number of views the slide show has received
     */
    public function getNumViews() {
        return $this->_numViews;
    }
}
```

The above pseudo-class only shows those methods which should be used by end-user developers. Other available methods are internal to the component.

When using the `Zend_Service_SlideShare` component, this data class will be used frequently to browse or add new slide shows to or from the web service.

25.3. Retrieving a single slide show

The simplest usage of the `Zend_Service_SlideShare` component is the retrieval of a single slide show by slide show ID provided by the slideshare.net application and is done by calling the `getSlideShow()` method of a `Zend_Service_SlideShare` object and using the resulting `Zend_Service_SlideShare_SlideShow` object as shown.

```
// Create a new instance of the component
$ss = new Zend_Service_SlideShare('APIKEY',
                                  'SHAREDSECRET',
                                  'USERNAME',
                                  'PASSWORD');

$slideshow = $ss->getSlideShow(123456);

print "Slide Show Title: {$slideshow->getTitle()}<br/>\n";
print "Number of views: {$slideshow->getNumViews()}<br/>\n";
```

25.4. Retrieving Groups of Slide Shows

If you do not know the specific ID of a slide show you are interested in retrieving, you can retrieving groups of slide shows by using one of three methods:

- *Slide shows from a specific account*

You can retrieve slide shows from a specific account by using the `getSlideShowsByUsername()` method and providing the username from which the slide shows should be retrieved

- *Slide shows which contain specific tags*

  You can retrieve slide shows which contain one or more specific tags by using the `getSlideShowsByTag` method and providing one or more tags which the slide show must have assigned to it in order to be retrieved

- *Slide shows by group*

  You can retrieve slide shows which are a member of a specific group using the `getSlideShowsByGroup` method and providing the name of the group which the slide show must belong to in order to be retrieved

Each of the above methods of retrieving multiple slide shows a similar approach is used. An example of using each method is shown below:

```
// Create a new instance of the component
$ss = new Zend_Service_SlideShare('APIKEY',
                                  'SHAREDSECRET',
                                  'USERNAME',
                                  'PASSWORD');

$starting_offset = 0;
$limit = 10;

// Retrieve the first 10 of each type
$ss_user = $ss->getSlideShowsByUser('username', $starting_offset, $limit);
$ss_tags = $ss->getSlideShowsByTag('zend', $starting_offset, $limit);
$ss_group = $ss->getSlideShowsByGroup('mygroup', $starting_offset, $limit);

// Iterate over the slide shows
foreach($ss_user as $slideshow) {
   print "Slide Show Title: {$slideshow->getTitle}<br/>\n";
}
```

25.5. Zend_Service_SlideShare Caching policies

By default, `Zend_Service_SlideShare` will cache any request against the web service automatically to the filesystem (default path `/tmp`) for 12 hours. If you desire to change this behavior, you must provide your own *Zend_Cache* object using the `setCacheObject` method as shown:

```
$frontendOptions = array(
                        'lifetime' => 7200,
                        'automatic_serialization' => true);
$backendOptions  = array(
                        'cache_dir' => '/webtmp/');

$cache = Zend_Cache::factory('Core',
                             'File',
                             $frontendOptions,
                             $backendOptions);

$ss = new Zend_Service_SlideShare('APIKEY',
```

```
                                    'SHAREDSECRET',
                                    'USERNAME',
                                    'PASSWORD');
$ss->setCacheObject($cache);

$ss_user = $ss->getSlideShowsByUser('username', $starting_offset, $limit);
```

25.6. Changing the behavior of the HTTP Client

If for whatever reason you would like to change the behavior of the HTTP client when making the web service request, you can do so by creating your own instance of the `Zend_Http_Client` object (see *Zend_Http*). This is useful for instance when it is desirable to set the timeout for the connection to something other then default as shown:

```
$client = new Zend_Http_Client();
$client->setConfig(array('timeout' => 5));

$ss = new Zend_Service_SlideShare('APIKEY',
                                    'SHAREDSECRET',
                                    'USERNAME',
                                    'PASSWORD');
$ss->setHttpClient($client);
$ss_user = $ss->getSlideShowsByUser('username', $starting_offset, $limit);
```

26. Zend_Service_StrikeIron

`Zend_Service_StrikeIron` provides a PHP 5 client to StrikeIron web services. See the following sections:

- Section 26, “Zend_Service_StrikeIron”
- Section 27, “Zend_Service_StrikeIron: Bundled Services”
- Section 28, “Zend_Service_StrikeIron: Advanced Uses”

26.1. Overview

StrikeIron offers hundreds of commercial data services ("Data as a Service") such as Online Sales Tax, Currency Rates, Stock Quotes, Geocodes, Global Address Verification, Yellow/White Pages, MapQuest Driving Directions, Dun & Bradstreet Business Credit Checks, and much, much more.

Each StrikeIron web service shares a standard SOAP (and REST) API, making it easy to integrate and manage multiple services. StrikeIron also manages customer billing for all services in a single account, making it perfect for solution providers. Get started with free web services at http://www.strikeiron.com/sdp.

StrikeIron's services may be used through the PHP 5 SOAP extension alone. However, using StrikeIron this way does not give an ideal PHP-like interface. The `Zend_Service_StrikeIron` component provides a lightweight layer on top of the SOAP extension for working with StrikeIron services in a more convenient, PHP-like manner.

The PHP 5 SOAP extension must be installed and enabled to use `Zend_Service_StrikeIron`.

The `Zend_Service_StrikeIron` component provides:

- A single point for configuring your StrikeIron authentication credentials that can be used across many StrikeIron services.
- A standard way of retrieving your StrikeIron subscription information such as license status and the number of hits remaining to a service.
- The ability to use any StrikeIron service from its WSDL without creating a PHP wrapper class, and the option of creating a wrapper for a more convenient interface.
- Wrappers for three popular StrikeIron services.

26.2. Registering with StrikeIron

Before you can get started with `Zend_Service_StrikeIron`, you must first register for a StrikeIron developer account.

After registering, you will receive a StrikeIron username and password. These will be used when connecting to StrikeIron using `Zend_Service_StrikeIron`.

You will also need to sign up for StrikeIron's Super Data Pack Web Service.

Both registration steps are free and can be done relatively quickly through the StrikeIron website.

26.3. Getting Started

Once you have registered for a StrikeIron account and signed up for the Super Data Pack, you're ready to start using `Zend_Service_StrikeIron`.

StrikeIron consists of hundreds of different web services. `Zend_Service_StrikeIron` can be used with many of these services but provides supported wrappers for three of them:

- ZIP Code Information
- US Address Verification
- Sales & Use Tax Basic

The class `Zend_Service_StrikeIron` provides a simple way of specifying your StrikeIron account information and other options in its constructor. It also has a factory method that will return clients for StrikeIron services:

```
$strikeIron = new Zend_Service_StrikeIron(array('username' => 'your-username',
                                                'password' => 'your-password'));

$taxBasic = $strikeIron->getService(array('class' => 'SalesUseTaxBasic'));
```

The `getService()` method will return a client for any StrikeIron service by the name of its PHP wrapper class. In this case, the name `SalesUseTaxBasic` refers to the wrapper class `Zend_Service_StrikeIron_SalesUseTaxBasic`. Wrappers are included for three services and described in Bundled Services.

The `getService()` method can also return a client for a StrikeIron service that does not yet have a PHP wrapper. This is explained in Using Services by WSDL.

26.4. Making Your First Query

Once you have used the `getService()` method to get a client for a particular StrikeIron service, you can utilize that client by calling methods on it just like any other PHP object.

```
$strikeIron = new Zend_Service_StrikeIron(array('username' => 'your-username',
                                                'password' => 'your-password'));

// Get a client for the Sales & Use Tax Basic service
$taxBasic = $strikeIron->getService(array('class' => 'SalesUseTaxBasic'));

// Query tax rate for Ontario, Canada
$rateInfo = $taxBasic->getTaxRateCanada(array('province' => 'ontario'));
echo $rateInfo->province;
echo $rateInfo->abbreviation;
echo $rateInfo->GST;
```

In the example above, the `getService()` method is used to return a client to the Sales & Use Tax Basic service. The client object is stored in `$taxBasic`.

The `getTaxRateCanada()` method is then called on the service. An associative array is used to supply keyword parameters to the method. This is the way that all StrikeIron methods are called.

The result from `getTaxRateCanada()` is stored in `$rateInfo` and has properties like `province` and `GST`.

Many of the StrikeIron services are as simple to use as the example above. See Bundled Services for detailed information on three StrikeIron services.

26.5. Examining Results

When learning or debugging the StrikeIron services, it's often useful to dump the result returned from a method call. The result will always be an object that is an instance of `Zend_Service_StrikeIron_Decorator`. This is a small decorator object that wraps the results from the method call.

The simplest way to examine a result from the service is to use the built-in PHP functions like print_r():

```
<?php
$strikeIron = new Zend_Service_StrikeIron(array('username' => 'your-username',
                                                'password' => 'your-password'));

$taxBasic = $strikeIron->getService(array('class' => 'SalesUseTaxBasic'));

$rateInfo = $taxBasic->getTaxRateCanada(array('province' => 'ontario'));
print_r($rateInfo);
?>
Zend_Service_StrikeIron_Decorator Object
(
    [_name:protected] => GetTaxRateCanadaResult
    [_object:protected] => stdClass Object
        (
            [abbreviation] => ON
            [province] => ONTARIO
            [GST] => 0.06
            [PST] => 0.08
            [total] => 0.14
            [HST] => Y
        )
)
```

In the output above, we see that the decorator (`$rateInfo`) wraps an object named `GetTaxRateCanadaResult`, the result of the call to `getTaxRateCanada()`.

This means that `$rateInfo` has public properties like `abbreviation`, `province`, and `GST`. These are accessed like `$rateInfo->province`.

StrikeIron result properties sometimes start with an uppercase letter such as `Foo` or `Bar` where most PHP object properties normally start with a lowercase letter as in `foo` or `bar`. The decorator will automatically do this inflection so you may read a property `Foo` as `foo`.

If you ever need to get the original object or its name out of the decorator, use the respective methods `getDecoratedObject()` and `getDecoratedObjectName()`.

26.6. Handling Errors

The previous examples are naive, i.e. no error handling was shown. It's possible that StrikeIron will return a fault during a method call. Events like bad account credentials or an expired subscription can cause StrikeIron to raise a fault.

An exception will be thrown when such a fault occurs. You should anticipate and catch these exceptions when making method calls to the service:

```
$strikeIron = new Zend_Service_StrikeIron(array('username' => 'your-username',
                                                'password' => 'your-password'));

$taxBasic = $strikeIron->getService(array('class' => 'SalesUseTaxBasic'));

try {

  $taxBasic->getTaxRateCanada(array('province' => 'ontario'));

} catch (Zend_Service_StrikeIron_Exception $e) {

  // error handling for events like connection
  // problems or subscription errors

}
```

The exceptions thrown will always be `Zend_Service_StrikeIron_Exception`.

It's important to understand the difference between exceptions and normal failed method calls. Exceptions occur for *exceptional* conditions, such as the network going down or your subscription expiring. Failed method calls that are a common occurrence, such as `getTaxRateCanada()` not finding the `province` you supplied, will not result an in exception.

Every time you make a method call to a StrikeIron service, you should check the response object for validity and also be prepared to catch an exception.

26.7. Checking Your Subscription

StrikeIron provides many different services. Some of these are free, some are available on a trial basis, and some are pay subscription only. When using StrikeIron, it's important to be aware of your subscription status for the services you are using and check it regularly.

Each StrikeIron client returned by the `getService` method has the ability to check the subscription status for that service using the `getSubscriptionInfo()` method of the client:

```
// Get a client for the Sales & Use Tax Basic service
$strikeIron = new Zend_Service_StrikeIron(array('username' => 'your-username',
                                                'password' => 'your-password'));

$taxBasic = $strikeIron->getService(array('class => 'SalesUseTaxBasic'));

// Check remaining hits for the Sales & Use Tax Basic service
$subscription = $taxBasic->getSubscriptionInfo();
echo $subscription->remainingHits;
```

The `getSubscriptionInfo()` method will return an object that typically has a `remainingHits` property. It's important to check the status on each service that you are using. If a method call is made to StrikeIron after the remaining hits have been used up, an exception will occur.

Checking your subscription to a service does not use any remaining hits to the service. Each time any method call to the service is made, the number of hits remaining will be cached and this cached value will be returned by `getSubscriptionInfo()` without connecting to the service again. To force `getSubscriptionInfo()` to override its cache and query the subscription information again, use `getSubscriptionInfo(true)`.

27. Zend_Service_StrikeIron: Bundled Services

`Zend_Service_StrikeIron` comes with wrapper classes for three popular StrikeIron services.

27.1. ZIP Code Information

`Zend_Service_StrikeIron_ZipCodeInfo` provides a client for StrikeIron's Zip Code Information Service. For more information on this service, visit these StrikeIron resources:

- Zip Code Information Service Page
- Zip Code Information Service WSDL

The service contains a `getZipCode()` method that will retrieve information about a United States ZIP code or Canadian postal code:

```
$strikeIron = new Zend_Service_StrikeIron(array('username' => 'your-username',
                                                'password' => 'your-password'));

// Get a client for the Zip Code Information service
$zipInfo = $strikeIron->getService(array('class' => 'ZipCodeInfo'));

// Get the Zip information for 95014
$response = $zipInfo->getZipCode(array('ZipCode' => 95014));
$zips = $response->serviceResult;

// Display the results
if ($zips->count == 0) {
    echo 'No results found';
} else {
    // a result with one single zip code is returned as an object,
    // not an array with one element as one might expect.
```

```
    if (! is_array($zips->zipCodes)) {
        $zips->zipCodes = array($zips->zipCodes);
    }

    // print all of the possible results
    foreach ($zips->zipCodes as $z) {
        $info = $z->zipCodeInfo;

        // show all properties
        print_r($info);

        // or just the city name
        echo $info->preferredCityName;
    }
}

// Detailed status information
// http://www.strikeiron.com/exampledata/StrikeIronZipCodeInformation_v3.pdf
$status = $response->serviceStatus;
```

27.2. U.S. Address Verification

`Zend_Service_StrikeIron_USAddressVerification` provides a client for StrikeIron's U.S. Address Verification Service. For more information on this service, visit these StrikeIron resources:

- U.S. Address Verification Service Page
- U.S. Address Verification Service WSDL

The service contains a `verifyAddressUSA()` method that will verify an address in the United States:

```
$strikeIron = new Zend_Service_StrikeIron(array('username' => 'your-username',
                                                'password' => 'your-password'));

// Get a client for the Zip Code Information service
$verifier = $strikeIron->getService(array('class' => 'USAddressVerification'));

// Address to verify. Not all fields are required but
// supply as many as possible for the best results.
$address = array('firm'          => 'Zend Technologies',
                 'addressLine1'  => '19200 Stevens Creek Blvd',
                 'addressLine2'  => '',
                 'city_state_zip' => 'Cupertino CA 95014');

// Verify the address
$result = $verifier->verifyAddressUSA($address);

// Display the results
if ($result->addressErrorNumber != 0) {
    echo $result->addressErrorNumber;
    echo $result->addressErrorMessage;
} else {
    // show all properties
    print_r($result);

    // or just the firm name
    echo $result->firm;
```

```
    // valid address?
    $valid = ($result->valid == 'VALID');
}
```

27.3. Sales & Use Tax Basic

`Zend_Service_StrikeIron_SalesUseTaxBasic` provides a client for StrikeIron's Sales & Use Tax Basic service. For more information on this service, visit these StrikeIron resources:

- Sales & Use Tax Basic Service Page
- Sales & Use Tax Basic Service WSDL

The service contains two methods, `getTaxRateUSA()` and `getTaxRateCanada()`, that will retrieve sales and use tax data for the United States and Canada, respectively.

```
$strikeIron = new Zend_Service_StrikeIron(array('username' => 'your-username',
                                                'password' => 'your-password'));

// Get a client for the Sales & Use Tax Basic service
$taxBasic = $strikeIron->getService(array('class' => 'SalesUseTaxBasic'));

// Query tax rate for Ontario, Canada
$rateInfo = $taxBasic->getTaxRateCanada(array('province' => 'foo'));
print_r($rateInfo);               // show all properties
echo $rateInfo->GST;              // or just the GST (Goods & Services Tax)

// Query tax rate for Cupertino, CA USA
$rateInfo = $taxBasic->getTaxRateUS(array('zip_code' => 95014));
print_r($rateInfo);               // show all properties
echo $rateInfo->state_sales_tax;  // or just the state sales tax
```

28. Zend_Service_StrikeIron: Advanced Uses

This section describes the more advanced uses of `Zend_Service_StrikeIron`.

28.1. Using Services by WSDL

Some StrikeIron services may have a PHP wrapper class available, such as those described in Bundled Services. However, StrikeIron offers hundreds of services and many of these may be usable even without creating a special wrapper class.

To try a StrikeIron service that does not have a wrapper class available, give the `wsdl` option to `getService()` instead of the `class` option:

```
$strikeIron = new Zend_Service_StrikeIron(array('username' => 'your-username',
                                                'password' => 'your-password'));

// Get a generic client to the Reverse Phone Lookup service
$phone = $strikeIron->getService(
    array('wsdl' => 'http://ws.strikeiron.com/ReversePhoneLookup?WSDL')
);

$result = $phone->lookup(array('Number' => '(408) 253-8800'));
echo $result->listingName;
```

```
// Zend Technologies USA Inc
```

Using StrikeIron services from the WSDL will require at least some understanding of the WSDL files. StrikeIron has many resources on its site to help with this. Also, Jan Schneider from the Horde project has written a small PHP routine that will format a WSDL file into more readable HTML.

Please note that only the services described in the Bundled Services section are officially supported.

28.2. Viewing SOAP Transactions

All communication with StrikeIron is done using the SOAP extension. It is sometimes useful to view the XML exchanged with StrikeIron for debug purposes.

Every StrikeIron client (subclass of `Zend_Service_StrikeIron_Base`) contains a `getSoapClient()` method to return the underlying instance of `SOAPClient` used to communicate with StrikeIron.

PHP's SOAPClient has a `trace` option that causes it to remember the XML exchanged during the last transaction. `Zend_Service_StrikeIron` does not enable the `trace` option by default but this can easily by changed by specifying the options that will be passed to the `SOAPClient` constructor.

To view a SOAP transaction, call the `getSoapClient()` method to get the `SOAPClient` instance and then call the appropriate methods like `__getLastRequest()` and `__getLastRequest()`:

```
$strikeIron =
    new Zend_Service_StrikeIron(array('username' => 'your-username',
                                      'password' => 'your-password',
                                      'options'  => array('trace' => true)));

// Get a client for the Sales & Use Tax Basic service
$taxBasic = $strikeIron->getService(array('class' => 'SalesUseTaxBasic'));

// Perform a method call
$taxBasic->getTaxRateCanada(array('province' => 'ontario'));

// Get SOAPClient instance and view XML
$soapClient = $taxBasic->getSoapClient();
echo $soapClient->__getLastRequest();
echo $soapClient->__getLastResponse();
```

29. Zend_Service_Technorati

29.1. Introduction

`Zend_Service_Technorati` provides an easy, intuitive and object-oriented interface for using the Technorati API. It provides access to all available Technorati API queries and returns the original XML response as a friendly PHP object.

Technorati is one of the most popular blog search engines. The API interface enables developers to retrieve information about a specific blog, search blogs matching a single tag or phrase and get information about a specific author (blogger). For a full list of available queries please see the Technorati API documentation or the Available Technorati queries section of this document.

29.2. Getting Started

Technorati requires a valid API key for usage. To get your own API Key you first need to create a new Technorati account, then visit the API Key section.

API Key limits

You can make up to 500 Technorati API calls per day, at no charge. Other usage limitations may apply, depending on the current Technorati API license.

Once you have a valid API key, you're ready to start using `Zend_Service_Technorati`.

29.3. Making Your First Query

In order to run a query, first you need a `Zend_Service_Technorati` instance with a valid API key. Then choose one of the available query methods, and call it providing required arguments.

Example 763. Sending your first query

```
// create a new Zend_Service_Technorati
// with a valid API_KEY
$technorati = new Zend_Service_Technorati('VALID_API_KEY');

// search Technorati for PHP keyword
$resultSet = $technorati->search('PHP');
```

Each query method accepts an array of optional parameters that can be used to refine your query.

Example 764. Refining your query

```
// create a new Zend_Service_Technorati
// with a valid API_KEY
$technorati = new Zend_Service_Technorati('VALID_API_KEY');

// filter your query including only results
// with some authority (Results from blogs with a handful of links)
$options = array('authority' => 'a4');

// search Technorati for PHP keyword
$resultSet = $technorati->search('PHP', $options);
```

A `Zend_Service_Technorati` instance is not a single-use object. That is, you don't need to create a new instance for each query call; simply use your current `Zend_Service_Technorati` object as long as you need it.

Example 765. Sending multiple queries with the same Zend_Service_Technorati instance

```
// create a new Zend_Service_Technorati
// with a valid API_KEY
$technorati = new Zend_Service_Technorati('VALID_API_KEY');

// search Technorati for PHP keyword
$search = $technorati->search('PHP');

// get top tags indexed by Technorati
$topTags = $technorati->topTags();
```

29.4. Consuming Results

You can get one of two types of result object in response to a query.

The first group is represented by `Zend_Service_Technorati_*ResultSet` objects. A result set object is basically a collection of result objects. It extends the basic `Zend_Service_Technorati_ResultSet` class and implements the `SeekableIterator` PHP interface. The best way to consume a result set object is to loop over it with the PHP `foreach` statement.

Example 766. Consuming a result set object

```
// create a new Zend_Service_Technorati
// with a valid API_KEY
$technorati = new Zend_Service_Technorati('VALID_API_KEY');

// search Technorati for PHP keyword
// $resultSet is an instance of Zend_Service_Technorati_SearchResultSet
$resultSet = $technorati->search('PHP');

// loop over all result objects
foreach ($resultSet as $result) {
    // $result is an instance of Zend_Service_Technorati_SearchResult
}
```

Because `Zend_Service_Technorati_ResultSet` implements the `SeekableIterator` interface, you can seek a specific result object using its position in the result collection.

Example 767. Seeking a specific result set object

```
// create a new Zend_Service_Technorati
// with a valid API_KEY
$technorati = new Zend_Service_Technorati('VALID_API_KEY');

// search Technorati for PHP keyword
// $resultSet is an instance of Zend_Service_Technorati_SearchResultSet
$resultSet = $technorati->search('PHP');

// $result is an instance of Zend_Service_Technorati_SearchResult
$resultSet->seek(1);
$result = $resultSet->current();
```

`SeekableIterator` works as an array and counts positions starting from index 0. Fetching position number 1 means getting the second result in the collection.

The second group is represented by special standalone result objects. `Zend_Service_Technorati_GetInfoResult`, `Zend_Service_Technorati_BlogInfoResult` and `Zend_Service_Technorati_KeyInfoResult` act as wrappers for additional objects, such as `Zend_Service_Technorati_Author` and `Zend_Service_Technorati_Weblog`.

Example 768. Consuming a standalone result object

```
// create a new Zend_Service_Technorati
// with a valid API_KEY
$technorati = new Zend_Service_Technorati('VALID_API_KEY');

// get info about weppos author
$result = $technorati->getInfo('weppos');

$author = $result->getAuthor();
echo '<h2>Blogs authored by ' . $author->getFirstName() . " " .
          $author->getLastName() . '</h2>';
echo '<ol>';
foreach ($result->getWeblogs() as $weblog) {
    echo '<li>' . $weblog->getName() . '</li>';
}
echo "</ol>";
```

Please read the Zend_Service_Technorati Classes section for further details about response classes.

29.5. Handling Errors

Each `Zend_Service_Technorati` query method throws a `Zend_Service_Technorati_Exception` exception on failure with a meaningful error message.

There are several reasons that may cause a `Zend_Service_Technorati` query to fail. `Zend_Service_Technorati` validates all parameters for any query request. If a parameter is invalid or it contains an invalid value, a new `Zend_Service_Technorati_Exception` exception is thrown. Additionally, the Technorati API interface could be temporally unavailable, or it could return a response that is not well formed.

You should always wrap a Technorati query with a `try`...`catch` block.

Example 769. Handling a Query Exception

```
$technorati = new Zend_Service_Technorati('VALID_API_KEY');
try {
    $resultSet = $technorati->search('PHP');
} catch(Zend_Service_Technorati_Exception $e) {
    echo "An error occurred: " $e->getMessage();
}
```

29.6. Checking Your API Key Daily Usage

From time to time you probably will want to check your API key daily usage. By default Technorati limits your API usage to 500 calls per day, and an exception is returned by `Zend_Service_Technorati` if you try to use it beyond this limit. You can get information about your API key usage using the `Zend_Service_Technorati::keyInfo()` method.

`Zend_Service_Technorati::keyInfo()` returns a `Zend_Service_Technorati_KeyInfoResult` object. For full details please see the API reference guide.

Example 770. Getting API key daily usage information

```
$technorati = new Zend_Service_Technorati('VALID_API_KEY');
$key = $technorati->keyInfo();

echo "API Key: " . $key->getApiKey() . "<br />";
echo "Daily Usage: " . $key->getApiQueries() . "/" .
     $key->getMaxQueries() . "<br />";
```

29.7. Available Technorati Queries

`Zend_Service_Technorati` provides support for the following queries:

- `Cosmos`
- `Search`
- `Tag`
- `DailyCounts`
- `TopTags`
- `BlogInfo`
- `BlogPostTags`
- `GetInfo`

29.7.1. Technorati Cosmos

`Cosmos` query lets you see what blogs are linking to a given URL. It returns a `Zend_Service_Technorati_CosmosResultSet` object. For full details please see `Zend_Service_Technorati::cosmos()` in the API reference guide.

Example 771. Cosmos Query

```
$technorati = new Zend_Service_Technorati('VALID_API_KEY');
$resultSet = $technorati->cosmos('http://devzone.zend.com/');

echo "<p>Reading " . $resultSet->totalResults() .
     " of " . $resultSet->totalResultsAvailable() .
     " available results</p>";
echo "<ol>";
foreach ($resultSet as $result) {
    echo "<li>" . $result->getWeblog()->getName() . "</li>";
}
echo "</ol>";
```

29.7.2. Technorati Search

The `Search` query lets you see what blogs contain a given search string. It returns a `Zend_Service_Technorati_SearchResultSet` object. For full details please see `Zend_Service_Technorati::search()` in the API reference guide.

Example 772. Search Query

```
$technorati = new Zend_Service_Technorati('VALID_API_KEY');
$resultSet = $technorati->search('zend framework');

echo "<p>Reading " . $resultSet->totalResults() .
     " of " . $resultSet->totalResultsAvailable() .
     " available results</p>";
echo "<ol>";
foreach ($resultSet as $result) {
    echo "<li>" . $result->getWeblog()->getName() . "</li>";
}
echo "</ol>";
```

29.7.3. Technorati Tag

The Tag query lets you see what posts are associated with a given tag. It returns a `Zend_Service_Technorati_TagResultSet` object. For full details please see `Zend_Service_Technorati::tag()` in the API reference guide.

Example 773. Tag Query

```
$technorati = new Zend_Service_Technorati('VALID_API_KEY');
$resultSet = $technorati->tag('php');

echo "<p>Reading " . $resultSet->totalResults() .
     " of " . $resultSet->totalResultsAvailable() .
     " available results</p>";
echo "<ol>";
foreach ($resultSet as $result) {
    echo "<li>" . $result->getWeblog()->getName() . "</li>";
}
echo "</ol>";
```

29.7.4. Technorati DailyCounts

The DailyCounts query provides daily counts of posts containing the queried keyword. It returns a `Zend_Service_Technorati_DailyCountsResultSet` object. For full details please see `Zend_Service_Technorati::dailyCounts()` in the API reference guide.

Example 774. DailyCounts Query

```
$technorati = new Zend_Service_Technorati('VALID_API_KEY');
$resultSet = $technorati->dailyCounts('php');

foreach ($resultSet as $result) {
    echo "<li>" . $result->getDate() .
         "(" . $result->getCount() . ")</li>";
}
echo "</ol>";
```

29.7.5. Technorati TopTags

The TopTags query provides information on top tags indexed by Technorati. It returns a `Zend_Service_Technorati_TagsResultSet` object. For full details please see `Zend_Service_Technorati::topTags()` in the API reference guide.

Example 775. TopTags Query

```
$technorati = new Zend_Service_Technorati('VALID_API_KEY');
$resultSet = $technorati->topTags();

echo "<p>Reading " . $resultSet->totalResults() .
     " of " . $resultSet->totalResultsAvailable() .
     " available results</p>";
echo "<ol>";
foreach ($resultSet as $result) {
    echo "<li>" . $result->getTag() . "</li>";
}
echo "</ol>";
```

29.7.6. Technorati BlogInfo

The BlogInfo query provides information on what blog, if any, is associated with a given URL. It returns a `Zend_Service_Technorati_BlogInfoResult` object. For full details please see `Zend_Service_Technorati::blogInfo()` in the API reference guide.

Example 776. BlogInfo Query

```
$technorati = new Zend_Service_Technorati('VALID_API_KEY');
$result = $technorati->blogInfo('http://devzone.zend.com/');

echo '<h2><a href="' . (string) $result->getWeblog()->getUrl() . '">' .
     $result->getWeblog()->getName() . '</a></h2>';
```

29.7.7. Technorati BlogPostTags

The BlogPostTags query provides information on the top tags used by a specific blog. It returns a `Zend_Service_Technorati_TagsResultSet` object. For full details please see `Zend_Service_Technorati::blogPostTags()` in the API reference guide.

Example 777. BlogPostTags Query

```
$technorati = new Zend_Service_Technorati('VALID_API_KEY');
$resultSet = $technorati->blogPostTags('http://devzone.zend.com/');

echo "<p>Reading " . $resultSet->totalResults() .
     " of " . $resultSet->totalResultsAvailable() .
     " available results</p>";
echo "<ol>";
foreach ($resultSet as $result) {
    echo "<li>" . $result->getTag() . "</li>";
}
echo "</ol>";
```

29.7.8. Technorati GetInfo

The GetInfo query tells you things that Technorati knows about a member. It returns a `Zend_Service_Technorati_GetInfoResult` object. For full details please see `Zend_Service_Technorati::getInfo()` in the API reference guide.

Example 778. GetInfo Query

```
$technorati = new Zend_Service_Technorati('VALID_API_KEY');
$result = $technorati->getInfo('weppos');

$author = $result->getAuthor();
echo "<h2>Blogs authored by " . $author->getFirstName() . " " .
     $author->getLastName() . "</h2>";
echo "<ol>";
foreach ($result->getWeblogs() as $weblog) {
    echo "<li>" . $weblog->getName() . "</li>";
}
echo "</ol>";
```

29.7.9. Technorati KeyInfo

The KeyInfo query provides information on daily usage of an API key. It returns a `Zend_Service_Technorati_KeyInfoResult` object. For full details please see `Zend_Service_Technorati::keyInfo()` in the API reference guide.

29.8. Zend_Service_Technorati Classes

The following classes are returned by the various Technorati queries. Each `Zend_Service_Technorati_*ResultSet` class holds a type-specific result set which can be easily iterated, with each result being contained in a type result object. All result set classes extend `Zend_Service_Technorati_ResultSet` class and implement the `SeekableIterator` interface, allowing for easy iteration and seeking to a specific result.

- `Zend_Service_Technorati_ResultSet`
- `Zend_Service_Technorati_CosmosResultSet`
- `Zend_Service_Technorati_SearchResultSet`
- `Zend_Service_Technorati_TagResultSet`
- `Zend_Service_Technorati_DailyCountsResultSet`
- `Zend_Service_Technorati_TagsResultSet`
- `Zend_Service_Technorati_Result`
- `Zend_Service_Technorati_CosmosResult`
- `Zend_Service_Technorati_SearchResult`
- `Zend_Service_Technorati_TagResult`
- `Zend_Service_Technorati_DailyCountsResult`
- `Zend_Service_Technorati_TagsResult`
- `Zend_Service_Technorati_GetInfoResult`
- `Zend_Service_Technorati_BlogInfoResult`

- Zend_Service_Technorati_KeyInfoResult

Zend_Service_Technorati_GetInfoResult, Zend_Service_Technorati_BlogInfoResult and Zend_Service_Technorati_KeyInfoResult represent exceptions to the above because they don't belong to a result set and they don't implement any interface. They represent a single response object and they act as a wrapper for additional Zend_Service_Technorati objects, such as Zend_Service_Technorati_Author and Zend_Service_Technorati_Weblog.

The Zend_Service_Technorati library includes additional convenient classes representing specific response objects. Zend_Service_Technorati_Author represents a single Technorati account, also known as a blog author or blogger. Zend_Service_Technorati_Weblog represents a single weblog object, along with all specific weblog properties such as feed URLs or blog name. For full details please see Zend_Service_Technorati in the API reference guide.

29.8.1. Zend_Service_Technorati_ResultSet

Zend_Service_Technorati_ResultSet is the most essential result set. The scope of this class is to be extended by a query-specific child result set class, and it should never be used to initialize a standalone object. Each of the specific result sets represents a collection of query-specific Zend_Service_Technorati_Result objects.

Zend_Service_Technorati_ResultSet implements the PHP SeekableIterator interface, and you can iterate all result objects via the PHP foreach statement.

Example 779. Iterating result objects from a resultset collection

```
// run a simple query
$technorati = new Zend_Service_Technorati('VALID_API_KEY');
$resultSet = $technorati->search('php');

// $resultSet is now an instance of
// Zend_Service_Technorati_SearchResultSet
// it extends Zend_Service_Technorati_ResultSet
foreach ($resultSet as $result) {
    // do something with your
    // Zend_Service_Technorati_SearchResult object
}
```

29.8.2. Zend_Service_Technorati_CosmosResultSet

Zend_Service_Technorati_CosmosResultSet represents a Technorati Cosmos query result set.

Zend_Service_Technorati_CosmosResultSet extends Zend_Service_Technorati_ResultSet.

29.8.3. Zend_Service_Technorati_SearchResultSet

Zend_Service_Technorati_SearchResultSet represents a Technorati Search query result set.

`Zend_Service_Technorati_SearchResultSet` extends Zend_Service_Technorati_ResultSet.

29.8.4. Zend_Service_Technorati_TagResultSet

`Zend_Service_Technorati_TagResultSet` represents a Technorati Tag query result set.

`Zend_Service_Technorati_TagResultSet` extends Zend_Service_Technorati_ResultSet.

29.8.5. Zend_Service_Technorati_DailyCountsResultSet

`Zend_Service_Technorati_DailyCountsResultSet` represents a Technorati DailyCounts query result set.

`Zend_Service_Technorati_DailyCountsResultSet` extends Zend_Service_Technorati_ResultSet.

29.8.6. Zend_Service_Technorati_TagsResultSet

`Zend_Service_Technorati_TagsResultSet` represents a Technorati TopTags or BlogPostTags queries result set.

`Zend_Service_Technorati_TagsResultSet` extends Zend_Service_Technorati_ResultSet.

29.8.7. Zend_Service_Technorati_Result

`Zend_Service_Technorati_Result` is the most essential result object. The scope of this class is to be extended by a query specific child result class, and it should never be used to initialize a standalone object.

29.8.8. Zend_Service_Technorati_CosmosResult

`Zend_Service_Technorati_CosmosResult` represents a single Technorati Cosmos query result object. It is never returned as a standalone object, but it always belongs to a valid Zend_Service_Technorati_CosmosResultSet object.

`Zend_Service_Technorati_CosmosResult` extends Zend_Service_Technorati_Result.

29.8.9. Zend_Service_Technorati_SearchResult

`Zend_Service_Technorati_SearchResult` represents a single Technorati Search query result object. It is never returned as a standalone object, but it always belongs to a valid Zend_Service_Technorati_SearchResultSet object.

`Zend_Service_Technorati_SearchResult` extends Zend_Service_Technorati_Result.

29.8.10. Zend_Service_Technorati_TagResult

`Zend_Service_Technorati_TagResult` represents a single Technorati Tag query result object. It is never returned as a standalone object, but it always belongs to a valid Zend_Service_Technorati_TagResultSet object.

`Zend_Service_Technorati_TagResult` extends Zend_Service_Technorati_Result.

29.8.11. Zend_Service_Technorati_DailyCountsResult

`Zend_Service_Technorati_DailyCountsResult` represents a single Technorati DailyCounts query result object. It is never returned as a standalone object, but it always belongs to a valid Zend_Service_Technorati_DailyCountsResultSet object.

`Zend_Service_Technorati_DailyCountsResult` extends Zend_Service_Technorati_Result.

29.8.12. Zend_Service_Technorati_TagsResult

`Zend_Service_Technorati_TagsResult` represents a single Technorati TopTags or BlogPostTags query result object. It is never returned as a standalone object, but it always belongs to a valid Zend_Service_Technorati_TagsResultSet object.

`Zend_Service_Technorati_TagsResult` extends Zend_Service_Technorati_Result.

29.8.13. Zend_Service_Technorati_GetInfoResult

`Zend_Service_Technorati_GetInfoResult` represents a single Technorati GetInfo query result object.

29.8.14. Zend_Service_Technorati_BlogInfoResult

`Zend_Service_Technorati_BlogInfoResult` represents a single Technorati BlogInfo query result object.

29.8.15. Zend_Service_Technorati_KeyInfoResult

`Zend_Service_Technorati_KeyInfoResult` represents a single Technorati KeyInfo query result object. It provides information about your Technorati API Key daily usage.

30. Zend_Service_Twitter

30.1. Introduction

`Zend_Service_Twitter` provides a client for the Twitter REST API. `Zend_Service_Twitter` allows you to query the public timeline. If you provide a username

and password for Twitter, it will allow you to get and update your status, reply to friends, direct message friends, mark tweets as favorite, and much more.

`Zend_Service_Twitter` is implementing a REST service, and all methods return an instance of `Zend_Rest_Client_Result`.

`Zend_Service_Twitter` is broken up into subsections so you can easily identify which type of call is being requested.

- `account` makes sure that your account credentials are valid, checks your API rate limit, and ends the current session for the authenticated user.
- `status` retrieves the public and user timelines and shows, updates, destroys, and retrieves replies for the authenticated user.
- `user` retrieves friends and followers for the authenticated user and returns extended information about a passed user.
- `directMessage` retrieves the authenticated user's received direct messages, deletes direct messages, and sends new direct messages.
- `friendship` creates and removes friendships for the authenticated user.
- `favorite` lists, creates, and removes favorite tweets.
- `block` blocks and unblocks users from following you.

30.2. Authentication

With the exception of fetching the public timeline, `Zend_Service_Twitter` requires authentication to work. Twitter currently uses HTTP Basic Authentication. You can pass in your username or registered email along with your password for Twitter to login.

Example 780. Creating the Twitter Class

The following code sample is how you create the Twitter service, pass in your username and password, and verify that they are correct.

```
$twitter = new Zend_Service_Twitter('myusername', 'mysecretpassword');

// verify your credentials with Twitter
$response = $twitter->account->verifyCredentials();
```

You can also pass in an array that contains the username and password as the first argument.

```
$userInfo   = array('username' => 'foo', 'password' => 'bar');
$twitter    = new Zend_Service_Twitter($userInfo);

// verify your credentials with Twitter
$response = $twitter->account->verifyCredentials();
```

30.3. Account Methods

- `verifyCredentials()` tests if supplied user credentials are valid with minimal overhead.

Example 781. Verifying credentials

```
$twitter    = new Zend_Service_Twitter('myusername', 'mysecretpassword');
$response   = $twitter->account->verifyCredentials();
```

- `endSession()` signs users out of client-facing applications.

  Example 782. Sessions ending

  ```
  $twitter    = new Zend_Service_Twitter('myusername', 'mysecretpassword');
  $response   = $twitter->account->endSession();
  ```

- `rateLimitStatus()` returns the remaining number of API requests available to the authenticating user before the API limit is reached for the current hour.

  Example 783. Rating limit status

  ```
  $twitter    = new Zend_Service_Twitter('myusername', 'mysecretpassword');
  $response   = $twitter->account->rateLimitStatus();
  ```

30.4. Status Methods

- `publicTimeline()` returns the 20 most recent statuses from non-protected users with a custom user icon. The public timeline is cached by Twitter for 60 seconds.

  Example 784. Retrieving public timeline

  ```
  $twitter    = new Zend_Service_Twitter('myusername', 'mysecretpassword');
  $response   = $twitter->status->publicTimeline();
  ```

- `friendsTimeline()` returns the 20 most recent statuses posted by the authenticating user and that user's friends.

  Example 785. Retrieving friends timeline

  ```
  $twitter    = new Zend_Service_Twitter('myusername', 'mysecretpassword');
  $response   = $twitter->status->friendsTimeline();
  ```

  The `friendsTimeline()` method accepts an array of optional parameters to modify the query.

  - `since` narrows the returned results to just those statuses created after the specified date/time (up to 24 hours old).
  - `page` specifies which page you want to return.

- `userTimeline()` returns the 20 most recent statuses posted from the authenticating user.

  Example 786. Retrieving user timeline

  ```
  $twitter    = new Zend_Service_Twitter('myusername', 'mysecretpassword');
  $response   = $twitter->status->userTimeline();
  ```

  The `userTimeline()` method accepts an array of optional parameters to modify the query.

- `id` specifies the ID or screen name of the user for whom to return the friends_timeline.
- `since` narrows the returned results to just those statuses created after the specified date/time (up to 24 hours old).
- `page` specifies which page you want to return.
- `count` specifies the number of statuses to retrieve. May not be greater than 200.

- `show()` returns a single status, specified by the `id` parameter below. The status' author will be returned inline.

Example 787. Showing user status

```
$twitter    = new Zend_Service_Twitter('myusername', 'mysecretpassword');
$response   = $twitter->status->show(1234);
```

- `update()` updates the authenticating user's status. This method requires that you pass in the status update that you want to post to Twitter.

Example 788. Updating user status

```
$twitter    = new Zend_Service_Twitter('myusername', 'mysecretpassword');
$response   = $twitter->status->update('My Great Tweet');
```

The `update()` method accepts a second additional parameter.

- `in_reply_to_status_id` specifies the ID of an existing status that the status to be posted is in reply to.

- `replies()` returns the 20 most recent @replies (status updates prefixed with @username) for the authenticating user.

Example 789. Showing user replies

```
$twitter    = new Zend_Service_Twitter('myusername', 'mysecretpassword');
$response   = $twitter->status->replies();
```

The `replies()` method accepts an array of optional parameters to modify the query.

- `since` narrows the returned results to just those statuses created after the specified date/time (up to 24 hours old).
- `page` specifies which page you want to return.
- `since_id` returns only statuses with an ID greater than (that is, more recent than) the specified ID.

- `destroy()` destroys the status specified by the required `id` parameter.

Example 790. Deleting user status

```
$twitter    = new Zend_Service_Twitter('myusername', 'mysecretpassword');
$response   = $twitter->status->destroy(12345);
```

30.5. User Methods

- `friends()`r eturns up to 100 of the authenticating user's friends who have most recently updated, each with current status inline.

  Example 791. Retrieving user friends

  ```
  $twitter    = new Zend_Service_Twitter('myusername', 'mysecretpassword');
  $response   = $twitter->user->friends();
  ```

  The `friends()` method accepts an array of optional parameters to modify the query.

  - `id` specifies the ID or screen name of the user for whom to return a list of friends.
  - `since` narrows the returned results to just those statuses created after the specified date/time (up to 24 hours old).
  - `page` specifies which page you want to return.

- `followers()` returns the authenticating user's followers, each with current status inline.

  Example 792. Retrieving user followers

  ```
  $twitter    = new Zend_Service_Twitter('myusername', 'mysecretpassword');
  $response   = $twitter->user->followers();
  ```

  The `followers()` method accepts an array of optional parameters to modify the query.

  - `id` specifies the ID or screen name of the user for whom to return a list of followers.
  - `page` specifies which page you want to return.

- `show()` returns extended information of a given user, specified by ID or screen name as per the required `id` parameter below.

  Example 793. Showing user informations

  ```
  $twitter    = new Zend_Service_Twitter('myusername', 'mysecretpassword');
  $response   = $twitter->user->show('myfriend');
  ```

30.6. Direct Message Methods

- `messages()` returns a list of the 20 most recent direct messages sent to the authenticating user.

  Example 794. Retrieving recent direct messages received

  ```
  $twitter    = new Zend_Service_Twitter('myusername', 'mysecretpassword');
  $response   = $twitter->directMessage->messages();
  ```

  The `message()` method accepts an array of optional parameters to modify the query.

  - `since_id` returns only direct messages with an ID greater than (that is, more recent than) the specified ID.

- `since` narrows the returned results to just those statuses created after the specified date/time (up to 24 hours old).
- `page` specifies which page you want to return.

- `sent()` returns a list of the 20 most recent direct messages sent by the authenticating user.

Example 795. Retrieving recent direct messages sent

```
$twitter    = new Zend_Service_Twitter('myusername', 'mysecretpassword');
$response   = $twitter->directMessage->sent();
```

The `sent()` method accepts an array of optional parameters to modify the query.

- `since_id` returns only direct messages with an ID greater than (that is, more recent than) the specified ID.
- `since` narrows the returned results to just those statuses created after the specified date/time (up to 24 hours old).
- `page` specifies which page you want to return.

- `new()` sends a new direct message to the specified user from the authenticating user. Requires both the user and text parameters below.

Example 796. Sending direct message

```
$twitter    = new Zend_Service_Twitter('myusername', 'mysecretpassword');
$response   = $twitter->directMessage->new('myfriend', 'mymessage');
```

- `destroy()` destroys the direct message specified in the required `id` parameter. The authenticating user must be the recipient of the specified direct message.

Example 797. Deleting direct message

```
$twitter    = new Zend_Service_Twitter('myusername', 'mysecretpassword');
$response   = $twitter->directMessage->destroy(123548);
```

30.7. Friendship Methods

- `create()` befriends the user specified in the `id` parameter with the authenticating user.

Example 798. Creating friend

```
$twitter    = new Zend_Service_Twitter('myusername', 'mysecretpassword');
$response   = $twitter->friendship->create('mynewfriend');
```

- `destroy()` discontinues friendship with the user specified in the `id` parameter and the authenticating user.

Example 799. Deleting friend

```
$twitter    = new Zend_Service_Twitter('myusername', 'mysecretpassword');
$response   = $twitter->friendship->destroy('myoldfriend');
```

- `exists()` tests if a friendship exists between the user specified in the `id` parameter and the authenticating user.

  Example 800. Checking friend existence

```
$twitter    = new Zend_Service_Twitter('myusername', 'mysecretpassword');
$response   = $twitter->friendship->exists('myfriend');
```

30.8. Favorite Methods

- `favorites()` returns the 20 most recent favorite statuses for the authenticating user or user specified by the `id` parameter.

  Example 801. Retrieving favorites

```
$twitter    = new Zend_Service_Twitter('myusername', 'mysecretpassword');
$response   = $twitter->favorite->favorites();
```

  The `favorites()` method accepts an array of optional parameters to modify the query.

  - `id` specifies the ID or screen name of the user for whom to request a list of favorite statuses.
  - `page` specifies which page you want to return.

- `create()` favorites the status specified in the `id` parameter as the authenticating user.

  Example 802. Creating favorites

```
$twitter    = new Zend_Service_Twitter('myusername', 'mysecretpassword');
$response   = $twitter->favorite->create(12351);
```

- `destroy()` un-favorites the status specified in the `id` parameter as the authenticating user.

  Example 803. Deleting favorites

```
$twitter    = new Zend_Service_Twitter('myusername', 'mysecretpassword');
$response   = $twitter->favorite->destroy(12351);
```

30.9. Block Methods

- `exists()` checks if the authenticating user is blocking a target user and can optionally return the blocked user's object if a block does exists.

  Example 804. Checking if block exists

```
$twitter = new Zend_Service_Twitter('myusername', 'mysecretpassword');

// returns true or false
$response = $twitter->block->exists('blockeduser');

// returns the blocked user's info if the user is blocked
$response2 = $twitter->block->exists('blockeduser', true);
```

  The `favorites()` method accepts a second optional parameter.

- `returnResult` specifies whether or not return the user object instead of just `TRUE` or `FALSE`.

- `create()` blocks the user specified in the `id` parameter as the authenticating user and destroys a friendship to the blocked user if one exists. Returns the blocked user in the requested format when successful.

  Example 805. Blocking a user

```
$twitter    = new Zend_Service_Twitter('myusername', 'mysecretpassword');
$response   = $twitter->block->create('usertoblock);
```

- `destroy()` un-blocks the user specified in the `id` parameter for the authenticating user. Returns the un-blocked user in the requested format when successful.

  Example 806. Removing a block

```
$twitter    = new Zend_Service_Twitter('myusername', 'mysecretpassword');
$response   = $twitter->block->destroy('blockeduser');
```

- `blocking()` returns an array of user objects that the authenticating user is blocking.

  Example 807. Who are you blocking

```
$twitter = new Zend_Service_Twitter('myusername', 'mysecretpassword');

// return the full user list from the first page
$response = $twitter->block->blocking();

// return an array of numeric user IDs from the second page
$response2 = $twitter->block->blocking(2, true);
```

  The `favorites()` method accepts two optional parameters.

  - `page` specifies which page ou want to return. A single page contains 20 IDs.

  - `returnUserIds` specifies whether to return an array of numeric user IDs the authenticating user is blocking instead of an array of user objects.

30.10. Zend_Service_Twitter_Search

30.10.1. Introduction

`Zend_Service_Twitter_Search` provides a client for the Twitter Search API. The Twitter Search service is use to search Twitter. Currently, it only returns data in Atom or JSON format, but a full REST service is in the future, which will support XML responses.

30.10.2. Twitter Trends

Returns the top ten queries that are currently trending on Twitter. The response includes the time of the request, the name of each trending topic, and the url to the Twitter Search results page for that topic. Currently the search API for trends only supports a JSON return so the function returns an array.

```
$twitterSearch  = new Zend_Service_Twitter_Search();
$twitterTrends  = $twitterSearch->trends();
```

```
foreach ($twitterTrends as $trend) {
    print $trend['name'] . ' - ' . $trend['url'] . PHP_EOL
}
```

The return array has two values in it:

- `name` is the name of trend.
- `url` is the URL to see the tweets for that trend.

30.10.3. Searching Twitter

Using the search method returns tweets that match a specific query. There are a number of Search Operators that you can use to query with.

The search method can accept six different optional URL parameters passed in as an array:

- `lang` restricts the tweets to a given language. `lang` must be given by an ISO 639-1 code.
- `rpp` is the number of tweets to return per page, up to a maximum of 100.
- `page` specifies the page number to return, up to a maximum of roughly 1500 results (based on rpp * page).
- `since_id` returns tweets with status IDs greater than the given ID.
- `show_user` specifies whether to add ">user<:" to the beginning of the tweet. This is useful for readers that do not display Atom's author field. The default is "FALSE".
- `geocode` returns tweets by users located within a given radius of the given latitude/longitude, where the user's location is taken from their Twitter profile. The parameter value is specified by "latitude,longitude,radius", where radius units must be specified as either "mi" (miles) or "km" (kilometers).

Example 808. JSON Search Example

The following code sample will return an array with the search results.

```
$twitterSearch  = new Zend_Service_Twitter_Search('json');
$searchResults  = $twitterSearch->search('zend', array('lang' => 'en'));
```

Example 809. ATOM Search Example

The following code sample will return a `Zend_Feed_Atom` object.

```
$twitterSearch  = new Zend_Service_Twitter_Search('atom');
$searchResults  = $twitterSearch->search('zend', array('lang' => 'en'));
```

30.10.4. Zend-specific Accessor Methods

While the Twitter Search API only specifies two methods, `Zend_Service_Twitter_Search` has additional methods that may be used for retrieving and modifying internal properties.

- `getResponseType()` and `setResponseType()` allow you to retrieve and modify the response type of the search between JSON and Atom.

31. Zend_Service_WindowsAzure

31.1. Introduction

Windows Azure is the name for Microsoft's Software + Services platform, an operating system in the cloud providing services for hosting, management, scalable storage with support for simple blobs, tables, and queues, as well as a management infrastructure for provisioning and geo-distribution of cloud-based services, and a development platform for the Azure Services layer.

31.2. Installing the Windows Azure SDK

There are two development scenario's when working with Windows Azure.

- You can develop your application using `Zend_Service_WindowsAzure` and the Windows Azure SDK, which provides a local developent environment of the services provided by Windows Azure's cloud infrastructure.
- You can develop your application using `Zend_Service_WindowsAzure`, working directly with the Windows Azure cloud infrastructure.

The first case requires you to install the Windows Azure SDK on your development machine. It is currently only available for Windows environments; progress is being made on a Java-based version of the SDK which can run on any platform.

The latter case requires you to have an account at Azure.com.

31.3. API Documentation

The `Zend_Service_WindowsAzure` class provides the PHP wrapper to the Windows Azure REST interface. Please consult the REST documentation for detailed description of the service. You will need to be familiar with basic concepts in order to use this service.

31.4. Features

`Zend_Service_WindowsAzure` provides the following functionality:

- PHP classes for Windows Azure Blobs, Tables and Queues (for CRUD operations)
- Helper Classes for HTTP transport, AuthN/AuthZ, REST and Error Management
- Manageability, Instrumentation and Logging support

31.5. Architecture

`Zend_Service_WindowsAzure` provides access to Windows Azure's storage, computation and management interfaces by abstracting the REST/XML interface Windows Azure provides into a simple PHP API.

An application built using `Zend_Service_WindowsAzure` can access Windows Azure's features, no matter if it is hosted on the Windows Azure platform or on an in-premise web server.

31.6. Zend_Service_WindowsAzure_Storage_Blob

Blob Storage stores sets of binary data. Blob storage offers the following three resources: the storage account, containers, and blobs. Within your storage account, containers provide a way to organize sets of blobs within your storage account.

Blob Storage is offered by Windows Azure as a REST API which is wrapped by the `Zend_Service_WindowsAzure_Storage_Blob` class in order to provide a native PHP interface to the storage account.

31.6.1. API Examples

This topic lists some examples of using the `Zend_Service_WindowsAzure_Storage_Blob` class. Other features are available in the download package, as well as a detailed API documentation of those features.

31.6.1.1. Creating a storage container

Using the following code, a blob storage container can be created on development storage.

Example 810. Creating a storage container

```
$storageClient = new Zend_Service_WindowsAzure_Storage_Blob();
$result = $storageClient->createContainer('testcontainer');

echo 'Container name is: ' . $result->Name;
```

31.6.1.2. Deleting a storage container

Using the following code, a blob storage container can be removed from development storage.

Example 811. Deleting a storage container

```
$storageClient = new Zend_Service_WindowsAzure_Storage_Blob();
$storageClient->deleteContainer('testcontainer');
```

31.6.1.3. Storing a blob

Using the following code, a blob can be uploaded to a blob storage container on development storage. Note that the container has already been created before.

Example 812. Storing a blob

```
$storageClient = new Zend_Service_WindowsAzure_Storage_Blob();

// upload /home/maarten/example.txt to Azure
$result = $storageClient->putBlob(
    'testcontainer', 'example.txt', '/home/maarten/example.txt'
);

echo 'Blob name is: ' . $result->Name;
```

31.6.1.4. Copying a blob

Using the following code, a blob can be copied from inside the storage account. The advantage of using this method is that the copy operation occurs in the Azure cloud and does not involve downloading the blob. Note that the container has already been created before.

Example 813. Copying a blob

```
$storageClient = new Zend_Service_WindowsAzure_Storage_Blob();

// copy example.txt to example2.txt
$result = $storageClient->copyBlob(
    'testcontainer', 'example.txt', 'testcontainer', 'example2.txt'
);

echo 'Copied blob name is: ' . $result->Name;
```

31.6.1.5. Downloading a blob

Using the following code, a blob can be downloaded from a blob storage container on development storage. Note that the container has already been created before and a blob has been uploaded.

Example 814. Downloading a blob

```
$storageClient = new Zend_Service_WindowsAzure_Storage_Blob();

// download file to /home/maarten/example.txt
$storageClient->getBlob(
    'testcontainer', 'example.txt', '/home/maarten/example.txt'
);
```

31.6.1.6. Making a blob publicly available

By default, blob storage containers on Windows Azure are protected from public viewing. If any user on the Internet should have access to a blob container, its ACL can be set to public. Note that this applies to a complete container and not to a single blob!

Using the following code, blob storage container ACL can be set on development storage. Note that the container has already been created before.

Example 815. Making a blob publicly available

```
$storageClient = new Zend_Service_WindowsAzure_Storage_Blob();

// make container publicly available
$storageClient->setContainerAcl('testcontainer', Zend_Service_WindowsAzure_Storage_Blob
```

31.6.2. Root container

Windows Azure Blob Storage provides support to work with a "root container". This means that a blob can be stored in the root of your storage account, i.e. `http://myaccount.blob.core.windows.net/somefile.txt`.

In order to work with the root container, it should first be created using the `createContainer()` method, naming the container `$root`. All other operations on the root container should be issued with the container name set to `$root`.

31.6.3. Blob storage stream wrapper

The Windows Azure SDK for PHP provides support for registering a blob storage client as a PHP file stream wrapper. The blob storage stream wrapper provides support for using regular

file operations on Windows Azure Blob Storage. For example, one can open a file from Windows Azure Blob Storage with the `fopen()` function:

Example 816. Example usage of blob storage stream wrapper

```
$fileHandle = fopen('azure://mycontainer/myfile.txt', 'r');

// ...

fclose($fileHandle);
```

In order to do this, the Windows Azure SDK for PHP blob storage client must be registered as a stream wrapper. This can be done by calling the `registerStreamWrapper()` method:

Example 817. Registering the blob storage stream wrapper

```
$storageClient = new Zend_Service_WindowsAzure_Storage_Blob();
$storageClient->registerStreamWrapper(); // registers azure:// on this storage client

// or:

$storageClient->registerStreamWrapper('blob://'); // regiters blob:// on this storage c
```

To unregister the stream wrapper, the `unregisterStreamWrapper()` method can be used.

31.6.4. Shared Access Signature

Windows Azure Bob Storage provides a feature called "Shared Access Signatures". By default, there is only one level of authorization possible in Windows Azure Blob Storage: either a container is private or it is public. Shared Access Signatures provide a more granular method of authorization: read, write, delete and list permissions can be assigned on a container or a blob and given to a specific client using an URL-based model.

An example would be the following signature:

```
http://phpstorage.blob.core.windows.net/phpazuretestshared1?st=2009-08-17T09%3A06%3A17Z&se=2009-08-17T09%3A5
```

The above signature gives write access to the "phpazuretestshared1" container of the "phpstorage" account.

31.6.4.1. Generating a Shared Access Signature

When you are the owner of a Windows Azure Bob Storage account, you can create and distribute a shared access key for any type of resource in your account. To do this, the `generateSharedAccessUrl()` method of the `Zend_Service_WindowsAzure_Storage_Blob` storage client can be used.

The following example code will generate a Shared Access Signature for write access in a container named "container1", within a timeframe of 3000 seconds.

Example 818. Generating a Shared Access Signature for a container

```
$storageClient   = new Zend_Service_WindowsAzure_Storage_Blob();
$sharedAccessUrl = storageClient->generateSharedAccessUrl(
    'container1',
    '',
    'c',
    'w',
    $storageClient ->isoDate(time() - 500),
    $storageClient ->isoDate(time() + 3000)
);
```

The following example code will generate a Shared Access Signature for read access in a blob named `test.txt` in a container named "container1" within a time frame of 3000 seconds.

Example 819. Generating a Shared Access Signature for a blob

```
$storageClient   = new Zend_Service_WindowsAzure_Storage_Blob();
$sharedAccessUrl = storageClient->generateSharedAccessUrl(
    'container1',
    'test.txt',
    'b',
    'r',
    $storageClient ->isoDate(time() - 500),
    $storageClient ->isoDate(time() + 3000)
);
```

31.6.4.2. Working with Shared Access Signatures from others

When you receive a Shared Access Signature from someone else, you can use the Windows Azure SDK for PHP to work with the addressed resource. For example, the following signature can be retrieved from the owner of a storage account:

```
http://phpstorage.blob.core.windows.net/phpazuretestshared1?st=2009-08-17T09%3A06%3A17Z&se=2009-08-17T09%3A5
```

The above signature gives write access to the "phpazuretestshared1" "container" of the phpstorage account. Since the shared key for the account is not known, the Shared Access Signature can be used to work with the authorized resource.

Example 820. Consuming a Shared Access Signature for a container

```
$storageClient = new Zend_Service_WindowsAzure_Storage_Blob(
    'blob.core.windows.net', 'phpstorage', ''
);
$storageClient->setCredentials(
    new Zend_Service_WindowsAzure_Credentials_SharedAccessSignature()
);
$storageClient->getCredentials()->setPermissionSet(array(
    'http://phpstorage.blob.core.windows.net/phpazuretestshared1?st=2009-08-17T09%3A06%3
));
$storageClient->putBlob(
    'phpazuretestshared1', 'NewBlob.txt', 'C:\Files\dataforazure.txt'
);
```

Note that there was no explicit permission to write to a specific blob. Instead, the Windows Azure SDK for PHP determined that a permission was required to either write to that specific blob, or

to write to its container. Since only a signature was available for the latter, the Windows Azure SDK for PHP chose those credentials to perform the request on Windows Azure blob storage.

31.7. Zend_Service_WindowsAzure_Storage_Table

The Table service offers structured storage in the form of tables.

Table Storage is offered by Windows Azure as a REST API which is wrapped by the `Zend_Service_WindowsAzure_Storage_Table` class in order to provide a native PHP interface to the storage account.

This topic lists some examples of using the `Zend_Service_WindowsAzure_Storage_Table` class. Other features are available in the download package, as well as a detailed API documentation of those features.

Note that development table storage (in the Windows Azure SDK) does not support all features provided by the API. Therefore, the examples listed on this page are to be used on Windows Azure production table storage.

31.7.1. Operations on tables

This topic lists some samples of operations that can be executed on tables.

31.7.1.1. Creating a table

Using the following code, a table can be created on Windows Azure production table storage.

Example 821. Creating a table

```
$storageClient = new Zend_Service_WindowsAzure_Storage_Table(
    'table.core.windows.net', 'myaccount', 'myauthkey'
);
$result = $storageClient->createTable('testtable');

echo 'New table name is: ' . $result->Name;
```

31.7.1.2. Listing all tables

Using the following code, a list of all tables in Windows Azure production table storage can be queried.

Example 822. Listing all tables

```
$storageClient = new Zend_Service_WindowsAzure_Storage_Table(
    'table.core.windows.net', 'myaccount', 'myauthkey'
);
$result = $storageClient->listTables();
foreach ($result as $table) {
    echo 'Table name is: ' . $table->Name . "\r\n";
}
```

31.7.2. Operations on entities

Tables store data as collections of entities. Entities are similar to rows. An entity has a primary key and a set of properties. A property is a named, typed-value pair, similar to a column.

The Table service does not enforce any schema for tables, so two entities in the same table may have different sets of properties. Developers may choose to enforce a schema on the client side. A table may contain any number of entities.

`Zend_Service_WindowsAzure_Storage_Table` provides 2 ways of working with entities:

- Enforced schema
- No enforced schema

All examples will make use of the following enforced schema class.

Example 823. Enforced schema used in samples

```
class SampleEntity extends Zend_Service_WindowsAzure_Storage_TableEntity
{
    /**
    * @azure Name
    */
    public $Name;

    /**
    * @azure Age Edm.Int64
    */
    public $Age;

    /**
    * @azure Visible Edm.Boolean
    */
    public $Visible = false;
}
```

Note that if no schema class is passed into table storage methods, `Zend_Service_WindowsAzure_Storage_Table` automatically works with `Zend_Service_WindowsAzure_Storage_DynamicTableEntity`.

31.7.2.1. Enforced schema entities

To enforce a schema on the client side using the `Zend_Service_WindowsAzure_Storage_Table` class, you can create a class which inherits `Zend_Service_WindowsAzure_Storage_TableEntity`. This class provides some basic functionality for the `Zend_Service_WindowsAzure_Storage_Table` class to work with a client-side schema.

Base properties provided by `Zend_Service_WindowsAzure_Storage_TableEntity` are:

- PartitionKey (exposed through `getPartitionKey()` and `setPartitionKey()`)
- RowKey (exposed through `getRowKey()` and `setRowKey()`)
- Timestamp (exposed through `getTimestamp()` and `setTimestamp()`)
- Etag value (exposed through `getEtag()` and `setEtag()`)

Here's a sample class inheriting `Zend_Service_WindowsAzure_Storage_TableEntity`:

Example 824. Sample enforced schema class

```
class SampleEntity extends Zend_Service_WindowsAzure_Storage_TableEntity
{
    /**
     * @azure Name
     */
    public $Name;

    /**
     * @azure Age Edm.Int64
     */
    public $Age;

    /**
     * @azure Visible Edm.Boolean
     */
    public $Visible = false;
}
```

The `Zend_Service_WindowsAzure_Storage_Table` class will map any class inherited from `Zend_Service_WindowsAzure_Storage_TableEntity` to Windows Azure table storage entities with the correct data type and property name. All there is to storing a property in Windows Azure is adding a docblock comment to a public property or public getter/setter, in the following format:

Example 825. Enforced property

```
/**
 * @azure <property name in Windows Azure> <optional property type>
 */
public $<property name in PHP>;
```

Let's see how to define a propety "Age" as an integer on Windows Azure table storage:

Example 826. Sample enforced property

```
/**
 * @azure Age Edm.Int64
 */
public $Age;
```

Note that a property does not necessarily have to be named the same on Windows Azure table storage. The Windows Azure table storage property name can be defined as well as the type.

The following data types are supported:

- `Edm.Binary` - An array of bytes up to 64 KB in size.
- `Edm.Boolean` - A boolean value.
- `Edm.DateTime` - A 64-bit value expressed as Coordinated Universal Time (UTC). The supported DateTime range begins from 12:00 midnight, January 1, 1601 A.D. (C.E.), Coordinated Universal Time (UTC). The range ends at December 31st, 9999.
- `Edm.Double` - A 64-bit floating point value.

- `Edm.Guid` - A 128-bit globally unique identifier.
- `Edm.Int32` - A 32-bit integer.
- `Edm.Int64` - A 64-bit integer.
- `Edm.String` - A UTF-16-encoded value. String values may be up to 64 KB in size.

31.7.2.2. No enforced schema entities (a.k.a. DynamicEntity)

To use the `Zend_Service_WindowsAzure_Storage_Table` class without defining a schema, you can make use of the `Zend_Service_WindowsAzure_Storage_DynamicTableEntity` class. This class inherits `Zend_Service_WindowsAzure_Storage_TableEntity` like an enforced schema class does, but contains additional logic to make it dynamic and not bound to a schema.

Base properties provided by `Zend_Service_WindowsAzure_Storage_DynamicTableEntity` are:

- PartitionKey (exposed through `getPartitionKey()` and `setPartitionKey()`)
- RowKey (exposed through `getRowKey()` and `setRowKey()`)
- Timestamp (exposed through `getTimestamp()` and `setTimestamp()`)
- Etag value (exposed through `getEtag()` and `setEtag()`)

Other properties can be added on the fly. Their Windows Azure table storage type will be determined on-the-fly:

Example 827. Dynamicaly adding properties to Zend_Service_WindowsAzure_Storage_DynamicTableEntity

```
$target = new Zend_Service_WindowsAzure_Storage_DynamicTableEntity(
    'partition1', '000001'
);
$target->Name = 'Name'; // Will add property "Name" of type "Edm.String"
$target->Age  = 25;     // Will add property "Age" of type "Edm.Int32"
```

Optionally, a property type can be enforced:

Example 828. Forcing property types on Zend_Service_WindowsAzure_Storage_DynamicTableEntity

```
$target = new Zend_Service_WindowsAzure_Storage_DynamicTableEntity(
    'partition1', '000001'
);
$target->Name = 'Name'; // Will add property "Name" of type "Edm.String"
$target->Age  = 25;     // Will add property "Age" of type "Edm.Int32"

// Change type of property "Age" to "Edm.Int32":
$target->setAzurePropertyType('Age', 'Edm.Int64');
```

The `Zend_Service_WindowsAzure_Storage_Table` class automatically works with `Zend_Service_WindowsAzure_Storage_TableEntity` if no specific class is passed into Table Storage methods.

31.7.2.3. Entities API examples

31.7.2.3.1. Inserting an entity

Using the following code, an entity can be inserted into a table named "testtable". Note that the table has already been created before.

Example 829. Inserting an entity

```
$entity = new SampleEntity ('partition1', 'row1');
$entity->FullName = "Maarten";
$entity->Age = 25;
$entity->Visible = true;

$storageClient = new Zend_Service_WindowsAzure_Storage_Table(
    'table.core.windows.net', 'myaccount', 'myauthkey'
);
$result = $storageClient->insertEntity('testtable', $entity);

// Check the timestamp and etag of the newly inserted entity
echo 'Timestamp: ' . $result->getTimestamp() . "\n";
echo 'Etag: ' . $result->getEtag() . "\n";
```

31.7.2.3.2. Retrieving an entity by partition key and row key

Using the following code, an entity can be retrieved by partition key and row key. Note that the table and entity have already been created before.

Example 830. Retrieving an entity by partition key and row key

```
$storageClient = new Zend_Service_WindowsAzure_Storage_Table(
    'table.core.windows.net', 'myaccount', 'myauthkey'
);
$entity= $storageClient->retrieveEntityById(
    'testtable', 'partition1', 'row1', 'SampleEntity'
);
```

31.7.2.3.3. Updating an entity

Using the following code, an entity can be updated. Note that the table and entity have already been created before.

Example 831. Updating an entity

```
$storageClient = new Zend_Service_WindowsAzure_Storage_Table(
    'table.core.windows.net', 'myaccount', 'myauthkey'
);
$entity = $storageClient->retrieveEntityById(
    'testtable', 'partition1', 'row1', 'SampleEntity'
);

$entity->Name = 'New name';
$result = $storageClient->updateEntity('testtable', $entity);
```

If you want to make sure the entity has not been updated before, you can make sure the Etag of the entity is checked. If the entity already has had an update, the update will fail to make sure you do not overwrite any newer data.

Example 832. Updating an entity (with Etag check)

```
$storageClient = new Zend_Service_WindowsAzure_Storage_Table(
    'table.core.windows.net', 'myaccount', 'myauthkey'
);
$entity = $storageClient->retrieveEntityById(
    'testtable', 'partition1', 'row1', 'SampleEntity'
);

$entity->Name = 'New name';

// last parameter instructs the Etag check:
$result = $storageClient->updateEntity('testtable', $entity, true);
```

31.7.2.3.4. Deleting an entity

Using the following code, an entity can be deleted. Note that the table and entity have already been created before.

Example 833. Deleting an entity

```
$storageClient = new Zend_Service_WindowsAzure_Storage_Table(
    'table.core.windows.net', 'myaccount', 'myauthkey'
);
$entity = $storageClient->retrieveEntityById(
    'testtable', 'partition1', 'row1', 'SampleEntity'
);
$result = $storageClient->deleteEntity('testtable', $entity);
```

31.7.2.4. Performing queries

Queries in `Zend_Service_WindowsAzure_Storage_Table` table storage can be performed in two ways:

- By manually creating a filter condition (involving learning a new query language)
- By using the fluent interface provided by the `Zend_Service_WindowsAzure_Storage_Table`

Using the following code, a table can be queried using a filter condition. Note that the table and entities have already been created before.

Example 834. Performing queries using a filter condition

```
$storageClient = new Zend_Service_WindowsAzure_Storage_Table(
    'table.core.windows.net', 'myaccount', 'myauthkey'
);
$entities = $storageClient->storageClient->retrieveEntities(
    'testtable',
    'Name eq \'Maarten\' and PartitionKey eq \'partition1\'',
    'SampleEntity'
);

foreach ($entities as $entity) {
    echo 'Name: ' . $entity->Name . "\n";
}
```

Using the following code, a table can be queried using a fluent interface. Note that the table and entities have already been created before.

Example 835. Performing queries using a fluent interface

```
$storageClient = new Zend_Service_WindowsAzure_Storage_Table(
    'table.core.windows.net', 'myaccount', 'myauthkey'
);
$entities = $storageClient->storageClient->retrieveEntities(
    'testtable',
    $storageClient->select()
                  ->from($tableName)
                  ->where('Name eq ?', 'Maarten')
                  ->andWhere('PartitionKey eq ?', 'partition1'),
    'SampleEntity'
);

foreach ($entities as $entity) {
    echo 'Name: ' . $entity->Name . "\n";
}
```

31.7.2.5. Batch operations

This topic demonstrates how to use the table entity group transaction features provided by Windows Azure table storage. Windows Azure table storage supports batch transactions on entities that are in the same table and belong to the same partition group. A transaction can include at most 100 entities.

The following example uses a batch operation (transaction) to insert a set of entities into the "testtable" table. Note that the table has already been created before.

Example 836. Executing a batch operation

```
$storageClient = new Zend_Service_WindowsAzure_Storage_Table(
    'table.core.windows.net', 'myaccount', 'myauthkey'
);

// Start batch
$batch = $storageClient->startBatch();

// Insert entities in batch
$entities = generateEntities();
foreach ($entities as $entity) {
    $storageClient->insertEntity($tableName, $entity);
}

// Commit
$batch->commit();
```

31.7.3. Table storage session handler

When running a PHP application on the Windows Azure platform in a load-balanced mode (running 2 Web Role instances or more), it is important that PHP session data can be shared between multiple Web Role instances. The Windows Azure SDK for PHP provides the `Zend_Service_WindowsAzure_SessionHandler` class, which uses Windows Azure Table Storage as a session handler for PHP applications.

To use the Zend_Service_WindowsAzure_SessionHandler session handler, it should be registered as the default session handler for your PHP application:

Example 837. Registering table storage session handler

```
$storageClient = new Zend_Service_WindowsAzure_Storage_Table(
    'table.core.windows.net', 'myaccount', 'myauthkey'
);

$sessionHandler = new Zend_Service_WindowsAzure_SessionHandler(
    $storageClient , 'sessionstable'
);
$sessionHandler->register();
```

The above classname registers the Zend_Service_WindowsAzure_SessionHandler session handler and will store sessions in a table called "sessionstable".

After registration of the Zend_Service_WindowsAzure_SessionHandler session handler, sessions can be started and used in the same way as a normal PHP session:

Example 838. Using table storage session handler

```
$storageClient = new Zend_Service_WindowsAzure_Storage_Table(
    'table.core.windows.net', 'myaccount', 'myauthkey'
);

$sessionHandler = new Zend_Service_WindowsAzure_SessionHandler(
    $storageClient , 'sessionstable'
);
$sessionHandler->register();

session_start();

if (!isset($_SESSION['firstVisit'])) {
    $_SESSION['firstVisit'] = time();
}

// ...
```

The Zend_Service_WindowsAzure_SessionHandler session handler should be registered before a call to session_start() is made!

31.8. Zend_Service_WindowsAzure_Storage_Queue

The Queue service stores messages that may be read by any client who has access to the storage account.

A queue can contain an unlimited number of messages, each of which can be up to 8 KB in size. Messages are generally added to the end of the queue and retrieved from the front of the queue, although first in/first out (FIFO) behavior is not guaranteed. If you need to store messages larger than 8 KB, you can store message data as a queue or in a table and then store a reference to the data as a message in a queue.

Queue Storage is offered by Windows Azure as a REST API which is wrapped by the Zend_Service_WindowsAzure_Storage_Queue class in order to provide a native PHP interface to the storage account.

31.8.1. API Examples

This topic lists some examples of using the `Zend_Service_WindowsAzure_Storage_Queue` class. Other features are available in the download package, as well as a detailed API documentation of those features.

31.8.1.1. Creating a queue

Using the following code, a queue can be created on development storage.

Example 839. Creating a queue

```
$storageClient = new Zend_Service_WindowsAzure_Storage_Queue();
$result = $storageClient->createQueue('testqueue');

echo 'Queue name is: ' . $result->Name;
```

31.8.1.2. Deleting a queue

Using the following code, a queue can be removed from development storage.

Example 840. Deleting a queue

```
$storageClient = new Zend_Service_WindowsAzure_Storage_Queue();
$storageClient->deleteQueue('testqueue');
```

31.8.1.3. Adding a message to a queue

Using the following code, a message can be added to a queue on development storage. Note that the queue has already been created before.

Example 841. Adding a message to a queue

```
$storageClient = new Zend_Service_WindowsAzure_Storage_Queue();

// 3600 = time-to-live of the message, if omitted defaults to 7 days
$storageClient->putMessage('testqueue', 'This is a test message', 3600);
```

31.8.1.4. Reading a message from a queue

Using the following code, a message can be read from a queue on development storage. Note that the queue and message have already been created before.

Example 842. Reading a message from a queue

```
$storageClient = new Zend_Service_WindowsAzure_Storage_Queue();

// retrieve 10 messages at once
$messages = $storageClient->getMessages('testqueue', 10);

foreach ($messages as $message) {
    echo $message->MessageText . "\r\n";
}
```

The messages that are read using `getMessages()` will be invisible in the queue for 30 seconds, after which the messages will re-appear in the queue. To mark a message as processed and remove it from the queue, use the `deleteMessage()` method.

Example 843. Marking a message as processed

```
$storageClient = new Zend_Service_WindowsAzure_Storage_Queue();

// retrieve 10 messages at once
$messages = $storageClient->getMessages('testqueue', 10);

foreach ($messages as $message) {
    echo $message . "\r\n";

    // Mark the message as processed
    $storageClient->deleteMessage('testqueue', $message);
}
```

31.8.1.5. Check if there are messages in a queue

Using the following code, a queue can be checked for new messages. Note that the queue and message have already been created before.

Example 844. Check if there are messages in a queue

```
$storageClient = new Zend_Service_WindowsAzure_Storage_Queue();

// retrieve 10 messages at once
$messages = $storageClient->peekMessages('testqueue', 10);

foreach ($messages as $message) {
    echo $message->MessageText . "\r\n";
}
```

Note that messages that are read using `peekMessages()` will not become invisible in the queue, nor can they be marked as processed using the `deleteMessage()` method. To do this, use `getMessages()` instead.

32. Zend_Service_Yahoo

32.1. Introduction

`Zend_Service_Yahoo` is a simple API for using many of the Yahoo! REST APIs. `Zend_Service_Yahoo` allows you to search Yahoo! Web search, Yahoo! News, Yahoo! Local, Yahoo! Images. In order to use the Yahoo! REST API, you must have a Yahoo! Application ID. To obtain an Application ID, please complete and submit the Application ID Request Form.

32.2. Searching the Web with Yahoo!

`Zend_Service_Yahoo` enables you to search the Web with Yahoo! using the `webSearch()` method, which accepts a string query parameter and an optional second parameter as an array of search options. For full details and an option list, please visit the Yahoo! Web Search Documentation. The `webSearch()` method returns a `Zend_Service_Yahoo_WebResultSet` object.

Example 845. Searching the Web with Yahoo!

```
$yahoo = new Zend_Service_Yahoo("YAHOO_APPLICATION_ID");
$results = $yahoo->webSearch('PHP');
foreach ($results as $result) {
    echo $result->Title .'<br />';
}
```

32.3. Finding Images with Yahoo!

You can search for Images with Yahoo using `Zend_Service_Yahoo`'s `imageSearch()` method. This method accepts a string query parameter and an optional array of search options, as for the `webSearch()` method. For full details and an option list, please visit the Yahoo! Image Search Documentation.

Example 846. Finding Images with Yahoo!

```
$yahoo = new Zend_Service_Yahoo("YAHOO_APPLICATION_ID");
$results = $yahoo->imageSearch('PHP');
foreach ($results as $result) {
    echo $result->Title .'<br />';
}
```

32.4. Finding videos with Yahoo!

You can search for videos with Yahoo using `Zend_Service_Yahoo`'s `videoSearch()` method. For full details and an option list, please visit the Yahoo! Video Search Documentation.

Example 847. Finding videos with Yahoo!

```
$yahoo = new Zend_Service_Yahoo("YAHOO_APPLICATION_ID");
$results = $yahoo->videoSearch('PHP');
foreach ($results as $result) {
    echo $result->Title .'<br />';
}
```

32.5. Finding Local Businesses and Services with Yahoo!

You can search for local businesses and services with Yahoo! by using the `localSearch()` method. For full details, please see the Yahoo! Local Search Documentation.

Example 848. Finding Local Businesses and Services with Yahoo!

```
$yahoo = new Zend_Service_Yahoo("YAHOO_APPLICATION_ID");
$results = $yahoo->localSearch('Apple Computers', array('zip' => '95014'));
foreach ($results as $result) {
    echo $result->Title .'<br />';
}
```

32.6. Searching Yahoo! News

Searching Yahoo! News is simple; just use the `newsSearch()` method, as in the following example. For full details, please see the Yahoo! News Search Documentation.

Example 849. Searching Yahoo! News

```
$yahoo = new Zend_Service_Yahoo("YAHOO_APPLICATION_ID");
$results = $yahoo->newsSearch('PHP');
foreach ($results as $result) {
    echo $result->Title .'<br />';
}
```

32.7. Searching Yahoo! Site Explorer Inbound Links

Searching Yahoo! Site Explorer Inbound Links is simple; just use the `inlinkDataSearch()` method, as in the following example. For full details, please see the Yahoo! Site Explorer Inbound Links Documentation.

Example 850. Searching Yahoo! Site Explorer Inbound Links

```
$yahoo = new Zend_Service_Yahoo("YAHOO_APPLICATION_ID");
$results = $yahoo->inlinkDataSearch('http://framework.zend.com/');
foreach ($results as $result) {
    echo $result->Title .'<br />';
}
```

32.8. Searching Yahoo! Site Explorer's PageData

Searching Yahoo! Site Explorer's PageData is simple; just use the `pageDataSearch()` method, as in the following example. For full details, please see the Yahoo! Site Explorer PageData Documentation.

Example 851. Searching Yahoo! Site Explorer's PageData

```
$yahoo = new Zend_Service_Yahoo("YAHOO_APPLICATION_ID");
$results = $yahoo->pageDataSearch('http://framework.zend.com/');
foreach ($results as $result) {
    echo $result->Title .'<br />';
}
```

32.9. Zend_Service_Yahoo Classes

The following classes are all returned by the various Yahoo! searches. Each search type returns a type-specific result set which can be easily iterated, with each result being contained in a type result object. All result set classes implement the `SeekableIterator` interface, allowing for easy iteration and seeking to a specific result.

- `Zend_Service_Yahoo_ResultSet`
- `Zend_Service_Yahoo_WebResultSet`
- `Zend_Service_Yahoo_ImageResultSet`
- `Zend_Service_Yahoo_VideoResultSet`
- `Zend_Service_Yahoo_LocalResultSet`
- `Zend_Service_Yahoo_NewsResultSet`

- Zend_Service_Yahoo_InlinkDataResultSet
- Zend_Service_Yahoo_PageDataResultSet
- Zend_Service_Yahoo_Result
- Zend_Service_Yahoo_WebResult
- Zend_Service_Yahoo_ImageResult
- Zend_Service_Yahoo_VideoResult
- Zend_Service_Yahoo_LocalResult
- Zend_Service_Yahoo_NewsResult
- Zend_Service_Yahoo_InlinkDataResult
- Zend_Service_Yahoo_PageDataResult
- Zend_Service_Yahoo_Image

32.9.1. Zend_Service_Yahoo_ResultSet

Each of the search specific result sets is extended from this base class.

Each of the specific result sets returns a search specific Zend_Service_Yahoo_Result objects.

32.9.1.1. Zend_Service_Yahoo_ResultSet::totalResults()

```
int totalResults();
```

Returns the number of results returned for the search.

32.9.1.2. Properties

Table 138. Zend_Service_Yahoo_ResultSet

Name	Type	Description
totalResultsAvailable	int	Total number of results found.
totalResultsReturned	int	Number of results in the current result set
firstResultPosition	int	Position of the first result in this set relative to the total number of results.

Back to Class List

32.9.2. Zend_Service_Yahoo_WebResultSet

`Zend_Service_Yahoo_WebResultSet` represents a Yahoo! Web Search result set.

`Zend_Service_Yahoo_WebResultSet` extends Zend_Service_Yahoo_ResultSet

Back to Class List

32.9.3. Zend_Service_Yahoo_ImageResultSet

`Zend_Service_Yahoo_ImageResultSet` represents a Yahoo! Image Search result set.

`Zend_Service_Yahoo_ImageResultSet` extends Zend_Service_Yahoo_ResultSet

Back to Class List

32.9.4. Zend_Service_Yahoo_VideoResultSet

`Zend_Service_Yahoo_VideoResultSet` represents a Yahoo! Video Search result set.

`Zend_Service_Yahoo_VideoResultSet` extends Zend_Service_Yahoo_ResultSet

Back to Class List

32.9.5. Zend_Service_Yahoo_LocalResultSet

`Zend_Service_Yahoo_LocalResultSet` represents a Yahoo! Local Search result set.

Table 139. Zend_Service_Yahoo_LocalResultSet Properties

Name	Type	Description
resultSetMapURL	string	The URL of a webpage containing a map graphic with all returned results plotted on it.

`Zend_Service_Yahoo_LocalResultSet` extends Zend_Service_Yahoo_ResultSet

Back to Class List

32.9.6. Zend_Service_Yahoo_NewsResultSet

`Zend_Service_Yahoo_NewsResultSet` represents a Yahoo! News Search result set.

`Zend_Service_Yahoo_NewsResultSet` extends Zend_Service_Yahoo_ResultSet

Back to Class List

32.9.7. Zend_Service_Yahoo_InlinkDataResultSet

`Zend_Service_Yahoo_InlinkDataResultSet` represents a Yahoo! Inbound Link Search result set.

`Zend_Service_Yahoo_InlinkDataResultSet` extends `Zend_Service_Yahoo_ResultSet`

Back to Class List

32.9.8. Zend_Service_Yahoo_PageDataResultSet

`Zend_Service_Yahoo_PageDataResultSet` represents a Yahoo! PageData Search result set.

`Zend_Service_Yahoo_PageDataResultSet` extends `Zend_Service_Yahoo_ResultSet`

Back to Class List

32.9.9. Zend_Service_Yahoo_Result

Each of the search specific results is extended from this base class.

32.9.9.1. Properties

Table 140. Zend_Service_Yahoo_Result Properties

Name	Type	Description
Title	string	Title of the Result item
Url	string	The URL of the result item
ClickUrl	string	The URL for linking to the result item

Back to Class List

32.9.10. Zend_Service_Yahoo_WebResult

Each Web Search result is returned as a `Zend_Service_Yahoo_WebResult` object.

32.9.10.1. Properties

Table 141. Zend_Service_Yahoo_WebResult Properties

Name	Type	Description
Summary	string	Result summary
MimeType	string	Result MIME type
ModificationDate	string	The last modification date of the result as a UNIX timestamp.
CacheUrl	string	Yahoo! web cache URL for the result, if it exists.
CacheSize	int	The size of the Cache entry

Back to Class List

32.9.11. Zend_Service_Yahoo_ImageResult

Each Image Search result is returned as a `Zend_Service_Yahoo_ImageResult` object.

32.9.11.1. Properties

Table 142. Zend_Service_Yahoo_ImageResult Properties

Name	Type	Description
Summary	string	Result summary
RefererUrl	string	The URL of the page which contains the image
FileSize	int	The size of the image file in bytes
FileFormat	string	The format of the image (bmp, gif, jpeg, png, etc.)
Height	int	The height of the image
Width	int	The width of the image
Thumbnail	Zend_Service_Yahoo_Image	Image thumbnail

Back to Class List

32.9.12. Zend_Service_Yahoo_VideoResult

Each Video Search result is returned as a `Zend_Service_Yahoo_VideoResult` object.

32.9.12.1. Properties

Table 143. Zend_Service_Yahoo_VideoResult Properties

Name	Type	Description
Summary	string	Result summary
RefererUrl	string	The URL of the page which contains the video
FileSize	int	The size of the video file in bytes
FileFormat	string	The format of the video (avi, flash, mpeg, msmedia, quicktime, realmedia, etc.)
Height	int	The height of the video in pixels
Width	int	The width of the video in pixels
Duration	int	The length of the video in seconds
Channels	int	Number of audio channels in the video
Streaming	boolean	Whether the video is streaming or not
Thumbnail	Zend_Service_Yahoo_Image	Image thumbnail

Back to Class List

32.9.13. Zend_Service_Yahoo_LocalResult

Each Local Search result is returned as a `Zend_Service_Yahoo_LocalResult` object.

32.9.13.1. Properties

Table 144. Zend_Service_Yahoo_LocalResult Properties

Name	Type	Description
Address	string	Street Address of the result
City	string	City in which the result resides in
State	string	State in which the result resides in
Phone	string	Phone number for the result
Rating	int	User submitted rating for the result
Distance	float	The distance to the result from your specified location
MapUrl	string	A URL of a map for the result
BusinessUrl	string	The URL for the business website, if known
BusinessClickUrl	string	The URL for linking to the business website, if known

Back to Class List

32.9.14. Zend_Service_Yahoo_NewsResult

Each News Search result is returned as a `Zend_Service_Yahoo_NewsResult` object.

32.9.14.1. Properties

Table 145. Zend_Service_Yahoo_NewsResult Properties

Name	Type	Description
Summary	string	Result summary
NewsSource	string	The company who distributed the article
NewsSourceUrl	string	The URL for the company who distributed the article
Language	string	The language the article is in
PublishDate	string	The date the article was published as a UNIX timestamp
ModificationDate	string	The date the article was last modified as a UNIX timestamp

Name	Type	Description
Thumbnail	Zend_Service_Yahoo_Image	Image Thumbnail for the article, if it exists

Back to Class List

32.9.15. Zend_Service_Yahoo_InlinkDataResult

Each Inbound Link Search result is returned as a `Zend_Service_Yahoo_InlinkDatabResult` object.

Back to Class List

32.9.16. Zend_Service_Yahoo_PageDataResult

Each Page Data Search result is returned as a `Zend_Service_Yahoo_PageDatabResult` object.

Back to Class List

32.9.17. Zend_Service_Yahoo_Image

All images returned either by the Yahoo! Image Search or the Yahoo! News Search are represented by `Zend_Service_Yahoo_Image` objects

32.9.17.1. Properties

Table 146. Zend_Service_Yahoo_Image Properties

Name	Type	Description
Url	string	Image URL
Width	int	Image Width
Height	int	Image Height

Back to Class List

Zend_Session

1. Introduction

The Zend Framework Auth team greatly appreciates your feedback and contributions on our email list: fw-auth@lists.zend.com

With web applications written using PHP, a *session* represents a logical, one-to-one connection between server-side, persistent state data and a particular user agent client (e.g., web browser). `Zend_Session` helps manage and preserve session data, a logical complement of cookie data, across multiple page requests by the same client. Unlike cookie data, session data are not stored on the client side and are only shared with the client when server-side source code voluntarily makes the data available in response to a client request. For the purposes of this component and documentation, the term "session data" refers to the server-side data stored in `$_SESSION`, managed by `Zend_Session`, and individually manipulated by `Zend_Session_Namespace` accessor objects. *Session namespaces* provide access to session data using classic namespaces implemented logically as named groups of associative arrays, keyed by strings (similar to normal PHP arrays).

`Zend_Session_Namespace` instances are accessor objects for namespaced slices of `$_SESSION`. The `Zend_Session` component wraps the existing PHP ext/session with an administration and management interface, as well as providing an API for `Zend_Session_Namespace` to persist session namespaces. `Zend_Session_Namespace` provides a standardized, object-oriented interface for working with namespaces persisted inside PHP's standard session mechanism. Support exists for both anonymous and authenticated (e.g., "login") session namespaces. `Zend_Auth`, the authentication component of Zend Framework, uses `Zend_Session_Namespace` to store some information associated with authenticated users. Since `Zend_Session` uses the normal PHP ext/session functions internally, all the familiar configuration options and settings apply (see http://www.php.net/session), with such bonuses as the convenience of an object-oriented interface and default behavior that provides both best practices and smooth integration with Zend Framework. Thus, a standard PHP session identifier, whether conveyed by cookie or within URLs, maintains the association between a client and session state data.

The default ext/session save handler does not maintain this association for server clusters under certain conditions because session data are stored to the filesystem of the server that responded to the request. If a request may be processed by a different server than the one where the session data are located, then the responding server has no access to the session data (if they are not available from a networked filesystem). A list of additional, appropriate save handlers will be provided, when available. Community members are encouraged to suggest and submit save handlers to the fw-auth@lists.zend.com list. A `Zend_Db` compatible save handler has been posted to the list.

2. Basic Usage

`Zend_Session_Namespace` instances provide the primary API for manipulating session data in the Zend Framework. Namespaces are used to segregate all session data, although a default namespace exists for those who only want one namespace for all their session data. `Zend_Session` utilizes ext/session and its special `$_SESSION` superglobal as the storage mechanism for session state data. While `$_SESSION` is still available in PHP's global namespace, developers should refrain from directly accessing it, so that `Zend_Session` and

Zend_Session_Namespace can most effectively and securely provide its suite of session related functionality.

Each instance of Zend_Session_Namespace corresponds to an entry of the $_SESSION superglobal array, where the namespace is used as the key.

```
$myNamespace = new Zend_Session_Namespace('myNamespace');

// $myNamespace corresponds to $_SESSION['myNamespace']
```

It is possible to use Zend_Session in conjunction with other code that uses $_SESSION directly. To avoid problems, however, it is highly recommended that such code only uses parts of $_SESSION that do not correspond to instances of Zend_Session_Namespace.

2.1. Tutorial Examples

If no namespace is specified when instantiating Zend_Session_Namespace, all data will be transparently stored in a namespace called "Default". Zend_Session is not intended to work directly on the contents of session namespace containers. Instead, we use Zend_Session_Namespace. The example below demonstrates use of this default namespace, showing how to count the number of client requests during a session:

Example 852. Counting Page Views

```
$defaultNamespace = new Zend_Session_Namespace('Default');

if (isset($defaultNamespace->numberOfPageRequests)) {
    // this will increment for each page load.
    $defaultNamespace->numberOfPageRequests++;
} else {
    $defaultNamespace->numberOfPageRequests = 1; // first time
}

echo "Page requests this session: ",
    $defaultNamespace->numberOfPageRequests;
```

When multiple modules use instances of Zend_Session_Namespace having different namespaces, each module obtains data encapsulation for its session data. The Zend_Session_Namespace constructor can be passed an optional $namespace argument, which allows developers to partition session data into separate namespaces. Namespacing provides an effective and popular way to secure session state data against changes due to accidental naming collisions.

Namespace names are restricted to character sequences represented as non-empty PHP strings that do not begin with an underscore ("_") character. Only core components included in Zend Framework should use namespace names starting with "Zend".

Example 853. New Way: Namespaces Avoid Collisions

```
// in the Zend_Auth component
$authNamespace = new Zend_Session_Namespace('Zend_Auth');
$authNamespace->user = "myusername";

// in a web services component
$webServiceNamespace = new Zend_Session_Namespace('Some_Web_Service');
$webServiceNamespace->user = "mywebusername";
```

The example above achieves the same effect as the code below, except that the session objects above preserve encapsulation of session data within their respective namespaces.

Example 854. Old Way: PHP Session Access

```
$_SESSION['Zend_Auth']['user'] = "myusername";
$_SESSION['Some_Web_Service']['user'] = "mywebusername";
```

2.2. Iterating Over Session Namespaces

`Zend_Session_Namespace` provides the full IteratorAggregate interface, including support for the `foreach` statement:

Example 855. Session Iteration

```
$aNamespace =
    new Zend_Session_Namespace('some_namespace_with_data_present');

foreach ($aNamespace as $index => $value) {
    echo "aNamespace->$index = '$value';\n";
}
```

2.3. Accessors for Session Namespaces

`Zend_Session_Namespace` implements the `__get()`, `__set()`, `__isset()`, and `__unset()` magic methods, which should not be invoked directly, except from within a subclass. Instead, the normal operators automatically invoke these methods, such as in the following example:

Example 856. Accessing Session Data

```
$namespace = new Zend_Session_Namespace(); // default namespace

$namespace->foo = 100;

echo "\$namespace->foo = $namespace->foo\n";

if (!isset($namespace->bar)) {
    echo "\$namespace->bar not set\n";
}

unset($namespace->foo);
```

3. Advanced Usage

While the basic usage examples are a perfectly acceptable way to utilize Zend Framework sessions, there are some best practices to consider. This section discusses the finer details of session handling and illustrates more advanced usage of the `Zend_Session` component.

3.1. Starting a Session

If you want all requests to have a session facilitated by `Zend_Session`, then start the session in the bootstrap file:

Example 857. Starting the Global Session

```
Zend_Session::start();
```

By starting the session in the bootstrap file, you avoid the possibility that your session might be started after headers have been sent to the browser, which results in an exception, and possibly a broken page for website viewers. Various advanced features require `Zend_Session::start()` first. (More on advanced features later.)

There are four ways to start a session, when using `Zend_Session`. Two are wrong.

1. Wrong: Do not enable PHP's `session.auto_start` setting. If you do not have the ability to disable this setting in php.ini, you are using mod_php (or equivalent), and the setting is already enabled in `php.ini`, then add the following to your `.htaccess` file (usually in your HTML document root directory):

   ```
   php_value session.auto_start 0
   ```

2. Wrong: Do not use PHP's `session_start()` function directly. If you use `session_start()` directly, and then start using `Zend_Session_Namespace`, an exception will be thrown by `Zend_Session::start()` ("session has already been started"). If you call `session_start()` after using `Zend_Session_Namespace` or calling `Zend_Session::start()`, an error of level `E_NOTICE` will be generated, and the call will be ignored.

3. Correct: Use `Zend_Session::start()`. If you want all requests to have and use sessions, then place this function call early and unconditionally in your bootstrap code. Sessions have some overhead. If some requests need sessions, but other requests will not need to use sessions, then:

   - Unconditionally set the `strict` option to `TRUE` using `Zend_Session::setOptions()` in your bootstrap.
   - Call `Zend_Session::start()` only for requests that need to use sessions and before any `Zend_Session_Namespace` objects are instantiated.
   - Use "`new Zend_Session_Namespace()`" normally, where needed, but make sure `Zend_Session::start()` has been called previously.

   The `strict` option prevents `new Zend_Session_Namespace()` from automatically starting the session using `Zend_Session::start()`. Thus, this option helps application developers enforce a design decision to avoid using sessions for certain requests, since it causes an exception to be thrown when `Zend_Session_Namespace` is instantiated before `Zend_Session::start()` is called. Developers should carefully consider the impact of using `Zend_Session::setOptions()`, since these options have global effect, owing to their correspondence to the underlying options for ext/session.

4. Correct: Just instantiate `Zend_Session_Namespace` whenever needed, and the underlying PHP session will be automatically started. This offers extremely simple usage that works well in most situations. However, you then become responsible for ensuring that the first `new Zend_Session_Namespace()` happens *before* any output (e.g., HTTP headers) has been sent by PHP to the client, if you are using the default, cookie-based sessions (strongly recommended). See Section 4.2, “Error: Headers Already Sent” for more information.

3.2. Locking Session Namespaces

Session namespaces can be locked, to prevent further alterations to the data in that namespace. Use `lock()` to make a specific namespace read-only, `unLock()` to make a read-only namespace read-write, and `isLocked()` to test if a namespace has been previously locked.

Locks are transient and do not persist from one request to the next. Locking the namespace has no effect on setter methods of objects stored in the namespace, but does prevent the use of the namespace's setter method to remove or replace objects stored directly in the namespace. Similarly, locking `Zend_Session_Namespace` instances does not prevent the use of symbol table aliases to the same data (see PHP references).

Example 858. Locking Session Namespaces

```
$userProfileNamespace = new Zend_Session_Namespace('userProfileNamespace');

// marking session as read only locked
$userProfileNamespace->lock();

// unlocking read-only lock
if ($userProfileNamespace->isLocked()) {
    $userProfileNamespace->unLock();
}
```

3.3. Namespace Expiration

Limits can be placed on the longevity of both namespaces and individual keys in namespaces. Common use cases include passing temporary information between requests, and reducing exposure to certain security risks by removing access to potentially sensitive information some time after authentication occurred. Expiration can be based on either elapsed seconds or the number of "hops", where a hop occurs for each successive request.

Example 859. Expiration Examples

```
$s = new Zend_Session_Namespace('expireAll');
$s->a = 'apple';
$s->p = 'pear';
$s->o = 'orange';

$s->setExpirationSeconds(5, 'a'); // expire only the key "a" in 5 seconds

// expire entire namespace in 5 "hops"
$s->setExpirationHops(5);

$s->setExpirationSeconds(60);
// The "expireAll" namespace will be marked "expired" on
// the first request received after 60 seconds have elapsed,
// or in 5 hops, whichever happens first.
```

When working with data expiring from the session in the current request, care should be used when retrieving them. Although the data are returned by reference, modifying the data will not make expiring data persist past the current request. In order to "reset" the expiration time, fetch the data into temporary variables, use the namespace to unset them, and then set the appropriate keys again.

3.4. Session Encapsulation and Controllers

Namespaces can also be used to separate session access by controllers to protect variables from contamination. For example, an authentication controller might keep its session state data separate from all other controllers for meeting security requirements.

Example 860. Namespaced Sessions for Controllers with Automatic Expiration

The following code, as part of a controller that displays a test question, initiates a boolean variable to represent whether or not a submitted answer to the test question should be accepted. In this case, the application user is given 300 seconds to answer the displayed question.

```
// ...
// in the question view controller
$testSpace = new Zend_Session_Namespace('testSpace');
// expire only this variable
$testSpace->setExpirationSeconds(300, 'accept_answer');
$testSpace->accept_answer = true;
//...
```

Below, the controller that processes the answers to test questions determines whether or not to accept an answer based on whether the user submitted the answer within the allotted time:

```
// ...
// in the answer processing controller
$testSpace = new Zend_Session_Namespace('testSpace');
if ($testSpace->accept_answer === true) {
    // within time
}
else {
    // not within time
}
// ...
```

3.5. Preventing Multiple Instances per Namespace

Although session locking provides a good degree of protection against unintended use of namespaced session data, `Zend_Session_Namespace` also features the ability to prevent the creation of multiple instances corresponding to a single namespace.

To enable this behavior, pass `TRUE` to the second constructor argument when creating the last allowed instance of `Zend_Session_Namespace`. Any subsequent attempt to instantiate the same namespace would result in a thrown exception.

Example 861. Limiting Session Namespace Access to a Single Instance

```
// create an instance of a namespace
$authSpaceAccessor1 = new Zend_Session_Namespace('Zend_Auth');

// create another instance of the same namespace, but disallow any
// new instances
$authSpaceAccessor2 = new Zend_Session_Namespace('Zend_Auth', true);

// making a reference is still possible
$authSpaceAccessor3 = $authSpaceAccessor2;

$authSpaceAccessor1->foo = 'bar';

assert($authSpaceAccessor2->foo, 'bar');

try {
    $aNamespaceObject = new Zend_Session_Namespace('Zend_Auth');
} catch (Zend_Session_Exception $e) {
    echo 'Cannot instantiate this namespace since ' .
         '$authSpaceAccessor2 was created\n';
}
```

The second parameter in the constructor above tells `Zend_Session_Namespace` that any future instances with the "`Zend_Auth`" namespace are not allowed. Attempting to create such an instance causes an exception to be thrown by the constructor. The developer therefore becomes responsible for storing a reference to an instance object (`$authSpaceAccessor1`, `$authSpaceAccessor2`, or `$authSpaceAccessor3` in the example above) somewhere, if access to the session namespace is needed at a later time during the same request. For example, a developer may store the reference in a static variable, add the reference to a registry (see *Zend_Registry*), or otherwise make it available to other methods that may need access to the session namespace.

3.6. Working with Arrays

Due to the implementation history of PHP magic methods, modifying an array inside a namespace may not work under PHP versions before 5.2.1. If you will only be working with PHP 5.2.1 or later, then you may skip to the next section.

Example 862. Modifying Array Data with a Session Namespace

The following illustrates how the problem may be reproduced:

```
$sessionNamespace = new Zend_Session_Namespace();
$sessionNamespace->array = array();

// may not work as expected before PHP 5.2.1
$sessionNamespace->array['testKey'] = 1;
echo $sessionNamespace->array['testKey'];
```

Example 863. Building Arrays Prior to Session Storage

If possible, avoid the problem altogether by storing arrays into a session namespace only after all desired array values have been set.

```
$sessionNamespace = new Zend_Session_Namespace('Foo');
$sessionNamespace->array = array('a', 'b', 'c');
```

If you are using an affected version of PHP and need to modify the array after assigning it to a session namespace key, you may use either or both of the following workarounds.

Example 864. Workaround: Reassign a Modified Array

In the code that follows, a copy of the stored array is created, modified, and reassigned to the location from which the copy was created, overwriting the original array.

```
$sessionNamespace = new Zend_Session_Namespace();

// assign the initial array
$sessionNamespace->array = array('tree' => 'apple');

// make a copy of the array
$tmp = $sessionNamespace->array;

// modfiy the array copy
$tmp['fruit'] = 'peach';

// assign a copy of the array back to the session namespace
$sessionNamespace->array = $tmp;

echo $sessionNamespace->array['fruit']; // prints "peach"
```

Example 865. Workaround: store array containing reference

Alternatively, store an array containing a reference to the desired array, and then access it indirectly.

```
$myNamespace = new Zend_Session_Namespace('myNamespace');
$a = array(1, 2, 3);
$myNamespace->someArray = array( &$a );
$a['foo'] = 'bar';
echo $myNamespace->someArray['foo']; // prints "bar"
```

3.7. Using Sessions with Objects

If you plan to persist objects in the PHP session, know that they will be serialized for storage. Thus, any object persisted with the PHP session must be unserialized upon retrieval from storage. The implication is that the developer must ensure that the classes for the persisted objects must have been defined before the object is unserialized from session storage. If an unserialized object's class is not defined, then it becomes an instance of `stdClass`.

3.8. Using Sessions with Unit Tests

Zend Framework relies on PHPUnit to facilitate testing of itself. Many developers extend the existing suite of unit tests to cover the code in their applications. The exception "*Zend_Session is currently marked as read-only*" is thrown while performing unit tests, if any write-related methods are used after ending the session. However, unit tests using `Zend_Session` require extra attention, because closing (`Zend_Session::writeClose()`), or destroying a session (`Zend_Session::destroy()`) prevents any further setting or unsetting of keys in any instance of `Zend_Session_Namespace`. This behavior is a direct result of the underlying ext/session mechanism and PHP's `session_destroy()` and `session_write_close()`, which have no "undo" mechanism to facilitate setup/teardown with unit tests.

To work around this, see the unit test `testSetExpirationSeconds()` in `SessionTest.php` and `SessionTestHelper.php`, both located in `tests/Zend/Session`, which make use of

PHP's `exec()` to launch a separate process. The new process more accurately simulates a second, successive request from a browser. The separate process begins with a "clean" session, just like any PHP script execution for a web request. Also, any changes to `$_SESSION` made in the calling process become available to the child process, provided the parent closed the session before using `exec()`.

Example 866. PHPUnit Testing Code Dependent on Zend_Session

```
// testing setExpirationSeconds()
$script = 'SessionTestHelper.php';
$s = new Zend_Session_Namespace('space');
$s->a = 'apple';
$s->o = 'orange';
$s->setExpirationSeconds(5);

Zend_Session::regenerateId();
$id = Zend_Session::getId();
session_write_close(); // release session so process below can use it
sleep(4); // not long enough for things to expire
exec($script . "expireAll $id expireAll", $result);
$result = $this->sortResult($result);
$expect = ';a === apple;o === orange;p === pear';
$this->assertTrue($result === $expect,
    "iteration over default Zend_Session namespace failed; " .
    "expecting result === '$expect', but got '$result'");

sleep(2); // long enough for things to expire (total of 6 seconds
          // waiting, but expires in 5)
exec($script . "expireAll $id expireAll", $result);
$result = array_pop($result);
$this->assertTrue($result === '',
    "iteration over default Zend_Session namespace failed; " .
    "expecting result === '', but got '$result')");
session_start(); // resume artificially suspended session

// We could split this into a separate test, but actually, if anything
// leftover from above contaminates the tests below, that is also a
// bug that we want to know about.
$s = new Zend_Session_Namespace('expireGuava');
$s->setExpirationSeconds(5, 'g'); // now try to expire only 1 of the
                                  // keys in the namespace
$s->g = 'guava';
$s->p = 'peach';
$s->p = 'plum';

session_write_close(); // release session so process below can use it
sleep(6); // not long enough for things to expire
exec($script . "expireAll $id expireGuava", $result);
$result = $this->sortResult($result);
session_start(); // resume artificially suspended session
$this->assertTrue($result === ';p === plum',
    "iteration over named Zend_Session namespace failed (result=$result)");
```

4. Global Session Management

The default behavior of sessions can be modified using the static methods of `Zend_Session`. All management and manipulation of global session management occurs using `Zend_Session`, including configuration of the usual options provided by ext/session, using

`Zend_Session::setOptions()`. For example, failure to insure the use of a safe `save_path` or a unique cookie name by ext/session using `Zend_Session::setOptions()` may result in security issues.

4.1. Configuration Options

When the first session namespace is requested, `Zend_Session` will automatically start the PHP session, unless already started with `Zend_Session::start()`. The underlying PHP session will use defaults from `Zend_Session`, unless modified first by `Zend_Session::setOptions()`.

To set a session configuration option, include the basename (the part of the name after "`session.`") as a key of an array passed to `Zend_Session::setOptions()`. The corresponding value in the array is used to set the session option value. If no options are set by the developer, `Zend_Session` will utilize recommended default options first, then the default php.ini settings. Community feedback about best practices for these options should be sent to fw-auth@lists.zend.com.

Example 867. Using Zend_Config to Configure Zend_Session

To configure this component using `Zend_Config_Ini`, first add the configuration options to the INI file:

```
; Accept defaults for production
[production]
; bug_compat_42
; bug_compat_warn
; cache_expire
; cache_limiter
; cookie_domain
; cookie_lifetime
; cookie_path
; cookie_secure
; entropy_file
; entropy_length
; gc_divisor
; gc_maxlifetime
; gc_probability
; hash_bits_per_character
; hash_function
; name should be unique for each PHP application sharing the same
; domain name
name = UNIQUE_NAME
; referer_check
; save_handler
; save_path
; serialize_handler
; use_cookies
; use_only_cookies
; use_trans_sid

; remember_me_seconds = <integer seconds>
; strict = on|off

; Development inherits configuration from production, but overrides
; several values
[development : production]
; Don't forget to create this directory and make it rwx (readable and
; modifiable) by PHP.
save_path = /home/myaccount/zend_sessions/myapp
use_only_cookies = on
; When persisting session id cookies, request a TTL of 10 days
remember_me_seconds = 864000
```

Next, load the configuration file and pass its array representation to `Zend_Session::setOptions()`:

```
$config = new Zend_Config_Ini('myapp.ini', 'development');

Zend_Session::setOptions($config->toArray());
```

Most options shown above need no explanation beyond that found in the standard PHP documentation, but those of particular interest are noted below.

- boolean `strict` - disables automatic starting of `Zend_Session` when using `new Zend_Session_Namespace()`.

- integer `remember_me_seconds` - how long should session id cookie persist, after user agent has ended (e.g., browser application terminated).

- string `save_path` - The correct value is system dependent, and should be provided by the developer using an *absolute path* to a directory readable and writable by the PHP process. If a writable path is not supplied, then `Zend_Session` will throw an exception when started (i.e., when `start()` is called).

Security Risk

If the path is readable by other applications, then session hijacking might be possible. If the path is writable by other applications, then session poisoning might be possible. If this path is shared with other users or other PHP applications, various security issues might occur, including theft of session content, hijacking of sessions, and collision of garbage collection (e.g., another user's application might cause PHP to delete your application's session files).

For example, an attacker can visit the victim's website to obtain a session cookie. Then, he edits the cookie path to his own domain on the same server, before visiting his own website to execute `var_dump($_SESSION)`. Armed with detailed knowledge of the victim's use of data in their sessions, the attacker can then modify the session state (poisoning the session), alter the cookie path back to the victim's website, and then make requests from the victim's website using the poisoned session. Even if two applications on the same server do not have read/write access to the other application's `save_path`, if the `save_path` is guessable, and the attacker has control over one of these two websites, the attacker could alter their website's `save_path` to use the other's save_path, and thus accomplish session poisoning, under some common configurations of PHP. Thus, the value for `save_path` should not be made public knowledge and should be altered to a secure location unique to each application.

- string `name` - The correct value is system dependent and should be provided by the developer using a value *unique* to the application.

Security Risk

If the `php.ini` setting for `session.name` is the same (e.g., the default "PHPSESSID"), and there are two or more PHP applications accessible through the same domain name then they will share the same session data for visitors to both websites. Additionally, possible corruption of session data may result.

- boolean `use_only_cookies` - In order to avoid introducing additional security risks, do not alter the default value of this option.

Security Risk

If this setting is not enabled, an attacker can easily fix victim's session ids, using links on the attacker's website, such as `http://www.example.com/index.php?PHPSESSID=fixed_session_id`. The fixation works, if the victim does not already have a session id cookie for example.com. Once a victim is using a known session id, the attacker can then attempt to hijack the session by pretending to be the victim, and emulating the victim's user agent.

4.2. Error: Headers Already Sent

If you see the error message, "Cannot modify header information - headers already sent", or, "You must call ... before any output has been sent to the browser; output started in ...", then carefully examine the immediate cause (function or method) associated with the message. Any actions that require sending HTTP headers, such as sending a cookie, must be done before sending normal output (unbuffered output), except when using PHP's output buffering.

- Using output buffering often is sufficient to prevent this issue, and may help improve performance. For example, in `php.ini`, "`output_buffering = 65535`" enables output buffering with a 64K buffer. Even though output buffering might be a good tactic on production servers to increase performance, relying only on buffering to resolve the "headers already sent" problem is not sufficient. The application must not exceed the buffer size, or the problem will occur whenever the output sent (prior to the HTTP headers) exceeds the buffer size.
- Alternatively, try rearranging the application logic so that actions manipulating headers are performed prior to sending any output whatsoever.
- If a `Zend_Session` method is involved in causing the error message, examine the method carefully, and make sure its use really is needed in the application. For example, the default usage of `destroy()` also sends an HTTP header to expire the client-side session cookie. If this is not needed, then use `destroy(false)`, since the instructions to set cookies are sent with HTTP headers.
- Alternatively, try rearranging the application logic so that all actions manipulating headers are performed prior to sending any output whatsoever.
- Remove any closing "?>" tags, if they occur at the end of a PHP source file. They are not needed, and newlines and other nearly invisible whitespace following the closing tag can trigger output to the client.

4.3. Session Identifiers

Introduction: Best practice in relation to using sessions with Zend Framework calls for using a browser cookie (i.e. a normal cookie stored in your web browser), instead of embedding a unique session identifier in URLs as a means to track individual users. By default this component uses only cookies to maintain session identifiers. The cookie's value is the unique identifier of your browser's session. PHP's ext/session uses this identifier to maintain a unique one-to-one relationship between website visitors, and persistent session data storage unique to each visitor. `Zend_Session*` wraps this storage mechanism (`$_SESSION`) with an object-oriented interface. Unfortunately, if an attacker gains access to the value of the cookie (the session id), an attacker might be able to hijack a visitor's session. This problem is not unique to PHP, or Zend Framework. The `regenerateId()` method allows an application to change the session id (stored in the visitor's cookie) to a new, random, unpredictable value. Note: Although not the same, to make this section easier to read, we use the terms "user agent" and "web browser" interchangeably.

Why?: If an attacker obtains a valid session identifier, an attacker might be able to impersonate a valid user (the victim), and then obtain access to confidential information or otherwise manipulate the victim's data managed by your application. Changing session ids helps protect against session hijacking. If the session id is changed, and an attacker does not know the new value, the attacker can not use the new session id in their attempts to hijack the visitor's session. Even if an attacker gains access to an old session id, `regenerateId()` also moves the session data from the old session id "handle" to the new one, so no data remains accessible via the old session id.

When to use regenerateId(): Adding `Zend_Session::regenerateId()` to your Zend Framework bootstrap yields one of the safest and most secure ways to regenerate session id's in

user agent cookies. If there is no conditional logic to determine when to regenerate the session id, then there are no flaws in that logic. Although regenerating on every request prevents several possible avenues of attack, not everyone wants the associated small performance and bandwidth cost. Thus, applications commonly try to dynamically determine situations of greater risk, and only regenerate the session ids in those situations. Whenever a website visitor's session's privileges are "escalated" (e.g. a visitor re-authenticates their identity before editing their personal "profile"), or whenever a security "sensitive" session parameter change occurs, consider using `regenerateId()` to create a new session id. If you call the `rememberMe()` function, then don't use `regenerateId()`, since the former calls the latter. If a user has successfully logged into your website, use `rememberMe()` instead of `regenerateId()`.

4.3.1. Session Hijacking and Fixation

Avoiding cross-site script (XSS) vulnerabilities helps preventing session hijacking. According to Secunia's statistics XSS problems occur frequently, regardless of the languages used to create web applications. Rather than expecting to never have a XSS problem with an application, plan for it by following best practices to help minimize damage, if it occurs. With XSS, an attacker does not need direct access to a victim's network traffic. If the victim already has a session cookie, Javascript XSS might allow an attacker to read the cookie and steal the session. For victims with no session cookies, using XSS to inject Javascript, an attacker could create a session id cookie on the victim's browser with a known value, then set an identical cookie on the attacker's system, in order to hijack the victim's session. If the victim visited an attacker's website, then the attacker can also emulate most other identifiable characteristics of the victim's user agent. If your website has an XSS vulnerability, the attacker might be able to insert an AJAX Javascript that secretly "visits" the attacker's website, so that the attacker knows the victim's browser characteristics and becomes aware of a compromised session at the victim website. However, the attacker can not arbitrarily alter the server-side state of PHP sessions, provided the developer has correctly set the value for the `save_path` option.

By itself, calling `Zend_Session::regenerateId()` when the user's session is first used, does not prevent session fixation attacks, unless you can distinguish between a session originated by an attacker emulating the victim. At first, this might sound contradictory to the previous statement above, until we consider an attacker who first initiates a real session on your website. The session is "first used" by the attacker, who then knows the result of the initialization (`regenerateId()`). The attacker then uses the new session id in combination with an XSS vulnerability, or injects the session id via a link on the attacker's website (works if `use_only_cookies = off`).

If you can distinguish between an attacker and victim using the same session id, then session hijacking can be dealt with directly. However, such distinctions usually involve some form of usability tradeoffs, because the methods of distinction are often imprecise. For example, if a request is received from an IP in a different country than the IP of the request when the session was created, then the new request probably belongs to an attacker. Under the following conditions, there might not be any way for a website application to distinguish between a victim and an attacker:

- attacker first initiates a session on your website to obtain a valid session id
- attacker uses XSS vulnerability on your website to create a cookie on the victim's browser with the same, valid session id (i.e. session fixation)
- both the victim and attacker originate from the same proxy farm (e.g. both are behind the same firewall at a large company, like AOL)

The sample code below makes it much harder for an attacker to know the current victim's session id, unless the attacker has already performed the first two steps above.

Example 868. Session Fixation

```
$defaultNamespace = new Zend_Session_Namespace();

if (!isset($defaultNamespace->initialized)) {
    Zend_Session::regenerateId();
    $defaultNamespace->initialized = true;
}
```

4.4. rememberMe(integer $seconds)

Ordinarily, sessions end when the user agent terminates, such as when an end user exits a web browser program. However, your application may provide the ability to extend user sessions beyond the lifetime of the client program through the use of persistent cookies. Use `Zend_Session::rememberMe()` before a session is started to control the length of time before a persisted session cookie expires. If you do not specify a number of seconds, then the session cookie lifetime defaults to `remember_me_seconds`, which may be set using `Zend_Session::setOptions()`. To help thwart session fixation/hijacking, use this function when a user successfully authenticates with your application (e.g., from a "login" form).

4.5. forgetMe()

This function complements `rememberMe()` by writing a session cookie that has a lifetime ending when the user agent terminates.

4.6. sessionExists()

Use this method to determine if a session already exists for the current user agent/request. It may be used before starting a session, and independently of all other `Zend_Session` and `Zend_Session_Namespace` methods.

4.7. destroy(bool $remove_cookie = true, bool $readonly = true)

`Zend_Session::destroy()` destroys all of the persistent data associated with the current session. However, no variables in PHP are affected, so your namespaced sessions (instances of `Zend_Session_Namespace`) remain readable. To complete a "logout", set the optional parameter to `TRUE` (the default) to also delete the user agent's session id cookie. The optional `$readonly` parameter removes the ability to create new `Zend_Session_Namespace` instances and for `Zend_Session` methods to write to the session data store.

If you see the error message, "Cannot modify header information - headers already sent", then either avoid using `TRUE` as the value for the first argument (requesting removal of the session cookie), or see Section 4.2, “Error: Headers Already Sent”. Thus, `Zend_Session::destroy(true)` must either be called before PHP has sent HTTP headers, or output buffering must be enabled. Also, the total output sent must not exceed the set buffer size, in order to prevent triggering sending the output before the call to `destroy()`.

Throws

By default, `$readonly` is enabled and further actions involving writing to the session data store will throw an exception.

4.8. stop()

This method does absolutely nothing more than toggle a flag in `Zend_Session` to prevent further writing to the session data store. We are specifically requesting feedback

on this feature. Potential uses/abuses might include temporarily disabling the use of `Zend_Session_Namespace` instances or `Zend_Session` methods to write to the session data store, while execution is transferred to view- related code. Attempts to perform actions involving writes via these instances or methods will throw an exception.

4.9. writeClose($readonly = true)

Shutdown the session, close writing and detach `$_SESSION` from the back-end storage mechanism. This will complete the internal data transformation on this request. The optional `$readonly` boolean parameter can remove write access by throwing an exception upon any attempt to write to the session via `Zend_Session` or `Zend_Session_Namespace`.

Throws

By default, `$readonly` is enabled and further actions involving writing to the session data store will throw an exception. However, some legacy application might expect `$_SESSION` to remain writable after ending the session via `session_write_close()`. Although not considered "best practice", the `$readonly` option is available for those who need it.

4.10. expireSessionCookie()

This method sends an expired session id cookie, causing the client to delete the session cookie. Sometimes this technique is used to perform a client-side logout.

4.11. setSaveHandler(Zend_Session_SaveHandler_Interface $interface)

Most developers will find the default save handler sufficient. This method provides an object-oriented wrapper for `session_set_save_handler()`.

4.12. namespaceIsset($namespace)

Use this method to determine if a session namespace exists, or if a particular index exists in a particular namespace.

Throws

An exception will be thrown if `Zend_Session` is not marked as readable (e.g., before `Zend_Session` has been started).

4.13. namespaceUnset($namespace)

Use `Zend_Session::namespaceUnset($namespace)` to efficiently remove an entire namespace and its contents. As with all arrays in PHP, if a variable containing an array is unset, and the array contains other objects, those objects will remain available, if they were also stored by reference in other array/objects that remain accessible via other variables. So `namespaceUnset()` does not perform a "deep" unsetting/deleting of the contents of the entries in the namespace. For a more detailed explanation, please see References Explained in the PHP manual.

Throws

An exception will be thrown if the namespace is not writable (e.g., after `destroy()`).

4.14. namespaceGet($namespace)

DEPRECATED: Use `getIterator()` in `Zend_Session_Namespace`. This method returns an array of the contents of `$namespace`. If you have logical reasons to keep this method publicly accessible, please provide feedback to the fw-auth@lists.zend.com mail list. Actually, all participation on any relevant topic is welcome :)

Throws

An exception will be thrown if `Zend_Session` is not marked as readable (e.g., before `Zend_Session` has been started).

4.15. getIterator()

Use `getIterator()` to obtain an array containing the names of all namespaces.

Throws

An exception will be thrown if `Zend_Session` is not marked as readable (e.g., before `Zend_Session` has been started).

5. Zend_Session_SaveHandler_DbTable

The basic setup for `Zend_Session_SaveHandler_DbTable` must at least have four columns, denoted in the config array or `Zend_Config` object: primary, which is the primary key and defaults to just the session id which by default is a string of length 32; modified, which is the unix timestamp of the last modified date; lifetime, which is the lifetime of the session (**modified + lifetime > time();**); and data, which is the serialized data stored in the session

Example 869. Basic Setup

```
CREATE TABLE `session` (
  `id` char(32),
  `modified` int,
  `lifetime` int,
  `data` text,
  PRIMARY KEY (`id`)
);
```

```
//get your database connection ready
$db = Zend_Db::factory('Pdo_Mysql', array(
    'host'        =>'example.com',
    'username'    => 'dbuser',
    'password'    => '******',
    'dbname'    => 'dbname'
));

//you can either set the Zend_Db_Table default adapter
//or you can pass the db connection straight to the save handler $config
Zend_Db_Table_Abstract::setDefaultAdapter($db);
$config = array(
    'name'           => 'session',
    'primary'        => 'id',
    'modifiedColumn' => 'modified',
    'dataColumn'     => 'data',
    'lifetimeColumn' => 'lifetime'
);

//create your Zend_Session_SaveHandler_DbTable and
//set the save handler for Zend_Session
Zend_Session::setSaveHandler(new Zend_Session_SaveHandler_DbTable($config));

//start your session!
Zend_Session::start();

//now you can use Zend_Session like any other time
```

You can also use Multiple Columns in your primary key for `Zend_Session_SaveHandler_DbTable`.

Example 870. Using a Multi-Column Primary Key

```
CREATE TABLE `session` (
    `session_id` char(32) NOT NULL,
    `save_path` varchar(32) NOT NULL,
    `name` varchar(32) NOT NULL DEFAULT '',
    `modified` int,
    `lifetime` int,
    `session_data` text,
    PRIMARY KEY (`Session_ID`, `save_path`, `name`)
);
```

```
//setup your DB connection like before
//NOTE: this config is also passed to Zend_Db_Table so anything specific
//to the table can be put in the config as well
$config = array(
    'name'              => 'session', //table name as per Zend_Db_Table
    'primary'           => array(
        'session_id',   //the sessionID given by PHP
        'save_path',    //session.save_path
        'name',         //session name
    ),
    'primaryAssignment' => array(
        //you must tell the save handler which columns you
        //are using as the primary key. ORDER IS IMPORTANT
        'sessionId', //first column of the primary key is of the sessionID
        'sessionSavePath', //second column of the primary key is the save path
        'sessionName', //third column of the primary key is the session name
    ),
    'modifiedColumn'    => 'modified',     //time the session should expire
    'dataColumn'        => 'session_data', //serialized data
    'lifetimeColumn'    => 'lifetime',     //end of life for a specific record
);

//Tell Zend_Session to use your Save Handler
Zend_Session::setSaveHandler(new Zend_Session_SaveHandler_DbTable($config));

//start your session
Zend_Session::start();

//use Zend_Session as normal
```

Zend_Soap

1. Zend_Soap_Server

`Zend_Soap_Server` class is intended to simplify Web Services server part development for PHP programmers.

It may be used in WSDL or non-WSDL mode, and using classes or functions to define Web Service API.

When `Zend_Soap_Server` component works in the WSDL mode, it uses already prepared WSDL document to define server object behavior and transport layer options.

WSDL document may be auto-generated with functionality provided by Zend_Soap_AutoDiscovery component or should be constructed manually using `Zend_Soap_Wsdl` class or any other XML generating tool.

If the non-WSDL mode is used, then all protocol options have to be set using options mechanism.

1.1. Zend_Soap_Server constructor

`Zend_Soap_Server` constructor should be used a bit differently for WSDL and non-WSDL modes.

1.1.1. Zend_Soap_Server constructor for the WSDL mode

`Zend_Soap_Server` constructor takes two optional parameters when it works in WSDL mode:

1. `$wsdl`, which is an URI of a WSDL file[1].
2. `$options` - options to create SOAP server object[2].

   The following options are recognized in the WSDL mode:

   - 'soap_version' ('soapVersion') - soap version to use (SOAP_1_1 or SOAP_1_2).
   - 'actor' - the actor URI for the server.
   - 'classmap' ('classMap') which can be used to map some WSDL types to PHP classes.

     The option must be an array with WSDL types as keys and names of PHP classes as values.
   - 'encoding' - internal character encoding (UTF-8 is always used as an external encoding).
   - 'wsdl' which is equivalent to `setWsdl($wsdlValue)` call.

1.1.2. Zend_Soap_Server constructor for the non-WSDL mode

The first constructor parameter *must* be set to `NULL` if you plan to use `Zend_Soap_Server` functionality in non-WSDL mode.

You also have to set 'uri' option in this case (see below).

The second constructor parameter (`$options`) is an array with options to create SOAP server object[3].

The following options are recognized in the non-WSDL mode:

- 'soap_version' ('soapVersion') - soap version to use (SOAP_1_1 or SOAP_1_2).
- 'actor' - the actor URI for the server.
- 'classmap' ('classMap') which can be used to map some WSDL types to PHP classes.

  The option must be an array with WSDL types as keys and names of PHP classes as values.
- 'encoding' - internal character encoding (UTF-8 is always used as an external encoding).
- 'uri' (required) - URI namespace for SOAP server.

1.2. Methods to define Web Service API

There are two ways to define Web Service API when your want to give access to your PHP code through SOAP.

The first one is to attach some class to the `Zend_Soap_Server` object which has to completely describe Web Service API:

```
...
class MyClass {
    /**
     * This method takes ...
     *
     * @param integer $inputParam
     * @return string
     */
    public function method1($inputParam) {
        ...
    }

    /**
     * This method takes ...
     *
     * @param integer $inputParam1
     * @param string  $inputParam2
     * @return float
     */
    public function method2($inputParam1, $inputParam2) {
        ...
    }

    ...
}
...
$server = new Zend_Soap_Server(null, $options);
// Bind Class to Soap Server
$server->setClass('MyClass');
// Bind already initialized object to Soap Server
$server->setObject(new MyClass());
...
```

[3] Options may be set later using `setOptions($options)` method.

```
$server->handle();
```

Important!

You should completely describe each method using method docblock if you plan to use autodiscover functionality to prepare corresponding Web Service WSDL.

The second method of defining Web Service API is using set of functions and `addFunction()` or `loadFunctions()` methods:

```
...
/**
 * This function ...
 *
 * @param integer $inputParam
 * @return string
 */
function function1($inputParam) {
    ...
}

/**
 * This function ...
 *
 * @param integer $inputParam1
 * @param string  $inputParam2
 * @return float
 */
function function2($inputParam1, $inputParam2) {
    ...
}
...
$server = new Zend_Soap_Server(null, $options);
$server->addFunction('function1');
$server->addFunction('function2');
...
$server->handle();
```

1.3. Request and response objects handling

Advanced

This section describes advanced request/response processing options and may be skipped.

`Zend_Soap_Server` component performs request/response processing automatically, but allows to catch it and do some pre- and post-processing.

1.3.1. Request processing

`Zend_Soap_Server::handle()` method takes request from the standard input stream ('php://input'). It may be overridden either by supplying optional parameter to the `handle()` method or by setting request using `setRequest()` method:

```
...
$server = new Zend_Soap_Server(...);
...
```

```
// Set request using optional $request parameter
$server->handle($request);
...
// Set request using setRequest() method
$server->setRequest();
$server->handle();
```

Request object may be represented using any of the following:

- DOMDocument (casted to XML)
- DOMNode (owner document is grabbed and casted to XML)
- SimpleXMLElement (casted to XML)
- stdClass (__toString() is called and verified to be valid XML)
- string (verified to be valid XML)

Last processed request may be retrieved using `getLastRequest()` method as an XML string:

```
...
$server = new Zend_Soap_Server(...);
...
$server->handle();
$request = $server->getLastRequest();
```

1.3.2. Response pre-processing

`Zend_Soap_Server::handle()` method automatically emits generated response to the output stream. It may be blocked using `setReturnResponse()` with TRUE or FALSE as a parameter[4]. Generated response is returned by `handle()` method in this case.

```
...
$server = new Zend_Soap_Server(...);
...
// Get a response as a return value of handle() method
// instead of emitting it to the standard output
$server->setReturnResponse(true);
...
$response = $server->handle();
...
```

Last response may be also retrieved by `getLastResponse()` method for some post-processing:

```
...
$server = new Zend_Soap_Server(...);
...
$server->handle();
$response = $server->getLastResponse();
...
```

2. Zend_Soap_Client

The `Zend_Soap_Client` class simplifies SOAP client development for PHP programmers.

[4] Current state of the Return Response flag may be requested with `setReturnResponse()` method.

It may be used in WSDL or non-WSDL mode.

Under the WSDL mode, the `Zend_Soap_Client` component uses a WSDL document to define transport layer options.

The WSDL description is usually provided by the web service the client will access. If the WSDL description is not made available, you may want to use `Zend_Soap_Client` in non-WSDL mode. Under this mode, all SOAP protocol options have to be set explicitly on the `Zend_Soap_Client` class.

2.1. Zend_Soap_Client Constructor

The `Zend_Soap_Client` constructor takes two parameters:

- `$wsdl` - the URI of a WSDL file.
- `$options` - options to create SOAP client object.

Both of these parameters may be set later using `setWsdl($wsdl)` and `setOptions($options)` methods respectively.

Important!

If you use `Zend_Soap_Client` component in non-WSDL mode, you *must* set the 'location' and 'uri' options.

The following options are recognized:

- 'soap_version' ('soapVersion') - soap version to use (SOAP_1_1 or SOAP_1_2).
- 'classmap' ('classMap') - can be used to map some WSDL types to PHP classes.

  The option must be an array with WSDL types as keys and names of PHP classes as values.
- 'encoding' - internal character encoding (UTF-8 is always used as an external encoding).
- 'wsdl' which is equivalent to `setWsdl($wsdlValue)` call.

  Changing this option may switch `Zend_Soap_Client` object to or from WSDL mode.
- 'uri' - target namespace for the SOAP service (required for non-WSDL-mode, doesn't work for WSDL mode).
- 'location' - the URL to request (required for non-WSDL-mode, doesn't work for WSDL mode).
- 'style' - request style (doesn't work for WSDL mode): `SOAP_RPC` or `SOAP_DOCUMENT`.
- 'use' - method to encode messages (doesn't work for WSDL mode): `SOAP_ENCODED` or `SOAP_LITERAL`.
- 'login' and 'password' - login and password for an HTTP authentication.
- 'proxy_host', 'proxy_port', 'proxy_login', and 'proxy_password' - an HTTP connection through a proxy server.
- 'local_cert' and 'passphrase' - HTTPS client certificate authentication options.

- 'compression' - compression options; it's a combination of SOAP_COMPRESSION_ACCEPT, SOAP_COMPRESSION_GZIP and SOAP_COMPRESSION_DEFLATE options which may be used like this:

```
// Accept response compression
$client = new Zend_Soap_Client("some.wsdl",
  array('compression' => SOAP_COMPRESSION_ACCEPT));
...

// Compress requests using gzip with compression level 5
$client = new Zend_Soap_Client("some.wsdl",
  array('compression' => SOAP_COMPRESSION_ACCEPT | SOAP_COMPRESSION_GZIP | 5));
...

// Compress requests using deflate compression
$client = new Zend_Soap_Client("some.wsdl",
  array('compression' => SOAP_COMPRESSION_ACCEPT | SOAP_COMPRESSION_DEFLATE));
```

2.2. Performing SOAP Requests

After we've created a Zend_Soap_Client object we are ready to perform SOAP requests.

Each web service method is mapped to the virtual Zend_Soap_Client object method which takes parameters with common PHP types.

Use it like in the following example:

```
//************************************************************
//                Server code
//************************************************************
// class MyClass {
//     /**
//      * This method takes ...
//      *
//      * @param integer $inputParam
//      * @return string
//      */
//     public function method1($inputParam) {
//         ...
//     }
//
//     /**
//      * This method takes ...
//      *
//      * @param integer $inputParam1
//      * @param string  $inputParam2
//      * @return float
//      */
//     public function method2($inputParam1, $inputParam2) {
//         ...
//     }
//
//     ...
// }
// ...
// $server = new Zend_Soap_Server(null, $options);
// $server->setClass('MyClass');
// ...
```

```
// $server->handle();
//
//************************************************************
//                End of server code
//************************************************************

$client = new Zend_Soap_Client("MyService.wsdl");
...

// $result1 is a string
$result1 = $client->method1(10);
...

// $result2 is a float
$result2 = $client->method2(22, 'some string');
```

3. WSDL Accessor

`Zend_Soap_Wsdl` class is used by `Zend_Soap_Server` component internally to operate with WSDL documents. Nevertheless, you could also use functionality provided by this class for your own needs. The `Zend_Soap_Wsdl` package contains both a parser and a builder of WSDL documents.

If you don't plan to do this, you can skip this documentation section.

3.1. Zend_Soap_Wsdl constructor

`Zend_Soap_Wsdl` constructor takes three parameters:

1. `$name` - name of the Web Service being described.
2. `$uri` - URI where the WSDL will be available (could also be a reference to the file in the filesystem.)
3. `$strategy` - optional flag used to identify the strategy for complex types (objects) detection. This was a boolean `$extractComplexTypes` before version 1.7 and can still be set as a boolean for backwards compatibility. By default the 1.6 detection behaviour is set. To read more on complex type detection strategies go to the section: Section 3.10.2, “Adding complex type information”.

3.2. addMessage() method

`addMessage($name, $parts)` method adds new message description to the WSDL document (/definitions/message element).

Each message correspond to methods in terms of `Zend_Soap_Server` and `Zend_Soap_Client` functionality.

`$name` parameter represents message name.

`$parts` parameter is an array of message parts which describe SOAP call parameters. It's an associative array: 'part name' (SOAP call parameter name) => 'part type'.

Type mapping management is performed using `addTypes()`, `addTypes()` and `addComplexType()` methods (see below).

Messages parts can use either 'element' or 'type' attribute for typing (see http://www.w3.org/TR/wsdl#_messages).

'element' attribute must refer to a corresponding element of data type definition. 'type' attribute refers to a corresponding complexType entry.

All standard XSD types have both 'element' and 'complexType' definitions (see http://schemas.xmlsoap.org/soap/encoding/).

All non-standard types, which may be added using `Zend_Soap_Wsdl::addComplexType()` method, are described using 'complexType' node of '/definitions/types/schema/' section of WSDL document.

So `addMessage()` method always uses 'type' attribute to describe types.

3.3. addPortType() method

`addPortType($name)` method adds new port type to the WSDL document (/definitions/portType) with the specified port type name.

It joins a set of Web Service methods defined in terms of `Zend_Soap_Server` implementation.

See http://www.w3.org/TR/wsdl#_porttypes for the details.

3.4. addPortOperation() method

`addPortOperation($portType, $name, $input = false, $output = false, $fault = false)` method adds new port operation to the specified port type of the WSDL document (/definitions/portType/operation).

Each port operation corresponds to a class method (if Web Service is based on a class) or function (if Web Service is based on a set of methods) in terms of `Zend_Soap_Server` implementation.

It also adds corresponding port operation messages depending on specified `$input`, `$output` and `$fault` parameters.

`Zend_Soap_Server` component generates two messages for each port operation while describing service based on `Zend_Soap_Server` class:

- input message with name `$methodName . 'Request'`.
- output message with name `$methodName . 'Response'`.

See http://www.w3.org/TR/wsdl#_request-response for the details.

3.5. addBinding() method

`addBinding($name, $portType)` method adds new binding to the WSDL document (/definitions/binding).

'binding' WSDL document node defines message format and protocol details for operations and messages defined by a particular portType (see http://www.w3.org/TR/wsdl#_bindings).

The method creates binding node and returns it. Then it may be used to fill with actual data.

`Zend_Soap_Server` implementation uses `$serviceName . 'Binding'` name for 'binding' element of WSDL document.

3.6. addBindingOperation() method

`addBindingOperation($binding, $name, $input = false, $output = false, $fault = false)` method adds an operation to a binding element (/definitions/binding/operation) with the specified name.

It takes `XML_Tree_Node` object returned by `addBinding()` as an input (`$binding` parameter) to add 'operation' element with input/output/false entries depending on specified parameters

`Zend_Soap_Server` implementation adds corresponding binding entry for each Web Service method with input and output entries defining 'soap:body' element as '<soap:body use="encoded" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>

See http://www.w3.org/TR/wsdl#_bindings for the details.

3.7. addSoapBinding() method

`addSoapBinding($binding, $style = 'document', $transport = 'http://schemas.xmlsoap.org/soap/http')` method adds SOAP binding ('soap:binding') entry to the binding element (which is already linked to some port type) with the specified style and transport (Zend_Soap_Server implementation uses RPC style over HTTP).

'/definitions/binding/soap:binding' element is used to signify that the binding is bound to the SOAP protocol format.

See http://www.w3.org/TR/wsdl#_bindings for the details.

3.8. addSoapOperation() method

`addSoapOperation($binding, $soap_action)` method adds SOAP operation ('soap:operation') entry to the binding element with the specified action. 'style' attribute of the 'soap:operation' element is not used since programming model (RPC-oriented or document-oriented) may be using `addSoapBinding()` method

'soapAction' attribute of '/definitions/binding/soap:operation' element specifies the value of the SOAPAction header for this operation. This attribute is required for SOAP over HTTP and *must not* be specified for other transports.

`Zend_Soap_Server` implementation uses `$serviceUri . '#' . $methodName` for SOAP operation action name.

See http://www.w3.org/TR/wsdl#_soap:operation for the details.

3.9. addService() method

`addService($name, $port_name, $binding, $location)` method adds '/definitions/service' element to the WSDL document with the specified Wed Service name, port name, binding, and location.

WSDL 1.1 allows to have several port types (sets of operations) per service. This ability is not used by `Zend_Soap_Server` implementation and not supported by `Zend_Soap_Wsdl` class.

`Zend_Soap_Server` implementation uses:

- `$name . 'Service'` as a Web Service name,
- `$name . 'Port'` as a port type name,
- `'tns:' . $name . 'Binding'` [5] as binding name,
- script URI[6] as a service URI for Web Service definition using classes.

where `$name` is a class name for the Web Service definition mode using class and script name for the Web Service definition mode using set of functions.

See http://www.w3.org/TR/wsdl#_services for the details.

3.10. Type mapping

`Zend_Soap` WSDL accessor implementation uses the following type mapping between PHP and SOAP types:

- PHP strings <-> `xsd:string`.
- PHP integers <-> `xsd:int`.
- PHP floats and doubles <-> `xsd:float`.
- PHP booleans <-> `xsd:boolean`.
- PHP arrays <-> `soap-enc:Array`.
- PHP object <-> `xsd:struct`.
- PHP class <-> based on complex type strategy (See: Section 3.10.2, “Adding complex type information”) [7].
- PHP void <-> empty type.
- If type is not matched to any of these types by some reason, then `xsd:anyType` is used.

Where `xsd:` is "http://www.w3.org/2001/XMLSchema" namespace, `soap-enc:` is a "http://schemas.xmlsoap.org/soap/encoding/" namespace, `tns:` is a "target namespace" for a service.

3.10.1. Retrieving type information

`getType($type)` method may be used to get mapping for a specified PHP type:

```
...
$wsdl = new Zend_Soap_Wsdl('My_Web_Service', $myWebServiceUri);

...
$soapIntType = $wsdl->getType('int');

...
class MyClass {
    ...
}
...
$soapMyClassType = $wsdl->getType('MyClass');
```

3.10.2. Adding complex type information

`addComplexType($type)` method is used to add complex types (PHP classes) to a WSDL document.

It's automatically used by `getType()` method to add corresponding complex types of method parameters or return types.

Its detection and building algorithm is based on the currently active detection strategy for complex types. You can set the detection strategy either by specifying the class name as string or instance of a `Zend_Soap_Wsdl_Strategy_Interface` implementation as the third parameter of the constructor or using the `setComplexTypeStrategy($strategy)` function of `Zend_Soap_Wsdl`. The following detection strategies currently exist:

- Class `Zend_Soap_Wsdl_Strategy_DefaultComplexType`: Enabled by default (when no third constructor parameter is set). Iterates over the public attributes of a class type and registers them as subtypes of the complex object type.
- Class `Zend_Soap_Wsdl_Strategy_AnyType`: Casts all complex types into the simple XSD type xsd:anyType. Be careful this shortcut for complex type detection can probably only be handled successfully by weakly typed languages such as PHP.
- Class `Zend_Soap_Wsdl_Strategy_ArrayOfTypeSequence`: This strategy allows to specify return parameters of the type: `int[]` or `string[]`. As of Zend Framework version 1.9 it can handle both simple PHP types such as int, string, boolean, float aswell as objects and arrays of objects.
- Class `Zend_Soap_Wsdl_Strategy_ArrayOfTypeComplex`: This strategy allows to detect very complex arrays of objects. Objects types are detected based on the `Zend_Soap_Wsdl_Strategy_DefaultComplexType` and an array is wrapped around that definition.
- Class `Zend_Soap_Wsdl_Strategy_Composite`: This strategy can combine all strategies by connecting PHP Complex types (Classnames) to the desired strategy via the `connectTypeToStrategy($type, $strategy)` method. A complete typemap can be given to the constructor as an array with `$type` -> `$strategy` pairs. The second parameter specifies the default strategy that will be used if an unknown type is requested for adding. This parameter defaults to the `Zend_Soap_Wsdl_Strategy_DefaultComplexType` strategy.

`addComplexType()` method creates '/definitions/types/xsd:schema/xsd:complexType' element for each described complex type with name of the specified PHP class.

Class property *MUST* have docblock section with the described PHP type to have property included into WSDL description.

`addComplexType()` checks if type is already described within types section of the WSDL document.

It prevents duplications if this method is called two or more times and recursion in the types definition section.

See http://www.w3.org/TR/wsdl#_types for the details.

3.11. addDocumentation() method

`addDocumentation($input_node, $documentation)` method adds human readable documentation using optional 'wsdl:document' element.

'/definitions/binding/soap:binding' element is used to signify that the binding is bound to the SOAP protocol format.

See http://www.w3.org/TR/wsdl#_documentation for the details.

3.12. Get finalized WSDL document

`toXML()`, `toDomDocument()` and `dump($filename = false)` methods may be used to get WSDL document as an XML, DOM structure or a file.

4. AutoDiscovery

4.1. AutoDiscovery Introduction

SOAP functionality implemented within Zend Framework is intended to make all steps required for SOAP communications more simple.

SOAP is language independent protocol. So it may be used not only for PHP-to-PHP communications.

There are three configurations for SOAP applications where Zend Framework may be utilized:

1. SOAP server PHP application <---> SOAP client PHP application
2. SOAP server non-PHP application <---> SOAP client PHP application
3. SOAP server PHP application <---> SOAP client non-PHP application

We always have to know, which functionality is provided by SOAP server to operate with it. WSDL is used to describe network service API in details.

WSDL language is complex enough (see http://www.w3.org/TR/wsdl for the details). So it's difficult to prepare correct WSDL description.

Another problem is synchronizing changes in network service API with already existing WSDL.

Both these problem may be solved by WSDL autogeneration. A prerequisite for this is a SOAP server autodiscovery. It constructs object similar to object used in SOAP server application, extracts necessary information and generates correct WSDL using this information.

There are two ways for using Zend Framework for SOAP server application:

- Use separated class.
- Use set of functions

Both methods are supported by Zend Framework Autodiscovery functionality.

The`Zend_Soap_AutoDiscover` class also supports datatypes mapping from PHP to XSD types.

Here is an example of common usage of the autodiscovery functionality. The `handle()` function generates the WSDL file and posts it to the browser.

```
class My_SoapServer_Class {
...
}
```

```
$autodiscover = new Zend_Soap_AutoDiscover();
$autodiscover->setClass('My_SoapServer_Class');
$autodiscover->handle();
```

If you need access to the generated WSDL file either to save it to a file or as an XML string you can use the `dump($filename)` or `toXml()` functions the AutoDiscover class provides.

Zend_Soap_Autodiscover is not a Soap Server

It is very important to note, that the class `Zend_Soap_AutoDiscover` does not act as a SOAP Server on its own. It only generates the WSDL and serves it to anyone accessing the url it is listening on.

As the SOAP Endpoint Uri is uses the default `'http://' . $_SERVER['HTTP_HOST'] . $_SERVER['SCRIPT_NAME']`, but this can be changed with the `setUri()` function or the Constructor parameter of `Zend_Soap_AutoDiscover` class. The endpoint has to provide a `Zend_Soap_Server` that listens to requests.

```
if(isset($_GET['wsdl'])) {
    $autodiscover = new Zend_Soap_AutoDiscover();
    $autodiscover->setClass('HelloWorldService');
    $autodiscover->handle();
} else {
    // pointing to the current file here
    $soap = new Zend_Soap_Server("http://example.com/soap.php?wsdl");
    $soap->setClass('HelloWorldService');
    $soap->handle();
}
```

4.2. Class autodiscovering

If class is used to provide SOAP server functionality, then the same class should be provided to `Zend_Soap_AutoDiscover` for WSDL generation:

```
$autodiscover = new Zend_Soap_AutoDiscover();
$autodiscover->setClass('My_SoapServer_Class');
$autodiscover->handle();
```

The following rules are used while WSDL generation:

- Generated WSDL describes an RPC style Web Service.
- Class name is used as a name of the Web Service being described.
- `'http://' .$_SERVER['HTTP_HOST'] . $_SERVER['SCRIPT_NAME']` is used as an URI where the WSDL is available by default but can be overwritten via `setUri()` method.

  It's also used as a target namespace for all service related names (including described complex types).
- Class methods are joined into one Port Type.

  `$className . 'Port'` is used as Port Type name.

- Each class method is registered as a corresponding port operation.
- Each method prototype generates corresponding Request/Response messages.

  Method may have several prototypes if some method parameters are optional.

Important!

WSDL autodiscovery utilizes the PHP docblocks provided by the developer to determine the parameter and return types. In fact, for scalar types, this is the only way to determine the parameter types, and for return types, this is the only way to determine them.

That means, providing correct and fully detailed docblocks is not only best practice, but is required for discovered class.

4.3. Functions autodiscovering

If set of functions are used to provide SOAP server functionality, then the same set should be provided to `Zend_Soap_AutoDiscovery` for WSDL generation:

```
$autodiscover = new Zend_Soap_AutoDiscover();
$autodiscover->addFunction('function1');
$autodiscover->addFunction('function2');
$autodiscover->addFunction('function3');
...
$autodiscover->handle();
```

The following rules are used while WSDL generation:

- Generated WSDL describes an RPC style Web Service.
- Current script name is used as a name of the Web Service being described.
- `'http://' .$_SERVER['HTTP_HOST'] . $_SERVER['SCRIPT_NAME']` is used as an URI where the WSDL is available.

  It's also used as a target namespace for all service related names (including described complex types).
- Functions are joined into one Port Type.

  `$functionName . 'Port'` is used as Port Type name.
- Each function is registered as a corresponding port operation.
- Each function prototype generates corresponding Request/Response messages.

  Function may have several prototypes if some method parameters are optional.

Important!

WSDL autodiscovery utilizes the PHP docblocks provided by the developer to determine the parameter and return types. In fact, for scalar types, this is the only way to determine the parameter types, and for return types, this is the only way to determine them.

That means, providing correct and fully detailed docblocks is not only best practice, but is required for discovered class.

4.4. Autodiscovering Datatypes

Input/output datatypes are converted into network service types using the following mapping:

- PHP strings <-> `xsd:string`.
- PHP integers <-> `xsd:int`.
- PHP floats and doubles <-> `xsd:float`.
- PHP booleans <-> `xsd:boolean`.
- PHP arrays <-> `soap-enc:Array`.
- PHP object <-> `xsd:struct`.
- PHP class <-> based on complex type strategy (See: Section 3.10.2, “Adding complex type information”) [8].
- type[] or object[] (ie. int[]) <-> based on complex type strategy
- PHP void <-> empty type.
- If type is not matched to any of these types by some reason, then `xsd:anyType` is used.

Where `xsd:` is "http://www.w3.org/2001/XMLSchema" namespace, `soap-enc:` is a "http://schemas.xmlsoap.org/soap/encoding/" namespace, `tns:` is a "target namespace" for a service.

4.5. WSDL Binding Styles

WSDL offers different transport mechanisms and styles. This affects the `soap:binding` and `soap:body` tags within the Binding section of WSDL. Different clients have different requirements as to what options really work. Therefore you can set the styles before you call any `setClass` or `addFunction` method on the AutoDiscover class.

```
$autodiscover = new Zend_Soap_AutoDiscover();
// Default is 'use' => 'encoded' and
// 'encodingStyle' => 'http://schemas.xmlsoap.org/soap/encoding/'
$autodiscover->setOperationBodyStyle(
                    array('use' => 'literal',
                          'namespace' => 'http://framework.zend.com')
                );

// Default is 'style' => 'rpc' and
// 'transport' => 'http://schemas.xmlsoap.org/soap/http'
$autodiscover->setBindingStyle(
                    array('style' => 'document',
                          'transport' => 'http://framework.zend.com')
                );
...
$autodiscover->addFunction('myfunc1');
$autodiscover->handle();
```

Zend_Tag

1. Introduction

Zend_Tag is a component suite which provides a facility to work with taggable Items. As its base, it provides two classes to work with Tags, Zend_Tag_Item and Zend_Tag_ItemList. Additionally, it comes with the interface Zend_Tag_Taggable, which allows you to use any of your models as a taggable item in conjunction with Zend_Tag.

Zend_Tag_Item is a basic taggable item implementation which comes with the essential functionality required to work with the Zend_Tag suite. A taggable item always consists of a title and a relative weight (e.g. number of occurrences). It also stores parameters which are used by the different sub-components of Zend_Tag.

To group multiple items together, Zend_Tag_ItemList exists as an array iterator and provides additional functionality to calculate absolute weight values based on the given relative weights of each item in it.

Example 871. Using Zend_Tag

This example illustrates how to create a list of tags and spread absolute weight values on them.

```
// Create the item list
$list = new Zend_Tag_ItemList();

// Assign tags to it
$list[] = new Zend_Tag_Item(array('title' => 'Code', 'weight' => 50));
$list[] = new Zend_Tag_Item(array('title' => 'Zend Framework', 'weight' => 1));
$list[] = new Zend_Tag_Item(array('title' => 'PHP', 'weight' => 5));

// Spread absolute values on the items
$list->spreadWeightValues(array(1, 2, 3, 4, 5, 6, 7, 8, 9, 10));

// Output the items with their absolute values
foreach ($list as $item) {
    printf("%s: %d\n", $item->getTitle(), $item->getParam('weightValue'));
}
```

This will output the three items Code, Zend Framework and PHP with the absolute values 10, 1 and 2.

2. Zend_Tag_Cloud

Zend_Tag_Cloud is the rendering part of Zend_Tag. By default it comes with a set of HTML decorators, which allow you to create tag clouds for a website, but also supplies you with two abstract classes to create your own decorators, to create tag clouds in PDF documents for example.

You can instantiate and configure Zend_Tag_Cloud either programatically or completely via an array or an instance of Zend_Config. The available options are:

- cloudDecorator: defines the decorator for the cloud. Can either be the name of the class which should be loaded by the pluginloader, an instance of

`Zend_Tag_Cloud_Decorator_Cloud` or an array containing the string `decorator` and optionally an array `options`, which will be passed to the decorators constructor.

- `tagDecorator`: defines the decorator for individual tags. This can either be the name of the class which should be loaded by the pluginloader, an instance of `Zend_Tag_Cloud_Decorator_Tag` or an array containing the string `decorator` and optionally an array `options`, which will be passed to the decorators constructor.

- `pluginLoader`: a different plugin loader to use. Must be an instance of `Zend_Loader_PluginLoader_Interface`.

- `prefixPath`: prefix paths to add to the plugin loader. Must be an array containing the keys prefix and path or multiple arrays containing the keys prefix and path. Invalid elements will be skipped.

- `itemList`: a different item list to use. Must be an instance of `Zend_Tag_ItemList`.

- `tags`: a list of tags to assign to the cloud. Each tag must either implement `Zend_Tag_Taggable` or be an array which can be used to instantiate `Zend_Tag_Item`.

Example 872. Using Zend_Tag_Cloud

This example illustrates a basic example of how to create a tag cloud, add multiple tags to it and finally render it.

```
// Create the cloud and assign static tags to it
$cloud = new Zend_Tag_Cloud(array(
    'tags' => array(
        array('title' => 'Code', 'weight' => 50,
              'params' => array('url' => '/tag/code')),
        array('title' => 'Zend Framework', 'weight' => 1,
              'params' => array('url' => '/tag/zend-framework')),
        array('title' => 'PHP', 'weight' => 5,
              'params' => array('url' => '/tag/php')),
    )
));

// Render the cloud
echo $cloud;
```

This will output the tag cloud with the three tags, spread with the default font-sizes.

2.1. Decorators

`Zend_Tag_Cloud` requires two types of decorators to be able to render a tag cloud. This includes a decorator which renders the single tags as well as a decorator which renders the surounding cloud. `Zend_Tag_Cloud` ships a default decorator set for formatting a tag cloud in HTML. This set will by default create a tag cloud as ul/li-list, spread with different font-sizes according to the weight values of the tags assigned to them.

2.1.1. HTML Tag decorator

The HTML tag decorator will by default render every tag in an anchor element, surounded by a li element. The anchor itself is fixed and cannot be changed, but the surounding element(s) can.

URL parameter

As the HTML tag decorator always surounds the tag title with an anchor, you should define an URL parameter for every tag used in it.

The tag decorator can either spread different font-sizes over the anchors or a defined list of classnames. When setting options for one of those possibilities, the corespondening one will automatically be enabled. The following configuration options are available:

- `fontSizeUnit`: defines the font-size unit used for all font-sizes. The possible values are: em, ex, px, in, cm, mm, pt, pc and %.
- `minFontSize`: the minimum font-size distributed through the tags (must be an integer).
- `maxFontSize`: the maximum font-size distributed through the tags (must be an integer).
- `classList`: an arry of classes distributed through the tags.
- `htmlTags`: an array of HTML tags surounding the anchor. Each element can either be a string, which is used as element type then, or an array containing an attribute list for the element, defined as key/value pair. In this case, the array key is used as element type.

2.1.2. HTML Cloud decorator

The HTML cloud decorator will suround the HTML tags with an ul-element by default and add no separation. Like in the tag decorator, you can define multiple surounding HTML tags and additionally define a separator. The available options are:

- `separator`: defines the separator which is placed between all tags.
- `htmlTags`: an array of HTML tags surounding all tags. Each element can either be a string, which is used as element type then, or an array containing an attribute list for the element, defined as key/value pair. In this case, the array key is used as element type.

Zend_Test

1. Introduction

`Zend_Test` provides tools to facilitate unit testing of your Zend Framework applications. At this time, we offer facilities to enable testing of your Zend Framework MVC applications

2. Zend_Test_PHPUnit

`Zend_Test_PHPUnit` provides a TestCase for MVC applications that contains assertions for testing against a variety of responsibilities. Probably the easiest way to understand what it can do is to see an example.

- When a user logs in, they should be redirected to their profile page, and that profile page should show relevant information

This example illustrates moving most of our bootstrapping to a plugin. This simplifies setup of the test case as it allows us to specify our environment succinctly, and also allows us to bootstrap the application in a single line. Also, our particular example is assuming that autoloading is setup so we do not need to worry about requiring

```
class UserControllerTest extends Zend_Test_PHPUnit_ControllerTestCase
{
    public function setUp()
    {
        $this->bootstrap = array($this, 'appBootstrap');
        parent::setUp();
    }

    public function appBootstrap()
    {
        $this->frontController
             ->registerPlugin(new Bugapp_Plugin_Initialize('development'));
    }

    public function testCallWithoutActionShouldPullFromIndexAction()
    {
        $this->dispatch('/user');
        $this->assertController('user');
        $this->assertAction('index');
    }

    public function testIndexActionShouldContainLoginForm()
    {
        $this->dispatch('/user');
        $this->assertAction('index');
        $this->assertQueryCount('form#loginForm', 1);
    }

    public function testValidLoginShouldGoToProfilePage()
    {
        $this->request->setMethod('POST')
              ->setPost(array(
                  'username' => 'foobar',
                  'password' => 'foobar'
              ));
        $this->dispatch('/user/login');
        $this->assertRedirectTo('/user/view');

        $this->resetRequest()
             ->resetResponse();

        $this->request->setMethod('GET')
             ->setPost(array());
        $this->dispatch('/user/view');
        $this->assertRoute('default');
        $this->assertModule('default');
        $this->assertController('user');
        $this->assertAction('view');
        $this->assertNotRedirect();
        $this->assertQuery('dl');
        $this->assertQueryContentContains('h2', 'User: foobar');
    }
}
```

This example could be written somewhat simpler -- not all the assertions shown are necessary, and are provided for illustration purposes only. Hopefully, it shows how simple it can be to test your applications.

2.1. Bootstrapping your TestCase

As noted in the Login example, all MVC test cases should extend `Zend_Test_PHPUnit_ControllerTestCase`. This class in turn extends `PHPUnit_Framework_TestCase`, and gives you all the structure and assertions you'd expect from PHPUnit -- as well as some scaffolding and assertions specific to Zend Framework's MVC implementation.

In order to test your MVC application, you will need to bootstrap it. There are several ways to do this, all of which hinge on the public `$bootstrap` property.

First, and probably most straight-forward, simply create a `Zend_Application` instance as you would in your `index.php`, and assign it to the `$bootstrap` property. Typically, you will do this in your `setUp()` method; you will need to call `parent::setUp()` when done:

```
class UserControllerTest extends Zend_Test_PHPUnit_ControllerTestCase
{
    public function setUp()
    {
        // Assign and instantiate in one step:
        $this->bootstrap = new Zend_Application(
            'testing',
            APPLICATION_PATH . '/configs/application.ini'
        );
        parent::setUp();
    }
}
```

Second, you can set this property to point to a file. If you do this, the file should *not* dispatch the front controller, but merely setup the front controller and any application specific needs.

```
class UserControllerTest extends Zend_Test_PHPUnit_ControllerTestCase
{
    public $bootstrap = '/path/to/bootstrap/file.php'

    // ...
}
```

Third, you can provide a PHP callback to execute in order to bootstrap your application. This method is seen in the Login example. If the callback is a function or static method, this could be set at the class level:

```
class UserControllerTest extends Zend_Test_PHPUnit_ControllerTestCase
{
    public $bootstrap = array('App', 'bootstrap');

    // ...
}
```

In cases where an object instance is necessary, we recommend performing this in your `setUp()` method:

```
class UserControllerTest extends Zend_Test_PHPUnit_ControllerTestCase
{
    public function setUp()
```

```
    {
        // Use the 'start' method of a Bootstrap object instance:
        $bootstrap = new Bootstrap('test');
        $this->bootstrap = array($bootstrap, 'start');
        parent::setUp();
    }
}
```

Note the call to `parent::setUp()`; this is necessary, as the `setUp()` method of `Zend_Test_PHPUnit_ControllerTestCase` will perform the remainder of the bootstrapping process (which includes calling the callback).

During normal operation, the `setUp()` method will bootstrap the application. This process first will include cleaning up the environment to a clean request state, resetting any plugins and helpers, resetting the front controller instance, and creating new request and response objects. Once this is done, it will then either `include()` the file specified in `$bootstrap`, or call the callback specified.

Bootstrapping should be as close as possible to how the application will be bootstrapped. However, there are several caveats:

- Do not provide alternate implementations of the Request and Response objects; they will not be used. `Zend_Test_PHPUnit_ControllerTestCase` uses custom request and response objects, `Zend_Controller_Request_HttpTestCase` and `Zend_Controller_Response_HttpTestCase`, respectively. These objects provide methods for setting up the request environment in targeted ways, and pulling response artifacts in specific ways.

- Do not expect to test server specifics. In other words, the tests are not a guarantee that the code will run on a specific server configuration, but merely that the application should run as expected should the router be able to route the given request. To this end, do not set server-specific headers in the request object.

Once the application is bootstrapped, you can then start creating your tests.

2.2. Testing your Controllers and MVC Applications

Once you have your bootstrap in place, you can begin testing. Testing is basically as you would expect in an PHPUnit test suite, with a few minor differences.

First, you will need to dispatch a URL to test, using the `dispatch()` method of the TestCase:

```
class IndexControllerTest extends Zend_Test_PHPUnit_ControllerTestCase
{
    // ...

    public function testHomePage()
    {
        $this->dispatch('/');
        // ...
    }
}
```

There will be times, however, that you need to provide extra information -- GET and POST variables, COOKIE information, etc. You can populate the request with that information:

```
class FooControllerTest extends Zend_Test_PHPUnit_ControllerTestCase
{
    // ...

    public function testBarActionShouldReceiveAllParameters()
    {
        // Set GET variables:
        $this->request->setQuery(array(
            'foo' => 'bar',
            'bar' => 'baz',
        ));

        // Set POST variables:
        $this->request->setPost(array(
            'baz'  => 'bat',
            'lame' => 'bogus',
        ));

        // Set a cookie value:
        $this->request->setCookie('user', 'matthew');
        // or many:
        $this->request->setCookies(array(
            'timestamp' => time(),
            'host'      => 'foobar',
        ));

        // Set headers, even:
        $this->request->setHeader('X-Requested-With', 'XmlHttpRequest');

        // Set the request method:
        $this->request->setMethod('POST');

        // Dispatch:
        $this->dispatch('/foo/bar');

        // ...
    }
}
```

Now that the request is made, it's time to start making assertions against it.

2.2.1. Controller Tests and the Redirector Action Helper

The redirect action helper issues an `exit()` statement when using the method `gotoAndExit()` and will then obviously also stops a test running for this method. For testability of your application dont use that method on the redirector!

Due to its nature the redirector action helper plugin issues a redirect and exists after this. Because you cannot test parts of an application that issue exit calls `Zend_Test_PHPUnit_ControllerTestCase` automatically disables the exit part of the redirector which can cause different behaviours in tests and the real application. To make sure redirect work correctly you should it them in the following way:

```
class MyController extends Zend_Controller_Action
{
    public function indexAction()
    {
```

```
        if($someCondition == true) {
            return $this->_redirect(...);
        } else if($anotherCondition == true) {
            $this->_redirector->gotoSimple("foo");
            return;
        }

        // do some stuff here
    }
}
```

Depending on your application this is not enough as additional action, `preDispatch()` or `postDispatch()` logic might be executed. This cannot be handled in a good way with Zend Test currently.

2.3. Assertions

Assertions are at the heart of Unit Testing; you use them to verify that the results are what you expect. To this end, `Zend_Test_PHPUnit_ControllerTestCase` provides a number of assertions to make testing your MVC apps and controllers simpler.

2.3.1. CSS Selector Assertions

CSS selectors are an easy way to verify that certain artifacts are present in the response content. They also make it trivial to ensure that items necessary for Javascript UIs and/or AJAX integration will be present; most JS toolkits provide some mechanism for pulling DOM elements based on CSS selectors, so the syntax would be the same.

This functionality is provided via Zend_Dom_Query, and integrated into a set of 'Query' assertions. Each of these assertions takes as their first argument a CSS selector, with optionally additional arguments and/or an error message, based on the assertion type. You can find the rules for writing the CSS selectors in the Zend_Dom_Query theory of operation chapter. Query assertions include:

- `assertQuery($path, $message = '')`: assert that one or more DOM elements matching the given CSS selector are present. If a `$message` is present, it will be prepended to any failed assertion message.
- `assertQueryContentContains($path, $match, $message = '')`: assert that one or more DOM elements matching the given CSS selector are present, and that at least one contains the content provided in `$match`. If a `$message` is present, it will be prepended to any failed assertion message.
- `assertQueryContentRegex($path, $pattern, $message = '')`: assert that one or more DOM elements matching the given CSS selector are present, and that at least one matches the regular expression provided in `$pattern`. If a `$message` is present, it will be prepended to any failed assertion message.
- `assertQueryCount($path, $count, $message = '')`: assert that there are exactly `$count` DOM elements matching the given CSS selector present. If a `$message` is present, it will be prepended to any failed assertion message.
- `assertQueryCountMin($path, $count, $message = '')`: assert that there are at least `$count` DOM elements matching the given CSS selector present. If a `$message` is

present, it will be prepended to any failed assertion message. *Note:* specifying a value of 1 for `$count` is the same as simply using `assertQuery()`.

- `assertQueryCountMax($path, $count, $message = '')`: assert that there are no more than `$count` DOM elements matching the given CSS selector present. If a `$message` is present, it will be prepended to any failed assertion message. *Note:* specifying a value of 1 for `$count` is the same as simply using `assertQuery()`.

Additionally, each of the above has a 'Not' variant that provides a negative assertion: `assertNotQuery()`, `assertNotQueryContentContains()`, `assertNotQueryContentRegex()`, and `assertNotQueryCount()`. (Note that the min and max counts do not have these variants, for what should be obvious reasons.)

2.3.2. XPath Assertions

Some developers are more familiar with XPath than with CSS selectors, and thus XPath variants of all the Query assertions are also provided. These are:

- `assertXpath($path, $message = '')`
- `assertNotXpath($path, $message = '')`
- `assertXpathContentContains($path, $match, $message = '')`
- `assertNotXpathContentContains($path, $match, $message = '')`
- `assertXpathContentRegex($path, $pattern, $message = '')`
- `assertNotXpathContentRegex($path, $pattern, $message = '')`
- `assertXpathCount($path, $count, $message = '')`
- `assertNotXpathCount($path, $count, $message = '')`
- `assertXpathCountMin($path, $count, $message = '')`
- `assertNotXpathCountMax($path, $count, $message = '')`

2.3.3. Redirect Assertions

Often an action will redirect. Instead of following the redirect, `Zend_Test_PHPUnit_ControllerTestCase` allows you to test for redirects with a handful of assertions.

- `assertRedirect($message = '')`: assert simply that a redirect has occurred.
- `assertNotRedirect($message = '')`: assert that no redirect has occurred.
- `assertRedirectTo($url, $message = '')`: assert that a redirect has occurred, and that the value of the Location header is the `$url` provided.
- `assertNotRedirectTo($url, $message = '')`: assert that a redirect has either NOT occurred, or that the value of the Location header is NOT the `$url` provided.
- `assertRedirectRegex($pattern, $message = '')`: assert that a redirect has occurred, and that the value of the Location header matches the regular expression provided by `$pattern`.

- `assertNotRedirectRegex($pattern, $message = '')`: assert that a redirect has either NOT occurred, or that the value of the Location header does NOT match the regular expression provided by `$pattern`.

2.3.4. Response Header Assertions

In addition to checking for redirect headers, you will often need to check for specific HTTP response codes and headers -- for instance, to determine whether an action results in a 404 or 500 response, or to ensure that JSON responses contain the appropriate Content-Type header. The following assertions are available.

- `assertResponseCode($code, $message = '')`: assert that the response resulted in the given HTTP response code.

- `assertHeader($header, $message = '')`: assert that the response contains the given header.

- `assertHeaderContains($header, $match, $message = '')`: assert that the response contains the given header and that its content contains the given string.

- `assertHeaderRegex($header, $pattern, $message = '')`: assert that the response contains the given header and that its content matches the given regex.

Additionally, each of the above assertions have a 'Not' variant for negative assertions.

2.3.5. Request Assertions

It's often useful to assert against the last run action, controller, and module; additionally, you may want to assert against the route that was matched. The following assertions can help you in this regard:

- `assertModule($module, $message = '')`: Assert that the given module was used in the last dispatched action.

- `assertController($controller, $message = '')`: Assert that the given controller was selected in the last dispatched action.

- `assertAction($action, $message = '')`: Assert that the given action was last dispatched.

- `assertRoute($route, $message = '')`: Assert that the given named route was matched by the router.

Each also has a 'Not' variant for negative assertions.

2.4. Examples

Knowing how to setup your testing infrastructure and how to make assertions is only half the battle; now it's time to start looking at some actual testing scenarios to see how you can leverage them.

```
    public function testCallWithoutActionShouldPullFromIndexAction()
    {
        $this->dispatch('/user');
        $this->assertController('user');
        $this->assertAction('index');
    }

    public function testLoginFormShouldContainLoginAndRegistrationForms()
    {
        $this->dispatch('/user');
        $this->assertQueryCount('form', 2);
    }

    public function testInvalidCredentialsShouldResultInRedisplayOfLoginForm()
    {
        $request = $this->getRequest();
        $request->setMethod('POST')
                ->setPost(array(
                    'username' => 'bogus',
                    'password' => 'reallyReallyBogus',
                ));
        $this->dispatch('/user/login');
        $this->assertNotRedirect();
        $this->assertQuery('form');
    }

    public function testValidLoginShouldRedirectToProfilePage()
    {
        $this->loginUser('foobar', 'foobar');
    }

    public function testAuthenticatedUserShouldHaveCustomizedProfilePage()
    {
        $this->loginUser('foobar', 'foobar');
        $this->request->setMethod('GET');
        $this->dispatch('/user/view');
        $this->assertNotRedirect();
        $this->assertQueryContentContains('h2', 'foobar');
    }

    public function
        testAuthenticatedUsersShouldBeRedirectedToProfileWhenVisitingLogin()
    {
        $this->loginUser('foobar', 'foobar');
        $this->request->setMethod('GET');
        $this->dispatch('/user');
        $this->assertRedirectTo('/user/view');
    }

    public function testUserShouldRedirectToLoginPageOnLogout()
    {
        $this->loginUser('foobar', 'foobar');
        $this->request->setMethod('GET');
        $this->dispatch('/user/logout');
        $this->assertRedirectTo('/user');
    }

    public function testRegistrationShouldFailWithInvalidData()
    {
        $data = array(
            'username' => 'this will not work',
            'email'    => 'this is an invalid email',
        );
        $request = $this->getRequest();
        $request->setMethod('POST')
                ->setPost($data);
        $this->dispatch('/user/register');
        $this->assertNotRedirect();
        $this->assertQuery('form .errors');
    }
```

This application *may* utilize a database. If so, you will probably need some scaffolding to ensure that the database is in a pristine, testable configuration at the beginning of each test. PHPUnit already provides functionality for doing so; read about it in the PHPUnit documentation. We recommend using a separate database for testing versus production, and in particular recommend either a SQLite file or in-memory database, as both options perform very well, do not require a separate server, and can utilize most SQL syntax. Also notice that we use the structure of the document our tests. For instance, in the final test, we look for a form that has a child with the class of `errors`; this allows us to test merely for the presence of form validation errors, and not worry about what specific errors might have been thrown. Notice that these are terse, and, for the most part, don't look for actual content. Instead, they look for artifacts within the response -- response codes and headers, and DOM nodes. This allows you to verify that the structure is as expected -- preventing your tests from choking every time new content is added to the site.

3. Zend_Test_PHPUnit_Db

Coupling of data-access and the domain model often requires the use of a database for testing purposes. But the database is persistent across different tests which leads to test results that can affect each other. Furthermore setting up the database to be able to run a test is quite some work. PHPUnit's Database extension simplifies testing with a database by offering a very simple mechanism to set up and teardown the database between different tests. This component extends the PHPUnit Database extension with Zend Framework specific code, such that writing database tests against a Zend Framework application is simplified.

Database Testing can be explained with two conceptual entities, DataSets and DataTables. Internally the PHPUnit Database extension can build up an object structure of a database, its tables and containing rows from configuration files or the real database content. This abstract object graph can then be compared using assertions. A common use-case in database testing is setting up some tables with seed data, then performing some operations, and finally asserting that the operated on database-state is equal to some predefined expected state. `Zend_Test_PHPUnit_Db` simplifies this task by allowing to generate DataSets and DataTables from existing `Zend_Db_Table_Abstract` or `Zend_Db_Table_Rowset_Abstract` instances.

Furthermore this component allows to integrate any `Zend_Db_Adapter_Abstract` for testing whereas the original extension only works with PDO. A Test Adapter implementation for `Zend_Db_Adapter_Abstract` is also included in this component. It allows to instantiate a Db Adapter that requires no database at all and acts as an SQL and result stack which is used by the API methods.

3.1. Quickstart

3.1.1. Setup a Database TestCase

We are now writting some database tests for the Bug Database example in the `Zend_Db_Table` documentation. First we begin to test that inserting a new bug is actually saved in the database correctly. First we have to setup a test-class that extends `Zend_Test_PHPUnit_DatabaseTestCase`. This class extends the PHPUnit Database Extension, which in turn extends the basic `PHPUnit_Framework_TestCase`. A database testcase contains two abstract methods that have to be implemented, one for the database connection and one for the initial dataset that should be used as seed or fixture.

You should be familiar with the PHPUnit Database extension to follow this quickstart easily. Although all the concepts are explained in this documentation it may be helpful to read the PHPUnit documentation first.

```
class BugsTest extends Zend_Test_PHPUnit_DatabaseTestCase
{
    private $_connectionMock;

    /**
     * Returns the test database connection.
     *
     * @return PHPUnit_Extensions_Database_DB_IDatabaseConnection
     */
    protected function getConnection()
    {
        if($this->_connectionMock == null) {
```

```
            $connection = Zend_Db::factory(...);
            $this->_connectionMock = $this->createZendDbConnection(
                $connection, 'zfunittests'
            );
            Zend_Db_Table_Abstract::setDefaultAdapter($connection);
        }
        return $this->_connectionMock;
    }

    /**
     * @return PHPUnit_Extensions_Database_DataSet_IDataSet
     */
    protected function getDataSet()
    {
        return $this->createFlatXmlDataSet(
            dirname(__FILE__) . '/_files/bugsSeed.xml'
        );
    }
}
```

Here we create the database connection and seed some data into the database. Some important details should be noted on this code:

- You cannot directly return a `Zend_Db_Adapter_Abstract` from the `getConnection()` method, but a PHPUnit specific wrapper which is generated with the `createZendDbConnection()` method.
- The database schema (tables and database) is not re-created on every testrun. The database and tables have to be created manually before running the tests.
- Database tests by default truncate the data during `setUp()` and then insert the seed data which is returned from the `getDataSet()` method.
- DataSets have to implement the interface `PHPUnit_Extensions_Database_DataSet_IDataSet`. There is a wide range of XML and YAML configuration file types included in PHPUnit which allows to specifiy how the tables and datasets should look like and you should look into the PHPUnit documentation to get the latest information on these dataset specifications.

3.1.2. Specify a seed dataset

In the previous setup for the database testcase we have specified a seed file for the database fixture. We now create this file specified in the Flat XML format:

```
<?xml version="1.0" encoding="UTF-8" ?>
<dataset>
    <zfbugs bug_id="1" bug_description="system needs electricity to run"
        bug_status="NEW" created_on="2007-04-01 00:00:00"
        updated_on="2007-04-01 00:00:00" reported_by="goofy"
        assigned_to="mmouse" verified_by="dduck" />
    <zfbugs bug_id="2" bug_description="Implement Do What I Mean function"
        bug_status="VERIFIED" created_on="2007-04-02 00:00:00"
        updated_on="2007-04-02 00:00:00" reported_by="goofy"
        assigned_to="mmouse" verified_by="dduck" />
    <zfbugs bug_id="3" bug_description="Where are my keys?" bug_status="FIXED"
        created_on="2007-04-03 00:00:00" updated_on="2007-04-03 00:00:00"
        reported_by="dduck" assigned_to="mmouse" verified_by="dduck" />
    <zfbugs bug_id="4" bug_description="Bug no product" bug_status="INCOMPLETE"
        created_on="2007-04-04 00:00:00" updated_on="2007-04-04 00:00:00"
```

```
        reported_by="mmouse" assigned_to="goofy" verified_by="dduck" />
</dataset>
```

We will work with this four entries in the database table "zfbugs" in the next examples. The required MySQL schema for this example is:

```
CREATE TABLE IF NOT EXISTS `zfbugs` (
    `bug_id` int(11) NOT NULL auto_increment,
    `bug_description` varchar(100) default NULL,
    `bug_status` varchar(20) default NULL,
    `created_on` datetime default NULL,
    `updated_on` datetime default NULL,
    `reported_by` varchar(100) default NULL,
    `assigned_to` varchar(100) default NULL,
    `verified_by` varchar(100) default NULL,
PRIMARY KEY  (`bug_id`)
) ENGINE=InnoDB AUTO_INCREMENT=1 ;
```

3.1.3. A few initial database tests

Now that we have implemented the two required abstract methods of the `Zend_Test_PHPUnit_DatabaseTestCase` and specified the seed database content, which will be re-created for each new test, we can go about to make our first assertion. This will be a test to insert a new bug.

```
class BugsTest extends Zend_Test_PHPUnit_DatabaseTestCase
{
    public function testBugInsertedIntoDatabase()
    {
        $bugsTable = new Bugs();

        $data = array(
            'created_on'      => '2007-03-22 00:00:00',
            'updated_on'      => '2007-03-22 00:00:00',
            'bug_description' => 'Something wrong',
            'bug_status'      => 'NEW',
            'reported_by'     => 'garfield',
            'verified_by'     => 'garfield',
            'assigned_to'     => 'mmouse',
        );

        $bugsTable->insert($data);

        $ds = new Zend_Test_PHPUnit_Db_DataSet_QueryDataSet(
            $this->getConnection()
        );
        $ds->addTable('zfbugs', 'SELECT * FROM zfbugs');

        $this->assertDataSetsEqual(
            $this->createFlatXmlDataSet(dirname(__FILE__)
                                        . "/_files/bugsInsertIntoAssertion.xml"),
            $ds
        );
    }
}
```

Now up to the `$bugsTable->insert($data);` everything looks familiar. The lines after that contain the assertion methodname. We want to verify that after inserting the new

bug the database has been updated correctly with the given data. For this we create a `Zend_Test_PHPUnit_Db_DataSet_QueryDataSet` instance and give it a database connection. We will then tell this dataset that it contains a table "zfbugs" which is given by an SQL statement. This current/actual state of the database is compared to the expected database state which is contained in another XML file "bugsInsertIntoAssertions.xml". This XML file is a slight deviation from the one given above and contains another row with the expected data:

```
<?xml version="1.0" encoding="UTF-8" ?>
<dataset>
    <!-- previous 4 rows -->
    <zfbugs bug_id="5" bug_description="Something wrong" bug_status="NEW"
        created_on="2007-03-22 00:00:00" updated_on="2007-03-22 00:00:00"
        reported_by="garfield" assigned_to="mmouse" verified_by="garfield" />
</dataset>
```

There are other ways to assert that the current database state equals an expected state. The "Bugs" table in the example already nows a lot about its inner state, so why not use this to our advantage? The next example will assert that deleting from the database is possible:

```
class BugsTest extends Zend_Test_PHPUnit_DatabaseTestCase
{
    public function testBugDelete()
    {
        $bugsTable = new Bugs();

        $bugsTable->delete(
            $bugsTable->getAdapter()->quoteInto("bug_id = ?", 4)
        );

        $ds = new Zend_Test_PHPUnit_Db_DataSet_DbTableDataSet();
        $ds->addTable($bugsTable);

        $this->assertDataSetsEqual(
            $this->createFlatXmlDataSet(dirname(__FILE__)
                                        . "/_files/bugsDeleteAssertion.xml"),
            $ds
        );
    }
}
```

We have created a `Zend_Test_PHPUnit_Db_DataSet_DbTableDataSet` dataset here, which takes any `Zend_Db_Table_Abstract` instance and adds it to the dataset with its table name, in this example "zfbugs". You could add several tables more if you wanted using the method `addTable()` if you want to check for expected database state in more than one table.

Here we only have one table and check against an expected database state in "bugsDeleteAssertion.xml" which is the original seed dataset without the row with id 4.

Since we have only checked that two specific tables (not datasets) are equal in the previous examples we should also look at how to assert that two tables are equal. Therefore we will add another test to our TestCase which verifies updating behaviour of a dataset.

```
class BugsTest extends Zend_Test_PHPUnit_DatabaseTestCase
{
    public function testBugUpdate()
    {
        $bugsTable = new Bugs();
```

```
        $data = array(
            'updated_on'      => '2007-05-23',
            'bug_status'      => 'FIXED'
        );

        $where = $bugsTable->getAdapter()->quoteInto('bug_id = ?', 1);

        $bugsTable->update($data, $where);

        $rowset = $bugsTable->fetchAll();

        $ds        = new Zend_Test_PHPUnit_Db_DataSet_DbRowset($rowset);
        $assertion = $this->createFlatXmlDataSet(
            dirname(__FILE__) . '/_files/bugsUpdateAssertion.xml'
        );
        $expectedRowsets = $assertion->getTable('zfbugs');

        $this->assertTablesEqual(
            $expectedRowsets, $ds
        );
    }
}
```

Here we create the current database state from a `Zend_Db_Table_Rowset_Abstract` instance in conjunction with the `Zend_Test_PHPUnit_Db_DataSet_DbRowset($rowset)` instance which creates an internal data-representation of the rowset. This can again be compared against another data-table by using the `$this->assertTablesEqual()` assertion.

3.2. Usage, API and Extensions Points

The Quickstart already gave a good introduction on how database testing can be done using PHPUnit and the Zend Framework. This section gives an overview over the API that the `Zend_Test_PHPUnit_Db` component comes with and how it works internally.

Some Remarks on Database Testing

Just as the Controller TestCase is testing an application at an integration level, the Database TestCase is an integration testing method. Its using several different application layers for testing purposes and therefore should be consumed with caution.

It should be noted that testing domain and business logic with integration tests such as Zend Framework's Controller and Database TestCases is a bad practice. The purpose of an Integration test is to check that several parts of an application work smoothly when wired together. These integration tests do not replace the need for a set of unit tests that test the domain and business logic at a much smaller level, the isolated class.

3.2.1. The Zend_Test_PHPUnit_DatabaseTestCase class

The `Zend_Test_PHPUnit_DatabaseTestCase` class derives from the `PHPUnit_Extensions_Database_TestCase` which allows to setup tests with a fresh database fixture on each run easily. The Zend implementation offers some additional convenience features over the PHPUnit Database extension when it comes to using `Zend_Db` resources inside your tests. The workflow of a database test-case can be described as follows.

1. For each test PHPUnit creates a new instance of the TestCase and calls the `setUp()` method.

2. The Database TestCase creates an instance of a Database Tester which handles the setting up and tearing down of the database.

3. The database tester collects the information on the database connection and initial dataset from `getConnection()` and `getDataSet()` which are both abstract methods and have to be implemented by any Database Testcase.

4. By default the database tester truncates the tables specified in the given dataset, and then inserts the data given as initial fixture.

5. When the database tester has finished setting up the database, PHPUnit runs the test.

6. After running the test, `tearDown()` is called. Because the database is wiped in `setUp()` before inserting the required initial fixture, no actions are executed by the database tester at this stage.

The Database TestCase expects the database schema and tables to be setup correctly to run the tests. There is no mechanism to create and tear down database tables.

The `Zend_Test_PHPUnit_DatabaseTestCase` class has some convenience functions that can help writing tests that interact with the database and the database testing extension.

The next table lists only the new methods compared to the `PHPUnit_Extensions_Database_TestCase`, whose API is documented in the PHPUnit Documentation.

Table 147. Zend_Test_PHPUnit_DatabaseTestCase API Methods

Method	Description
`createZendDbConnection(Zend_Db_Adapter_Abstract $connection, $schema)`	Create a PHPUnit Database Extension compatible Connection instance from a `Zend_Db_Adapter_Abstract` instance. This method should be used in for testcase setup when implementing the abstract `getConnection()` method of the database testcase.
`getAdapter()`	Convenience method to access the underlying `Zend_Db_Adapter_Abstract` instance which is nested inside the PHPUnit database connection created with `getConnection()`.
`createDbRowset(Zend_Db_Table_Rowset_Abstract $rowset, $tableName = null)`	Create a DataTable Object that is filled with the data from a given `Zend_Db_Table_Rowset_Abstract` instance. The table the rowset is connected to is chosen when `$tableName` is `NULL`.
`createDbTable(Zend_Db_Table_Abstract $table, $where = null, $order = null, $count = null, $offset = null)`	Create a DataTable object that represents the data contained in a `Zend_Db_Table_Abstract` instance. For

Method	Description
	retrieving the data `fetchAll()` is used, where the optional parameters can be used to restrict the data table to a certain subset.
`createDbTableDataSet(array $tables=array())`	Create a DataSet containing the given `$tables`, an array of `Zend_Db_Table_Abstract` instances.

3.2.2. Integrating Database Testing with the ControllerTestCase

Because PHP does not support multiple inheritance it is not possible to use the Controller and Database testcases in conjunction. However you can use the `Zend_Test_PHPUnit_Db_SimpleTester` database tester in your controller test-case to setup a database enviroment fixture for each new controller test. The Database TestCase in general is only a set of convenience functions which can also be accessed and used without the test case.

Example 875. Database integration example

This example extends the User Controller Test from the `Zend_Test_PHPUnit_ControllerTestCase` documentation to include a database setup.

```
class UserControllerTest extends Zend_Test_PHPUnit_ControllerTestCase
{
    public function setUp()
    {
        $this->setupDatabase();
        $this->bootstrap = array($this, 'appBootstrap');
        parent::setUp();
    }

    public function setupDatabase()
    {
        $db = Zend_Db::factory(...);
        $connection = new Zend_Test_PHPUnit_Db_Connection($db,
                                                          'database_schema_name');
        $databaseTester = new Zend_Test_PHPUnit_Db_SimpleTester($connection);

        $databaseFixture =
                    new PHPUnit_Extensions_Database_DataSet_FlatXmlDataSet(
                        dirname(__FILE__) . '/_files/initialUserFixture.xml'
                    );

        $databaseTester->setupDatabase($databaseFixture);
    }
}
```

Now the Flat XML dataset "initialUserFixture.xml" is used to set the database into an initial state before each test, exactly as the DatabaseTestCase works internally.

3.3. Using the Database Testing Adapter

There are times when you don't want to test parts of your application with a real database, but are forced to because of coupling. The `Zend_Test_DbAdapter` offers a convenient way to

use a implementation of `Zend_Db_Adapter_Abstract` without having to open a database connection. Furthermore this Adapter is very easy to mock from within your PHPUnit testsuite, since it requires no constructor arguments.

The Test Adapter acts as a stack for various database results. Its order of results have to be userland implemented, which might be a tedious task for tests that call many different database queries, but its just the right helper for tests where only a handful of queries are executed and you know the exact order of the results that have to be returned to your userland code.

```
$adapter   = new Zend_Test_DbAdapter();
$stmt1Rows = array(array('foo' => 'bar'), array('foo' => 'baz'));
$stmt1     = Zend_Test_DbStatement::createSelectStatement($stmt1Rows);
$adapter->appendStatementToStack($stmt1);

$stmt2Rows = array(array('foo' => 'bar'), array('foo' => 'baz'));
$stmt2     = Zend_Test_DbStatement::createSelectStatement($stmt2Rows);
$adapter->appendStatementToStack($stmt2);

$rs = $adapter->query('SELECT ...'); // Returns Statement 2
while ($row = $rs->fetch()) {
    echo $rs['foo']; // Prints "Bar", "Baz"
}
$rs = $adapter->query('SELECT ...'); // Returns Statement 1
```

Behaviour of any real database adapter is simulated as much as possible such that methods like `fetchAll()`, `fetchObject()`, `fetchColumn` and more are working for the test adapter.

You can also put INSERT, UPDATE and DELETE statement onto the result stack, these however only return a statement which allows to specifiy the result of `$stmt->rowCount()`.

```
$adapter = new Zend_Test_DbAdapter();
$adapter->appendStatementToStack(
    Zend_Test_DbStatement::createInsertStatement(1)
);
$adapter->appendStatementToStack(
    Zend_Test_DbStatement::createUpdateStatement(2)
);
$adapter->appendStatementToStack(
    Zend_Test_DbStatement::createDeleteStatement(10
));
```

By default the query profiler is enabled, so that you can retrieve the executed SQL statements and their bound parameters to check for the correctness of the execution.

```
$adapter = new Zend_Test_DbAdapter();
$stmt = $adapter->query("SELECT * FROM bugs");

$qp = $adapter->getProfiler()->getLastQueryProfile();

echo $qp->getQueY(); // SELECT * FROM bugs
```

The test adapter never checks if the query specified is really of the type SELECT, DELETE, INSERT or UPDATE which is returned next from the stack. The correct order of returning the data has to be implemented by the user of the test adapter.

The Test adapter also specifies methods to simulate the use of the methods `listTables()`, `describeTables()` and `lastInsertId()`. Additionally using the

`setQuoteIdentifierSymbol()` you can specify which symbol should be used for quoting, by default none is used.

Zend_Text

1. Zend_Text_Figlet

`Zend_Text_Figlet` is a component which enables developers to create a so called FIGlet text. A FIGlet text is a string, which is represented as ASCII art. FIGlets use a special font format, called FLT (FigLet Font). By default, one standard font is shipped with `Zend_Text_Figlet`, but you can download additional fonts at http://www.figlet.org.

Compressed fonts

`Zend_Text_Figlet` supports gzipped fonts. This means that you can take an `.flf` file and gzip it. To allow `Zend_Text_Figlet` to recognize this, the gzipped font must have the extension `.gz`. Further, to be able to use gzipped fonts, you have to have enabled the GZIP extension of PHP.

Encoding

`Zend_Text_Figlet` expects your strings to be UTF-8 encoded by default. If this is not the case, you can supply the character encoding as second parameter to the `render()` method.

You can define multiple options for a FIGlet. When instantiating `Zend_Text_Figlet`, you can supply an array or an instance of `Zend_Config`.

- `font` - Defines the font which should be used for rendering. If not defines, the built-in font will be used.
- `outputWidth` - Defines the maximum width of the output string. This is used for word-wrap as well as justification. Beware of too small values, they may result in an undefined behaviour. The default value is 80.
- `handleParagraphs` - A boolean which indicates, how new lines are handled. When set to TRUE, single new lines are ignored and instead treated as single spaces. Only multiple new lines will be handled as such. The default value is FALSE.
- `justification` - May be one of the values of `Zend_Text_Figlet::JUSTIFICATION_*`. There is `JUSTIFICATION_LEFT`, `JUSTIFICATION_CENTER` and `JUSTIFICATION_RIGHT` The default justification is defined by the `rightToLeft` value.
- `rightToLeft` - Defines in which direction the text is written. May be either `Zend_Text_Figlet::DIRECTION_LEFT_TO_RIGHT` or `Zend_Text_Figlet::DIRECTION_RIGHT_TO_LEFT`. By default the setting of the font file is used. When justification is not defined, a text written from right-to-left is automatically right-aligned.
- `smushMode` - An integer bitfield which defines, how the single characters are smushed together. Can be the sum of multiple values from `Zend_Text_Figlet::SM_*`. There are the following smush modes: SM_EQUAL, SM_LOWLINE, SM_HIERARCHY, SM_PAIR, SM_BIGX, SM_HARDBLANK, SM_KERN and SM_SMUSH. A value of 0 doesn't disable the entire smushing, but forces SM_KERN to be applied, while a value of -1 disables it. An explanation of the different smush modes can be found here. By default the setting of the font

file is used. The smush mode option is normally used only by font designers testing the various layoutmodes with a new font.

Example 876. Using Zend_Text_Figlet

This example illustrates the basic use of `Zend_Text_Figlet` to create a simple FIGlet text:

```
$figlet = new Zend_Text_Figlet();
echo $figlet->render('Zend');
```

Assuming you are using a monospace font, this would look as follows:

```
 _____    _____    _  __   _____
|__  //   |  ___||  | \| || |  __ \\
  / //    | ||__    |  ' || | |  \ ||
 / //__   | ||___   | .  || | |__/ ||
/_____||  |_____||  |_|\_|| |_____//
`-----`'  `-----`   `-` -`'  -----`
```

2. Zend_Text_Table

`Zend_Text_Table` is a component to create text based tables on the fly with different decorators. This can be helpful, if you either want to send structured data in text emails, which are used to have mono-spaced fonts, or to display table information in a CLI application. `Zend_Text_Table` supports multi-line columns, colspan and align as well.

Encoding

`Zend_Text_Table` expects your strings to be UTF-8 encoded by default. If this is not the case, you can either supply the character encoding as a parameter to the `constructor` or the `setContent` method of `Zend_Text_Table_Column`. Alternatively if you have a different encoding in the entire process, you can define the standard input charset with `Zend_Text_Table::setInputCharset($charset)`. In case you need another output charset for the table, you can set this with `Zend_Text_Table::setOutputCharset($charset)`.

A `Zend_Text_Table` object consists of rows, which contain columns, represented by `Zend_Text_Table_Row` and `Zend_Text_Table_Column`. When creating a table, you can supply an array with options for the table. Those are:

- `columnWidths` (required): An array defining all columns width their widths in characters.
- `decorator`: The decorator to use for the table borders. The default is `unicode`, but you may also specify `ascii` or give an instance of a custom decorator object.
- `padding`: The left and right padding withing the columns in characters. The default padding is zero.
- `AutoSeparate`: The way how the rows are separated with horizontal lines. The default is a separation between all rows. This is defined as a bitmask containing one ore more of the following constants of `Zend_Text_Table`:
  - `Zend_Text_Table::AUTO_SEPARATE_NONE`
  - `Zend_Text_Table::AUTO_SEPARATE_HEADER`

- `Zend_Text_Table::AUTO_SEPARATE_FOOTER`
- `Zend_Text_Table::AUTO_SEPARATE_ALL`

Where header is always the first row, and the footer is always the last row.

Rows are simply added to the table by creating a new instance of `Zend_Text_Table_Row`, and appending it to the table via the `appendRow` method. Rows themselves have no options. You can also give an array to directly to the `appendRow` method, which then will automatically converted to a row object, containing multiple column objects.

The same way you can add columns to the rows. Create a new instance of `Zend_Text_Table_Column` and then either set the column options in the constructor or later with the `set*` methods. The first parameter is the content of the column which may have multiple lines, which in the best case are separated by just the `\n` character. The second parameter defines the align, which is `left` by default and can be one of the class constants of `Zend_Text_Table_Column`:

- `ALIGN_LEFT`
- `ALIGN_CENTER`
- `ALIGN_RIGHT`

The third parameter is the colspan of the column. For example, when you choose "2" as colspan, the column will span over two columns of the table. The last parameter defines the encoding of the content, which should be supplied, if the content is neither ASCII nor UTF-8. To append the column to the row, you simply call `appendColumn` in your row object with the column object as parameter. Alternatively you can directly give a string to the `appendColumn` method.

To finally render the table, you can either use the `render` method of the table, or use the magic method `__toString` by doing `echo $table;` or `$tableString = (string) $table`.

Example 877. Using Zend_Text_Table

This example illustrates the basic use of `Zend_Text_Table` to create a simple table:

```
$table = new Zend_Text_Table(array('columnWidths' => array(10, 20)));

// Either simple
$table->appendRow(array('Zend', 'Framework'));

// Or verbose
$row = new Zend_Text_Table_Row();

$row->appendColumn(new Zend_Text_Table_Column('Zend'));
$row->appendColumn(new Zend_Text_Table_Column('Framework'));

$table->appendRow($row);

echo $table;
```

This will result in the following output:

```
##################################
#Zend      #Framework           #
##################################
```

Zend_TimeSync

1. Introduction

`Zend_TimeSync` is able to receive internet or network time from a time server using the *NTP* or *SNTP* protocol. With `Zend_TimeSync`, Zend Framework is able to act independently from the time settings of the server where it is running.

To be independent from the actual time of the server, `Zend_TimeSync` works with the difference of the real time which is sent through NTP or SNTP and the internal server's time.

Background

`Zend_TimeSync` is not able to change the server's time, but it will return a Zend_Date instance from which the difference from the server's time can be worked with.

1.1. Why Zend_TimeSync ?

So why would someone use `Zend_TimeSync` ?

Normally every server within a multi-server farm will have a service running which synchronizes its own time with a time server. So within a standard environment it should not be necessary to use `Zend_TimeSync`. But it can become handy if there is no service available and if you don't have the right to install such a service.

Here are some example use cases, for which `Zend_TimeSync` is perfect suited:

- *Server without time service*

  If your application is running on a server and this server does not have any time service running, it may make sense to use `Zend_TimeSync` in your application.

- *Separate database server*

  If your database is running on a different server and this server is not connected with *NTP* or *SNTP* to the application server, you might have problems using storing and using time stamp data.

- *Multiple servers*

  If your application is running on more than one server and these servers' time bases are not syncronized, you can expect problems within your application when part of the application is coming from one server and another part from another server.

- *Batch processing*

  If you want to work with a time service within a batch file or within a command line application, `Zend_TimeSync` may be of use.

`Zend_TimeSync` may provide a good solution in all of these cases and can be used if you are unable to run any services on your server.

1.2. What is NTP ?

The Network Time Protocol (*NTP*) is a protocol for synchronizing multiple systems' clocks over packet-switched, variable-latency data networks. NTP uses UDP port 123 as its transport layer. See the wikipedia article for details about this protocol.

1.3. What is SNTP?

The Simple Network Time Protocol (*SNTP*) is a protocol synchronizing multiple systems' clocks over packet-switched, variable-latency data networks. SNTP uses UDP port 37 as its transport layer. It is closely related to the Network Time Protocol, but simpler.

1.4. Problematic usage

Be warned that when you are using `Zend_TimeSync` you will have to think about some details related to the structure of time sync and the internet itself. Correct usage and best practices will be described here. Read carefully before you begin using `Zend_TimeSync`.

1.5. Decide which server to use

You should select the time server that you want to use very carefully according to the following criteria:

- Distance

  The distance from your application server to the time server. If your server is in Europe, it would make little sense to select a time server in Tahiti. Always select a server which is not far away. This reduces the request time and overall network load.

- Speed

  How long it takes to receive the request is also relevant. Try different servers to get the best result. If you are requesting a server which is never accessible, you will always have an unnecessary delay.

- Splitting

  Do not always use the same server. All time servers will lock out requests from servers that are flooding the server. If your application requires heavy use of time servers, you should consider one of the pools described later.

So where can you find a time server? Generally you can use any timeserver you can connect to. This can be a time server within your LAN or any public time server you have access to. If you decide to use a public time server, you should consider using a server pool. Server pools are public addresses from which you will get a random, pooled time server by requesting the time. This way you will not have to split your requests. There are public server pools available for many regions which you may use to avoid problems mentioned above.

See pool.ntp.org to find your nearest server pool. For example, if your server is located within Germany you can connect to *0.europe.pool.ntp.org*.

2. Working with Zend_TimeSync

`Zend_TimeSync` can return the actual time from any given *NTP* or *SNTP* time server. It can automatically handle multiple servers and provides a simple interface.

All examples in this chapter use a public, generic time server: *0.europe.pool.ntp.org*. You should use a public, generic time server which is close to your application server. See http://www.pool.ntp.org for information.

2.1. Generic Time Server Request

Requesting the time from a time server is simple. First, you provide the time server from which you want to request the time.

```
$server = new Zend_TimeSync('0.pool.ntp.org');

print $server->getDate()->getIso();
```

So what is happening in the background of `Zend_TimeSync`? First the syntax of the time server is checked. In our example, '*0.pool.ntp.org*' is checked and recognised as a possible address for a time server. Then when calling `getDate()` the actual set time server is requested and it will return its own time. `Zend_TimeSync` then calculates the difference to the actual time of the server running the script and returns a `Zend_Date` object with the correct time.

For details about `Zend_Date` and its methods see the `Zend_Date` documentation.

2.2. Multiple Time Servers

Not all time servers are always available to return their time. Servers may be unavailable during maintenance, for example. When the time cannot be requested from the time server, you will get an exception.

`Zend_TimeSync` is a simple solution that can handle multiple time servers and supports an automatic fallback mechanism. There are two supported ways; you can either specify an array of time servers when creating the instance, or you can add additional time servers to the instance using the `addServer()` method.

```
$server = new Zend_TimeSync(array('0.pool.ntp.org',
                                  '1.pool.ntp.org',
                                  '2.pool.ntp.org'));
$server->addServer('3.pool.ntp.org');

print $server->getDate()->getIso();
```

There is no limit to the number of time servers you can add. When a time server can not be reached, `Zend_TimeSync` will fallback and try to connect to the next time server.

When you supply more than one time server- which is considered a best practice for `Zend_TimeSync`- you should name each server. You can name your servers with array keys, with the second parameter at instantiation, or with the second parameter when adding another time server.

```
$server = new Zend_TimeSync(array('generic'  => '0.pool.ntp.org',
                                  'fallback' => '1.pool.ntp.org',
                                  'reserve'  => '2.pool.ntp.org'));
$server->addServer('3.pool.ntp.org', 'additional');

print $server->getDate()->getIso();
```

Naming the time servers allows you to request a specific time server as we will see later in this chapter.

2.3. Protocols of Time Servers

There are different types of time servers. Most public time servers use the *NTP* protocol. But there are other time synchronization protocols available.

You set the proper protocol in the address of the time server. There are two protocols which are supported by `Zend_TimeSync`: *NTP* and *SNTP*. The default protocol is *NTP*. If you are using *NTP*, you can omit the protocol in the address as demonstrated in the previous examples.

```
$server = new Zend_TimeSync(array('generic'  => 'ntp:\\0.pool.ntp.org',
                                  'fallback' => 'ntp:\\1.pool.ntp.org',
                                  'reserve'  => 'ntp:\\2.pool.ntp.org'));
$server->addServer('sntp:\\internal.myserver.com', 'additional');

print $server->getDate()->getIso();
```

`Zend_TimeSync` can handle mixed time servers. So you are not restricted to only one protocol; you can add any server independently from its protocol.

2.4. Using Ports for Time Servers

As with every protocol within the world wide web, the *NTP* and *SNTP* protocols use standard ports. NTP uses port *123* and SNTP uses port *37*.

But sometimes the port that the protocols use differs from the standard one. You can define the port which has to be used for each server within the address. Just add the number of the port after the address. If no port is defined, then `Zend_TimeSync` will use the standard port.

```
$server = new Zend_TimeSync(array('generic'  => 'ntp:\\0.pool.ntp.org:200',
                                  'fallback' => 'ntp:\\1.pool.ntp.org'));
$server->addServer('sntp:\\internal.myserver.com:399', 'additional');

print $server->getDate()->getIso();
```

2.5. Time Servers Options

There is only one option within `Zend_TimeSync` which will be used internally: *timeout*. You can set any self-defined option you are in need of and request it, however.

The option *timeout* defines the number of seconds after which a connection is detected as broken when there was no response. The default value is *1*, which means that `Zend_TimeSync` will fallback to the next time server if the requested time server does not respond in one second.

With the `setOptions()` method, you can set any option. This function accepts an array where the key is the option to set and the value is the value of that option. Any previously set option will be overwritten by the new value. If you want to know which options are set, use the `getOptions()` method. It accepts either a key which returns the given option if specified, or, if no key is set, it will return all set options.

```
Zend_TimeSync::setOptions(array('timeout' => 3, 'myoption' => 'timesync'));
$server = new Zend_TimeSync(array('generic'  => 'ntp:\\0.pool.ntp.org',
                                  'fallback' => 'ntp:\\1.pool.ntp.org'));
$server->addServer('sntp:\\internal.myserver.com', 'additional');
```

```
print $server->getDate()->getIso();
print_r(Zend_TimeSync::getOptions();
print "Timeout = " . Zend_TimeSync::getOptions('timeout');
```

As you can see, the options for `Zend_TimeSync` are static. Each instance of `Zend_TimeSync` will use the same options.

2.6. Using Different Time Servers

`Zend_TimeSync`'s default behavior for requesting a time is to request it from the first given server. But sometimes it is useful to set a different time server from which to request the time. This can be done with the `setServer()` method. To define the used time server set the alias as a parameter within the method. To get the actual used time server call the `getServer()` method. It accepts an alias as a parameter which defines the time server to be returned. If no parameter is given, the current time server will be returned.

```
$server = new Zend_TimeSync(array('generic'  => 'ntp:\\0.pool.ntp.org',
                                  'fallback' => 'ntp:\\1.pool.ntp.org'));
$server->addServer('sntp:\\internal.myserver.com', 'additional');

$actual = $server->getServer();
$server = $server->setServer('additional');
```

2.7. Information from Time Servers

Time servers not only offer the time itself, but also additional information. You can get this information with the `getInfo()` method.

```
$server = new Zend_TimeSync(array('generic'  => 'ntp:\\0.pool.ntp.org',
                                  'fallback' => 'ntp:\\1.pool.ntp.org'));

print_r ($server->getInfo());
```

The returned information differs with the protocol used and can also differ with the server used.

2.8. Handling Exceptions

Exceptions are collected for all time servers and returned as an array. So you can iterate through all thrown exceptions as shown in the following example:

```
$serverlist = array(
        // invalid servers
        'invalid_a'  => 'ntp://a.foo.bar.org',
        'invalid_b'  => 'sntp://b.foo.bar.org',
);

$server = new Zend_TimeSync($serverlist);

try {
    $result = $server->getDate();
    echo $result->getIso();
} catch (Zend_TimeSync_Exception $e) {

    $exceptions = $e->get();
```

```
    foreach ($exceptions as $key => $myException) {
        echo $myException->getMessage();
        echo '<br />';
    }
}
```

Zend_Tool

1. Using Zend_Tool On The Command Line

The CLI, or command line tool (internally known as the console tool), is currently the primary interface for dispatching `Zend_Tool` requests. With the CLI tool, developers can issue tooling requests inside the "command line window", also commonly known as a "terminal" window. This environment is predominant in the *nix environment, but also has a common implementation in windows with the `cmd.exe`, console2 and also with the Cygwin project.

1.1. Installation

1.1.1. Download And Go

First download Zend Framework. This can be done by going to framework.zend.com and downloading the latest release. After you've downloaded the package and placed it on your system. The next step is to make the zf command available to your system. The easiest way to do this, is to copy the proper files from the bin/ directory of the download, and place these files within the *same* directory as the location of the php cli binary.

1.1.2. Installing Via Pear

To install via PEAR, you must use the 3rd party zfcampus.org site to retrieve the latest Zend Framework PEAR package. These packages are typically built within a day of an official Zend Framework release. The benefit of installing via the PEAR package manager is that during the install process, the ZF library will end up on the include_path, and the zf.php and zf scripts will end up in a place on your system that will allow you to run them without any additional setup.

```
pear discover-channel pear.zfcampus.org
pear install zfcampus/zf
```

That is it. After the initial install, you should be able to continue on by running the zf command. Go good way to check to see if it't there is to run zf --help

1.1.3. Installing by Hand

Installing by hand refers to the process of forcing the zf.php and Zend Framework library to work together when they are placed in non-convential places, or at least, in a place that your system cannot dispatch from easily (typical of programs in your system PATH).

If you are on a *nix or mac system, you can also create a link from somewhere in your path to the zf.sh file. If you do this, you do not need to worry about having Zend Framework's library on your include_path, as the zf.php and zf.sh files will be able to access the library relative to where they are (meaning the ./bin/ files are ../library/ relative to the Zend Framework library).

There are a number of other options available for setting up the zf.php and library on your system. These options revolve around setting specific environment variables. These are described in the later section on "customizing the CLI environement". The environment variables for setting the zf.php include_path, ZF_INCLUDE_PATH and ZF_INCLUDE_PATH_PREPEND, are the ones of most interest.

1.2. General Purpose Commands

1.2.1. Version

This will show the current version number of the copy of Zend Framework the zf.php tool is using.

```
zf show version
```

1.2.2. Built-in Help

The built-in help system is the primary place where you can get up-to-date information on what your system is capable of doing. The help system is dynamic in that as providers are added to your system, they are automatically dispatchable, and as such, the parameters required to run them will be in the help screen. The easiest way to retrieve the help screen is the following:

```
zf --help
```

This will give you an overview of the various capabilities of the system. Sometimes, there are more finite commands than can be run, and to gain more information about these, you might have to run a more specialized help command. For specialized help, simply replace any of the elements of the command with a "?". This will tell the help system that you want more information about what commands can go in place of the question mark. For example:

```
zf ? controller
```

The above means "show me all 'actions' for the provider 'controller'"; while the following:

```
zf show ?
```

means "show me all providers that support the 'show' action". This works for drilling down into options as well as you can see in the following examples:

```
zf show version.? (show any specialties)
zf show version ? (show any options)
```

1.2.3. Manifest

This will show what information is in the tooling systems manifest. This is more important for provider developers than casual users of the tooling system.

```
zf show manifest
```

1.3. Project Specific Commands

1.3.1. Project

The project provider is the first command you might want to run. This will setup the basic structure of your application. This is required before any of the other providers can be executed.

```
zf create project MyProjectName
```

This will create a project in a directory called ./MyProjectName. From this point on, it is important to note that any subsequent commands on the command line must be issued from within the project directory you had just created. So, after creation, changing into that directory is required.

1.3.2. Project

The module provider allows for the easy creation of a Zend Framework module. A module follows the hMVC pattern loosely. When creating modules, it will take the same structure used at the application/ level, and duplicate it inside of the chosen name for your module, inside of the "modules" directory of the application/ directory without duplicating the modules directory itself. For example:

```
zf create module Blog
```

This will create a module named Blog at application/modules/Blog, and all of the artifacts that a module will need.

1.3.3. Controller

The controller provider is responsible for creating (mostly) empty controllers as well as their corresponding view script directories and files. To utilize it to create an 'Auth' controlller, for example, execute:

```
zf create controller Auth
```

This will create a controller named Auth, specifically it will create a file at application/controllers/AuthController.php with the AuthController inside. If you wish to create a controller for a module, use any of the following:

```
zf create controller Post 1 Blog
zf create controller Post -m Blog
zf create controller Post --module=Blog
```

Note: In the first command, 1 is the value for the "includeIndexAction" flag.

1.3.4. Action

To create an action within an existing controller:

```
zf create action login Auth
zf create action login -c Auth
zf create action login --controller-name=Auth
```

1.3.5. View

To create a view outside of the normal controller/action creation, you would use one of the following:

```
zf create view Auth my-script-name
zf create view -c Auth -a my-script-name
```

This will create a view script in the controller folder of Auth.

1.3.6. Model

The model provider is only responsible for creating the proper model files, with the proper name inside the application folder. For example

```
zf create model User
```

If you wish to create a model within a specific module:

```
zf create model Post -m Blog
```

The above will create a 'Post' model inside of the 'Blog' module.

1.3.7. Form

The form provider is only responsible for creating the proper form file and init() method, with the proper name inside the application folder. For example:

```
zf create form Auth
```

If you wish to create a model within a specific module:

```
zf create form Comment -m Blog
```

The above will create a 'Comment' form inside of the 'Blog' module.

1.3.8. DbAdapter

To configure a DbAdapter, you will need to provide the information as a url encoded string. This string needs to be in quotes on the command line.

For example, to enter the following information:

- adapter: Pdo_Mysql
- username: test
- password: test
- dbname: test

The following will have to be run on the command line:

```
zf configure dbadapter "adapter=Pdo_Mysql&username=test&password=test&dbname=test"
```

This assumes you wish to store this information inside of the 'production' space of the application configuration file. The following will demonstrate an sqlite configuration, in the 'development' section of the application config file.

```
zf configure dbadapter "adapter=Pdo_Sqlite&dbname=../data/test.db" development
zf configure dbadapter "adapter=Pdo_Sqlite&dbname=../data/test.db" -s development
```

1.3.9. DbTable

The DbTable provider is responsible for creating Zend_Db_Table model/data access files for your application to consume, with the proper class name, and in the proper location in the application. The two important pieces of information are the *DbTable name*, and the *actual database table name*. For example:

```
zf create dbtable User user
zf create dbtable User -a user

// also accepts a force option to overwrite existing files
zf create dbtable User user -f
zf create dbtable User user --force-override
```

The DbTable provider is also capable of creating the proper files by scanning the database configured with the above DbAdapter provider.

```
zf create dbtable.from-database
```

When executing the above, it might make sense to use the pretend / "-p" flag first so that you can see what would be done, and what tables can be found in the database.

```
zf -p create dbtable.from-database
```

1.3.10. Layout

Currently, the only supported action for layouts is simply to enable them will setup the proper keys in the application.ini file for the application resource to work, and create the proper directories and layout.phtml file.

```
zf enable layout
```

1.4. Environment Customization

1.4.1. The Storage Directory

The storage directory is important so that providers may have a place to find custom user generated logic that might change the way they behave. One example can be found below is the placement of a custom project profile file.

```
zf --setup storage-directory
```

1.4.2. The Configuration File

This will create the proper zf.ini file. This *should* be run after `zf --setup storage-directory`. If it is not, it will be located inside the users home directory. If it is, it will be located inside the users storage directory.

```
zf --setup config-file
```

1.4.3. Environment Locations

These should be set if you wish to override the default places where zf will attempt to read their values.

- ZF_HOME
  - the directory this tool will look for a home directory
  - directory must exist

    - search order:
        - ZF_HOME environment variable
        - HOME environment variable
        - then HOMEPATH environment variable
- ZF_STORAGE_DIRECTORY
    - where this tool will look for a storage directory
    - directory must exist
    - search order:
        - ZF_STORAGE_DIRECTORY environment variable
        - $homeDirectory/.zf/ directory
- ZF_CONFIG_FILE
    - where this tool will look for a configuration file
    - search order:
        - ZF_CONFIG_FILE environment variable
        - $homeDirectory/.zf.ini file if it exists
        - $storageDirectory/zf.ini file if it exists
- ZF_INCLUDE_PATH
    - set the include_path for this tool to use this value
    - original behavior:
        - use php's include_path to find ZF
        - use the ZF_INCLUDE_PATH environment variable
        - use the path ../library (relative to zf.php) to find ZF
- ZF_INCLUDE_PATH_PREPEND
    - prepend the current php.ini include_path with this value

2. Extending Zend_Tool

2.1. Overview of Zend_Tool

`Zend_Tool_Framework` is a framework for exposing common functionalities such as the creation of project scaffolds, code generation, search index generation, and much more. Functionality may be written and exposed via PHP classes dropped into the PHP include_path, providing incredible flexibility of implementation. The functionality may then be consumed by

writing implementation and/or protocol-specific clients -- such as console clients, XML-RPC, SOAP, and much more.

`Zend_Tool_Project` builds on and extends the capabilities of `Zend_Tool_Framework` to that of managing a "project". In general, a "project" is a planned endeavor or an initiative. In the computer world, projects generally are a collection of resources. These resources can be files, directories, databases, schemas, images, styles, and more.

2.2. Zend_Tool_Framework Extensions

2.2.1. Overall Architecture

`Zend_Tool_Framework` provides the following:

- *Common interfaces and abstracts* that allow developers to create functionality and capabilities that are dispatchable by tooling clients.
- *Base client functionality* and a concrete console implementation that connect external tools and interfaces to the `Zend_Tool_Framework`. The Console client may be used in CLI environments such as unix shells and the Windows console.
- *"Provider" and "Manifest" interfaces* that can be utilized by the tooling system. "Providers" represent the functional aspect of the framework, and define the actions that tooling clients may call. "Manifests" act as metadata registries that provide additional context for the various defined providers.
- *An introspective loading system* that will scan the environment for providers and determine what is required to dispatch them.
- *A standard set of system providers* that allow the system to report what the full capabilities of the system are as well as provide useful feedback. This also includes a comprehensive "Help System".

Definitions that you should be aware of through this manual with respect to `Zend_Tool_Framework` include:

- `Zend_Tool_Framework` - The framework which exposes tooling capabilities.
- *Tooling Client* - A developer tool that connects to and consumes `Zend_Tool_Framework`.
- *Client* - The subsystem of `Zend_Tool_Framework` that exposes an interface such that tooling clients can connect, query and execute commands.
- *Console Client / Command Line Interface / `zf.php`* - The tooling client for the command line.
- *Provider* - A subsystem and a collection of built-in functionality that the framework exports.
- *Manifest* - A subsystem for defining, organizing, and disseminating provider requirement data.
- `Zend_Tool_Project` Provider - A set of providers specifically for creating and maintaining Zend Framework-based projects.

2.2.2. Understanding the CLI Client

The CLI, or command line tool (internally known as the console tool), is currently the primary interface for dispatching `Zend_Tool` requests. With the CLI tool, developers can issue tooling

requests inside the "command line windows", also commonly known as a "terminal" window. This environment is predominant in the *nix environment, but also has a common implementation in windows with the `cmd.exe`, console2 and also with the Cygwin project.

2.2.2.1. Setting up the CLI tool

To issue tooling requests via the command line client, you first need to set up the client so that your system can handle the "zf" command. The command line client, for all intents and purposes, is the `.sh` or `.bat` file that is provided with your Zend Framework distribution. In trunk, it can be found here: http://framework.zend.com/svn/framework/standard/trunk/bin/.

As you can see, there are 3 files in the `/bin/` directory: a `zf.php`, `zf.sh`, and `zf.bat`. The `zf.sh` and the `zf.bat` are the operating system specific client wrappers: `zf.sh` for the *nix environment, and `zf.bat` for the Win32 environment. These client wrappers are responsible for finding the proper `php.exe`, finding the `zf.php`, and passing on the client request. The `zf.php` is the responsible for handling understanding your environment, constructing the proper include_path, and passing what is provided on the command line to the proper library component for dispatching.

Ultimately, you want to ensure two things to make everything work regardless of the operating system you are on:

1. `zf.sh/zf.bat` is reachable from your system path. This is the ability to call **zf** from anywhere on your command line, regardless of what your current working directory is.

2. `ZendFramework/library` is in your include_path.

Note: while the above are the most ideal requirements, you can simply download Zend Framework and expect it to work as `./path/to/zf.php` some command.

2.2.2.2. Setting up the CLI tool on Unix-like Systems

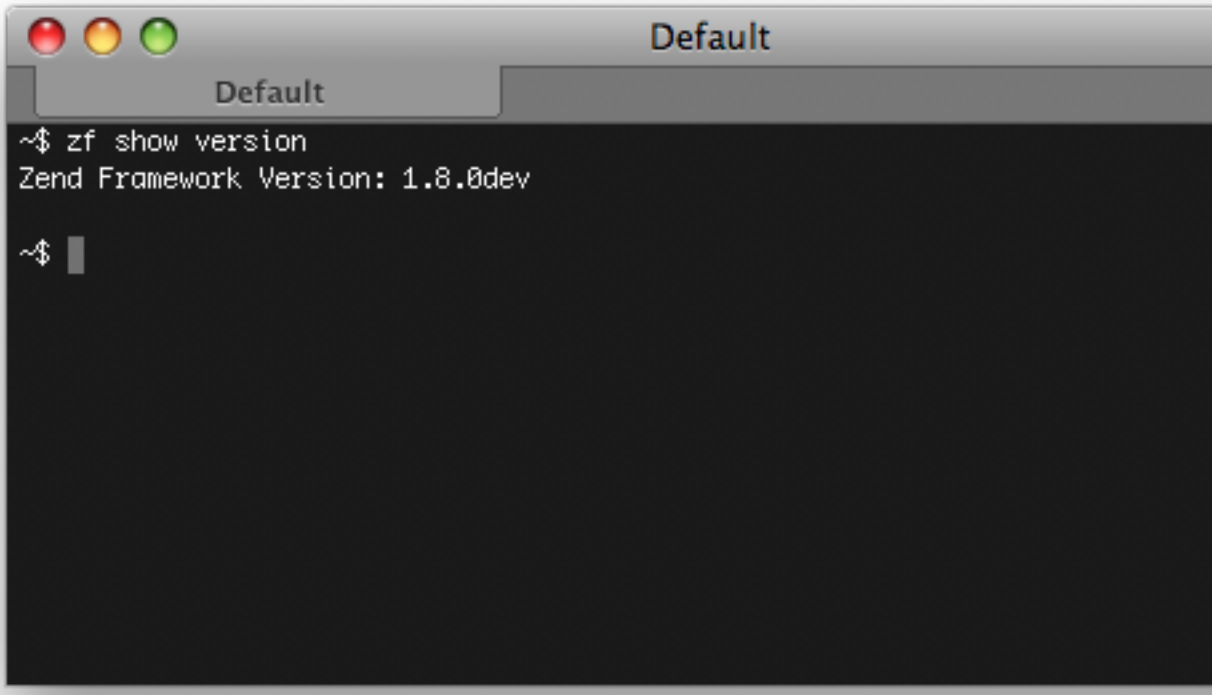
The most common setup in the *nix environment, is to copy the `zf.sh` and `zf.php` into the same directory as your PHP binary. This can generally be found in one of the following places:

```
/usr/bin
/usr/local/bin
/usr/local/ZendServer/bin/
/Applications/ZendServer/bin/
```

To find out the location of your PHP binary, you can execute 'which php' on the command line. This will return the location of the PHP binary you will be using to run PHP scripts in this environment.

The next order of business is to ensure that Zend Framework library is set up correctly inside of the system PHP include_path. To find out where your include_path is located, you can execute **php -i** and look for the include_path variable, or more succinctly, execute **php -i | grep include_path**. Once you have found where your include_path is located (this will generally be something like `/usr/lib/php`, `/usr/share/php`, `/usr/local/lib/php`, or similar), ensure that the contents of the `/library/` directory are put inside your include_path specified directory.

Once you have done those two things, you should be able to issue a command and get back the proper response like this:



If you do not see this type of output, go back and check your setup to ensure you have all of the necessary pieces in the proper place.

There are a couple of alternative setups you might want to employ depending on your servers configuration, your level of access, or for other reasons.

Alternative Setup involves keeping the Zend Framework download together as is, and creating a link from a PATH location to the `zf.sh`. What this means is you can place the contents of the ZendFramework download into a location such as `/usr/local/share/ZendFramework`, or more locally like `/home/username/lib/ZendFramework`, and creating a symbolic link to the `zf.sh`.

Assuming you want to put the link inside `/usr/local/bin` (this could also work for placing the link inside `/home/username/bin/` for example) you would issue a command similar to this:

```
ln -s /usr/local/share/ZendFramework/bin/zf.sh /usr/local/bin/zf

# OR (for example)
ln -s /home/username/lib/ZendFramework/bin/zf.sh /home/username/bin/zf
```

This will create a link which you should be able to access globally on the command line.

2.2.2.3. Setting up the CLI tool on Windows

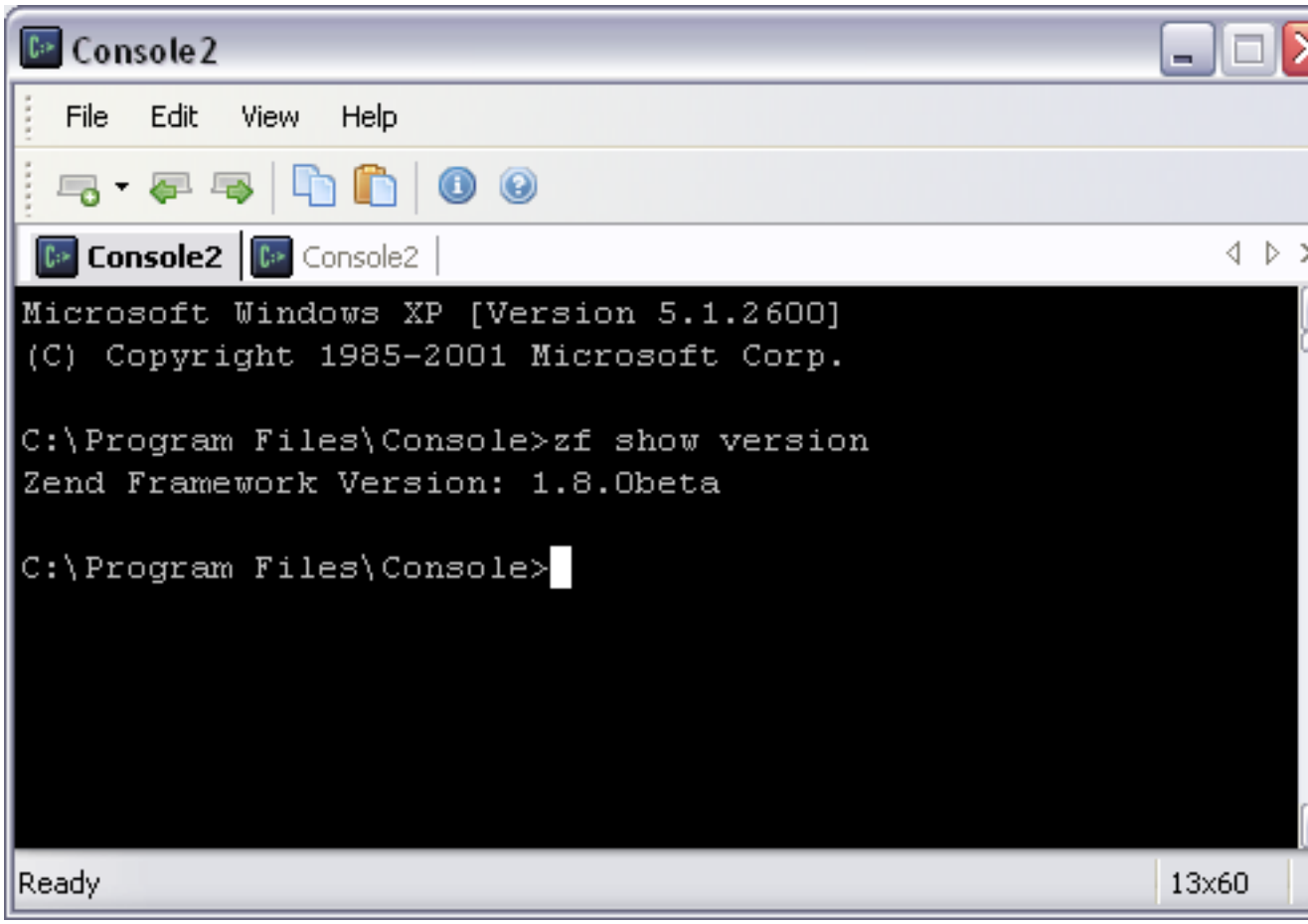
The most common setup in the Windows Win32 environment, is to copy the `zf.bat` and `zf.php` into the same directory as your PHP binary. This can generally be found in one of the following places:

```
C:\PHP
C:\Program Files\ZendServer\bin\
C:\WAMP\PHP\bin
```

You should be able to run `php.exe` on the command line. If you are not able to, first check the documentation that came with your PHP distribution, or ensure that the path to `php.exe` is in your Windows PATH environment variable.

The next order of business is to ensure that Zend Framework library is set up correctly inside of the system PHP include_path. To find out where your include_path is located, you can type **php -i** and look for the include_path variable, or more succinctly execute **php -i | grep include_path** if you have Cygwin setup with grep available. Once you have found where your include_path is located (this will generally be something like `C:\PHP\pear`, `C:\PHP\share`, `C:\Program %20Files\ZendServer\share` or similar), ensure that the contents of the library/ directory are put inside your include_path specified directory.

Once you have done those two things, you should be able to issue a command and get back the proper response like this:



If you do not see this type of output, go back and check your setup to ensure you have all of the necessary pieces in the proper place.

There are a couple of alternative setups you might want to employ depending on your server's configuration, your level of access, or for other reasons.

Alternative Setup involves keeping the Zend Framework download together as is, and altering both your system PATH as well as the `php.ini` file. In your user's environment, make sure to add `C:\Path\To\ZendFramework\bin`, so that your `zf.bat` file is executable. Also, alter the `php.ini` file to ensure that `C:\Path\To\ZendFramework\library` is in your include_path.

2.2.2.4. Other Setup Considerations

If for some reason you do not want Zend Framework library inside your include_path, there is another option. There are two special environment variables that `zf.php` will utilize to determine the location of your Zend Framework installation.

The first is `ZEND_TOOL_INCLUDE_PATH_PREPEND`, which will prepend the value of this environment variable to the system (`php.ini`) include_path before loading the client.

Alternatively, you might want to use `ZEND_TOOL_INCLUDE_PATH` to completely *replace* the system include_path for one that makes sense specifically for the **zf** command line tool.

2.2.3. Creating Providers

In general, a provider, on its own, is nothing more than the shell for a developer to bundle up some capabilities they wish to dispatch with the command line (or other) clients. It is an analogue to what a "controller" is inside of your MVC application.

2.2.3.1. How Zend Tool finds your Providers

By default Zend Tool uses the BasicLoader to find all the providers that you can run. It recursivly iterates all include path directories and opens all files that end with "Manifest.php" or "Provider.php". All classes in these files are inspected if they implement either `Zend_Tool_Framework_Provider_Interface` or `Zend_Tool_Framework_Manifest_ProviderManifestable`. Instances of the provider interface make up for the real functionality and all their public methods are accessible as provider actions. The ProviderManifestable interface however requires the implementation of a method `getProviders()` which returns an array of instantiated provider interface instances.

The following naming rules apply on how you can access the providers that were found by the IncludePathLoader:

- The last part of your classname split by underscore is used for the provider name, e.g. "My_Provider_Hello" leads to your provider being accessible by the name "hello".
- If your provider has a method `getName()` it will be used instead of the previous method to determine the name.
- If your provider has "Provider" as prefix, e.g. it is called `My_HelloProvider` it will be stripped from the name so that the provider will be called "hello".

The IncludePathLoader does not follow symlinks, that means you cannot link provider functionality into your include paths, they have to be physically present in the include paths.

Example 878. Exposing Your Providers with a Manifest

You can expose your providers to Zend Tool by offering a manifest with a special filename ending with "Manifest.php". A Provider Manifest is an implementation of the Zend_Tool_Framework_Manifest_ProviderManifestable and requires the `getProviders()` method to return an array of instantiated providers. In anticipation of our first own provider `My_Component_HelloProvider` we will create the following manifest:

```
class My_Component_Manifest
    implements Zend_Tool_Framework_Manifest_ProviderManifestable
{
    public function getProviders()
    {
        return array(
            new My_Component_HelloProvider()
        );
    }
}
```

2.2.3.2. Basic Instructions for Creating Providers

As an example, if a developer wants to add the capability of showing the version of a datafile that his 3rd party component is working from, there is only one class the developer would need to implement. Assuming the component is called `My_Component`, he would create a class named `My_Component_HelloProvider` in a file named `HelloProvider.php` somewhere on the include_path. This class would implement `Zend_Tool_Framework_Provider_Interface`, and the body of this file would only have to look like the following:

```
class My_Component_HelloProvider
    implements Zend_Tool_Framework_Provider_Interface
{
    public function say()
    {
        echo 'Hello from my provider!';
    }
}
```

Given that code above, and assuming the developer wishes to access this functionality through the console client, the call would look like this:

```
% zf say hello
Hello from my provider!
```

2.2.3.3. The response object

As discussed in the architecture section Zend Tool allows to hook different clients for using your Zend Tool providers. To keep compliant with different clients you should use the response object to return messages from your providers instead of using `echo()` or a similiar output mechanism. Rewritting our hello provider with this knowledge it looks like:

```
class My_Component_HelloProvider
    extends Zend_Tool_Framework_Provider_Abstract
{
    public function say()
```

```
    {
        $this->_registry->getResponse
                        ->appendContent("Hello from my provider!");
    }
}
```

As you can see one has to extend the `Zend_Tool_Framework_Provider_Abstract` to gain access to the Registry which holds the `Zend_Tool_Framework_Client_Response` instance.

2.2.3.4. Advanced Development Information

2.2.3.4.1. Passing Variables to a Provider

The above "Hello World" example is great for simple commands, but what about something more advanced? As your scripting and tooling needs grow, you might find that you need the ability to accept variables. Much like function signatures have parameters, your tooling requests can also accept parameters.

Just as each tooling request can be isolated to a method within a class, the parameters of a tooling request can also be isolated in a very well known place. Parameters of the action methods of a provider can include the same parameters you want your client to utilize when calling that provider and action combination. For example, if you wanted to accept a name in the above example, you would probably do this in OO code:

```
class My_Component_HelloProvider
    implements Zend_Tool_Framework_Provider_Interface
{
    public function say($name = 'Ralph')
    {
        echo 'Hello' . $name . ', from my provider!';
    }
}
```

The above example can then be called via the command line **zf say hello Joe**. "Joe" will be supplied to the provider as a parameter of the method call. Also note, as you see that the parameter is optional, that means it is also optional on the command line, so that **zf say hello** will still work, and default to the name "Ralph".

2.2.3.4.2. Prompt the User for Input

There are cases when the workflow of your provider requires to prompt the user for input. This can be done by requesting the client to ask for more the required input by calling:

```
class My_Component_HelloProvider
    extends Zend_Tool_Framework_Provider_Abstract
{
    public function say($name = 'Ralph')
    {
        $nameResponse = $this->_registry
                             ->getClient()
                             ->promptInteractiveInput("Whats your name?");
        $name = $name->getContent();

        echo 'Hello' . $name . ', from my provider!';
    }
}
```

This command throws an exception if the current client is not able to handle interactive requests. In case of the default Console Client however you will be asked to enter the name.

2.2.3.4.3. Pretending to execute a Provider Action

Another interesting feature you might wish to implement is *pretendability*. Pretendabilty is the ability for your provider to "pretend" as if it is doing the requested action and provider combination and give the user as much information about what it *would* do without actually doing it. This might be an important notion when doing heavy database or filesystem modifications that the user might not otherwise want to do.

Pretendability is easy to implement. There are two parts to this feature: 1) marking the provider as having the ability to "pretend", and 2) checking the request to ensure the current request was indeed asked to be "pretended". This feature is demonstrated in the code sample below.

```
class My_Component_HelloProvider
    extends    Zend_Tool_Framework_Provider_Abstract
    implements Zend_Tool_Framework_Provider_Pretendable
{
    public function say($name = 'Ralph')
    {
        if ($this->_registry->getRequest()->isPretend()) {
            echo 'I would say hello to ' . $name . '.';
        } else {
            echo 'Hello' . $name . ', from my provider!';
        }
    }
}
```

To run the provider in pretend mode just call:

```
% zf --pretend say hello Ralph
I would say hello Ralph.
```

2.2.3.4.4. Verbose and Debug modes

You can also run your provider actions in "verbose" or "debug" modes. The semantics in regard to this actions have to be implemented by you in the context of your provider. You can access debug or verbose modes with:

```
class My_Component_HelloProvider
    implements Zend_Tool_Framework_Provider_Interface
{
    public function say($name = 'Ralph')
    {
        if($this->_registry->getRequest()->isVerbose()) {
            echo "Hello::say has been called\n";
        }
        if($this->_registry->getRequest()->isDebug()) {
            syslog(LOG_INFO, "Hello::say has been called\n");
        }
    }
}
```

2.2.3.4.5. Accessing User Config and Storage

Using the Enviroment variable ZF_CONFIG_FILE or the .zf.ini in your home directory you can inject configuration parameters into any Zend Tool provider. Access to this

configuration is available via the registry that is passed to your provider if you extend `Zend_Tool_Framework_Provider_Abstract`.

```
class My_Component_HelloProvider
    extends Zend_Tool_Framework_Provider_Abstract
{
    public function say()
    {
        $username = $this->_registry->getConfig()->username;
        if(!empty($username)) {
            echo "Hello $username!";
        } else {
            echo "Hello!";
        }
    }
}
```

The returned configuration is of the type `Zend_Tool_Framework_Client_Config` but internally the `__get()` and `__set()` magic methods proxy to a `Zend_Config` of the given configuration type.

The storage allows to save arbitrary data for later reference. This can be useful for batch processing tasks or for re-runs of your tasks. You can access the storage in a similar way like the configuration:

```
class My_Component_HelloProvider
    extends Zend_Tool_Framework_Provider_Abstract
{
    public function say()
    {
        $aValue = $this->_registry->getStorage()->get("myUsername");
        echo "Hello $aValue!";
    }
}
```

The API of the storage is very simple:

```
class Zend_Tool_Framework_Client_Storage
{
    public function setAdapter($adapter);
    public function isEnabled();
    public function put($name, $value);
    public function get($name, $defaultValue=null);
    public function has($name);
    public function remove($name);
    public function getStreamUri($name);
}
```

When designing your providers that are config or storage aware remember to check if the required user-config or storage keys really exist for a user. You won't run into fatal errors when none of these are provided though, since empty ones are created upon request.

2.3. Zend_Tool_Project Extensions

Zend_Tool_Project exposes a rich set of functionality and capabilities that make the task of creating new providers, specficially those targetting project easier and more manageable.

2.3.1. Overall Architecture

This same concept applies to Zend Framework projects. In Zend Framework projects, you have controllers, actions, views, models, databases and so on and so forth. In terms of `Zend_Tool`, we need a way to track these types of resources - thus `Zend_Tool_Project`.

`Zend_Tool_Project` is capable of tracking project resources throughout the development of a project. So, for example, if in one command you created a controller, and in the next command you wish to create an action within that controller, `Zend_Tool_Project` is gonna have to *know* about the controller file you created so that you can (in the next action), be able to append that action to it. This is what keeps our projects up to date and *stateful*.

Another important point to understand about projects is that typically, resources are organized in a hierarchical fashion. With that in mind, `Zend_Tool_Project` is capable of serializing the current project into a internal representation that allows it to keep track of not only *what* resources are part of a project at any given time, but also *where* they are in relation to one another.

2.3.2. Creating Providers

Project specific providers are created in the same fashion as plain framework providers, with one exception: project providers must extend the `Zend_Tool_Project_Provider_Abstract`. This class comes with some significant functionality that helps developers load existing project, obtian the profile object, and be able to search the profile, then later store any changes to the current project profile.

```
class My_Component_HelloProvider
    extends Zend_Tool_Project_Provider_Abstract
{
    public function say()
    {
        $profile = $this->_loadExistingProfile();

        /* ... do project stuff here */

        $this->_storeProfile();
    }
}
```

Zend_Tool_Framework

1. Introduction

`Zend_Tool_Framework` is a framework for exposing common functionalities such as the creation of project scaffolds, code generation, search index generation, and much more. Functionality may be written and exposed via PHP classes dropped into the PHP include_path, providing incredible flexibility of implementation. The functionality may then be consumed by writing implementation and/or protocol-specific clients -- such as console clients, XML-RPC, SOAP, and much more.

`Zend_Tool_Framework` provides the following:

- *Common interfaces and abstracts* that allow developers to create functionality and capabilities that are dispatchable by tooling clients.
- *Base client functionality* and a concrete console implementation that connect external tools and interfaces to the `Zend_Tool_Framework`. The Console client may be used in CLI environments such as unix shells and the Windows console.
- *"Provider" and "Manifest" interfaces* that can be utilized by the tooling system. "Providers" represent the functional aspect of the framework, and define the actions that tooling clients may call. "Manifests" act as metadata registries that provide additional context for the various defined providers.
- *An introspective loading system* that will scan the environment for providers and determine what is required to dispatch them.
- *A standard set of system providers* that allow the system to report what the full capabilities of the system are as well as provide useful feedback. This also includes a comprehensive "Help System".

Definitions that you should be aware of through this manual with respect to `Zend_Tool_Framework` include:

- `Zend_Tool_Framework` - The framework which exposes tooling capabilities.
- *Tooling Client* - A developer tool that connects to and consumes `Zend_Tool_Framework`.
- *Client* - The subsystem of `Zend_Tool_Framework` that exposes an interface such that tooling clients can connect, query and execute commands.
- *Console Client / Command Line Interface / `zf.php`* - The tooling client for the command line.
- *Provider* - A subsystem and a collection of built-in functionality that the framework exports.
- *Manifest* - A subsystem for defining, organizing, and disseminating provider requirement data.
- `Zend_Tool_Project` Provider - A set of providers specifically for creating and maintaining Zend Framework-based projects.

2. Using the CLI Tool

The CLI, or command line tool (internally known as the console tool), is currently the primary interface for dispatching `Zend_Tool` requests. With the CLI tool, developers can issue tooling

requests inside the "command line windows", also commonly known as a "terminal" window. This environment is predominant in the *nix environment, but also has a common implementation in windows with the `cmd.exe`, console2 and also with the Cygwin project.

2.1. Setting up the CLI tool

To issue tooling requests via the command line client, you first need to set up the client so that your system can handle the "zf" command. The command line client, for all intents and purposes, is the `.sh` or `.bat` file that is provided with your Zend Framework distribution. In trunk, it can be found here: http://framework.zend.com/svn/framework/standard/trunk/bin/.

As you can see, there are 3 files in the `/bin/` directory: a `zf.php`, `zf.sh`, and `zf.bat`. The `zf.sh` and the `zf.bat` are the operating system specific client wrappers: `zf.sh` for the *nix environment, and `zf.bat` for the Win32 environment. These client wrappers are responsible for finding the proper `php.exe`, finding the `zf.php`, and passing on the client request. The `zf.php` is the responsible for handling understanding your environment, constructing the proper include_path, and passing what is provided on the command line to the proper library component for dispatching.

Ultimately, you want to ensure two things to make everything work regardless of the operating system you are on:

1. `zf.sh/zf.bat` is reachable from your system path. This is the ability to call **zf** from anywhere on your command line, regardless of what your current working directory is.

2. `ZendFramework/library` is in your include_path.

Note: while the above are the most ideal requirements, you can simply download Zend Framework and expect it to work as `./path/to/zf.php` some command.

2.2. Setting up the CLI tool on Unix-like Systems

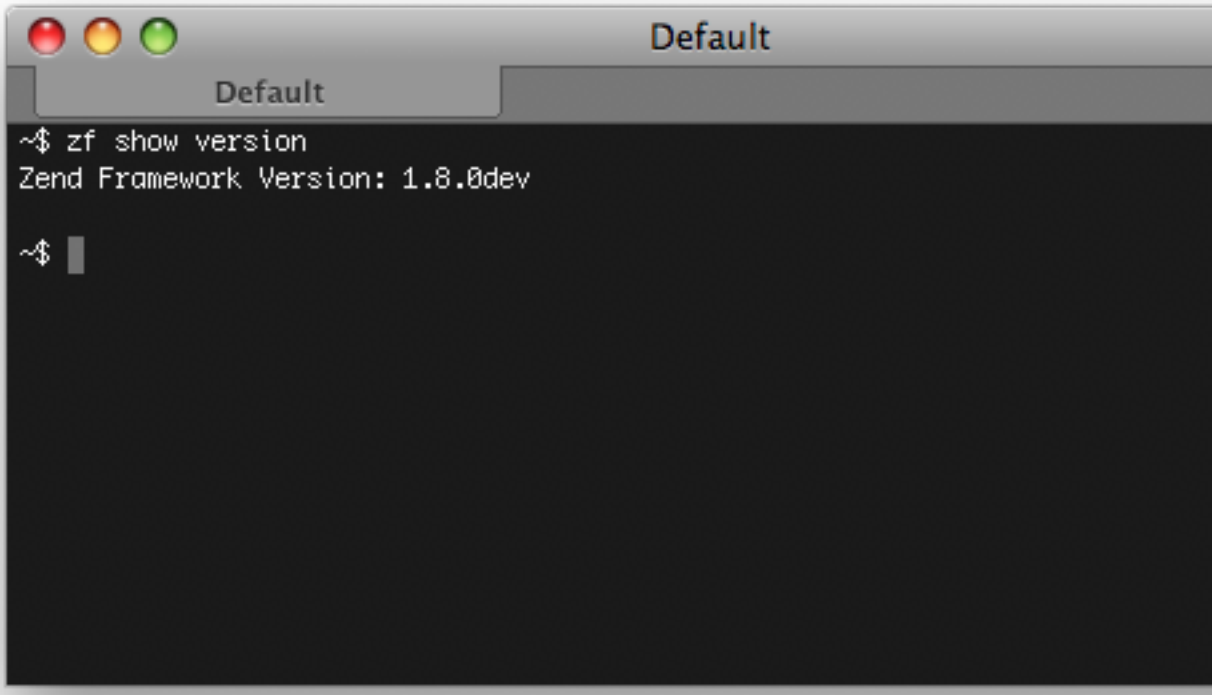
The most common setup in the *nix environment, is to copy the `zf.sh` and `zf.php` into the same directory as your PHP binary. This can generally be found in one of the following places:

```
/usr/bin
/usr/local/bin
/usr/local/ZendServer/bin/
/Applications/ZendServer/bin/
```

To find out the location of your PHP binary, you can execute 'which php' on the command line. This will return the location of the PHP binary you will be using to run PHP scripts in this environment.

The next order of business is to ensure that Zend Framework library is set up correctly inside of the system PHP include_path. To find out where your include_path is located, you can execute **php -i** and look for the include_path variable, or more succinctly, execute **php -i | grep include_path**. Once you have found where your include_path is located (this will generally be something like `/usr/lib/php`, `/usr/share/php`, `/usr/local/lib/php`, or similar), ensure that the contents of the `/library/` directory are put inside your include_path specified directory.

Once you have done those two things, you should be able to issue a command and get back the proper response like this:



If you do not see this type of output, go back and check your setup to ensure you have all of the necessary pieces in the proper place.

There are a couple of alternative setups you might want to employ depending on your servers configuration, your level of access, or for other reasons.

Alternative Setup involves keeping the Zend Framework download together as is, and creating a link from a `PATH` location to the `zf.sh`. What this means is you can place the contents of the ZendFramework download into a location such as `/usr/local/share/ZendFramework`, or more locally like `/home/username/lib/ZendFramework`, and creating a symbolic link to the `zf.sh`.

Assuming you want to put the link inside `/usr/local/bin` (this could also work for placing the link inside `/home/username/bin/` for example) you would issue a command similar to this:

```
ln -s /usr/local/share/ZendFramework/bin/zf.sh /usr/local/bin/zf

# OR (for example)
ln -s /home/username/lib/ZendFramework/bin/zf.sh /home/username/bin/zf
```

This will create a link which you should be able to access globally on the command line.

2.3. Setting up the CLI tool on Windows

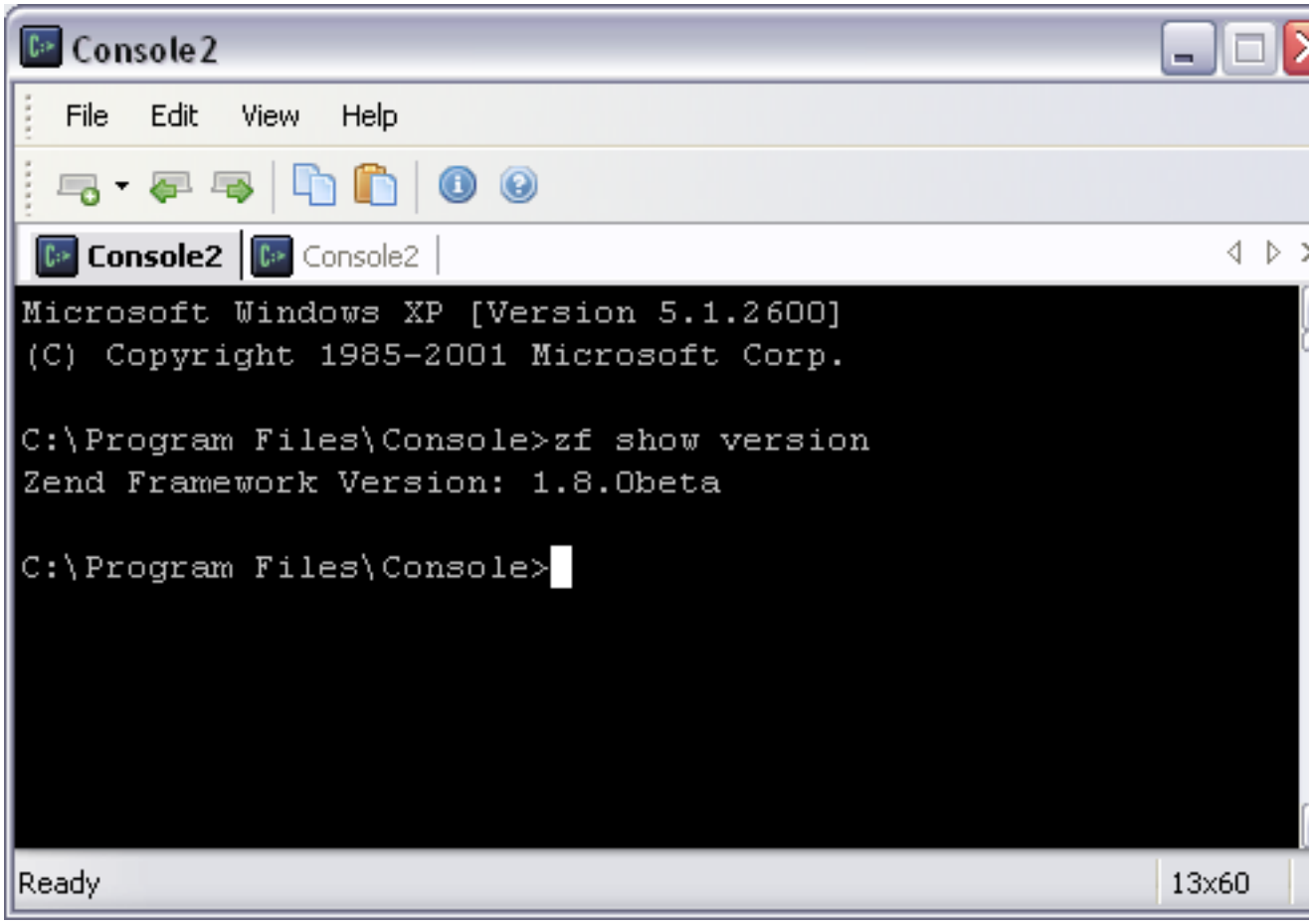
The most common setup in the Windows Win32 environment, is to copy the `zf.bat` and `zf.php` into the same directory as your PHP binary. This can generally be found in one of the following places:

```
C:\PHP
C:\Program Files\ZendServer\bin\
C:\WAMP\PHP\bin
```

You should be able to run `php.exe` on the command line. If you are not able to, first check the documentation that came with your PHP distribution, or ensure that the path to `php.exe` is in your Windows PATH environment variable.

The next order of business is to ensure that Zend Framework library is set up correctly inside of the system PHP include_path. To find out where your include_path is located, you can type **php -i** and look for the include_path variable, or more succinctly execute **php -i | grep include_path** if you have Cygwin setup with grep available. Once you have found where your include_path is located (this will generally be something like `C:\PHP\pear`, `C:\PHP\share`, `C:\Program %20Files\ZendServer\share` or similar), ensure that the contents of the library/ directory are put inside your include_path specified directory.

Once you have done those two things, you should be able to issue a command and get back the proper response like this:



If you do not see this type of output, go back and check your setup to ensure you have all of the necessary pieces in the proper place.

There are a couple of alternative setups you might want to employ depending on your server's configuration, your level of access, or for other reasons.

Alternative Setup involves keeping the Zend Framework download together as is, and altering both your system `PATH` as well as the `php.ini` file. In your user's environment, make sure to add `C:\Path\To\ZendFramework\bin`, so that your `zf.bat` file is executable. Also, alter the `php.ini` file to ensure that `C:\Path\To\ZendFramework\library` is in your include_path.

2.4. Other Setup Considerations

If for some reason you do not want Zend Framework library inside your include_path, there is another option. There are two special environment variables that `zf.php` will utilize to determine the location of your Zend Framework installation.

The first is `ZEND_TOOL_INCLUDE_PATH_PREPEND`, which will prepend the value of this environment variable to the system (`php.ini`) include_path before loading the client.

Alternatively, you might want to use `ZEND_TOOL_INCLUDE_PATH` to completely *replace* the system include_path for one that makes sense specifically for the **zf** command line tool.

2.5. Where To Go Next?

At this point, you should be set up to start initiating some more "interesting" commands. To get going, you can issue the **zf --help** command to see what is available to you.

```
~/tmp/scratch$ php ~/Projects/ScratchPHP/scripts/test-cli-prototyper.php
~/temp$ zf --help
Usage:
  zf [--global-options] action-name [--action-options] provider-name [--provide
    Note: You may use "?" in any place of the above usage string to ask for more
    Example: "zf ? project" will provide all available actions for the project pr
Providers and their actions:
  Model
    zf create model [model name]

  Version
    zf show version

  Profile
    zf show profile

  Phpinfo
    zf show phpinfo [section name]

  Project
    zf create project [project path] [alternate profile]

  Controller
    zf create controller [controller name] [view included]

  View
    zf create view [controller name] [action or script name]

  Action
    zf create action [controller name] [view included]

  DbAdapter
    zf configure db-adapter [adapter type] [parameters]
    zf test db-adapter.connection
```

Continue on to the `Zend_Tool_Project` "Create Project" section to understand how to use the **zf** script for project creation.

3. Architecture

3.1. Registry

Because providers and manifests may come from anywhere in the include_path, a registry is provided to simplify access to the various pieces of the toolchain. This registry is injected into registry-aware components, which may then pull dependencies from them as necessary. Most dependencies registered with the registry will be sub-component-specific repositories.

The interface for the registry consists of the following definition:

```
interface Zend_Tool_Framework_Registry_Interface
{
    public function setClient(Zend_Tool_Framework_Client_Abstract $client);
    public function getClient();
    public function setLoader(Zend_Tool_Framework_Loader_Abstract $loader);
    public function getLoader();
    public function setActionRepository(
        Zend_Tool_Framework_Action_Repository $actionRepository
    );
    public function getActionRepository();
    public function setProviderRepository(
        Zend_Tool_Framework_Provider_Repository $providerRepository
    );
    public function getProviderRepository();
    public function setManifestRepository(
        Zend_Tool_Framework_Manifest_Repository $manifestRepository
    );
    public function getManifestRepository();
    public function setRequest(Zend_Tool_Framework_Client_Request $request);
    public function getRequest();
    public function setResponse(Zend_Tool_Framework_Client_Response $response);
    public function getResponse();
}
```

The various objects the registry manages will be discussed in their appropriate sections.

Classes that should be registry-aware should implement `Zend_Tool_Framework_Registry_EnabledInterface`. This interface merely allows initialization of the registry in the target class.

```
interface Zend_Tool_Framework_Registry_EnabledInterface
{
    public function setRegistry(
        Zend_Tool_Framework_Registry_Interface $registry
    );
}
```

3.2. Providers

`Zend_Tool_Framework_Provider` represents the functional or "capability" aspect of the framework. Fundamentally, `Zend_Tool_Framework_Provider` will provide the interfaces necessary to produce "providers", or bits of tooling functionality that can be called and used inside the `Zend_Tool_Framework` toolchain. The simplistic nature of implementing this provider interface allows the developer a "one-stop-shop" of adding functionality or capabilities to `Zend_Tool_Framework`.

The provider interface is an empty interface and enforces no methods (this is the Marker Interface pattern):

```
interface Zend_Tool_Framework_Provider_Interface
{}
```

Or, if you wish, you can implement the base (or abstract) Provider which will give you access to the `Zend_Tool_Framework_Registry`:

```
abstract class Zend_Tool_Framework_Provider_Abstract
    implements Zend_Tool_Framework_Provider_Interface,
               Zend_Tool_Registry_EnabledInterface
{
    protected $_registry;
    public function setRegistry(
        Zend_Tool_Framework_Registry_Interface $registry
    );
}
```

3.3. Loaders

The purpose of a Loader is to find Providers and Manifest files that contain classes which implement either `Zend_Tool_Framework_Provider_Interface` or `Zend_Tool_Framework_Manifest_Interface`. Once these files are found by a loader, providers are loaded into the Provider Repository and manifest metadata is loaded into the Manifest Repository.

To implement a loader, one must extend the following abstract class:

```
abstract class Zend_Tool_Framework_Loader_Abstract
{

    abstract protected function _getFiles();

    public function load()
    {
        /** ... */
    }
}
```

The `_getFiles()` method should return an array of files (absolute paths). The built-in loader supplied with Zend Framework is called the IncludePath loader. By default, the Tooling framework will use an include_path based loader to find files that might include Providers or Manifest Metadata objects. `Zend_Tool_Framework_Loader_IncludePathLoader`, without any other options, will search for files inside the include path that end in `Mainfest.php`, `Tool.php` or `Provider.php`. Once found, they will be tested (by the `load()` method of the `Zend_Tool_Framework_Loader_Abstract`) to determine if they implement any of the supported interfaces. If they do, an instance of the found class is instantiated, and it is appended to the proper repository.

```
class Zend_Tool_Framework_Loader_IncludePathLoader
    extends Zend_Tool_Framework_Loader_Abstract
{

    protected $_filterDenyDirectoryPattern = '.*(/|\\\\).svn';
    protected $_filterAcceptFilePattern = '.*(?:Manifest|Provider)\.php$';

    protected function _getFiles()
    {
        /** ... */
    }
}
```

As you can see, the IncludePath loader will search all include_paths for the files that match the `$_filterAcceptFilePattern` and *not* match the `$_filterDenyDirectoryPattern`.

3.4. Manifests

In short, the Manifest shall contain specific or arbitrary metadata that is useful to any provider or client, as well as be responsible for loading any additional providers into the provider repository.

To introduce metadata into the manifest repository, all one must do is implement the empty `Zend_Tool_Framework_Manifest_Interface`, and provide a `getMetadata()` method which shall return an array of objects that implement `Zend_Tool_Framework_Manifest_Metadata`.

```
interface Zend_Tool_Framework_Manifest_Interface
{
    public function getMetadata();
}
```

Metadata objects are loaded (by a loader defined below) into the Manifest Repository (`Zend_Tool_Framework_Manifest_Repository`). Manifests will be processed after all Providers have been found to be loaded into the provider repository. This shall allow Manifests to create Metadata objects based on what is currently inside the provider repository.

There are a few different metadata classes that can be used to describe metadata. The `Zend_Tool_Framework_Manifest_Metadata` is the base metadata object. As you can see by the following code snippet, the base metadata class is fairly lightweight and abstract in nature:

```
class Zend_Tool_Framework_Metadata_Basic
{

    protected $_type        = 'Global';
    protected $_name        = null;
    protected $_value       = null;
    protected $_reference   = null;

    public function getType();
    public function getName();
    public function getValue();
    public function getReference();
    /** ... */
}
```

There are other built in metadata classes as well for describing more specialized metadata: `ActionMetadata` and `ProviderMetadata`. These classes will help you describe in more detail metadata that is specific to either actions or providers, and the reference is expected to be a reference to an action or a provider respectively. These classes are described in the following code snippet.

```
class Zend_Tool_Framework_Manifest_ActionMetadata
    extends Zend_Tool_Framework_Manifest_Metadata
{

    protected $_type = 'Action';
    protected $_actionName = null;

    public function getActionName();
    /** ... */
}
```

```
class Zend_Tool_Framework_Manifest_ProviderMetadata
    extends Zend_Tool_Framework_Manifest_Metadata
{

    protected $_type = 'Provider';
    protected $_providerName  = null;
    protected $_actionName    = null;
    protected $_specialtyName = null;

    public function getProviderName();
    public function getActionName();
    public function getSpecialtyName();
    /** ... */
}
```

'Type' in these classes is used to describe the type of metadata the object is responsible for. In the cases of the `ActionMetadata`, the type would be 'Action', and conversely in the case of the `ProviderMetadata` the type is 'Provider'. These metadata types will also include additional structured information about both the "thing" they are describing as well as the object (the `getReference()`) they are referencing with this new metadata.

In order to create your own metadata type, all one must do is extend the base `Zend_Tool_Framework_Manifest_Metadata` class and return these new metadata objects via a local Manifest class or object. These user based classes will live in the Manifest Repository

Once these metadata objects are in the repository, there are then two different methods that can be used in order to search for them in the repository.

```
class Zend_Tool_Framework_Manifest_Repository
{
    /**
     * To use this method to search, $searchProperties should contain the names
     * and values of the key/value pairs you would like to match within the
     * manifest.
     *
     * For Example:
     *     $manifestRepository->findMetadatas(array(
     *         'action' => 'Foo',
     *         'name'   => 'cliActionName'
     *         ));
     *
     * Will find any metadata objects that have a key with name 'action' value
     * of 'Foo', AND a key named 'name' value of 'cliActionName'
     *
     * Note: to either exclude or include name/value pairs that exist in the
     * search criteria but do not appear in the object, pass a bool value to
     * $includeNonExistentProperties
     */
    public function findMetadatas(Array $searchProperties = array(),
                                  $includeNonExistentProperties = true);

    /**
     * The following will return exactly one of the matching search criteria,
     * regardless of how many have been returned. First one in the manifest is
     * what will be returned.
     */
    public function findMetadata(Array $searchProperties = array(),
                                 $includeNonExistentProperties = true)
    {
```

```
        $metadatas = $this->getMetadatas($searchProperties,
                                         $includeNonExistentProperties);
        return array_shift($metadatas);
    }
}
```

Looking at the search methods above, the signatures allow for extremely flexible searching. In order to find a metadata object, simply pass in an array of matching constraints via an array. If the data is accessible through the Property accessor (the `getSomething()` methods implemented on the metadata object), then it will be passed back to the user as a "found" metadata object.

3.5. Clients

Clients are the interface which bridges a user or external tool into the `Zend_Tool_Framework` system. Clients can come in all shapes and sizes: RPC endpoints, Command Line Interface, or even a web interface. `Zend_Tool` has implemented the command line interface as the default interface for interacting with the `Zend_Tool_Framework` system.

To implement a client, one would need to extend the following abstract class:

```
abstract class Zend_Tool_Framework_Client_Abstract
{
    /**
     * This method should be implemented by the client implementation to
     * construct and set custom loaders, request and response objects.
     *
     * (not required, but suggested)
     */
    protected function _preInit();

    /**
     * This method should be implemented by the client implementation to parse
     * out and set up the request objects action, provider and parameter
     * information.
     */
    abstract protected function _preDispatch();

    /**
     * This method should be implemented by the client implementation to take
     * the output of the response object and return it (in an client specific
     * way) back to the Tooling Client.
     *
     * (not required, but suggested)
     */
    abstract protected function _postDispatch();
}
```

As you can see, there 1 method is required to fulfill the needs of a client (two others suggested), the initialization, prehandling and post handling. For a more in depth study of how the command line client works, please see the source code.

4. Creating Providers to use with Zend_Tool_Framework

In general, a provider, on its own, is nothing more than the shell for a developer to bundle up some capabilities they wish to dispatch with the command line (or other) clients. It is an analogue to what a "controller" is inside of your MVC application.

4.1. How Zend Tool finds your Providers

By default Zend Tool uses the IncludePathLoader to find all the providers that you can run. It recursivly iterates all include path directories and opens all files that end with "Manifest.php" or "Provider.php". All classes in these files are inspected if they implement either `Zend_Tool_Framework_Provider_Interface` or `Zend_Tool_Framework_Manifest_ProviderManifestable`. Instances of the provider interface make up for the real functionality and all their public methods are accessible as provider actions. The ProviderManifestable interface however requires the implementation of a method `getProviders()` which returns an array of instantiated provider interface instances.

The following naming rules apply on how you can access the providers that were found by the IncludePathLoader:

- The last part of your classname split by underscore is used for the provider name, e.g. "My_Provider_Hello" leads to your provider being accessible by the name "hello".
- If your provider has a method `getName()` it will be used instead of the previous method to determine the name.
- If your provider has "Provider" as prefix, e.g. it is called `My_HelloProvider` it will be stripped from the name so that the provider will be called "hello".

The IncludePathLoader does not follow symlinks, that means you cannot link provider functionality into your include paths, they have to be physically present in the include paths.

Example 879. Exposing Your Providers with a Manifest

You can expose your providers to Zend Tool by offering a manifest with a special filename ending with "Manifest.php". A Provider Manifest is an implementation of the Zend_Tool_Framework_Manifest_ProviderManifestable and requires the `getProviders()` method to return an array of instantiated providers. In anticipation of our first own provider `My_Component_HelloProvider` we will create the following manifest:

```
class My_Component_Manifest
    implements Zend_Tool_Framework_Manifest_ProviderManifestable
{
    public function getProviders()
    {
        return array(
            new My_Component_HelloProvider()
        );
    }
}
```

4.2. Basic Instructions for Creating Providers

As an example, if a developer wants to add the capability of showing the version of a datafile that his 3rd party component is working from, there is only one class the developer would need to implement. Assuming the component is called `My_Component`, he would create a class named `My_Component_HelloProvider` in a file named `HelloProvider.php` somewhere on the include_path. This class would implement `Zend_Tool_Framework_Provider_Interface`, and the body of this file would only have to look like the following:

```
class My_Component_HelloProvider
    implements Zend_Tool_Framework_Provider_Interface
{
    public function say()
    {
        echo 'Hello from my provider!';
    }
}
```

Given that code above, and assuming the developer wishes to access this functionality through the console client, the call would look like this:

```
% zf say hello
Hello from my provider!
```

4.3. The response object

As discussed in the architecture section Zend Tool allows to hook different clients for using your Zend Tool providers. To keep compliant with different clients you should use the response object to return messages from your providers instead of using `echo()` or a similiar output mechanism. Rewritting our hello provider with this knowledge it looks like:

```
class My_Component_HelloProvider
    extends Zend_Tool_Framework_Provider_Abstract
{
    public function say()
    {
        $this->_registry->getResponse
                        ->appendContent("Hello from my provider!");
    }
}
```

As you can see one has to extend the `Zend_Tool_Framework_Provider_Abstract` to gain access to the Registry which holds the `Zend_Tool_Framework_Client_Response` instance.

4.4. Advanced Development Information

4.4.1. Passing Variables to a Provider

The above "Hello World" example is great for simple commands, but what about something more advanced? As your scripting and tooling needs grow, you might find that you need the ability to accept variables. Much like function signatures have parameters, your tooling requests can also accept parameters.

Just as each tooling request can be isolated to a method within a class, the parameters of a tooling request can also be isolated in a very well known place. Parameters of the action methods of a provider can include the same parameters you want your client to utilize when calling that provider and action combination. For example, if you wanted to accept a name in the above example, you would probably do this in OO code:

```
class My_Component_HelloProvider
    implements Zend_Tool_Framework_Provider_Interface
{
    public function say($name = 'Ralph')
    {
        echo 'Hello' . $name . ', from my provider!';
```

```
    }
}
```

The above example can then be called via the command line **zf say hello Joe**. "Joe" will be supplied to the provider as a parameter of the method call. Also note, as you see that the parameter is optional, that means it is also optional on the command line, so that **zf say hello** will still work, and default to the name "Ralph".

4.4.2. Prompt the User for Input

There are cases when the workflow of your provider requires to prompt the user for input. This can be done by requesting the client to ask for more the required input by calling:

```
class My_Component_HelloProvider
    extends Zend_Tool_Framework_Provider_Abstract
{
    public function say($name = 'Ralph')
    {
        $nameResponse = $this->_registry
                             ->getClient()
                             ->promptInteractiveInput("Whats your name?");
        $name = $name->getContent();

        echo 'Hello' . $name . ', from my provider!';
    }
}
```

This command throws an exception if the current client is not able to handle interactive requests. In case of the default Console Client however you will be asked to enter the name.

4.4.3. Pretending to execute a Provider Action

Another interesting feature you might wish to implement is *pretendability*. Pretendabilty is the ability for your provider to "pretend" as if it is doing the requested action and provider combination and give the user as much information about what it *would* do without actually doing it. This might be an important notion when doing heavy database or filesystem modifications that the user might not otherwise want to do.

Pretendability is easy to implement. There are two parts to this feature: 1) marking the provider as having the ability to "pretend", and 2) checking the request to ensure the current request was indeed asked to be "pretended". This feature is demonstrated in the code sample below.

```
class My_Component_HelloProvider
    extends    Zend_Tool_Framework_Provider_Abstract
    implements Zend_Tool_Framework_Provider_Pretendable
{
    public function say($name = 'Ralph')
    {
        if ($this->_registry->getRequest()->isPretend()) {
            echo 'I would say hello to ' . $name . '.';
        } else {
            echo 'Hello' . $name . ', from my provider!';
        }
    }
}
```

To run the provider in pretend mode just call:

```
% zf --pretend say hello Ralph
I would say hello Ralph.
```

4.4.4. Verbose and Debug modes

You can also run your provider actions in "verbose" or "debug" modes. The semantics in regard to this actions have to be implemented by you in the context of your provider. You can access debug or verbose modes with:

```
class My_Component_HelloProvider
    implements Zend_Tool_Framework_Provider_Interface
{
    public function say($name = 'Ralph')
    {
        if($this->_registry->getRequest()->isVerbose()) {
            echo "Hello::say has been called\n";
        }
        if($this->_registry->getRequest()->isDebug()) {
            syslog(LOG_INFO, "Hello::say has been called\n");
        }
    }
}
```

4.4.5. Accessing User Config and Storage

Using the Enviroment variable ZF_CONFIG_FILE or the .zf.ini in your home directory you can inject configuration parameters into any Zend Tool provider. Access to this configuration is available via the registry that is passed to your provider if you extend `Zend_Tool_Framework_Provider_Abstract`.

```
class My_Component_HelloProvider
    extends Zend_Tool_Framework_Provider_Abstract
{
    public function say()
    {
        $username = $this->_registry->getConfig()->username;
        if(!empty($username)) {
            echo "Hello $username!";
        } else {
            echo "Hello!";
        }
    }
}
```

The returned configuration is of the type `Zend_Tool_Framework_Client_Config` but internally the `__get()` and `__set()` magic methods proxy to a `Zend_Config` of the given configuration type.

The storage allows to save arbitrary data for later reference. This can be useful for batch processing tasks or for re-runs of your tasks. You can access the storage in a similar way like the configuration:

```
class My_Component_HelloProvider
    extends Zend_Tool_Framework_Provider_Abstract
{
    public function say()
    {
```

```
        $aValue = $this->_registry->getStorage()->get("myUsername");
        echo "Hello $aValue!";
    }
}
```

The API of the storage is very simple:

```
class Zend_Tool_Framework_Client_Storage
{
    public function setAdapter($adapter);
    public function isEnabled();
    public function put($name, $value);
    public function get($name, $defaultValue=null);
    public function has($name);
    public function remove($name);
    public function getStreamUri($name);
}
```

When designing your providers that are config or storage aware remember to check if the required user-config or storage keys really exist for a user. You won't run into fatal errors when none of these are provided though, since empty ones are created upon request.

5. Shipped System Providers

In addition to the more useful project based providers that come shipped with `Zend_Tool_Project`, there are also some more basic, but interesting providers that come built into `Zend_Tool_Framework`. Some of these exist for the purpose of providing a means via the command line to extract information, such as the version, while others are intended to aid the developer when creating additional providers.

5.1. The Version Provider

The Version provider is included so that you may determine which version of the framework that the **zf** or `Zend_Tool` is currently set to work with.

Through the command line, simply run **zf show version**.

5.2. The Manifest Provider

The Manifest provider is included so that you may determine what kind of "manifest" information is available during the `Zend_Tool` runtime. Manifest data is information that is attached to specific objects during `Zend_Tool`'s runtime. Inside the manifest you will find the console specific namings that you are expected to use when calling certain commands. Data found in the manifest can be used by any provider or client on an as-needed basis.

Through the command line, simply run **zf show manifest**.

6. Extending and Configuring Zend_Tool_Framework

6.1. Customizing Zend_Tool Console Client

As of Zend Framework 1.9, `Zend_Tool_Framework` allows developers to store information, provider specific configuration values, and custom files in a special location on the developers

machine. These configuration values and files can be used by providers to extend functionality, customize functionality, or any other reasons a provider sees fit.

The primary purpose, and the purpose most immediately used by existing providers is to allow developers to customize the way the "out of the box" providers do work.

One of the more commonly requested features is to be able to provide custom project profiles to Zend_Tool_Project's Project Provider. This would allow developers to store a custom profile in a special place that can be used repeatedly by the Zend_Tool system in order to build custom profiles. Another commonly requested feature is to be able to configure the behavior of providers with a configuration setting. In order to achieve this, not only do we have to have a Zend_Tool configuration file, but we also have to have a place to find this configuration file.

6.1.1. The Home Directory

Before the Console Client can start searching for a Zend_Tool configuration file or a local storage directory, it must first be able to identify where the "home directory" is located.

On *nix-based machines, PHP will be populated with an environment variable named HOME with a path to the current users home directory. Typically, this path will be very similar to /home/myusername.

On Windows-based machines, PHP will typically be populated with an environment variable named HOMEPATH with the current users home directory. This directory is usually found in either C:\Documents and Settings\Username\, or in Vista at C:\Users\Username.

If either a home directory cannot be found, or you wish to change the location of where Zend_Tool_Framework Console Client finds the home directory, you can provide an environment variable named ZF_HOME to specify where to find the home directory.

6.1.2. Local Storage

Once a home directory can be located, Zend_Tool_Framework's Console Client can either autodiscover the local storage directory, or it can be told where to expect the local storage directory.

Assuming the home directory has been found (here noted as $HOME), the Console Client will then look for the local storage directory in $HOME/.zf/. If found, it will set the local storage directory to this location.

If the directory cannot be found, or the developer wishes to override this location, that can be done by setting an environment variable. Regardless if $HOME has been previously set or not, the developer may supply the environment variable ZF_STORAGE_DIR.

Once the path to a local storage directory is found, the directory *must* exist for it to be passed into the Zend_Tool_Framework runtime, as it will not be created for you.

6.1.3. User Configuration

Like local storage, once a home directory can be located, Zend_Tool_Framework's Console Client can then either attempt to autodiscover the path to a configuration file, or it can be told specifically where to find the configuration file.

Assuming the home directory has been found (here noted as $HOME), the Console Client will then attempt to look for the existence of a configuration file located at $HOME/.zf.ini. This file, if found, will be used as the configuration file for Zend_Tool_Framework.

If that location does not exist, but a local storage directory does, then the Console Client will then attempt to locate the configuration file within the local storage directory. Assuming the local storage directory exists in `$LOCAL_STORAGE`, then if a file exists as `$LOCAL_STORAGE/zf.ini`, it will be found by the Console Client and utilized as the `Zend_Tool_Framework` configuration file.

If the file cannot be autodiscovered or the developer wishes to specify the location of location of the configuration file, the developer can do so by setting an environment variable. If the environment variable `ZF_CONFIG_FILE` is set, then its value will be used as the location of the configuration file to use with the Console Client. The `ZF_CONFIG_FILE` can point to any Zend_Config readable INI, XML or PHP File.

If the file does not exist in either the autodiscovered or the provided location, it will not be used as `Zend_Tool_Framework` does not attempt to create the file automatically.

6.1.4. User Configuration File Content

The configuration file should be structured as a `Zend_Config` configuration file, in ini format, and without any sections being defined. First level keys should be used by the provider searching for a specific value. For example, if the "Project" provider is expecting a "profiles" directory, then it should typically be understood that it will search for the following ini key value pair:

```
project.profile = some/path/to/some-directory
```

The only reserved ini prefix is the value "php". The "php" prefix to values will be reserved to store names and values of runtime settable php values, such as include_path or error_reporting. To override the include_path and error_reporting with an ini value, a developer would set:

```
php.include_path = "/path/to/includes1:/path/to/includes2"
php.error_reporting = 1
```

The reserved prefix "php" only works with INI files. You can't set PHP INI values with PHP or XML config.

Zend_Tool_Project

1. Introduction

`Zend_Tool_Project` builds on and extends the capabilities of `Zend_Tool_Framework` to that of managing a "project". In general, a "project" is a planned endeavor or an initiative. In the computer world, projects generally are a collection of resources. These resources can be files, directories, databases, schemas, images, styles, and more.

This same concept applies to Zend Framework projects. In Zend Framework projects, you have controllers, actions, views, models, databases and so on and so forth. In terms of `Zend_Tool`, we need a way to track these types of resources - thus `Zend_Tool_Project`.

`Zend_Tool_Project` is capable of tracking project resources throughout the development of a project. So, for example, if in one command you created a controller, and in the next command you wish to create an action within that controller, `Zend_Tool_Project` is gonna have to *know* about the controller file you created so that you can (in the next action), be able to append that action to it. This is what keeps our projects up to date and *stateful*.

Another important point to understand about projects is that typically, resources are organized in a hierarchical fashion. With that in mind, `Zend_Tool_Project` is capable of serializing the current project into a internal representation that allows it to keep track of not only *what* resources are part of a project at any given time, but also *where* they are in relation to one another.

2. Create A Project

The following examples will assume you have the command line interface of `Zend_Tool_Framework` available to you.

To issue any of the commands for `Zend_Tool_Project` with CLI, you must be in the directory where the project was initially created.

To get started with `Zend_Tool_Project`, you simply need to create a project. Creating a project is simple: go to a place on your filesystem, create a directory, change to that directory, then issue the following command:

/tmp/project$ zf create project

Optionally, you can create a directory anywhere by the following:

$ zf create project /path/to/non-existent-dir

The following table will describe the capabilities of providers that are available to you. As you can see in this table, there is a "Project" provider. The Project provider has a couple of actions associated to it, and with those actions a number of options that can be used to modify the behavior of the action and provider.

Table 148. Project Provider Options

Provider Name	Available Actions	Parameters	CLI Usage
Project	Create / Show	create - [path=null, profile='default']	**zf create project some/path**

3. Zend Tool Project Providers

Below is a table of all of the providers shipped with `Zend_Tool_Project`.

Table 149. Project Provider Options

Provider Name	Available Actions	Parameters	CLI Usage
Controller	Create	create - [name, indexActionIncluded=true]	**zf create controller foo**
Action	Create	create - [name, controllerName=index, viewIncluded=true]	**zf create action bar foo** (or **zf create action --name bar --controlller-name=foo**)
Controller	Create	create - [name, indexActionIncluded=true]	**zf create controller foo**
Profile	Show	show - []	**zf show profile**
View	Create	create - [controllerName,actionName]	**zf create view foo bar** (or **zf create view -c foo -a bar**)
Test	Create / Enable / Disable	create - [libraryClassName]	**zf create test My_Foo_Baz** / **zf disable test** / **zf enable test**

4. Zend_Tool_Project Internals

4.1. Zend_Tool_Project Internal Xml Structure

4.2. Zend_Tool_Project Internal Extending

Zend_Translate

1. Introduction

`Zend_Translate` is Zend Framework's solution for multilingual applications.

In multilingual applications, the content must be translated into several languages and display content depending on the user's language. PHP offers already several ways to handle such problems, however the PHP solution has some problems:

- *Inconsistent API:* There is no single API for the different source formats. The usage of gettext for example is very complicated.
- *PHP supports only gettext and native array:* PHP itself offers only support for array or gettext. All other source formats have to be coded manually, because there is no native support.
- *No detection of the default language:* The default language of the user cannot be detected without deeper knowledge of the backgrounds for the different web browsers.
- *Gettext is not thread-safe:* PHP's gettext library is not thread safe, and it should not be used in a multithreaded environment. This is due to problems with gettext itself, not PHP, but it is an existing problem.

`Zend_Translate` does not have the above problems. This is why we recommend using `Zend_Translate` instead of PHP's native functions. The benefits of `Zend_Translate` are:

- *Supports multiple source formats:* `Zend_Translate` supports several source formats, including those supported by PHP, and other formats including TMX and CSV files.
- *Thread-safe gettext:* The gettext reader of `Zend_Translate` is thread-safe. There are no problems using it in multi-threaded environments.
- *Easy and generic API:* The API of `Zend_Translate` is very simple and requires only a handful of functions. So it's easy to learn and easy to maintain. All source formats are handled the same way, so if the format of your source files change from Gettext to TMX, you only need to change one line of code to specify the storage adapter.
- *Detection of the user's standard language:* The preferred language of the user accessing the site can be detected and used by `Zend_Translate`.
- *Automatic source detection:* `Zend_Translate` is capable of detecting and integrating multiple source files and additionally detect the locale to be used depending on directory or filenames.

1.1. Starting multi-lingual

So let's get started with multi-lingual business. What we want to do is translate our string output so the view produces the translated output. Otherwise we would have to write one view for each language, and no one would like to do this. Generally, multi-lingual sites are very simple in their design. There are only four steps you would have to do:

1. Decide which adapter you want to use;
2. Create your view and integrate `Zend_Translate` in your code;
3. Create the source file from your code;

4. Translate your source file to the desired language.

The following sections guide you through all four steps. Read through the next few pages to create your own multi-lingual web application.

2. Adapters for Zend_Translate

`Zend_Translate` can handle different adapters for translation. Each adapter has its own advantages and disadvantages. Below is a comprehensive list of all supported adapters for translation source files.

Table 150. Adapters for Zend_Translate

Adapter	Description	Usage
Array	Use PHP arrays	Small pages; simplest usage; only for programmers
Csv	Use comma separated (*.csv/*.txt) files	Simple text file format; fast; possible problems with unicode characters
Gettext	Use binary gettext (*.mo) files	GNU standard for linux; thread-safe; needs tools for translation
Ini	Use simple ini (*.ini) files	Simple text file format; fast; possible problems with unicode characters
Tbx	Use termbase exchange (*.tbx/*.xml) files	Industry standard for inter application terminology strings; XML format
Tmx	Use tmx (*.tmx/*.xml) files	Industry standard for inter application translation; XML format; human readable
Qt	Use qt linguist (*.ts) files	Cross platform application framework; XML format; human readable
Xliff	Use xliff (*.xliff/*.xml) files	A simpler format as `TMX` but related to it; XML format; human readable
XmlTm	Use xmltm (*.xml) files	Industry standard for XML document translation memory; XML format; human readable
Others	*.sql	Different other adapters may be implemented in the future

2.1. How to decide which translation adapter to use

You should decide which Adapter you want to use for `Zend_Translate`. Frequently, external criteria such as a project requirement or a customer requirement determines this for you, but if you are in the position to do this yourself, the following hints may simplify your decision.

When deciding your adapter you should also be aware of the used encoding. Even if Zend Framework declares UTF-8 as default encoding you will sometimes

be in the need of other encoding. `Zend_Translate` will not change any encoding which is defined within the source file which means that if your Gettext source is build upon ISO-8859-1 it will also return strings in this encoding without converting them. There is only one restriction:

When you use a xml based source format like TMX or XLIFF you must define the encoding within the xml files header because xml files without defined encoding will be treated as UTF-8 by any xml parser by default. You should also be aware that actually the encoding of xml files is limited to the encodings supported by PHP which are UTF-8, ISO-8859-1 and US-ASCII.

2.1.1. Zend_Translate_Adapter_Array

The Array Adapter is the Adapter which is simplest to use for programmers. But when you have numerous translation strings or many languages you should think about another Adapter. For example, if you have 5000 translation strings, the Array Adapter is possibly not the best choice for you.

You should only use this Adapter for small sites with a handful of languages, and if you or your programmer team creates the translations yourselves.

2.1.2. Zend_Translate_Adapter_Csv

The Csv Adapter is the Adapter which is simplest to use for customers. CSV files are readable by standard text editors, but text editors often do not support utf8 character sets.

You should only use this Adapter if your customer wants to do translations himself.

Beware that the Csv Adapter has problems when your Csv files are encoded differently than the locale setting of your environment. This is due to a Bug of PHP itself which will not be fixed before PHP 6.0 (http://bugs.php.net/bug.php?id=38471). So you should be aware that the Csv Adapter due to PHP restrictions is not locale aware.

2.1.3. Zend_Translate_Adapter_Gettext

The Gettext Adapter is the Adapter which is used most frequently. Gettext is a translation source format which was introduced by GNU, and is now used worldwide. It is not human readable, but there are several freeware tools (for instance, POEdit), which are very helpful. The `Zend_Translate` Gettext Adapter is not implemented using PHP's gettext extension. You can use the Gettext Adapter even if you do not have the PHP gettext extension installed. Also the Adapter is thread-safe and the PHP gettext extension is currently not thread-safe.

Most people will use this adapter. With the available tools, professional translation is very simple. But gettext data are is stored in a machine-readable format, which is not readable without tools.

2.1.4. Zend_Translate_Adapter_Ini

The Ini Adapter is a very simple Adapter which can even be used directly by customers. INI files are readable by standard text editors, but text editors often do not support utf8 character sets.

You should only use this Adapter when your customer wants to do translations himself. Do not use this adapter as generic translation source.

Regression in PHP 5.3

Prior to PHP 5.3, `parse_ini_file()` and `parse_ini_string()` handled non-ASCII characters within INI option keys worked without an issue. However, starting with PHP 5.3, any such keys will now be silently dropped in the returned array from either function. If you had keys utilizing UTF-8 or Latin-1 characters, you may find your translations no longer work when using the INI adapter. If this is the case, we recommend utilizing a different adapter.

2.1.5. Zend_Translate_Adapter_Tbx

The Tbx Adapter is an Adapter which will be used by customers which already use the TBX format for their internal translation system. Tbx is no standard translation format but more a collection of already translated and pre translated source strings. When you use this adapter you have to be sure that all your needed source string are translated. TBX is a XML file based format and a completely new format. XML files are human-readable, but the parsing is not as fast as with gettext files.

This adapter is perfect for companies when pre translated source files already exist. The files are human readable and system-independent.

2.1.6. Zend_Translate_Adapter_Tmx

The Tmx Adapter is the Adapter which will be used by most customers which have multiple systems which use the same translation source, or when the translation source must be system-independent. TMX is a XML file based format, which is announced to be the next industry standard. XML files are human-readable, but the parsing is not as fast as with gettext files.

Most medium to large companies use this adapter. The files are human readable and system-independent.

2.1.7. Zend_Translate_Adapter_Qt

The Qt Adapter is for all customers which have TS files as their translation source which are made by QtLinguist. QT is a XML file based format. XML files are human-readable, but the parsing is not as fast as with gettext files.

Several big players have build software upon the QT framework. The files are human readable and system-independent.

2.1.8. Zend_Translate_Adapter_Xliff

The Xliff Adapter is the Adapter which will be used by most customers which want to have XML files but do not have tools for TMX. XLIFF is a XML file based format, which is related to TMX but simpler as it does not support all possibilities of it. XML files are human-readable, but the parsing is not as fast as with gettext files.

Most medium companies use this adapter. The files are human readable and system-independent.

2.1.9. Zend_Translate_Adapter_XmlTm

The XmlTm Adapter is the Adapter which will be used by customers which do their layout themself. XmlTm is a format which allows the complete html source to be included in the translation source, so the translation is coupled with the layout. XLIFF is a XML file based format, which is related to XLIFF but its not as simple to read.

This adapter should only be used when source files already exist. The files are human readable and system-independent.

2.2. Integrate self written Adapters

`Zend_Translate` allows you to integrate and use self written Adapter classes. They can be used like the standard Adapter classes which are already included within `Zend_Translate`.

Any adapter class you want to use with `Zend_Translate` must be a subclass of `Zend_Translate_Adapter`. `Zend_Translate_Adapter` is an abstract class which already defines all what is needed for translation. What has to be done by you, is the definition of the reader for translation datas.

The usage of the prefix "Zend" should be limited to Zend Framework. If you extend `Zend_Translate` with your own adapter, you should name it like "Company_Translate_Adapter_MyFormat". The following code shows an example of how a self written adapter class could be implemented:

```
try {
    $translate = new Zend_Translate('Company_Translate_Adapter_MyFormat',
                                    '/path/to/translate.xx',
                                    'en',
                                    array('myoption' => 'myvalue'));
} catch (Exception $e) {
    // File not found, no adapter class...
    // General failure
}
```

2.3. Speedup all Adapters

`Zend_Translate` allows you use internally `Zend_Cache` to fasten the loading of translation sources. This comes very handy if you use many translation sources or extensive source formats like XML based files.

To use caching you will just have to give a cache object to the `Zend_Translate::setCache()` method. It takes a instance of `Zend_Cache` as only parameter. Also if you use any adapter direct you can use the `setCache()` method. For convenience there are also the static methods `getCache()`, `hasCache()`, `clearCache()` and `removeCache()`.

```
$cache = Zend_Cache::factory('Core',
                             'File',
                             $frontendOptions,
                             $backendOptions);
Zend_Translate::setCache($cache);
$translate = new Zend_Translate('gettext',
                                '/path/to/translate.mo',
                                'en');
```

You must set the cache *before* you use or initiate any adapter or instance of `Zend_Translate`. Otherwise your translation source will not be cached until you add a new source with the `addTranslation()` method.

3. Using Translation Adapters

The next step is to use the adapter within your code.

Example 880. Example of single-language PHP code

```
print "Example\n";
print "=======\n";
print "Here is line one\n";
print "Today is the " . date("d.m.Y") . "\n";
print "\n";
print "Here is line two\n";
```

The example above shows some output with no support for translation. You probably write your code in your native language. Generally you need to translate not only the output, but also error and log messages.

The next step is to integrate Zend Translate into your existing code. Of course it is much easier if you had already written your code with translation in mind, than changing your code afterwards.

Example 881. Example of multi-lingual PHP code

```
$translate = new Zend_Translate('gettext', '/my/path/source-de.mo', 'de');
$translate->addTranslation('/path/to/translation/fr-source.mo', 'fr');

print $translate->_("Example") . "\n";
print "=======\n";
print $translate->_("Here is line one") . "\n";
printf($translate->_("Today is the %1\$s") . "\n", date('d.m.Y'));
print "\n";

$translate->setLocale('fr');
print $translate->_("Here is line two") . "\n";
```

Now let's take a deeper look into what has been done and how to integrate `Zend_Translate` into your own code.

Create a new `Zend_Translate` object and define the base adapter:

```
$translate = new Zend_Translate
    'gettext',
    '/path/to/translation/source-de.mo',
    'de'
);
```

In this example we chose the *Gettext Adapter*. We place our file *source-de.mo* into the directory */path/to/translation*. The gettext file will have German translation included, and we also added another language source for French.

The next step is to wrap all strings which are to be translated. The simplest approach is to have only simple strings or sentences like this:

```
print $translate->_("Example") . "\n";
print "=======\n";
print $translate->_("Here is line one") . "\n";
```

Some strings do not needed to be translated. The separating line is always a separating line, even in other languages.

Having data values integrated into a translation string is also supported through the use of embedded parameters.

```
printf($translate->_("Today is the %1\$s") . "\n", date("d.m.Y"));
```

Instead of `print()`, use the `printf()` function and replace all parameters with `%1\$s` parts. The first is `%1\$s`, the second is `%2\$s`, and so on. This way a translation can be done without knowing the exact value. In our example, the date is always the actual day, but the string can be translated without the knowledge of the actual day.

Each string is identified in the translation storage by a message ID. You can use message IDs instead of strings in your code, like this:

```
print $translate->_(1) . "\n";
print "=======\n";
print $translate->_(2) . "\n";
```

But doing this has several disadvantages:

You can not see what your code should output just by viewing your code.

Also you will have problems if some strings are not translated. You must always keep in mind how translation works. First `Zend_Translate` checks whether the specified language has a translation for the given message ID or string. If no translation string has been found it refers to the next lower level language as defined within `Zend_Locale`. So "*de_AT*" becomes "*de*" only. If there is no translation found for "*de*" either, then the original message is returned. This way you always have an output, even in case the message translation does not exist in your message storage. `Zend_Translate` never throws an error or exception when translating strings.

3.1. Translation Source Structures

Your next step is to create the translation sources for the languages you want to translate. Every adapter is created its own way as described here, but there are common features applicable for all adapters.

You have to decide where to store your translation source files. Using `Zend_Translate` you are not restricted in any way. The following structures are preferable:

- Single structured source

```
/application/
/languages/
/languages/lang.en
/languages/lang.de
/library/
```

  Positive: all source files for every languages are stored in one directory. No splitting of related files.

- Language structured source

```
/application/
/languages/
/languages/en/
/languages/en/first.en
/languages/en/second.en
/languages/de/
/languages/de/first.de
/languages/de/second.de
/library
```

Positive: Every language is stored in their own directories. Easy translation, as every language team has to translate only one directory. Also the usage of multiple files is transparent.

- Application structured source

```
/application/
/application/languages/
/application/languages/first.en
/application/languages/first.de
/application/languages/second.en
/application/languages/second.de
/library/
```

Positive: all source files for every language are stored in one directory. No splitting of related files.

Negative: having multiple files for the same language can be problematic.

- Gettext structured source

```
/application/
/languages/
/languages/de/
/languages/de/LC_MESSAGES/
/languages/de/LC_MESSAGES/first.mo
/languages/de/LC_MESSAGES/second.mo
/languages/en/
/languages/en/LC_MESSAGES/
/languages/en/LC_MESSAGES/first.mo
/languages/en/LC_MESSAGES/second.mo
/library/
```

Positive: existing gettext sources can be used without changing structure.

Negative: having sub-sub directories may be confusing for people who have not used gettext before.

- File structured source

```
/application/
/application/models/
/application/models/MyModel.php
/application/models/MyModel.de
/application/models/MyModel.en
/application/controllers/
/application/controllers/MyController.php
/application/controllers/MyController.de
/application/controllers/MyController.en
/library/
```

Positive: translation files are localted near their source.

Negative: too many and also small translation files result in being tendious to translate. Also every file has to be added as translation source.

Single structured and language structured source files are most usable for `Zend_Translate`.

So now, that we know which structure we want to have, we should create our translation source files.

4. Creating source files

Below you will find a description of the different source formats which can be used with `Zend_Translate`.

Note that most of the described formats should be created by using a tool or a generation process. These Tools and processes are not part of Zend Framework and for most of the described formats free tools are available.

4.1. Creating Array source files

Array source files are plain arrays. But you have to define them manually since there is no tool to aid this. But because they are so simple, it's the fastest way to look up messages if your code works as expected. It's generally the best adapter to get started with translation business.

```
$english = array(
    'message1' => 'message1',
    'message2' => 'message2',
    'message3' => 'message3');

$german = array(
    'message1' => 'Nachricht1',
    'message2' => 'Nachricht2',
    'message3' => 'Nachricht3');

$translate = new Zend_Translate('array', $english, 'en');
$translate->addTranslation($deutsch, 'de');
```

Since release 1.5 it is also supported to have arrays included within an external file. You just have to provide the filename and `Zend_Translate` will automatically include it and look for the array. See the following example for details:

```
// myarray.php
return array(
    'message1' => 'Nachricht1',
    'message2' => 'Nachricht2',
    'message3' => 'Nachricht3');

// controller
$translate = new Zend_Translate('array', '/path/to/myarray.php', 'de');
```

Files which do not return an array will fail to be included. Also any output within this file will be ignored and suppressed.

4.2. Creating Gettext source files

Gettext source files are created by GNU's gettext library. There are several free tools available that can parse your code files and create the needed gettext source files. These have the extension **.mo* and they are binary files. An open source tool for creating the files is poEdit. This tool also supports you during the translation process itself.

```
// We accume that we have created the mo files and translated them
$translate = new Zend_Translate('gettext', '/path/to/english.mo', 'en');
$translate->addTranslation('/path/to/german.mo', 'de');
```

As you can see the adapters are used exactly the same way, with one small difference: change *array* to *gettext*. All other usages are exactly the same as with all other adapters. With the gettext adapter you no longer have to be aware of gettext's standard directory structure, bindtextdomain and textdomain. Just give the path and filename to the adapter.

You should always use UTF-8 as source encoding. Otherwise you will have problems when using two different source encodings. E.g. one of your source files is encoded with ISO-8815-11 and another one with CP815. You can set only one encoding for your source file, so one of your languages probably will not display correctly.

UTF-8 is a portable format which supports all languages. When using UTF-8 for all languages, you will eliminate the problem of incompatible encodings.

Many gettext editors add adapter informations as empty translation string. This is the reason why empty strings are not translated when using the gettext adapter. Instead they are erased from the translation table and provided by the `getAdapterInfo()` method. It will return the adapter informations for all added gettext files as array using the filename as key.

```
// Getting the adapter informations
$translate = new Zend_Translate('gettext', '/path/to/english.mo', 'en');
print_r($translate->getAdapterInfo());
```

4.3. Creating TMX source files

TMX source files are a new industry standard. They have the advantage of being XML files and so they are readable by every editor and of course by humans. You can either create TMX files manually with a text editor, or you can use a special tool. But most tools currently available for creating TMX source files are not freely available.

Example 882. Example TMX file

```
<?xml version="1.0" ?>
<!DOCTYPE tmx SYSTEM "tmx14.dtd">
<tmx version="1.4">
 <header creationtoolversion="1.0.0" datatype="winres" segtype="sentence"
         adminlang="en-us" srclang="de-at" o-tmf="abc"
         creationtool="XYZTool" >
 </header>
 <body>
  <tu tuid='message1'>
   <tuv xml:lang="de"><seg>Nachricht1</seg></tuv>
   <tuv xml:lang="en"><seg>message1</seg></tuv>
  </tu>
  <tu tuid='message2'>
   <tuv xml:lang="en"><seg>message2</seg></tuv>
   <tuv xml:lang="de"><seg>Nachricht2</seg></tuv>
  </tu>
```

```
$translate = new Zend_Translate('tmx', 'path/to/mytranslation.tmx', 'en');
```

TMX files can have several languages within the same file. All other included languages are added automatically, so you do not have to call `addLanguage()`.

If you want to have only specified languages from the source translated you can set the option '`defined_language`' to `TRUE`. With this option you can add the wished languages explicitly with `addLanguage()`. The default value for this option is to add all languages.

4.4. Creating CSV source files

CSV source files are small and human readable. If your customers want to translate their own, you will probably use the CSV adapter.

Example 883. Example CSV file

```
#Example csv file
message1;Nachricht1
message2;Nachricht2
```

```
$translate = new Zend_Translate('csv', '/path/to/mytranslation.csv', 'de');
$translate->addTranslation('path/to/other.csv', 'fr');
```

There are three different options for the CSV adapter. You can set '`delimiter`', '`limit`' and '`enclosure`'.

The default delimiter for CSV string is '`;`', but with the option '`delimiter`' you can decide to use another one.

The default limit for a line within a CSV file is '`0`'. This means that the end of a CSV line is searched automatically. If you set '`limit`' to any value, then the CSV file will be read faster, but any line exceeding this limit will be truncated.

The default enclosure to use for CSV files is '"'. You can set a different one using the option '`enclosure`'.

Example 884. Second CSV file example

```
# Example CSV file
"message,1",Nachricht1
message2,"Nachricht,2"
"message3,",Nachricht3
```

```
$translate = new Zend_Translate(
    'csv',
    '/path/to/mytranslation.csv',
    'de',
    array('delimiter' => ','));

$translate->addTranslation('/path/to/other.csv', 'fr');
```

When you are using non-ASCII characters within your CSV file, like umlauts or UTF-8 chars, then you should always use enclosure. Omitting the enclosure can lead to missing characters in your translation.

4.5. Creating INI source files

INI source files are human readable but normally not very small as they also include other data beside translations. If you have data which shall be editable by your customers you can use the INI adapter.

Example 885. Example INI file

```
[Test]
;TestPage Comment
Message_1="Nachricht 1 (de)"
Message_2="Nachricht 2 (de)"
Message_3="Nachricht :3 (de)"
```

```
$translate = new Zend_Translate('ini', '/path/to/mytranslation.ini', 'de');
$translate->addTranslation('/path/to/other.ini', 'it');
```

INI files have several restrictions. If a value in the ini file contains any non-alphanumeric characters it needs to be enclosed in double-quotes ("). There are also reserved words which must not be used as keys for ini files. These include: `NULL`, `yes`, `no`, `TRUE`, and `FALSE`. Values `NULL`, `no` and `FALSE` results in `""`, `yes` and `TRUE` results in `1`. Characters `{}|&~![()"` must not be used anywhere in the key and have a special meaning in the value. Do not use them as it will produce unexpected behaviour.

5. Additional features for translation

There are several additional features which are supported by `Zend_Translate`. Read here for these additional informations.

5.1. Options for adapters

Options can be used with all adapters. Of course the options are different for all adapters. You can set options when you create the adapter. Actually there is one option which is available to all adapters: '`clear`' sets if translation data should be added to existing one or not. Standard behaviour is to add new translation data to existing one. But the translation data is only cleared for the selected language. So other languages remain untouched.

You can set options temporarily when using `addTranslation($data, $locale, array $options = array())` as third and optional parameter. And you can use the method `setOptions()` to set the options permanently.

Example 886. Using translation options

```
// define ':' as separator for the translation source files
$options = array('delimiter' => ':');
$translate = new Zend_Translate(
    'csv',
    '/path/to/mytranslation.csv',
    'de',
    $options);

...

// clear the defined language and use new translation data
$options = array('clear' => true);
$translate->addTranslation('/path/to/new.csv', 'fr', $options);
```

Here you can find all available options for the different adapters with a description of their usage:

Table 151. Options for translation adapters

Option	Adapter	Description	Default value
clear	all	If set to TRUE, the already read translations will be cleared. This can be used instead of creating a new instance when reading new translation data	*FALSE*
disableNotices	all	If set to TRUE, all notices regarding not available translations will be disabled. You should set this option to TRUE in production environment	*FALSE*
ignore	all	All directories and files beginning with this prefix will be ignored when searching for files. This value defaults to *'.'* which leads to the behavior that all hidden files will be ignored. Setting this value to `'tmp'` would mean that directories and files like `'tmpImages'` and `'tmpFiles'` would be ignored as well as all subsequent directories	*.*
log	all	An instance of `Zend_Log` where untranslated messages and notices will be written to	*NULL*
logMessage	all	The message which will be written into the log	*Untranslated message within '%locale%': %message%*
logUntranslated	all	When this option is set to TRUE, all message IDs which can not be translated will be written into the attached log	*FALSE*

Option	Adapter	Description	Default value
reload	all	When this option is set to TRUE, then files are reloaded into the cache. This option can be used to recreate the cache, or to add translations to already cached data after the cache has already been created.	*FALSE*
scan	all	If set to NULL, no scanning of the directory structure will be done. If set to Zend_Translate::LOCALE_DIRECTORY the locale will be detected within the directory. If set to Zend_Translate::LOCALE_FILENAME the locale will be detected within the filename. See Section 5.3, "Automatic source detection" for details	*NULL*
delimiter	Csv	Defines which sign is used as delimiter for separating source and translation	*;*
enclosure	Csv	Defines the enclosure character to be used. Defaults to a doublequote	*"*
length	Csv	Defines the maximum length of a csv line. When set to 0 it will be detected automatically	*0*
useId	Xliff	If you set this option to FALSE, then the source string will be used as message Id. The default for this option is TRUE, which means that the Id from the trans-unit element will be used as message Id	*TRUE*

When you want to have self defined options, you are also able to use them within all adapters. The `setOptions()` method can be used to define your option. `setOptions()` needs an array

with the options you want to set. If an given option exists it will be signed over. You can define as much options as needed as they will not be checked by the adapter. Just make sure not to overwrite any existing option which is used by an adapter.

To return the option you can use the `getOptions()` method. When `getOptions()` is called without a parameter it will return all options set. When the optional parameter is given you will only get the specified option.

5.2. Handling languages

When working with different languages there are a few methods which will be useful.

The `getLocale()` method can be used to get the currently set language. It can either hold an instance of `Zend_Locale` or the identifier of a locale.

The `setLocale()` method sets a new standard language for translation. This prevents the need of setting the optional language parameter more than once to the `translate()` method. If the given language does not exist, or no translation data is available for the language, `setLocale()` tries to downgrade to the language without the region if any was given. A language of `en_US` would be downgraded to `en`. When even the downgraded language can not be found an exception will be thrown.

The `isAvailable()` method checks if a given language is already available. It returns `TRUE` if data for the given language exist.

And finally the `getList()` method can be used to get all currently set languages for an adapter returned as array.

Example 887. Handling languages with adapters

```
// returns the currently set language
$actual = $translate->getLocale();

// you can use the optional parameter while translating
echo $translate->_("my_text", "fr");
// or set a new language
$translate->setLocale("fr");
echo $translate->_("my_text");
// refer to the base language
// fr_CH will be downgraded to fr
$translate->setLocale("fr_CH");
echo $translate->_("my_text");

// check if this language exist
if ($translate->isAvailable("fr")) {
    // language exists
}
```

5.2.1. Automatical handling of languages

Note that as long as you only add new translation sources with the `addTranslation()` method `Zend_Translate` will automatically set the best fitting language for your environment when you use one of the automatic locales which are '`auto`' or '`browser`'. So normally you will not need to call `setLocale()`. This should only be used in conjunction with automatic source detection.

The algorithm will search for the best fitting locale depending on the user's browser and your environment. See the following example for details:

Example 888. Automatically language detection

```
// Let's expect the browser returns these language settings:
// HTTP_ACCEPT_LANGUAGE = "de_AT=1;fr=1;en_US=0.8";

// Example 1:
// When no fitting language is found, the message ID is returned
$translate = new Zend_Translate(
    'gettext',
    'my_it.mo',
    'auto',
    array('scan' => Zend_Translate::LOCALE_FILENAME));

// Example 2:
// Best found fitting language is 'fr'
$translate = new Zend_Translate(
    'gettext',
    'my_fr.mo',
    'auto',
    array('scan' => Zend_Translate::LOCALE_FILENAME));

// Example 3:
// Best found fitting language is 'de' ('de_AT' will be degraded)
$translate = new Zend_Translate(
    'gettext',
    'my_de.mo',
    'auto',
    array('scan' => Zend_Translate::LOCALE_FILENAME));

// Example 4:
// Returns 'it' as translation source and overrides the automatic settings
$translate = new Zend_Translate(
    'gettext',
    'my_it.mo',
    'auto',
    array('scan' => Zend_Translate::LOCALE_FILENAME));

$translate->addTranslation('my_ru.mo', 'ru');
$translate->setLocale('it_IT');
```

After setting a language manually with the `setLocale()` method the automatic detection will be switched off and overridden.

If you want to use it again, you can set the language *auto* with `setLocale()` which will reactivate the automatic detection for `Zend_Translate`.

Since Zend Framework 1.7.0 `Zend_Translate` also recognises an application wide locale. You can simply set a `Zend_Locale` instance to the registry like shown below. With this notation you can forget about setting the locale manually with each instance when you want to use the same locale multiple times.

```
// in your bootstrap file
$locale = new Zend_Locale();
Zend_Registry::set('Zend_Locale', $locale);

// default language when requested language is not available
$defaultlanguage = 'en';
```

```
// somewhere in your application
$translate = new Zend_Translate('gettext', 'my_de.mo');

if (!$translate->isAvailable($locale->getLanguage())) {
    // not available languages are rerouted to another language
    $translate->setLocale($defaultlanguage);
}

$translate->getLocale();
```

5.3. Automatic source detection

Zend_Translate can detect translation sources automatically. So you don't have to declare each source file manually. You can let Zend_Translate do this job and scan the complete directory structure for source files.

Automatic source detection is available since Zend Framework version 1.5 .

The usage is quite the same as initiating a single translation source with one difference. You must give a directory which has to be scanned instead a file.

Example 889. Scanning a directory structure for sources

```
// assuming we have the following structure
//  /language/
//  /language/login/login.tmx
//  /language/logout/logout.tmx
//  /language/error/loginerror.tmx
//  /language/error/logouterror.tmx

$translate = new Zend_Translate('tmx', '/language');
```

So Zend_Translate does not only search the given directory, but also all subdirectories for translation source files. This makes the usage quite simple. But Zend_Translate will ignore all files which are not sources or which produce failures while reading the translation data. So you have to make sure that all of your translation sources are correct and readable because you will not get any failure if a file is bogus or can not be read.

Depending on how deep your directory structure is and how much files are within this structure it can take a long time for Zend_Translate to complete.

In our example we have used the TMX format which includes the language to be used within the source. But many of the other source formats are not able to include the language within the file. Even this sources can be used with automatic scanning if you do some pre-requisits as described below:

5.3.1. Language through naming directories

One way to include automatic language detection is to name the directories related to the language which is used for the sources within this directory. This is the easiest way and is used for example within standard gettext implementations.

`Zend_Translate` needs the '`scan`' option to know that it should search the names of all directories for languages. See the following example for details:

Example 890. Directory scanning for languages

```
// assuming we have the following structure
//  /language/
//  /language/de/login/login.mo
//  /language/de/error/loginerror.mo
//  /language/en/login/login.mo
//  /language/en/error/loginerror.mo

$translate = new Zend_Translate(
    'gettext',
    '/language',
    null,
    array('scan' => Zend_Translate::LOCALE_DIRECTORY));
```

This works only for adapters which do not include the language within the source file. Using this option for example with TMX will be ignored. Also language definitions within the filename will be ignored when using this option.

You should be aware if you have several subdirectories under the same structure. Assuming we have a structure like `/language/module/de/en/file.mo`. In this case the path contains multiple strings which would be detected as locale. It could be either `de` or `en`. In such a case the behaviour is undefined and it is recommended to use file detection in such situations.

5.3.2. Language through filenames

Another way to detect the language automatically is to use special filenames. You can either name the complete file or parts of a file after the used language. To use this way of detection you will have to set the '`scan`' option at initiation. There are several ways of naming the sourcefiles which are described below:

Example 891. Filename scanning for languages

```
// assuming we have the following structure
//  /language/
//  /language/login/login_en.mo
//  /language/login/login_de.mo
//  /language/error/loginerror_en.mo
//  /language/error/loginerror_de.mo

$translate = new Zend_Translate(
    'gettext',
    '/language',
    null,
    array('scan' => Zend_Translate::LOCALE_FILENAME));
```

5.3.2.1. Complete filename

Having the whole file named after the language is the simplest way but only viable if you have only one file per language.

```
/languages/
/languages/en.mo
/languages/de.mo
/languages/es.mo
```

5.3.2.2. Extension of the file

Another simple way to use the extension of the file for language detection. But this may be confusing since you will no longer have an idea which extension the file originally had.

```
/languages/
/languages/view.en
/languages/view.de
/languages/view.es
```

5.3.2.3. Filename tokens

`Zend_Translate` is also capable of detecting the language if it is included within the filename. But if you go this way you will have to separate the language with a token. There are three supported tokens which can be used: a dot '.', an underscore '_', or a hyphen '-'.

```
/languages/
/languages/view_en.mo -> detects english
/languages/view_de.mo -> detects german
/languages/view_it.mo -> detects italian
```

The first found string delimited by a token which can be interpreted as a locale will be used. See the following example for details.

```
/languages/
/languages/view_en_de.mo -> detects english
/languages/view_en_es.mo -> detects english and overwrites the first file
/languages/view_it_it.mo -> detects italian
```

All three tokens are used to detect the locale. When the filename contains multiple tokens, the first found token depends on the order of the tokens which are used. See the following example for details.

```
/languages/
/languages/view_en-it.mo -> detects english because '_' will be used before '-'
/languages/view-en_it.mo -> detects italian because '_' will be used before '-'
/languages/view_en.it.mo -> detects italian because '.' will be used before '_'
```

5.4. Checking for translations

Normally text will be translated without any computation. But sometimes it is necessary to know if a text is translated or not, therefor the `isTranslated()` method can be used.

`isTranslated($messageId, $original = false, $locale = null)` takes the text you want to check as its first parameter, and as optional third parameter the locale for which you want to do the check. The optional second parameter declares whether translation is fixed to the declared language or a lower set of translations can be used. If you have a text which can be returned for 'en' but not for 'en_US' you will normally get the translation returned, but by setting `$original` to `TRUE`, `isTranslated()` will return `FALSE`.

Example 892. Checking if a text is translatable

```
$english = array(
    'message1' => 'Nachricht 1',
    'message2' => 'Nachricht 2',
    'message3' => 'Nachricht 3');

$translate = new Zend_Translate('array', $english, 'de_AT');

if ($translate->isTranslated('message1')) {
    print "'message1' can be translated";
}

if (!($translate->isTranslated('message1', true, 'de'))) {
    print "'message1' can not be translated to 'de'"
        . " as it's available only in 'de_AT'";
}

if ($translate->isTranslated('message1', false, 'de')) {
    print "'message1' can be translated in 'de_AT' as it falls back to 'de'";
}
```

5.5. How to log not found translations

When you have a bigger site or you are creating the translation files manually, you often have the problem that some messages are not translated. But there is an easy solution for you when you are using `Zend_Translate`.

You have to follow two or three simple steps. First, you have to create an instance of `Zend_Log`. Then you have to attach this instance to `Zend_Translate`. See the following example:

Example 893. Log translations

```
$translate = new Zend_Translate('gettext', $path, 'de');

// Create a log instance
$writer = new Zend_Log_Writer_Stream('/path/to/file.log');
$log    = new Zend_Log($writer);

// Attach it to the translation instance
$translate->setOptions(array(
    'log'             => $log,
    'logUntranslated' => true));

$translate->translate('unknown string');
```

Now you will have a new notice in the log: `Untranslated message within 'de': unknown string`.

You should note that any translation which can not be found will be logged. This means all translations when a user requests a language which is not supported. Also every request for a message which can not be translated will be logged. Be aware, that 100 people requesting the same translation, will result 100 logged notices.

This feature can not only be used to log messages but also to attach this untranslated messages into an empty translation file. To do so you will have to write your own log writer which writes the format you want to have and strips the prepending "Untranslated message".

You can also set the '`logMessage`' option when you want to have your own log message. Use the '`%message%`' token for placing the messageId within your log message, and the '`%locale%`' token for the requested locale. See the following example for a self defined log message:

Example 894. Self defined log messages

```
$translate = new Zend_Translate('gettext', $path, 'de');

// Create a log instance
$writer = new Zend_Log_Writer_Stream('/path/to/file.log');
$log    = new Zend_Log($writer);

// Attach it to the translation instance
$translate->setOptions(array(
    'log'             => $log,
    'logMessage'      => "Missing '%message%' within locale '%locale%'",
    'logUntranslated' => true));

$translate->translate('unknown string');
```

5.6. Accessing source data

Sometimes it is useful to have access to the translation source data. Therefor the following two functions are provided.

The `getMessageIds($locale = null)` method returns all known message IDs as array.

The `getMessages($locale = null)` method returns the complete translation source as an array. The message ID is used as key and the translation data as value.

Both methods accept an optional parameter `$locale` which, if set, returns the translation data for the specified language. If this parameter is not given, the actual set language will be used. Keep in mind that normally all translations should be available in all languages. Which means that in a normal situation you will not have to set this parameter.

Additionally the `getMessages()` method can be used to return the complete translation dictionary using the pseudo-locale 'all'. This will return all available translation data for each added locale.

Attention: the returned array can be *very big*, depending on the number of added locales and the amount of translation data.

Example 895. Handling languages with adapters

```
// returns all known message IDs
$messageIds = $translate->getMessageIds();
print_r($messageIds);

// or just for the specified language
$messageIds = $translate->getMessageIds('en_US');
print_r($messageIds);

// returns all the complete translation data
$source = $translate->getMessages();
print_r($source);
```

6. Plural notations for Translation

As of Zend Framework 1.9, `Zend_Translate` is able to provide plural support. Professional translation will always have the need to use plurals as they are native in almost all languages.

So what are plurals? Generally spoken plurals are words which take into account numeric meanings. But as you may imagine each language has it's own definition of plurals. English, for example, supports one plural. We have a singular definition, for example "car", which means implicit one car, and we have the plural definition, "cars" which could mean more than one car but also zero cars. Other languages like russian or polish have more plurals and also the rules for plurals are different.

When you want to use plurals with `Zend_Translate` you must not need to know how the plurals are defined, only the translator must know as he does the translation. The only information you need to have is the language.

There are two ways for using plurals... the traditional one, which means that you use a own method, and a modern one, which allows you to do plural translations with the same method as normal translations.

6.1. Traditional plural translations

People who worked with gettext in past will be more common with traditional plural translations. There is a own `plural()` method which can be used for plural translations.

Example 896. Example of traditional plural translations

The `plural()` method accepts 4 parameters. The first parameter is the singular messageId, the second is the plural messageId and the third is the number or amount.

The number will be used to detect the plural which has to be returned. A optional forth parameter can be used to give a locale which will be used to return the translation.

```
$translate = new Zend_Translate('gettext', '/path/to/german.mo', 'de');
$translate->plural('Car', 'Cars', $number);
```

6.2. Modern plural translations

As traditional plural translations are restricted to source code using english plurals we added a new way for plural translations. It allows to use the same `translate()` for standard and for plural translations.

To use plural translations with `translate()` you need to give an array as messageId instead of an string. This array must have the original plural messageId's, then the amount and at last an optional locale when your given messageId's are not in english notation.

Example 897. Example of modern plural translations

When we want to translate the same plural definitions like in the previous our example would have to be defined like below.

```
$translate = new Zend_Translate('gettext', '/path/to/german.mo', 'de');
$translate->translate(array('Car', 'Cars', $number));
```

Using modern plural translations it is also possible to use any language as source for messageId's.

Example 898. Example of modern plural translations using a different source language

Let's expect we want to use russian and let's also expect that the given messageId's are russian and not english.

```
$translate = new Zend_Translate('gettext', '/path/to/german.mo', 'de');
$translate->translate(array('Car',
                            'Cars first plural',
                            'Cars second plural',
                            $number,
                            'ru'));
```

As you can see you can give more than just the one english plural. But you must give the source language in this case so `Zend_Translate` knows which plural rules it has to apply.

When you omit the plural language then english will be used per default and any additional plural definition will be ignored.

6.3. Plural source files

Not all source formats support plural forms. Look into this list for details:

Table 152. Plural support

Adapter	Plurals supported		
Array	*yes*		
Csv	*yes*		
Gettext	*yes*		
Ini	*no*		
Qt	*no*		
Tbx	*no*		
Tmx	*no*		
Xliff	*no*		
XmlTm	*no*		

Below you can find examples of plural defined source files.

6.3.1. Array source with plural definitions

An array with plural definitions has to look like the following example.

```
array(
    'plural_0' => array(
        'plural_0 (ru)',
        'plural_1 (ru)',
        'plural_2 (ru)',
        'plural_3 (ru)'
    ),
    'plural_1' => ''
);
```

In the above example `plural_0` and `plural_1` are the plural definitions from the source code. And the array at `plural_0` has all translated plural forms available. Take a look at the following example with real content and translation from english source to german.

```
array(
    'Car' => array(
        'Auto',
        'Autos'
    ),
    'Cars' => ''
);
```

When your translated language supports more plural forms then simply add them to the array below the first plural form. When your source language suppors more plural forms, than simply add a new empty translation.

6.3.2. Csv source with plural definitions

A csv file with plural definitions has to look like the following example.

```
"plural_0";"plural_0 (ru)";"plural_1 (ru)";"plural_2 (ru)";"plural_3 (ru)"
"plural_1";
```

All translated plural forms have to be added after the first plural of the source language. And all further plural forms of the source language have to be added below but without translation. Note that you must add a delimiter to empty source plurals.

6.3.3. Gettext source with plural definitions

Gettext sources support plural forms out of the box. There is no need for adoption as the `*.mo` file will contain all necessary data.

Note that gettext does not support the usage of source languages which are not using english plural forms. When you plan to use a source language which supports other plural forms like russian for example, then you can not use gettext sources.

6.4. Custom plural rules

In rare cases it could be useful to be able to define own plural rules. See chinese for example. This language defines two plural rules. Per default it does not use plurals. But in rare cases it uses a rule like *(number == 1) ? 0 : 1*.

Also when you want to use a language which has no known plural rules, and would want to define your own rules.

This can be done by using `Zend_Translate_Plural::setRule()`. The method expects two parameters which must be given. A rule, which is simply a callback to a self defined method. And a locale for which the rule will be used.

Your rule could look like this:

```
public function MyRule($number) {
    return ($number == 10) ? 0 : 1;
}
```

As you see, your rule must accept one parameter. It is the number which you will use to return which plural the translation has to use. In our example we defined that when we get a '10' the plural definition 0 has to be used, in all other cases we're using 1.

Your rules can be as simple or as complicated as you want. You must only return an integer value. The plural definition 0 stands for singular translation, and 1 stands for the first plural rule.

To activate your rule, and to link it to the wished locale, you have to call it like this:

```
Zend_Translate_Plural::setPlural('MyPlural', 'zh');
```

Now we linked our plural definition to the chinese language.

You can define one plural rule for every language. But you should be aware that you set the plural rules before you are doing translations.

Define custom plurals only when needed

`Zend_Translate` defines plurals for most known languages. You should not define own plurals when you are not in need. The default rules work most of time.

Zend_Uri

1. Zend_Uri

1.1. Overview

`Zend_Uri` is a component that aids in manipulating and validating Uniform Resource Identifiers (URIs). `Zend_Uri` exists primarily to service other components, such as `Zend_Http_Client`, but is also useful as a standalone utility.

URIs always begin with a scheme, followed by a colon. The construction of the many different schemes varies significantly. The `Zend_Uri` class provides a factory that returns a subclass of itself which specializes in each scheme. The subclass will be named `Zend_Uri_<scheme>`, where `<scheme>` is the scheme, lowercased with the first letter capitalized. An exception to this rule is HTTPS, which is also handled by `Zend_Uri_Http`.

1.2. Creating a New URI

`Zend_Uri` will build a new URI from scratch if only a scheme is passed to `Zend_Uri::factory()`.

Example 899. Creating a New URI with Zend_Uri::factory()

```
// To create a new URI from scratch, pass only the scheme.
$uri = Zend_Uri::factory('http');

// $uri instanceof Zend_Uri_Http
```

To create a new URI from scratch, pass only the scheme to `Zend_Uri::factory()`[1]. If an unsupported scheme is passed, a `Zend_Uri_Exception` will be thrown.

If the scheme or URI passed is supported, `Zend_Uri::factory()` will return a subclass of itself that specializes in the scheme to be created.

1.3. Manipulating an Existing URI

To manipulate an existing URI, pass the entire URI to `Zend_Uri::factory()`.

Example 900. Manipulating an Existing URI with Zend_Uri::factory()

```
// To manipulate an existing URI, pass it in.
$uri = Zend_Uri::factory('http://www.zend.com');

// $uri instanceof Zend_Uri_Http
```

The URI will be parsed and validated. If it is found to be invalid, a `Zend_Uri_Exception` will be thrown immediately. Otherwise, `Zend_Uri::factory()` will return a subclass of itself that specializes in the scheme to be manipulated.

1.4. URI Validation

The `Zend_Uri::check()` method can only be used if validation of an existing URI is needed.

[1] At the time of writing, `Zend_Uri` only supports the HTTP and HTTPS schemes.

Example 901. URI Validation with Zend_Uri::check()

```
// Validate whether a given URI is well formed
$valid = Zend_Uri::check('http://uri.in.question');

// $valid is TRUE for a valid URI, or FALSE otherwise.
```

`Zend_Uri::check()` returns a boolean, which is more convenient than using `Zend_Uri::factory()` and catching the exception.

1.4.1. Allowing "Unwise" characters in URIs

By default, `Zend_Uri` will not accept the following characters: `"{"`, `"}"`, `"|"`, `"\"`, `"^"`, `"`"`. These characters are defined by the RFC as "unwise" and invalid; however, many implementations do accept these characters as valid.

`Zend_Uri` can be set to accept these "unwise" characters by setting the 'allow_unwise' option to boolean `TRUE` using `Zend_Uri::setConfig()`:

Example 902. Allowing special characters in URIs

```
// Contains '|' symbol
// Normally, this would return false:
$valid = Zend_Uri::check('http://example.com/?q=this|that');

// However, you can allow "unwise" characters
Zend_Uri::setConfig(array('allow_unwise' => true));

// will return 'true'
$valid = Zend_Uri::check('http://example.com/?q=this|that');

// Reset the 'allow_unwise' value to the default FALSE
Zend_Uri::setConfig(array('allow_unwise' => false));
```

`Zend_Uri::setConfig()` sets configuration options globally. It is recommended to reset the 'allow_unwise' option to 'FALSE', like in the example above, unless you are certain you want to always allow unwise characters globally.

1.5. Common Instance Methods

Every instance of a `Zend_Uri` subclass (e.g. `Zend_Uri_Http`) has several instance methods that are useful for working with any kind of URI.

1.5.1. Getting the Scheme of the URI

The scheme of the URI is the part of the URI that precedes the colon. For example, the scheme of `http://www.zend.com` is `http`.

Example 903. Getting the Scheme from a Zend_Uri_* Object

```
$uri = Zend_Uri::factory('http://www.zend.com');

$scheme = $uri->getScheme();  // "http"
```

The `getScheme()` instance method returns only the scheme part of the URI object.

1.5.2. Getting the Entire URI

Example 904. Getting the Entire URI from a Zend_Uri_* Object

```
$uri = Zend_Uri::factory('http://www.zend.com');

echo $uri->getUri();  // "http://www.zend.com"
```

The `getUri()` method returns the string representation of the entire URI.

1.5.3. Validating the URI

`Zend_Uri::factory()` will always validate any URI passed to it and will not instantiate a new `Zend_Uri` subclass if the given URI is found to be invalid. However, after the `Zend_Uri` subclass is instantiated for a new URI or an existing valid one, it is possible that the URI can later become invalid after it is manipulated.

Example 905. Validating a Zend_Uri_* Object

```
$uri = Zend_Uri::factory('http://www.zend.com');

$isValid = $uri->valid();  // TRUE
```

The `valid()` instance method provides a means to check that the URI object is still valid.

Zend_Validate

1. Introduction

The Zend_Validate component provides a set of commonly needed validators. It also provides a simple validator chaining mechanism by which multiple validators may be applied to a single datum in a user-defined order.

1.1. What is a validator?

A validator examines its input with respect to some requirements and produces a boolean result - whether the input successfully validates against the requirements. If the input does not meet the requirements, a validator may additionally provide information about which requirement(s) the input does not meet.

For example, a web application might require that a username be between six and twelve characters in length and may only contain alphanumeric characters. A validator can be used for ensuring that usernames meet these requirements. If a chosen username does not meet one or both of the requirements, it would be useful to know which of the requirements the username fails to meet.

1.2. Basic usage of validators

Having defined validation in this way provides the foundation for Zend_Validate_Interface, which defines two methods, isValid() and getMessages(). The isValid() method performs validation upon the provided value, returning TRUE if and only if the value passes against the validation criteria.

If isValid() returns FALSE, the getMessages() returns an array of messages explaining the reason(s) for validation failure. The array keys are short strings that identify the reasons for validation failure, and the array values are the corresponding human-readable string messages. The keys and values are class-dependent; each validation class defines its own set of validation failure messages and the unique keys that identify them. Each class also has a const definition that matches each identifier for a validation failure cause.

The getMessages() methods return validation failure information only for the most recent isValid() call. Each call to isValid() clears any messages and errors caused by a previous isValid() call, because it's likely that each call to isValid() is made for a different input value.

The following example illustrates validation of an e-mail address:

```
$validator = new Zend_Validate_EmailAddress();

if ($validator->isValid($email)) {
    // email appears to be valid
} else {
    // email is invalid; print the reasons
    foreach ($validator->getMessages() as $messageId => $message) {
        echo "Validation failure '$messageId': $message\n";
```

```
    }
}
```

1.3. Customizing messages

Validate classes provide a `setMessage()` method with which you can specify the format of a message returned by `getMessages()` in case of validation failure. The first argument of this method is a string containing the error message. You can include tokens in this string which will be substituted with data relevant to the validator. The token *%value%* is supported by all validators; this is substituted with the value you passed to `isValid()`. Other tokens may be supported on a case-by-case basis in each validation class. For example, *%max%* is a token supported by `Zend_Validate_LessThan`. The `getMessageVariables()` method returns an array of variable tokens supported by the validator.

The second optional argument is a string that identifies the validation failure message template to be set, which is useful when a validation class defines more than one cause for failure. If you omit the second argument, `setMessage()` assumes the message you specify should be used for the first message template declared in the validation class. Many validation classes only have one error message template defined, so there is no need to specify which message template you are changing.

```
$validator = new Zend_Validate_StringLength(8);

$validator->setMessage(
    'The string \'%value%\' is too short; it must be at least %min% ' .
    'characters',
    Zend_Validate_StringLength::TOO_SHORT);

if (!$validator->isValid('word')) {
    $messages = $validator->getMessages();
    echo current($messages);

    // "The string 'word' is too short; it must be at least 8 characters"
}
```

You can set multiple messages using the `setMessages()` method. Its argument is an array containing key/message pairs.

```
$validator = new Zend_Validate_StringLength(array('min' => 8, 'max' => 12));

$validator->setMessages( array(
    Zend_Validate_StringLength::TOO_SHORT =>
        'The string \'%value%\' is too short',
    Zend_Validate_StringLength::TOO_LONG  =>
        'The string \'%value%\' is too long'
));
```

If your application requires even greater flexibility with which it reports validation failures, you can access properties by the same name as the message tokens supported by a given validation class. The value property is always available in a validator; it is the value you specified as the argument of `isValid()`. Other properties may be supported on a case-by-case basis in each validation class.

```
$validator = new Zend_Validate_StringLength(array('min' => 8, 'max' => 12));
```

```
if (!validator->isValid('word')) {
    echo 'Word failed: '
         . $validator->value
         . '; its length is not between '
         . $validator->min
         . ' and '
         . $validator->max
         . "\n";
}
```

1.4. Using the static is() method

If it's inconvenient to load a given validation class and create an instance of the validator, you can use the static method `Zend_Validate::is()` as an alternative invocation style. The first argument of this method is a data input value, that you would pass to the `isValid()` method. The second argument is a string, which corresponds to the basename of the validation class, relative to the `Zend_Validate` namespace. The `is()` method automatically loads the class, creates an instance, and applies the `isValid()` method to the data input.

```
if (Zend_Validate::is($email, 'EmailAddress')) {
    // Yes, email appears to be valid
}
```

You can also pass an array of constructor arguments, if they are needed for the validator.

```
if (Zend_Validate::is($value, 'Between', array('min' => 1, 'max' => 12))) {
    // Yes, $value is between 1 and 12
}
```

The `is()` method returns a boolean value, the same as the `isValid()` method. When using the static `is()` method, validation failure messages are not available.

The static usage can be convenient for invoking a validator ad hoc, but if you have the need to run a validator for multiple inputs, it's more efficient to use the non-static usage, creating an instance of the validator object and calling its `isValid()` method.

Also, the `Zend_Filter_Input` class allows you to instantiate and run multiple filter and validator classes on demand to process sets of input data. See Section 5, “Zend_Filter_Input”.

1.4.1. Namespaces

When working with self defined validators you can give a forth parameter to `Zend_Validate::is()` which is the namespace where your validator can be found.

```
if (Zend_Validate::is($value, 'MyValidator', array('min' => 1, 'max' => 12),
                      array('FirstNamespace', 'SecondNamespace')) {
    // Yes, $value is ok
}
```

`Zend_Validate` allows also to set namespaces as default. This means that you can set them once in your bootstrap and have not to give them again for each call of `Zend_Validate::is()`. The following code snippet is identical to the above one.

```
Zend_Validate::setDefaultNamespaces(array('FirstNamespace', 'SecondNamespace'));
if (Zend_Validate::is($value, 'MyValidator', array('min' => 1, 'max' => 12)) {
```

```
    // Yes, $value is ok
}

if (Zend_Validate::is($value, 'OtherValidator', array('min' => 1, 'max' => 12)) {
    // Yes, $value is ok
}
```

For your convinience there are following methods which allow the handling of namespaces:

- *`Zend_Validate::getDefaultNamespaces()`*: Returns all set default namespaces as array.
- *`Zend_Validate::setDefaultNamespaces()`*: Sets new default namespaces and overrides any previous set. It accepts either a string for a single namespace of an array for multiple namespaces.
- *`Zend_Validate::addDefaultNamespaces()`*: Adds additional namespaces to already set ones. It accepts either a string for a single namespace of an array for multiple namespaces.
- *`Zend_Validate::hasDefaultNamespaces()`*: Returns TRUE when one or more default namespaces are set, and FALSE when no default namespaces are set.

1.5. Translating messages

Validate classes provide a `setTranslator()` method with which you can specify a instance of `Zend_Translate` which will translate the messages in case of a validation failure. The `getTranslator()` method returns the set translator instance.

```
$validator = new Zend_Validate_StringLength(array('min' => 8, 'max' => 12));
$translate = new Zend_Translate(
    'array',
    array(Zend_Validate_StringLength::TOO_SHORT => 'Translated \'%value%\''),
    'en'
);

$validator->setTranslator($translate);
```

With the static `setDefaultTranslator()` method you can set a instance of `Zend_Translate` which will be used for all validation classes, and can be retrieved with `getDefaultTranslator()`. This prevents you from setting a translator manually for all validator classes, and simplifies your code.

```
$translate = new Zend_Translate(
    'array',
    array(Zend_Validate_StringLength::TOO_SHORT => 'Translated \'%value%\''),
    'en'
);
Zend_Validate::setDefaultTranslator($translate);
```

When you have set an application wide locale within your registry, then this locale will be used as default translator.

Sometimes it is necessary to disable the translator within a validator. To archive this you can use the `setDisableTranslator()` method, which accepts a boolean parameter, and `translatorIsDisabled()` to get the set value.

```
$validator = new Zend_Validate_StringLength(array('min' => 8, 'max' => 12));
if (!$validator->isTranslatorDisabled()) {
    $validator->setDisableTranslator();
}
```

It is also possible to use a translator instead of setting own messages with `setMessage()`. But doing so, you should keep in mind, that the translator works also on messages you set your own.

2. Standard Validation Classes

Zend Framework comes with a standard set of validation classes, which are ready for you to use.

2.1. Alnum

Returns TRUE if and only if `$value` contains only alphabetic and digit characters. This validator includes an option to also consider white space characters as valid.

The alphabetic characters mean characters that makes up words in each language. However, the English alphabet is treated as the alphabetic characters in following languages: Chinese, Japanese, Korean. The language is specified by `Zend_Locale`.

2.2. Alpha

Returns TRUE if and only if `$value` contains only alphabetic characters. This validator includes an option to also consider white space characters as valid.

2.3. Barcode

`Zend_Validate_Barcode` allows you to check if a given value can be represented as barcode.

`Zend_Validate_Barcode` supports multiple barcode standards and can be extended with proprietary barcode implementations very easily. The following barcode standards are supported:

- *CODE25*: Often called "two of five" or "Code25 Industrial".

  This barcode has no length limitation. It supports only digits, and the last digit can be an optional checksum which is calculated with modulo 10. This standard is very old and nowadays not often used. Common usecases are within the industry.

- *CODE25INTERLEAVED*: Often called "Code 2 of 5 Interleaved".

  This standard is a variant of CODE25. It has no length limitation, but it must contain an even amount of characters. It supports only digits, and the last digit can be an optional checksum which is calculated with modulo 10. It is used worldwide and common on the market.

- *CODE39*: CODE39 is one of the oldest available codes.

  This barcode has a variable length. It supports digits, upper cased alphabetical characters and 7 special characters like whitespace, point and dollar sign. It can have an optional checksum which is calculated with modulo 43. This standard is used worldwide and common within the industry.

- *CODE39EXT*: CODE39EXT is an extension of CODE39.

  This barcode has the same properties as CODE39. Additionally it allows the usage of all 128 ASCII characters. This standard is used worldwide and common within the industry.

- *CODE93*: CODE93 is the successor of CODE39.

  This barcode has a variable length. It supports digits, alphabetical characters and 7 special characters. It has an optional checksum which is calculated with modulo 47 and contains 2 characters. This standard produces a denser code than CODE39 and is more secure.

- *CODE93EXT*: CODE93EXT is an extension of CODE93.

  This barcode has the same properties as CODE93. Additionally it allows the usage of all 128 ASCII characters. This standard is used worldwide and common within the industry.

- *EAN2*: EAN is the shortcut for "European Article Number".

  These barcode must have 2 characters. It supports only digits and does not have a checksum. This standard is mainly used as addition to EAN13 (ISBN) when printed on books.

- *EAN5*: EAN is the shortcut for "European Article Number".

  These barcode must have 5 characters. It supports only digits and does not have a checksum. This standard is mainly used as addition to EAN13 (ISBN) when printed on books.

- *EAN8*: EAN is the shortcut for "European Article Number".

  These barcode can have 7 or 8 characters. It supports only digits. When it has a length of 8 characters it includes a checksum. This standard is used worldwide but has a very limited range. It can be found on small articles where a longer barcode could not be printed.

- *EAN12*: EAN is the shortcut for "European Article Number".

  This barcode must have a length of 12 characters. It supports only digits, and the last digit is always a checksum which is calculated with modulo 10. This standard is used within the USA and common on the market. It has been superceded by EAN13.

- *EAN13*: EAN is the shortcut for "European Article Number".

  This barcode must have a length of 13 characters. It supports only digits, and the last digit is always a checksum which is calculated with modulo 10. This standard is used worldwide and common on the market.

- *EAN14*: EAN is the shortcut for "European Article Number".

  This barcode must have a length of 14 characters. It supports only digits, and the last digit is always a checksum which is calculated with modulo 10. This standard is used worldwide and common on the market. It is the successor for EAN13.

- *EAN18*: EAN is the shortcut for "European Article Number".

  This barcode must have a length of 18 characters. It support only digits. The last digit is always a checksum digit which is calculated with modulo 10. This code is often used for the identification of shipping containers.

- *GTIN12*: GTIN is the shortcut for "Global Trade Item Number".

This barcode uses the same standard as EAN12 and is its successor. It's commonly used within the USA.

- *GTIN13*: GTIN is the shortcut for "Global Trade Item Number".

  This barcode uses the same standard as EAN13 and is its successor. It is used worldwide by industry.

- *GTIN14*: GTIN is the shortcut for "Global Trade Item Number".

  This barcode uses the same standard as EAN14 and is its successor. It is used worldwide and common on the market.

- *IDENTCODE*: Identcode is used by Deutsche Post and DHL. It's an specialized implementation of Code25.

  This barcode must have a length of 12 characters. It supports only digits, and the last digit is always a checksum which is calculated with modulo 10. This standard is mainly used by the companies DP and DHL.

- *INTELLIGENTMAIL*: Intelligent Mail is a postal barcode.

  This barcode can have a length of 20, 25, 29 or 31 characters. It supports only digits, and contains no checksum. This standard is the successor of PLANET and POSTNET. It is mainly used by the United States Postal Services.

- *ISSN*: ISSN is the abbreviation for International Standard Serial Number.

  This barcode can have a length of 8 or 13 characters. It supports only digits, and the last digit must be a checksum digit which is calculated with modulo 11. It is used worldwide for printed publications.

- *ITF14*: ITF14 is the GS1 implementation of an Interleaved Two of Five bar code.

  This barcode is a special variant of Interleaved 2 of 5. It must have a length of 14 characters and is based on GTIN14. It supports only digits, and the last digit must be a checksum digit which is calculated with modulo 10. It is used worldwide and common within the market.

- *LEITCODE*: Leitcode is used by Deutsche Post and DHL. It's an specialized implementation of Code25.

  This barcode must have a length of 14 characters. It supports only digits, and the last digit is always a checksum which is calculated with modulo 10. This standard is mainly used by the companies DP and DHL.

- *PLANET*: Planet is the abbreviation for Postal Alpha Numeric Encoding Technique.

  This barcode can have a length of 12 or 14 characters. It supports only digits, and the last digit is always a checksum. This standard is mainly used by the United States Postal Services.

- *POSTNET*: Postnet is used by the US Postal Service.

  This barcode can have a length of 6, 7, 10 or 12 characters. It supports only digits, and the last digit is always a checksum. This standard is mainly used by the United States Postal Services.

- *ROYALMAIL*: Royalmail is used by Royal Mail.

This barcode has no defined length. It supports digits, uppercased letters, and the last digit is always a checksum. This standard is mainly used by Royal Mail for their Cleanmail Service. It is also called RM4SCC.

- *SSCC*: SSCC is the shortcut for "Serial Shipping Container Code".

  This barcode is a variant of EAN barcode. It must have a length of 18 characters and supports only digits. The last digit must be a checksum digit which is calculated with modulo 10. It is commonly used by the transport industry.

- *UPCA*: UPC is the shortcut for "Univeral Product Code".

  This barcode preceeded EAN13. It must have a length of 12 characters and supports only digits. The last digit must be a checksum digit which is calculated with modulo 10. It is commonly used within the USA.

- *UPCE*: UPCE is the short variant from UPCA.

  This barcode is a smaller variant of UPCA. It can have a length of 6, 7 or 8 characters and supports only digits. When the barcode is 8 chars long it includes a checksum which is calculated with modulo 10. It is commonly used with small products where a UPCA barcode would not fit.

2.3.1. Basic usage

To validate if a given string is a barcode you just need to know its type. See the following example for an EAN13 barcode:

```
$valid = new Zend_Validate_Barcode('EAN13');
if ($valid->isValid($input)) {
    // input appears to be valid
} else {
    // input is invalid
}
```

2.3.2. Optional checksum

Some barcodes can be provided with an optional checksum. These barcodes would be valid even without checksum. Still, when you provide a checksum, then you should also validate it. By default, these barcode types perform no checksum validation. By using the checksum option you can define if the checksum will be validated or ignored.

```
$valid = new Zend_Validate_Barcode(array(
    'adapter'  => 'EAN13',
    'checksum' => false,
));
if ($valid->isValid($input)) {
    // input appears to be valid
} else {
    // input is invalid
}
```

Reduced security by disabling checksum validation

By switching off checksum validation you will also reduce the security of the used barcodes. Additionally you should note that you can also turn off the

checksum validation for those barcode types which must contain a checksum value. Barcodes which would not be valid could then be returned as valid even if they are not.

2.3.3. Writing custom adapters

You may write custom barcode validators for usage with `Zend_Validate_Barcode`; this is often necessary when dealing with proprietary barcode types. To write your own barcode validator, you need the following information.

- *Length*: The length your barcode must have. It can have one of the following values:
  - *Integer*: A value greater 0, which means that the barcode must have this length.
  - *-1*: There is no limitation for the length of this barcode.
  - *"even"*: The length of this barcode must have a even amount of digits.
  - *"odd"*: The length of this barcode must have a odd amount of digits.
  - *array*: An array of integer values. The length of this barcode must have one of the set array values.
- *Characters*: A string which contains all allowed characters for this barcode. Also the integer value 128 is allowed, which means the first 128 characters of the ASCII table.
- *Checksum*: A string which will be used as callback for a method which does the checksum validation.

Your custom barcode validator must extend `Zend_Validate_Barcode_AdapterAbstract` or implement Zend_Validate_Barcode_AdapterInterface.

As an example, let's create a validator that expects an even number of characters that include all digits and the letters 'ABCDE', and which requires a checksum.

```
class My_Barcode_MyBar extends Zend_Validate_Barcode_AdapterAbstract
{
    protected $_length     = 'even';
    protected $_characters = '0123456789ABCDE';
    protected $_checksum   = '_mod66';

    protected function _mod66($barcode)
    {
        // do some validations and return a boolean
    }
}

$valid = new Zend_Validate_Barcode('My_Barcode_MyBar');
if ($valid->isValid($input)) {
    // input appears to be valid
} else {
    // input is invalid
}
```

2.4. Between

Returns TRUE if and only if `$value` is between the minimum and maximum boundary values. The comparison is inclusive by default (`$value` may equal a boundary value), though this may

be overridden in order to do a strict comparison, where `$value` must be strictly greater than the minimum and strictly less than the maximum.

2.5. Callback

`Zend_Validate_Callback` allows you to provide a callback with which to validate a given value.

2.5.1. Basic usage

The simplest usecase is to have a single function and use it as a callback. Let's expect we have the following function.

```
function myMethod($value)
{
    // some validation
    return true;
}
```

To use it within `Zend_Validate_Callback` you just have to call it this way:

```
$valid = new Zend_Validate_Callback('myMethod');
if ($valid->isValid($input)) {
    // input appears to be valid
} else {
    // input is invalid
}
```

2.5.2. Usage with closures

PHP 5.3 introduces closures, which are basically self-contained or *anonymous* functions. PHP considers closures another form of callback, and, as such, may be used with `Zend_Validate_Callback`. As an example:

```
$valid = new Zend_Validate_Callback(function($value){
    // some validation
    return true;
});

if ($valid->isValid($input)) {
    // input appears to be valid
} else {
    // input is invalid
}
```

2.5.3. Usage with class-based callbacks

Of course it's also possible to use a class method as callback. Let's expect we have the following class method:

```
class MyClass
{
    public function myMethod($value)
    {
        // some validation
        return true;
    }
```

```
}
```

The definition of the callback is in this case almost the same. You have just to create an instance of the class before the method and create an array describing the callback:

```
$object = new MyClass;
$valid = new Zend_Validate_Callback(array($object, 'myMethod'));
if ($valid->isValid($input)) {
    // input appears to be valid
} else {
    // input is invalid
}
```

You may also define a static method as a callback. Consider the following class definition and validator usage:

```
class MyClass
{
    public static function test($value)
    {
        // some validation
        return true;
    }
}

$valid = new Zend_Validate_Callback(array('MyClass, 'test'));
if ($valid->isValid($input)) {
    // input appears to be valid
} else {
    // input is invalid
}
```

Finally, if you are using PHP 5.3, you may define the magic method `__invoke()` in your class. If you do so, simply providing an instance of the class as the callback will also work:

```
class MyClass
{
    public function __invoke($value)
    {
        // some validation
        return true;
    }
}

$object = new MyClass();
$valid = new Zend_Validate_Callback($object);
if ($valid->isValid($input)) {
    // input appears to be valid
} else {
    // input is invalid
}
```

2.5.4. Adding options

`Zend_Validate_Callback` also allows the usage of options which are provided as additional arguments to the callback.

Consider the following class and method definition:

```
class MyClass
{
    function myMethod($value, $option)
    {
        // some validation
        return true;
    }
}
```

There are two ways to inform the validator of additional options: pass them in the constructor, or pass them to the `setOptions()` method.

To pass them to the constructor, you would need to pass an array containing two keys, "callback" and "options":

```
$valid = new Zend_Validate_Callback(array(
    'callback' => array('MyClass', 'myMethod'),
    'options'  => $option,
));

if ($valid->isValid($input)) {
    // input appears to be valid
} else {
    // input is invalid
}
```

Otherwise, you may pass them to the validator after instantiation:

```
$valid = new Zend_Validate_Callback(array('MyClass', 'myMethod'));
$valid->setOptions($option);

if ($valid->isValid($input)) {
    // input appears to be valid
} else {
    // input is invalid
}
```

When there are additional values given to `isValid()` then these values will be added immediately after `$value`.

```
$valid = new Zend_Validate_Callback(array('MyClass', 'myMethod'));
$valid->setOptions($option);

if ($valid->isValid($input, $additional)) {
    // input appears to be valid
} else {
    // input is invalid
}
```

When making the call to the callback, the value to be validated will always be passed as the first argument to the callback followed by all other values given to `isValid()`; all other options will follow it. The amount and type of options which can be used is not limited.

2.6. CreditCard

`Zend_Validate_CreditCard` allows you to validate if a given value could be a credit card number.

A creditcard contains several items of metadata, including a hologram, account number, logo, expiration date, security code and the card holder name. The algorithms for verifying the combination of metadata are only known to the issuing company, and should be verified with them for purposes of payment. However, it's often useful to know whether or not a given number actually falls within the ranges of possible numbers *prior* to performing such verification, and, as such, `Zend_Validate_CreditCard` simply verifies that the credit card number provided is well-formed.

For those cases where you have a service that can perform comprehensive verification, `Zend_Validate_CreditCard` also provides the ability to attach a service callback to trigger once the credit card number has been deemed valid; this callback will then be triggered, and its return value will determine overall validity.

The following issuing institutes are accepted:

- *American Express*

  China UnionPay

  Diners Club Card Blanche

  Diners Club International

  Diners Club US & Canada

  Discover Card

  JCB

  Laser

  Maestro

  MasterCard

  Solo

  Visa

  Visa Electron

Invalid institutes

The institutes *Bankcard* and *Diners Club enRoute* do not exist anymore. Therefore they are treated as invalid.

Switch has been rebranded to *Visa* and is therefore also treated as invalid.

2.6.1. Basic usage

There are several credit card institutes which can be validated by `Zend_Validate_CreditCard`. Per default, all known institutes will be accepted. See the folowing example:

```
$valid = new Zend_Validate_CreditCard();
if ($valid->isValid($input)) {
```

```
    // input appears to be valid
} else {
    // input is invalid
}
```

The above example would validate against all known credit card institutes.

2.6.2. Accepting defined credit cards

Sometimes it is necessary to accept only defined credit card institutes instead of all; e.g., when you have a webshop which accepts only Visa and American Express cards. `Zend_Validate_CreditCard` allows you to do exactly this by limiting it to exactly these institutes.

To use a limitation you can either provide specific institutes at initiation, or afterwards by using `setType()`. Each can take several arguments.

You can provide a single institute:

```
$valid = new Zend_Validate_CreditCard(
    Zend_Validate_CreditCard::AMERICAN_EXPRESS
);
```

When you want to allow multiple institutes, then you can provide them as array:

```
$valid = new Zend_Validate_CreditCard(array(
    Zend_Validate_CreditCard::AMERICAN_EXPRESS,
    Zend_Validate_CreditCard::VISA
));
```

And as with all validators, you can also pass an associative array of options or an instance of `Zend_Config`. In this case you have to provide the institutes with the type array key as simulated here:

```
$valid = new Zend_Validate_CreditCard(array(
    'type' => array(Zend_Validate_CreditCard::AMERICAN_EXPRESS)
));
```

Table 153. Constants for credit card institutes

Institute	Constant			
American Express	`AMERICAN_EXPRESS`			
China UnionPay	`UNIONPAY`			
Diners Club Card Blanche	`DINERS_CLUB`			
Diners Club International	`DINERS_CLUB`			
Diners Club US & Canada	`DINERS_CLUB_US`			
Discover Card	`DISCOVER`			
JCB	`JCB`			

Institute	Constant			
Laser	LASER			
Maestro	MAESTRO			
MasterCard	MASTERCARD			
Solo	SOLO			
Visa	VISA			
Visa Electron	VISA			

You can also set or add institutes afterward instantiation by using the methods `setType()`, `addType()` and `getType()`.

```
$valid = new Zend_Validate_CreditCard();
$valid->setType(array(
    Zend_Validate_CreditCard::AMERICAN_EXPRESS,
    Zend_Validate_CreditCard::VISA
));
```

Default institute

When no institute is given at initiation then `ALL` will be used, which sets all institutes at once.

In this case the usage of `addType()` is useless because all institutes are already added.

2.6.3. Validation by using foreign APIs

As said before `Zend_Validate_CreditCard` will only validate the credit card number. Fortunately, some institutes provide online APIs which can validate a credit card number by using algorithms which are not available to the public. Most of these services are paid services. Therefore, this check is deactivated per default.

When you have access to such an API, then you can use it as an addon for `Zend_Validate_CreditCard` and increase the security of the validation.

To do so, you simply need to give a callback which will be called when the generic validation has passed. This prevents the API from being called for invalid numbers, which increases the performance of the application.

`setService()` sets a new service, and `getService()` returns the set service. As a configuration option, you can give the array key 'service' at initiation. For details about possible options take a look into Callback.

```
// Your service class
class CcService
{
    public function checkOnline($cardnumber, $types)
    {
        // some online validation
    }
}

// The validation
```

```
$service = new CcService();
$valid   = new Zend_Validate_CreditCard(Zend_Validate_CreditCard::VISA);
$valid->setService(array($service, 'checkOnline'));
```

As you can see the callback method will be called with the creditcard number as the first parameter, and the accepted types as the second parameter.

2.7. Ccnum

Returns TRUE if and only if `$value` follows the Luhn algorithm (mod-10 checksum) for credit card numbers.

The `Ccnum` validator has been deprecated in favor of the `CreditCard` validator. For security reasons you should use CreditCard instead of Ccnum.

2.8. Date

Returns TRUE if `$value` is a valid date of the format 'YYYY-MM-DD'. If the optional locale option is set then the date will be validated according to the set locale. And if the optional format option is set this format is used for the validation. for details about the optional parameters see Zend_Date::isDate().

2.9. Db_RecordExists and Db_NoRecordExists

`Zend_Validate_Db_RecordExists` and `Zend_Validate_Db_NoRecordExists` provide a means to test whether a record exists in a given table of a database, with a given value.

2.9.1. Basic usage

An example of basic usage of the validators:

```
//Check that the email address exists in the database
$validator = new Zend_Validate_Db_RecordExists(
    array(
        'table' => 'users',
        'field' => 'emailaddress'
    )
);

if ($validator->isValid($emailaddress)) {
    // email address appears to be valid
} else {
    // email address is invalid; print the reasons
    foreach ($validator->getMessages() as $message) {
        echo "$message\n";
    }
}
```

The above will test that a given email address is in the database table. If no record is found containing the value of `$emailaddress` in the specified column, then an error message is displayed.

```
//Check that the username is not present in the database
$validator = new Zend_Validate_Db_NoRecordExists(
    array(
```

```
        'table' => 'users',
        'field' => 'username'
    )
);
if ($validator->isValid($username)) {
    // username appears to be valid
} else {
    // username is invalid; print the reason
    $messages = $validator->getMessages();
    foreach ($messages as $message) {
        echo "$message\n";
    }
}
```

The above will test that a given username is not in the database table. If a record is found containing the value of `$username` in the specified column, then an error message is displayed.

2.9.2. Excluding records

`Zend_Validate_Db_RecordExists` and `Zend_Validate_Db_NoRecordExists` also provide a means to test the database, excluding a part of the table, either by providing a where clause as a string, or an array with the keys "field" and "value".

When providing an array for the exclude clause, the `!=` operator is used, so you can check the rest of a table for a value before altering a record (for example on a user profile form)

```
//Check no other users have the username
$user_id   = $user->getId();
$validator = new Zend_Validate_Db_NoRecordExists(
    array(
        'table' => 'users',
        'field' => 'username',
        'exclude' => array(
            'field' => 'id',
            'value' => $user_id
        )
    )
);

if ($validator->isValid($username)) {
    // username appears to be valid
} else {
    // username is invalid; print the reason
    $messages = $validator->getMessages();
    foreach ($messages as $message) {
        echo "$message\n";
    }
}
```

The above example will check the table to ensure no records other than the one where `id = $user_id` contains the value $username.

You can also provide a string to the exclude clause so you can use an operator other than `!=`. This can be useful for testing against composite keys.

```
$post_id   = $post->getId();
$clause    = $db->quoteInto('post_id = ?', $category_id);
$validator = new Zend_Validate_Db_RecordExists(
```

```
    array(
        'table'   => 'posts_categories',
        'field'   => 'post_id',
        'exclude' => $clause
    )
);

if ($validator->isValid($username)) {
    // username appears to be valid
} else {
    // username is invalid; print the reason
    $messages = $validator->getMessages();
    foreach ($messages as $message) {
        echo "$message\n";
    }
}
```

The above example will check the `posts_categories` table to ensure that a record with the `post_id` has a value matching `$category_id`

2.9.3. Database Adapters

You can also specify an adapter. This will allow you to work with applications using multiple database adapters, or where you have not set a default adapter. As in the example below:

```
$validator = new Zend_Validate_Db_RecordExists(
    array(
        'table' => 'users',
        'field' => 'id',
        'adapter' => $dbAdapter
    )
);
```

2.9.4. Database Schemas

You can specify a schema within your database for adapters such as PostgreSQL and DB/2 by simply supplying an array with `table` and `schema` keys. As in the example below:

```
$validator = new Zend_Validate_Db_RecordExists(
    array(
        'table'  => 'users',
        'schema' => 'my',
        'field'  => 'id'
    )
);
```

2.10. Digits

Returns `TRUE` if and only if `$value` only contains digit characters.

2.11. EmailAddress

`Zend_Validate_EmailAddress` allows you to validate an email address. The validator first splits the email address on local-part @ hostname and attempts to match these against known specifications for email addresses and hostnames.

2.11.1. Basic usage

A basic example of usage is below:

```
$validator = new Zend_Validate_EmailAddress();
if ($validator->isValid($email)) {
    // email appears to be valid
} else {
    // email is invalid; print the reasons
    foreach ($validator->getMessages() as $message) {
        echo "$message\n";
    }
}
```

This will match the email address `$email` and on failure populate `$validator->getMessages()` with useful error messages.

2.11.2. Options for validating Email Addresses

`Zend_Validate_EmailAddress` supports several options which can either be set at initiation, by giving an array with the related options, or afterwards, by using `setOptions()`. The following options are supported:

- *allow*: Defines which type of domain names are accepted. This option is used in conjunction with the hostname option to set the hostname validator. For more informations about possible values of this option, look at Hostname and possible `ALLOW*` constants. This option defaults to `ALLOW_DNS`.
- *hostname*: Sets the hostname validator with which the domain part of the email address will be validated.
- *mx*: Defines if the MX records from the server should be detected. If this option is defined to `TRUE` then the MX records are used to verify if the server accepts emails. This option defaults to `FALSE`.
- *deep*: Defines if the servers MX records should be verified by a deep check. When this option is set to `TRUE` then additionally to MX records also the A, A6 and `AAAA` records are used to verify if the server accepts emails. This option defaults to `FALSE`.
- *domain*: Defines if the domain part should be checked. When this option is set to `FALSE`, then only the local part of the email address will be checked. In this case the hostname validator will not be called. This option defaults to `TRUE`.

```
$validator = new Zend_Validate_EmailAddress();
$validator->setOptions(array('domain' => false));
```

2.11.3. Complex local parts

`Zend_Validate_EmailAddress` will match any valid email address according to RFC2822. For example, valid emails include `bob@domain.com`, `bob+jones@domain.us`, `"bob@jones"@domain.com` and `"bob jones"@domain.com`

Some obsolete email formats will not currently validate (e.g. carriage returns or a "\" character in an email address).

2.11.4. Validating only the local part

If you need `Zend_Validate_EmailAddress` to check only the local part of an email address, and want to disable validation of the hostname, you can set the domain option to FALSE. This forces `Zend_Validate_EmailAddress` not to validate the hostname part of the email address.

```
$validator = new Zend_Validate_EmailAddress();
$validator->setOptions(array('domain' => FALSE));
```

2.11.5. Validating different types of hostnames

The hostname part of an email address is validated against `Zend_Validate_Hostname`. By default only DNS hostnames of the form `domain.com` are accepted, though if you wish you can accept IP addresses and Local hostnames too.

To do this you need to instantiate `Zend_Validate_EmailAddress` passing a parameter to indicate the type of hostnames you want to accept. More details are included in `Zend_Validate_Hostname`, though an example of how to accept both DNS and Local hostnames appears below:

```
$validator = new Zend_Validate_EmailAddress(
                    Zend_Validate_Hostname::ALLOW_DNS |
                    Zend_Validate_Hostname::ALLOW_LOCAL);
if ($validator->isValid($email)) {
    // email appears to be valid
} else {
    // email is invalid; print the reasons
    foreach ($validator->getMessages() as $message) {
        echo "$message\n";
    }
}
```

2.11.6. Checking if the hostname actually accepts email

Just because an email address is in the correct format, it doesn't necessarily mean that email address actually exists. To help solve this problem, you can use MX validation to check whether an MX (email) entry exists in the DNS record for the email's hostname. This tells you that the hostname accepts email, but doesn't tell you the exact email address itself is valid.

MX checking is not enabled by default. To enable MX checking you can pass a second parameter to the `Zend_Validate_EmailAddress` constructor.

```
$validator = new Zend_Validate_EmailAddress(
    array(
        'allow' => Zend_Validate_Hostname::ALLOW_DNS,
        'mx'    => true
    )
);
```

MX Check under Windows

Within Windows environments MX checking is only available when PHP 5.3 or above is used. Below PHP 5.3 MX checking will not be used even if it's activated within the options.

Alternatively you can either pass TRUE or FALSE to `$validator->setValidateMx()` to enable or disable MX validation.

By enabling this setting network functions will be used to check for the presence of an MX record on the hostname of the email address you wish to validate. Please be aware this will likely slow your script down.

Sometimes validation for MX records returns FALSE, even if emails are accepted. The reason behind this behaviour is, that servers can accept emails even if they do not provide a MX record. In this case they can provide A, A6 or AAAA records. To allow `Zend_Validate_EmailAddress` to check also for these other records, you need to set deep MX validation. This can be done at initiation by setting the deep option or by using `setOptions()`.

```
$validator = new Zend_Validate_EmailAddress(
    array(
        'allow' => Zend_Validate_Hostname::ALLOW_DNS,
        'mx'    => true,
        'deep'  => true
    )
);
```

Performance warning

You should be aware that enabling MX check will slow down you script because of the used network functions. Enabling deep check will slow down your script even more as it searches the given server for 3 additional types.

Disallowed IP addresses

You should note that MX validation is only accepted for external servers. When deep MX validation is enabled, then local IP addresses like **192.168.*** or **169.254.*** are not accepted.

2.11.7. Validating International Domains Names

`Zend_Validate_EmailAddress` will also match international characters that exist in some domains. This is known as International Domain Name (IDN) support. This is enabled by default, though you can disable this by changing the setting via the internal `Zend_Validate_Hostname` object that exists within `Zend_Validate_EmailAddress`.

```
$validator->getHostnameValidator()->setValidateIdn(false);
```

More information on the usage of `setValidateIdn()` appears in the `Zend_Validate_Hostname` documentation.

Please note IDNs are only validated if you allow DNS hostnames to be validated.

2.11.8. Validating Top Level Domains

By default a hostname will be checked against a list of known TLDs. This is enabled by default, though you can disable this by changing the setting via the internal `Zend_Validate_Hostname` object that exists within `Zend_Validate_EmailAddress`.

```
$validator->getHostnameValidator()->setValidateTld(false);
```

More information on the usage of `setValidateTld()` appears in the `Zend_Validate_Hostname` documentation.

Please note TLDs are only validated if you allow DNS hostnames to be validated.

2.11.9. Setting messages

`Zend_Validate_EmailAddress` makes also use of `Zend_Validate_Hostname` to check the hostname part of a given email address. As with Zend Framework 1.10 you can simply set messages for `Zend_Validate_Hostname` from within `Zend_Validate_EmailAddress`.

```
$validator = new Zend_Validate_EmailAddress();
$validator->setMessages(
    array(
        Zend_Validate_Hostname::UNKNOWN_TLD => 'I don't know the TLD you gave'
    )
);
```

Before Zend Framework 1.10 you had to attach the messages to your own `Zend_Validate_Hostname`, and then set this validator within `Zend_Validate_EmailAddress` to get your own messages returned.

2.12. Float

Returns TRUE if and only if `$value` is a floating-point value. Since Zend Framework 1.8 this validator takes into account the actual locale from browser, environment or application wide set locale. You can of course use the get/setLocale accessors to change the used locale or give it while creating a instance of this validator.

2.13. GreaterThan

Returns TRUE if and only if `$value` is greater than the minimum boundary.

2.14. Hex

Returns TRUE if and only if `$value` contains only hexadecimal digit characters.

2.15. Hostname

`Zend_Validate_Hostname` allows you to validate a hostname against a set of known specifications. It is possible to check for three different types of hostnames: a DNS Hostname (i.e. domain.com), IP address (i.e. 1.2.3.4), and Local hostnames (i.e. localhost). By default only DNS hostnames are matched.

Basic usage

A basic example of usage is below:

```
$validator = new Zend_Validate_Hostname();
if ($validator->isValid($hostname)) {
    // hostname appears to be valid
} else {
    // hostname is invalid; print the reasons
    foreach ($validator->getMessages() as $message) {
        echo "$message\n";
```

```
    }
}
```

This will match the hostname `$hostname` and on failure populate `getMessages()` with useful error messages.

Validating different types of hostnames

You may find you also want to match IP addresses, Local hostnames, or a combination of all allowed types. This can be done by passing a parameter to `Zend_Validate_Hostname` when you instantiate it. The parameter should be an integer which determines what types of hostnames are allowed. You are encouraged to use the `Zend_Validate_Hostname` constants to do this.

The Zend_Validate_Hostname constants are: `ALLOW_DNS` to allow only DNS hostnames, `ALLOW_IP` to allow IP addresses, `ALLOW_LOCAL` to allow local network names, and `ALLOW_ALL` to allow all three types. To just check for IP addresses you can use the example below:

```
$validator = new Zend_Validate_Hostname(Zend_Validate_Hostname::ALLOW_IP);
if ($validator->isValid($hostname)) {
    // hostname appears to be valid
} else {
    // hostname is invalid; print the reasons
    foreach ($validator->getMessages() as $message) {
        echo "$message\n";
    }
}
```

As well as using `ALLOW_ALL` to accept all hostnames types you can combine these types to allow for combinations. For example, to accept DNS and Local hostnames instantiate your Zend_Validate_Hostname object as so:

```
$validator = new Zend_Validate_Hostname(Zend_Validate_Hostname::ALLOW_DNS |
                                        Zend_Validate_Hostname::ALLOW_IP);
```

Validating International Domains Names

Some Country Code Top Level Domains (ccTLDs), such as 'de' (Germany), support international characters in domain names. These are known as International Domain Names (IDN). These domains can be matched by `Zend_Validate_Hostname` via extended characters that are used in the validation process.

Until now more than 50 ccTLDs support IDN domains.

To match an IDN domain it's as simple as just using the standard Hostname validator since IDN matching is enabled by default. If you wish to disable IDN validation this can be done by either passing a parameter to the `Zend_Validate_Hostname` constructor or via the `setValidateIdn()` method.

You can disable IDN validation by passing a second parameter to the Zend_Validate_Hostname constructor in the following way.

```
$validator =
    new Zend_Validate_Hostname(
        array(
            'allow' => Zend_Validate_Hostname::ALLOW_DNS,
            'idn'   => false
```

```
        )
    );
```

Alternatively you can either pass TRUE or FALSE to `setValidateIdn()` to enable or disable IDN validation. If you are trying to match an IDN hostname which isn't currently supported it is likely it will fail validation if it has any international characters in it. Where a ccTLD file doesn't exist in Zend/Validate/Hostname specifying the additional characters a normal hostname validation is performed.

Please note IDNs are only validated if you allow DNS hostnames to be validated.

Validating Top Level Domains

By default a hostname will be checked against a list of known TLDs. If this functionality is not required it can be disabled in much the same way as disabling IDN support. You can disable TLD validation by passing a third parameter to the Zend_Validate_Hostname constructor. In the example below we are supporting IDN validation via the second parameter.

```
$validator =
    new Zend_Validate_Hostname(
        array(
            'allow' => Zend_Validate_Hostname::ALLOW_DNS,
            'idn'   => true,
            'tld'   => false
        )
    );
```

Alternatively you can either pass TRUE or FALSE to `setValidateTld()` to enable or disable TLD validation.

Please note TLDs are only validated if you allow DNS hostnames to be validated.

2.16. Iban

Returns TRUE if and only if `$value` contains a valid IBAN (International Bank Account Number). IBAN numbers are validated against the country where they are used and by a checksum.

There are two ways to validate IBAN numbers. As first way you can give a locale which represents a country. Any given IBAN number will then be validated against this country.

```
$validator = new Zend_Validate_Iban('de_AT');
$iban = 'AT611904300234573201';
if ($validator->isValid($iban)) {
    // IBAN appears to be valid
} else {
    // IBAN is invalid
    foreach ($validator->getMessages() as $message) {
        echo "$message\n";
    }
}
```

This should be done when you want to validate IBAN numbers for a single countries. The simpler way of validation is not to give a locale like shown in the next example.

```
$validator = new Zend_Validate_Iban();
$iban = 'AT611904300234573201';
```

```
if ($validator->isValid($iban)) {
    // IBAN appears to be valid
} else {
    // IBAN is invalid
}
```

But this shows one big problem: When you have to accept only IBAN numbers from one single country, for example france, then IBAN numbers from other countries would also be valid. Therefor just remember: When you have to validate a IBAN number against a defined country you should give the locale. And when you accept all IBAN numbers regardless of any country omit the locale for simplicity.

2.17. Identical

`Zend_Validate_Identical` allows you to validate if a given value is identical with an set haystack.

2.17.1. Basic usage

To validate if two values are identical you need to set the origin value as haystack. See the following example which validates two strings.

```
$valid = new Zend_Validate_Identical('origin');
if ($valid->isValid($value) {
    return true;
}
```

The validation will only then return TRUE when both values are 100% identical. In our example, when `$value` is 'origin'.

You can set the wished token also afterwards by using the method `setToken()` and `getToken()` to get the actual set token.

2.17.2. Identical objects

Of course `Zend_Validate_Identical` can not only validate strings, but also any other variable type like Boolean, Integer, Float, Array or even Objects. As already noted Haystack and Value must be identical.

```
$valid = new Zend_Validate_Identical(123);
if ($valid->isValid($input)) {
    // input appears to be valid
} else {
    // input is invalid
}
```

Type comparison

You should be aware that also the type of a variable is used for validation. This means that the string *'3'* is not identical with the integer *3*.

This is also the case for Form Elements. They are objects or arrays. So you can't simply compare a Textfield which contains a password with an textual password from another source. The Element itself is given as array which also contains additional informations.

2.17.3. Configuration

As all other validators also `Zend_Validate_Identical` supports the usage of configuration settings as input parameter. This means that you can configure this validator with an `Zend_Config` object.

But this adds one case which you have to be aware. When you are using an array as haystack then you should wrap it within an 'token' key when it could contain only one element.

```
$valid = new Zend_Validate_Identical(array('token' => 123));
if ($valid->isValid($input)) {
    // input appears to be valid
} else {
    // input is invalid
}
```

The above example validates the integer 123. The reason for this special case is, that you can configure the token which has to be used by giving the 'token' key.

So, when your haystack contains one element and this element is named 'token' then you have to wrap it like shown in the example below.

```
$valid = new Zend_Validate_Identical(array('token' => array('token' => 123)));
if ($valid->isValid($input)) {
    // input appears to be valid
} else {
    // input is invalid
}
```

2.18. InArray

`Zend_Validate_InArray` allows you to validate if a given value is contained within an array. It is also able to validate multidimensional arrays.

2.18.1. Simple array validation

The simplest way, is just to give the array which should be searched against at initiation:

```
$validator = new Zend_Validate_InArray(array('key' => 'value',
                                             'otherkey' => 'othervalue'));
if ($validator->isValid('value')) {
    // value found
} else {
    // no value found
}
```

This will behave exactly like PHP's `in_array()` method.

Per default this validation is not strict nor can it validate multidimensional arrays.

Of course you can give the array to validate against also afterwards by using the `setHaystack()` method. `getHaystack()` returns the actual set haystack array.

```
$validator = new Zend_Validate_InArray();
$validator->setHaystack(array('key' => 'value', 'otherkey' => 'othervalue'));

if ($validator->isValid('value')) {
    // value found
} else {
    // no value found
}
```

2.18.2. Strict array validation

As mentioned before you can also do a strict validation within the array. Per default there would be no difference between the integer value *0* and the string *"0"*. When doing a strict validation this difference will also be validated and only same types are accepted.

A strict validation can also be done by using two different ways. At initiation and by using a method. At initiation you have to give an array with the following structure:

```
$validator = new Zend_Validate_InArray(
    array(
        'haystack' => array('key' => 'value', 'otherkey' => 'othervalue'),
        'strict'   => true
    )
);

if ($validator->isValid('value')) {
    // value found
} else {
    // no value found
}
```

The *haystack* key contains your array to validate against. And by setting the *strict* key to TRUE, the validation is done by using a strict type check.

Of course you can also use the `setStrict()` method to change this setting afterwards and `getStrict()` to get the actual set state.

Note that the *strict* setting is per default FALSE.

2.18.3. Recursive array validation

In addition to PHP's `in_array()` method this validator can also be used to validate multidimensional arrays.

To validate multidimensional arrays you have to set the *recursive* option.

```
$validator = new Zend_Validate_InArray(
    array(
        'haystack' => array(
            'firstDimension' => array('key' => 'value',
                                      'otherkey' => 'othervalue'),
            'secondDimension' => array('some' => 'real',
                                       'different' => 'key')),
        'recursive' => true
```

```
    )
);

if ($validator->isValid('value')) {
    // value found
} else {
    // no value found
}
```

Your array will then be validated recursive to see if the given value is contained. Additionally you could use `setRecursive()` to set this option afterwards and `getRecursive()` to retrieve it.

```
$validator = new Zend_Validate_InArray(
    array(
        'firstDimension' => array('key' => 'value',
                                  'otherkey' => 'othervalue'),
        'secondDimension' => array('some' => 'real',
                                   'different' => 'key')
    )
);
$validator->setRecursive(true);

if ($validator->isValid('value')) {
    // value found
} else {
    // no value found
}
```

Default setting for recursion

Per default the recursive validation is turned off.

Option keys within the haystack

When you are using the keys 'haystack', 'strict' or 'recursive' within your haystack, then you must wrap the haystack key.

2.19. Int

Returns TRUE if and only if `$value` is a valid integer. Since Zend Framework 1.8 this validator takes into account the actual locale from browser, environment or application wide set locale. You can of course use the get/setLocale accessors to change the used locale or give it while creating a instance of this validator.

2.20. Ip

`Zend_Validate_Ip` allows you to validate if a given value is an IP address. It supports the IPv4 and also the IPv6 standard.

2.20.1. Basic usage

A basic example of usage is below:

```
$validator = new Zend_Validate_Ip();
if ($validator->isValid($ip)) {
```

```
    // ip appears to be valid
} else {
    // ip is invalid; print the reasons
}
```

Invalid IP addresses

Keep in mind that `Zend_Validate_Ip` only validates IP addresses. Addresses like 'mydomain.com' or '192.168.50.1/index.html' are no valid IP addresses. They are either hostnames or valid URLs but not IP addresses.

IPv6 validation

`Zend_Validate_Ip` validates IPv6 addresses with regex. The reason is that the filters and methods from PHP itself don't follow the RFC. Many other available classes also don't follow it.

2.20.2. Validate IPv4 or IPV6 alone

Sometimes it's useful to validate only one of the supported formats. For example when your network only supports IPv4. In this case it would be useless to allow IPv6 within this validator.

To limit `Zend_Validate_Ip` to one protocol you can set the options allowipv4 or allowipv6 to FALSE. You can do this either by giving the option to the constructor or by using `setOptions()` afterwards.

```
$validator = new Zend_Validate_Ip(array('allowipv6' => false);
if ($validator->isValid($ip)) {
    // ip appears to be valid ipv4 address
} else {
    // ip is no ipv4 address
}
```

Default behaviour

The default behaviour which `Zend_Validate_Ip` follows is to allow both standards.

2.21. Isbn

`Zend_Validate_Isbn` allows you to validate an ISBN-10 or ISBN-13 value.

2.21.1. Basic usage

A basic example of usage is below:

```
$validator = new Zend_Validate_Isbn();
if ($validator->isValid($isbn)) {
    // isbn is valid
} else {
    // isbn is not valid
}
```

This will validate any ISBN-10 and ISBN-13 without separator.

2.21.2. Setting an explicit ISBN validation type

An example of an ISBN type restriction is below:

```
$validator = new Zend_Validate_Isbn();
$validator->setType(Zend_Validate_Isbn::ISBN13);
// OR
$validator = new Zend_Validate_Isbn(array(
    'type' => Zend_Validate_Isbn::ISBN13,
));

if ($validator->isValid($isbn)) {
    // this is a valid ISBN-13 value
} else {
    // this is an invalid ISBN-13 value
}
```

The above will validate only ISBN-13 values.

Valid types include:

- `Zend_Validate_Isbn::AUTO` (default)
- `Zend_Validate_Isbn::ISBN10`
- `Zend_Validate_Isbn::ISBN13`

2.21.3. Specifying a separator restriction

An example of separator restriction is below:

```
$validator = new Zend_Validate_Isbn();
$validator->setSeparator('-');
// OR
$validator = new Zend_Validate_Isbn(array(
    'separator' => '-',
));

if ($validator->isValid($isbn)) {
    // this is a valid ISBN with separator
} else {
    // this is an invalid ISBN with separator
}
```

Values without separator

This will return `FALSE` if `$isbn` doesn't contain a separator *or* if it's an invalid ISBN value.

Valid separators include:

- "" (empty) (default)
- "-" (hyphen)
- " " (space)

2.22. LessThan

Returns TRUE if and only if `$value` is less than the maximum boundary.

2.23. NotEmpty

This validator allows you to validate if a given value is not empty. This is often useful when working with form elements or other user input, where you can use it to ensure required elements have values associated with them.

2.23.1. Default behaviour for Zend_Validate_NotEmpty

By default, this validator works differently than you would expect when you've worked with PHP's `empty()` function. In particular, this validator will evaluate both the integer *0* and string '*0*' as empty.

```
$valid = new Zend_Validate_NotEmpty();
$value  = '';
$result = $valid->isValid($value);
// returns false
```

Default behaviour differs from PHP

Without providing configuration, `Zend_Validate_NotEmpty`'s behaviour differs from PHP.

2.23.2. Changing behaviour for Zend_Validate_NotEmpty

Some projects have differing opinions of what is considered an "empty" value: a string with only whitespace might be considered empty, or *0* may be considered non-empty (particularly for boolean sequences). To accomodate differing needs, `Zend_Validate_NotEmpty` allows you to configure which types should be validated as empty and which not.

The following types can be handled:

- *boolean*: Returns FALSE when the boolean value is FALSE.
- *integer*: Returns FALSE when an integer *0* value is given. Per default this validation is not activated and returns TRUE on any integer values.
- *float*: Returns FALSE when an float *0.0* value is given. Per default this validation is not activated and returns TRUE on any float values.
- *string*: Returns FALSE when an empty string *''* is given.
- *zero*: Returns FALSE when the single character zero (*'0'*) is given.
- *empty_array*: Returns FALSE when an empty *array* is given.
- *null*: Returns FALSE when an NULL value is given.
- *php*: Returns FALSE on the same reasons where PHP method `empty()` would return TRUE.
- *space*: Returns FALSE when an string is given which contains only whitespaces.

- *all*: Returns FALSE on all above types.

All other given values will return TRUE per default.

There are several ways to select which of the above types are validated. You can give one or multiple types and add them, you can give an array, you can use constants, or you can give a textual string. See the following examples:

```
// Returns false on 0
$validator = new Zend_Validate_NotEmpty(Zend_Validate_NotEmpty::INTEGER);

// Returns false on 0 or '0'
$validator = new Zend_Validate_NotEmpty(
    Zend_Validate_NotEmpty::INTEGER + Zend_NotEmpty::ZERO
);

// Returns false on 0 or '0'
$validator = new Zend_Validate_NotEmpty(array(
    Zend_Validate_NotEmpty::INTEGER,
    Zend_Validate_NotEmpty::ZERO
));

// Returns false on 0 or '0'
$validator = new Zend_Validate_NotEmpty(array(
    'integer',
    'zero',
));
```

You can also provide an instance of `Zend_Config` to set the desired types. To set types after instantiation, use the `setType()` method.

2.24. PostCode

`Zend_Validate_PostCode` allows you to determine if a given value is a valid postal code. Postal codes are specific to cities, and in some locales termed ZIP codes.

`Zend_Validate_PostCode` knows more than 160 different postal code formates. To select the correct format there are 2 ways. You can either use a fully qualified locale or you can set your own format manually.

Using a locale is more convenient as Zend Framework already knows the appropriate postal code format for each locale; however, you need to use the fully qualified locale (one containing a region specifier) to do so. For instance, the locale "de" is a locale but could not be used with `Zend_Validate_PostCode` as it does not include the region; "de_AT", however, would be a valid locale, as it specifies the region code ("AT", for Austria).

```
$validator = new Zend_Validate_PostCode('de_AT');
```

When you don't set a locale yourself, then `Zend_Validate_PostCode` will use the application wide set locale, or, when there is none, the locale returned by `Zend_Locale`.

```
// application wide locale within your bootstrap
$locale = new Zend_Locale('de_AT');
Zend_Registry::set('Zend_Locale', $locale);

$validator = new Zend_Validate_PostCode();
```

You can also change the locale afterwards by calling `setLocale()`. And of course you can get the actual used locale by calling `getLocale()`.

```
$validator = new Zend_Validate_PostCode('de_AT');
$validator->setLocale('en_GB');
```

Postal code formats themself are simply regular expression strings. When the international postal code format, which is used by setting the locale, does not fit your needs, then you can also manually set a format by calling `setFormat()`.

```
$validator = new Zend_Validate_PostCode('de_AT');
$validator->setFormat('AT-\d{5}');
```

Conventions for self defined formats

When using self defined formats you should omit the starting (**'/^'**) and ending tags (**'$/'**). They are attached automatically.

You should also be aware that postcode values are always be validated in a strict way. This means that they have to be written standalone without additional characters when they are not covered by the format.

2.24.1. Constructor options

At it's most basic, you may pass either a `Zend_Locale` object or a string representing a fully qualified locale to the constructor of `Zend_Validate_PostCode`.

```
$validator = new Zend_Validate_PostCode('de_AT');
$validator = new Zend_Validate_PostCode($locale);
```

Additionally, you may pass either an array or a `Zend_Config` object to the constructor. When you do so, you must include either the key "locale" or "format"; these will be used to set the appropriate values in the validator object.

```
$validator = new Zend_Validate_PostCode(array(
    'locale' => 'de_AT',
    'format' => 'AT_\d+'
));
```

2.25. Regex

Returns TRUE if and only if `$value` matches against a regular expression pattern.

2.26. Sitemap Validators

The following validators conform to the Sitemap XML protocol.

2.26.1. Sitemap_Changefreq

Validates whether a string is valid for using as a 'changefreq' element in a Sitemap XML document. Valid values are: 'always', 'hourly', 'daily', 'weekly', 'monthly', 'yearly', or 'never'.

Returns TRUE if and only if the value is a string and is equal to one of the frequencies specified above.

2.26.2. Sitemap_Lastmod

Validates whether a string is valid for using as a 'lastmod' element in a Sitemap XML document. The lastmod element should contain a W3C date string, optionally discarding information about time.

Returns TRUE if and only if the given value is a string and is valid according to the protocol.

Example 906. Sitemap Lastmod Validator

```
$validator = new Zend_Validate_Sitemap_Lastmod();

$validator->isValid('1999-11-11T22:23:52-02:00'); // true
$validator->isValid('2008-05-12T00:42:52+02:00'); // true
$validator->isValid('1999-11-11'); // true
$validator->isValid('2008-05-12'); // true

$validator->isValid('1999-11-11t22:23:52-02:00'); // false
$validator->isValid('2008-05-12T00:42:60+02:00'); // false
$validator->isValid('1999-13-11'); // false
$validator->isValid('2008-05-32'); // false
$validator->isValid('yesterday'); // false
```

2.26.3. Sitemap_Loc

Validates whether a string is valid for using as a 'loc' element in a Sitemap XML document. This uses `Zend_Form::check()` internally. Read more at URI Validation.

2.26.4. Sitemap_Priority

Validates whether a value is valid for using as a 'priority' element in a Sitemap XML document. The value should be a decimal between 0.0 and 1.0. This validator accepts both numeric values and string values.

Example 907. Sitemap Priority Validator

```
$validator = new Zend_Validate_Sitemap_Priority();

$validator->isValid('0.1'); // true
$validator->isValid('0.789'); // true
$validator->isValid(0.8); // true
$validator->isValid(1.0); // true

$validator->isValid('1.1'); // false
$validator->isValid('-0.4'); // false
$validator->isValid(1.00001); // false
$validator->isValid(0xFF); // false
$validator->isValid('foo'); // false
```

2.27. StringLength

Returns TRUE if and only if the string length of `$value` is at least a minimum and no greater than a maximum (when the max option is not `NULL`). The `setMin()` method throws an exception if the minimum length is set to a value greater than the set maximum length, and the `setMax()` method throws an exception if the maximum length is set to a value less than the set minimum length. This class supports UTF-8 and other character encodings, based on the current value

of iconv.internal_encoding. If you need a different encoding you can set it with the accessor methods getEncoding and setEncoding.

3. Validator Chains

Often multiple validations should be applied to some value in a particular order. The following code demonstrates a way to solve the example from the introduction, where a username must be between 6 and 12 alphanumeric characters:

```
// Create a validator chain and add validators to it
$validatorChain = new Zend_Validate();
$validatorChain->addValidator(
                    new Zend_Validate_StringLength(array('min' => 6,
                                                         'max' => 12)))
               ->addValidator(new Zend_Validate_Alnum());

// Validate the username
if ($validatorChain->isValid($username)) {
    // username passed validation
} else {
    // username failed validation; print reasons
    foreach ($validatorChain->getMessages() as $message) {
        echo "$message\n";
    }
}
```

Validators are run in the order they were added to `Zend_Validate`. In the above example, the username is first checked to ensure that its length is between 6 and 12 characters, and then it is checked to ensure that it contains only alphanumeric characters. The second validation, for alphanumeric characters, is performed regardless of whether the first validation, for length between 6 and 12 characters, succeeds. This means that if both validations fail, `getMessages()` will return failure messages from both validators.

In some cases it makes sense to have a validator break the chain if its validation process fails. `Zend_Validate` supports such use cases with the second parameter to the `addValidator()` method. By setting `$breakChainOnFailure` to `TRUE`, the added validator will break the chain execution upon failure, which avoids running any other validations that are determined to be unnecessary or inappropriate for the situation. If the above example were written as follows, then the alphanumeric validation would not occur if the string length validation fails:

```
$validatorChain->addValidator(
                    new Zend_Validate_StringLength(array('min' => 6,
                                                         'max' => 12)),
                    true)
               ->addValidator(new Zend_Validate_Alnum());
```

Any object that implements `Zend_Validate_Interface` may be used in a validator chain.

4. Writing Validators

`Zend_Validate` supplies a set of commonly needed validators, but inevitably, developers will wish to write custom validators for their particular needs. The task of writing a custom validator is described in this section.

`Zend_Validate_Interface` defines two methods, `isValid()` and `getMessages()`, that may be implemented by user classes in order to create custom validation objects. An

object that implements Zend_Validate_Interface interface may be added to a validator chain with Zend_Validate::addValidator(). Such objects may also be used with Zend_Filter_Input.

As you may already have inferred from the above description of Zend_Validate_Interface, validation classes provided with Zend Framework return a boolean value for whether or not a value validates successfully. They also provide information about *why* a value failed validation. The availability of the reasons for validation failures may be valuable to an application for various purposes, such as providing statistics for usability analysis.

Basic validation failure message functionality is implemented in Zend_Validate_Abstract. To include this functionality when creating a validation class, simply extend Zend_Validate_Abstract. In the extending class you would implement the isValid() method logic and define the message variables and message templates that correspond to the types of validation failures that can occur. If a value fails your validation tests, then isValid() should return FALSE. If the value passes your validation tests, then isValid() should return TRUE.

In general, the isValid() method should not throw any exceptions, except where it is impossible to determine whether or not the input value is valid. A few examples of reasonable cases for throwing an exception might be if a file cannot be opened, an LDAP server could not be contacted, or a database connection is unavailable, where such a thing may be required for validation success or failure to be determined.

Example 908. Creating a Simple Validation Class

The following example demonstrates how a very simple custom validator might be written. In this case the validation rules are simply that the input value must be a floating point value.

```
class MyValid_Float extends Zend_Validate_Abstract
{
    const FLOAT = 'float';

    protected $_messageTemplates = array(
        self::FLOAT => "'%value%' is not a floating point value"
    );

    public function isValid($value)
    {
        $this->_setValue($value);

        if (!is_float($value)) {
            $this->_error();
            return false;
        }

        return true;
    }
}
```

The class defines a template for its single validation failure message, which includes the built-in magic parameter, *%value%*. The call to _setValue() prepares the object to insert the tested value into the failure message automatically, should the value fail validation. The call to _error() tracks a reason for validation failure. Since this class only defines one failure message, it is not necessary to provide _error() with the name of the failure message template.

Example 909. Writing a Validation Class having Dependent Conditions

The following example demonstrates a more complex set of validation rules, where it is required that the input value be numeric and within the range of minimum and maximum boundary values. An input value would fail validation for exactly one of the following reasons:

- The input value is not numeric.
- The input value is less than the minimum allowed value.
- The input value is more than the maximum allowed value.

These validation failure reasons are then translated to definitions in the class:

```
class MyValid_NumericBetween extends Zend_Validate_Abstract
{
    const MSG_NUMERIC = 'msgNumeric';
    const MSG_MINIMUM = 'msgMinimum';
    const MSG_MAXIMUM = 'msgMaximum';

    public $minimum = 0;
    public $maximum = 100;

    protected $_messageVariables = array(
        'min' => 'minimum',
        'max' => 'maximum'
    );

    protected $_messageTemplates = array(
        self::MSG_NUMERIC => "'%value%' is not numeric",
        self::MSG_MINIMUM => "'%value%' must be at least '%min%'",
        self::MSG_MAXIMUM => "'%value%' must be no more than '%max%'"
    );

    public function isValid($value)
    {
        $this->_setValue($value);

        if (!is_numeric($value)) {
            $this->_error(self::MSG_NUMERIC);
            return false;
        }

        if ($value < $this->minimum) {
            $this->_error(self::MSG_MINIMUM);
            return false;
        }

        if ($value > $this->maximum) {
            $this->_error(self::MSG_MAXIMUM);
            return false;
        }

        return true;
    }
}
```

The public properties `$minimum` and `$maximum` have been established to provide the minimum and maximum boundaries, respectively, for a value to successfully validate. The class also defines two message variables that correspond to the public properties and allow min and max to be used in message templates as magic parameters, just as with value.

Note that if any one of the validation checks in `isValid()` fails, an appropriate failure message is prepared, and the method immediately returns FALSE. These validation rules are therefore sequentially dependent. That is, if one test should fail, there is no need to test any subsequent validation rules. This need not be the case, however. The following example illustrates how to write a class having independent validation rules, where the validation object may return multiple reasons why a particular validation attempt failed.

Example 910. Validation with Independent Conditions, Multiple Reasons for Failure

Consider writing a validation class for password strength enforcement - when a user is required to choose a password that meets certain criteria for helping secure user accounts. Let us assume that the password security criteria enforce that the password:

- is at least 8 characters in length,
- contains at least one uppercase letter,
- contains at least one lowercase letter,
- and contains at least one digit character.

The following class implements these validation criteria:

```
class MyValid_PasswordStrength extends Zend_Validate_Abstract
{
    const LENGTH = 'length';
    const UPPER  = 'upper';
    const LOWER  = 'lower';
    const DIGIT  = 'digit';

    protected $_messageTemplates = array(
        self::LENGTH => "'%value%' must be at least 8 characters in length",
        self::UPPER  => "'%value%' must contain at least one uppercase letter",
        self::LOWER  => "'%value%' must contain at least one lowercase letter",
        self::DIGIT  => "'%value%' must contain at least one digit character"
    );

    public function isValid($value)
    {
        $this->_setValue($value);

        $isValid = true;

        if (strlen($value) < 8) {
            $this->_error(self::LENGTH);
            $isValid = false;
        }

        if (!preg_match('/[A-Z]/', $value)) {
            $this->_error(self::UPPER);
            $isValid = false;
        }

        if (!preg_match('/[a-z]/', $value)) {
            $this->_error(self::LOWER);
            $isValid = false;
        }

        if (!preg_match('/\d/', $value)) {
            $this->_error(self::DIGIT);
            $isValid = false;
        }

        return $isValid;
    }
}
```

Note that the four criteria tests in `isValid()` do not immediately return FALSE. This allows the validation class to provide *all* of the reasons that the input password failed to meet the validation requirements. if, for example, a user were to input the string "#$%" as a password, `isValid()` would cause all four validation failure messages to be returned by a subsequent call to `getMessages()`.

5. Validation Messages

Each validator which is based on `Zend_Validate` provides one or multiple messages in the case of a failed validation. You can use this information to set your own messages, or to translate existing messages which a validator could return to something different.

These validation messages are constants which can be found at top of each validator class. Let's look into `Zend_Validate_GreaterThan` for an descriptive example:

```
protected $_messageTemplates = array(
    self::NOT_GREATER => "'%value%' is not greater than '%min%'",
);
```

As you can see the constant `self::NOT_GREATER` refers to the failure and is used as key, and the message itself is used as value of the message array.

You can retrieve all message templates from a validator by using the `getMessageTemplates()` method. It returns you the above array which contains all messages a validator could return in the case of a failed validation.

```
$validator = new Zend_Validate_GreaterThan();
$messages  = $validator->getMessageTemplates();
```

Using the `setMessage()` method you can set another message to be returned in case of the specified failure.

```
$validator = new Zend_Validate_GreaterThan();
$validator->setMessage('Please enter a lower value', Zend_Validate_GreaterThan::NOT_GREATER
```

The second parameter defines the failure which will be overridden. When you omit this parameter, then the given message will be set for all possible failures of this validator.

5.1. Using pre-translated validation messages

Zend Framework is shipped with more than 45 different validators with more than 200 failure messages. It can be a tendious task to translate all of these messages. But for your convinience Zend Framework comes with already pre-translated validation messages. You can find them within the path `/resources/languages` in your Zend Framework installation.

Used path

The resource files are outside of the library path because all of your translations should also be outside of this path.

So to translate all validation messages to german for example, all you have to do is to attach a translator to `Zend_Validate` using these resource files.

```
$translator = new Zend_Translate(
    'array',
    '/resources/languages',
    $language,
    array('scan' => Zend_Locale::LOCALE_DIRECTORY)
);
Zend_Validate_Abstract::setDefaultTranslator($translator);
```

Used translation adapter

As translation adapter Zend Framework choosed the array adapter. It is simple to edit and created very fast.

Supported languages

This feature is very young, so the amount of supported languages may not be complete. New languages will be added with each release. Additionally feel free to use the existing resource files to make your own translations.

You could also use these resource files to rewrite existing translations. So you are not in need to create these files manually yourself.

5.2. Limit the size of a validation message

Sometimes it is necessary to limit the maximum size a validation message can have. For example when your view allows a maximum size of 100 chars to be rendered on one line. To simplify the usage, `Zend_Validate` is able to automatically limit the maximum returned size of a validation message.

To get the actual set size use `Zend_Validate::getMessageLength()`. If it is -1, then the returned message will not be truncated. This is default behaviour.

To limit the returned message size use `Zend_Validate::setMessageLength()`. Set it to any integer size you need. When the returned message exceeds the set size, then the message will be truncated and the string '...' will be added instead of the rest of the message.

```
Zend_Validate::setMessageLength(100);
```

Where is this parameter used?

The set message length is used for all validators, even for self defined ones, as long as they extend `Zend_Validate_Abstract`.

Zend_Version

1. Getting the Zend Framework Version

`Zend_Version` provides a class constant `Zend_Version::VERSION` that contains a string identifying the version number of your Zend Framework installation. `Zend_Version::VERSION` might contain "1.7.4", for example.

The static method `Zend_Version::compareVersion($version)` is based on the PHP function `version_compare()`. This method returns -1 if the specified version is older than the installed Zend Framework version, 0 if they are the same and +1 if the specified version is newer than the version of the Zend Framework installation.

Example 911. Example of the compareVersion() Method

```
// returns -1, 0 or 1
$cmp = Zend_Version::compareVersion('2.0.0');
```

Zend_View

1. Introduction

Zend_View is a class for working with the "view" portion of the model-view-controller pattern. That is, it exists to help keep the view script separate from the model and controller scripts. It provides a system of helpers, output filters, and variable escaping.

Zend_View is template system agnostic; you may use PHP as your template language, or create instances of other template systems and manipulate them within your view script.

Essentially, using Zend_View happens in two major steps: 1. Your controller script creates an instance of Zend_View and assigns variables to that instance. 2. The controller tells the Zend_View to render a particular view, thereby handing control over the view script, which generates the view output.

1.1. Controller Script

As a simple example, let us say your controller has a list of book data that it wants to have rendered by a view. The controller script might look something like this:

```
// use a model to get the data for book authors and titles.
$data = array(
    array(
        'author' => 'Hernando de Soto',
        'title' => 'The Mystery of Capitalism'
    ),
    array(
        'author' => 'Henry Hazlitt',
        'title' => 'Economics in One Lesson'
    ),
    array(
        'author' => 'Milton Friedman',
        'title' => 'Free to Choose'
    )
);

// now assign the book data to a Zend_View instance
Zend_Loader::loadClass('Zend_View');
$view = new Zend_View();
$view->books = $data;

// and render a view script called "booklist.php"
echo $view->render('booklist.php');
```

1.2. View Script

Now we need the associated view script, "booklist.php". This is a PHP script like any other, with one exception: it executes inside the scope of the Zend_View instance, which means that references to $this point to the Zend_View instance properties and methods. (Variables assigned to the instance by the controller are public properties of the Zend_View instance). Thus, a very basic view script could look like this:

```
if ($this->books): ?>
```

```
    <!-- A table of some books. -->
    <table>
        <tr>
            <th>Author</th>
            <th>Title</th>
        </tr>

        <?php foreach ($this->books as $key => $val): ?>
        <tr>
            <td><?php echo $this->escape($val['author']) ?></td>
            <td><?php echo $this->escape($val['title']) ?></td>
        </tr>
        <?php endforeach; ?>
    </table>

<?php else: ?>
    <p>There are no books to display.</p>

<?php endif;?>
```

Note how we use the "escape()" method to apply output escaping to variables.

1.3. Options

`Zend_View` has several options that may be set to configure the behaviour of your view scripts.

- basePath: indicate a base path from which to set the script, helper, and filter path. It assumes a directory structure of:

  ```
  base/path/
      helpers/
      filters/
      scripts/
  ```

  This may be set via `setBasePath()`, `addBasePath()`, or the basePath option to the constructor.

- encoding: indicate the character encoding to use with `htmlentities()`, `htmlspecialchars()`, and other operations. Defaults to ISO-8859-1 (latin1). May be set via `setEncoding()` or the encoding option to the constructor.

- escape: indicate a callback to be used by `escape()`. May be set via `setEscape()` or the escape option to the constructor.

- filter: indicate a filter to use after rendering a view script. May be set via `setFilter()`, `addFilter()`, or the filter option to the constructor.

- strictVars: force `Zend_View` to emit notices and warnings when uninitialized view variables are accessed. This may be set by calling `strictVars(true)` or passing the strictVars option to the constructor.

1.4. Short Tags with View Scripts

In our examples, we make use of PHP long tags: *<?php*. We also favor the use of alternate syntax for control structures. These are convenient shorthands to use when writing view scripts, as they make the constructs more terse, keep statements on single lines, and eliminate the need to hunt for brackets within HTML.

In previous versions, we often recommended using short tags (<? and <?=), as they make the view scripts slightly less verbose. However, the default for the `php.ini short_open_tag` setting is typically off in production or on shared hosts -- making their use not terribly portable. If you use template XML in view scripts, short open tags will cause the templates to fail validation. Finally, if you use short tags when `short_open_tag` is off, the view scripts will either cause errors or simply echo PHP code back to the viewer.

If, despite these warnings, you wish to use short tags but they are disabled, you have two options:

- Turn on short tags in your `.htaccess` file:

```
php_value "short_open_tag" "on"
```

  This will only be possible if you are allowed to create and utilize `.htaccess` files. This directive can also be added to your `httpd.conf` file.

- Enable an optional stream wrapper to convert short tags to long tags on the fly:

```
$view->setUseStreamWrapper(true);
```

  This registers `Zend_View_Stream` as a stream wrapper for view scripts, and will ensure that your code continues to work as if short tags were enabled.

View Stream Wrapper Degrades Performance

Usage of the stream wrapper *will* degrade performance of your application, though actual benchmarks are unavailable to quantify the amount of degradation. We recommend that you either enable short tags, convert your scripts to use full tags, or have a good partial and/or full page content caching strategy in place.

1.5. Utility Accessors

Typically, you'll only ever need to call on `assign()`, `render()`, or one of the methods for setting/adding filter, helper, and script paths. However, if you wish to extend `Zend_View` yourself, or need access to some of its internals, a number of accessors exist:

- `getVars()` will return all assigned variables.
- `clearVars()` will clear all assigned variables; useful when you wish to re-use a view object, but want to control what variables are available.
- `getScriptPath($script)` will retrieve the resolved path to a given view script.
- `getScriptPaths()` will retrieve all registered script paths.
- `getHelperPath($helper)` will retrieve the resolved path to the named helper class.
- `getHelperPaths()` will retrieve all registered helper paths.
- `getFilterPath($filter)` will retrieve the resolved path to the named filter class.
- `getFilterPaths()` will retrieve all registered filter paths.

2. Controller Scripts

The controller is where you instantiate and configure `Zend_View`. You then assign variables to the view, and tell the view to render output using a particular script.

2.1. Assigning Variables

Your controller script should assign necessary variables to the view before it hands over control to the view script. Normally, you can do assignments one at a time by assigning to property names of the view instance:

```
$view = new Zend_View();
$view->a = "Hay";
$view->b = "Bee";
$view->c = "Sea";
```

However, this can be tedious when you have already collected the values to be assigned into an array or object.

The assign() method lets you assign from an array or object "in bulk". The following examples have the same effect as the above one-by-one property assignments.

```
$view = new Zend_View();

// assign an array of key-value pairs, where the
// key is the variable name, and the value is
// the assigned value.
$array = array(
    'a' => "Hay",
    'b' => "Bee",
    'c' => "Sea",
);
$view->assign($array);

// do the same with an object's public properties;
// note how we cast it to an array when assigning.
$obj = new StdClass;
$obj->a = "Hay";
$obj->b = "Bee";
$obj->c = "Sea";
$view->assign((array) $obj);
```

Alternatively, you can use the assign method to assign one-by-one by passing a string variable name, and then the variable value.

```
$view = new Zend_View();
$view->assign('a', "Hay");
$view->assign('b', "Bee");
$view->assign('c', "Sea");
```

2.2. Rendering a View Script

Once you have assigned all needed variables, the controller should tell `Zend_View` to render a particular view script. Do so by calling the render() method. Note that the method will return the rendered view, not print it, so you need to print or echo it yourself at the appropriate time.

```
$view = new Zend_View();
$view->a = "Hay";
$view->b = "Bee";
$view->c = "Sea";
echo $view->render('someView.php');
```

2.3. View Script Paths

By default, `Zend_View` expects your view scripts to be relative to your calling script. For example, if your controller script is at "/path/to/app/controllers" and it calls $view->render('someView.php'), `Zend_View` will look for "/path/to/app/controllers/someView.php".

Obviously, your view scripts are probably located elsewhere. To tell `Zend_View` where it should look for view scripts, use the setScriptPath() method.

```
$view = new Zend_View();
$view->setScriptPath('/path/to/app/views');
```

Now when you call $view->render('someView.php'), it will look for "/path/to/app/views/someView.php".

In fact, you can "stack" paths using the addScriptPath() method. As you add paths to the stack, `Zend_View` will look at the most-recently-added path for the requested view script. This allows you override default views with custom views so that you may create custom "themes" or "skins" for some views, while leaving others alone.

```
$view = new Zend_View();
$view->addScriptPath('/path/to/app/views');
$view->addScriptPath('/path/to/custom/');

// now when you call $view->render('booklist.php'), Zend_View will
// look first for "/path/to/custom/booklist.php", then for
// "/path/to/app/views/booklist.php", and finally in the current
// directory for "booklist.php".
```

Never use user input to set script paths

`Zend_View` uses script paths to lookup and render view scripts. As such, these directories should be known before-hand, and under your control. *Never* set view script paths based on user input, as you can potentially open yourself up to Local File Inclusion vulnerability if the specified path includes parent directory traversals. For example, the following input could trigger the issue:

```
// $_GET['foo'] == '../../../etc'
$view->addScriptPath($_GET['foo']);
$view->render('passwd');
```

While this example is contrived, it does clearly show the potential issue. If you *must* rely on user input to set your script path, properly filter the input and check to ensure it exists under paths controlled by your application.

3. View Scripts

Once your controller has assigned variables and called `render()`, `Zend_View` then includes the requested view script and executes it "inside" the scope of the `Zend_View` instance. Therefore, in your view scripts, references to $this actually point to the `Zend_View` instance itself.

Variables assigned to the view from the controller are referred to as instance properties. For example, if the controller were to assign a variable 'something', you would refer to it as $this->something in the view script. (This allows you to keep track of which values were assigned to the script, and which are internal to the script itself.)

By way of reminder, here is the example view script from the Zend_View introduction.

```
<?php if ($this->books): ?>
    <!-- A table of some books. -->
    <table>
        <tr>
            <th>Author</th>
            <th>Title</th>
        </tr>

        <?php foreach ($this->books as $key => $val): ?>
        <tr>
            <td><?php echo $this->escape($val['author']) ?></td>
            <td><?php echo $this->escape($val['title']) ?></td>
        </tr>
        <?php endforeach; ?>
    </table>

<?php else: ?>
    <p>There are no books to display.</p>

<?php endif;?>
```

3.1. Escaping Output

One of the most important tasks to perform in a view script is to make sure that output is escaped properly; among other things, this helps to avoid cross-site scripting attacks. Unless you are using a function, method, or helper that does escaping on its own, you should always escape variables when you output them.

Zend_View comes with a method called escape() that does such escaping for you.

```
// bad view-script practice:
echo $this->variable;

// good view-script practice:
echo $this->escape($this->variable);
```

By default, the escape() method uses the PHP htmlspecialchars() function for escaping. However, depending on your environment, you may wish for escaping to occur in a different way. Use the setEscape() method at the controller level to tell Zend_View what escaping callback to use.

```
// create a Zend_View instance
$view = new Zend_View();

// tell it to use htmlentities as the escaping callback
$view->setEscape('htmlentities');

// or tell it to use a static class method as the callback
$view->setEscape(array('SomeClass', 'methodName'));

// or even an instance method
$obj = new SomeClass();
$view->setEscape(array($obj, 'methodName'));

// and then render your view
```

```
echo $view->render(...);
```

The callback function or method should take the value to be escaped as its first parameter, and all other parameters should be optional.

3.2. Using Alternate Template Systems

Although PHP is itself a powerful template system, many developers feel it is too powerful or complex for their template designers and will want to use an alternate template engine. `Zend_View` provides two mechanisms for doing so, the first through view scripts, the second by implementing `Zend_View_Interface`.

3.2.1. Template Systems Using View Scripts

A view script may be used to instantiate and manipulate a separate template object, such as a PHPLIB-style template. The view script for that kind of activity might look something like this:

```
include_once 'template.inc';
$tpl = new Template();

if ($this->books) {
    $tpl->setFile(array(
        "booklist" => "booklist.tpl",
        "eachbook" => "eachbook.tpl",
    ));

    foreach ($this->books as $key => $val) {
        $tpl->set_var('author', $this->escape($val['author']);
        $tpl->set_var('title', $this->escape($val['title']);
        $tpl->parse("books", "eachbook", true);
    }

    $tpl->pparse("output", "booklist");
} else {
    $tpl->setFile("nobooks", "nobooks.tpl")
    $tpl->pparse("output", "nobooks");
}
```

These would be the related template files:

```
<!-- booklist.tpl -->
<table>
    <tr>
        <th>Author</th>
        <th>Title</th>
    </tr>
    {books}
</table>

<!-- eachbook.tpl -->
    <tr>
        <td>{author}</td>
        <td>{title}</td>
    </tr>

<!-- nobooks.tpl -->
<p>There are no books to display.</p>
```

3.2.2. Template Systems Using Zend_View_Interface

Some may find it easier to simply provide a `Zend_View`-compatible template engine. `Zend_View_Interface` defines the minimum interface needed for compatability:

```
/**
 * Return the actual template engine object
 */
public function getEngine();

/**
 * Set the path to view scripts/templates
 */
public function setScriptPath($path);

/**
 * Set a base path to all view resources
 */
public function setBasePath($path, $prefix = 'Zend_View');

/**
 * Add an additional base path to view resources
 */
public function addBasePath($path, $prefix = 'Zend_View');

/**
 * Retrieve the current script paths
 */
public function getScriptPaths();

/**
 * Overloading methods for assigning template variables as object
 * properties
 */
public function __set($key, $value);
public function __isset($key);
public function __unset($key);

/**
 * Manual assignment of template variables, or ability to assign
 * multiple variables en masse.
 */
public function assign($spec, $value = null);

/**
 * Unset all assigned template variables
 */
public function clearVars();

/**
 * Render the template named $name
 */
public function render($name);
```

Using this interface, it becomes relatively easy to wrap a third-party template engine as a `Zend_View`-compatible class. As an example, the following is one potential wrapper for Smarty:

```
class Zend_View_Smarty implements Zend_View_Interface
{
```

```
    /**
     * Smarty object
     * @var Smarty
     */
    protected $_smarty;

    /**
     * Constructor
     *
     * @param string $tmplPath
     * @param array $extraParams
     * @return void
     */
    public function __construct($tmplPath = null, $extraParams = array())
    {
        $this->_smarty = new Smarty;

        if (null !== $tmplPath) {
            $this->setScriptPath($tmplPath);
        }

        foreach ($extraParams as $key => $value) {
            $this->_smarty->$key = $value;
        }
    }

    /**
     * Return the template engine object
     *
     * @return Smarty
     */
    public function getEngine()
    {
        return $this->_smarty;
    }

    /**
     * Set the path to the templates
     *
     * @param string $path The directory to set as the path.
     * @return void
     */
    public function setScriptPath($path)
    {
        if (is_readable($path)) {
            $this->_smarty->template_dir = $path;
            return;
        }

        throw new Exception('Invalid path provided');
    }

    /**
     * Retrieve the current template directory
     *
     * @return string
     */
    public function getScriptPaths()
    {
        return array($this->_smarty->template_dir);
```

```
    }

    /**
     * Alias for setScriptPath
     *
     * @param string $path
     * @param string $prefix Unused
     * @return void
     */
    public function setBasePath($path, $prefix = 'Zend_View')
    {
        return $this->setScriptPath($path);
    }

    /**
     * Alias for setScriptPath
     *
     * @param string $path
     * @param string $prefix Unused
     * @return void
     */
    public function addBasePath($path, $prefix = 'Zend_View')
    {
        return $this->setScriptPath($path);
    }

    /**
     * Assign a variable to the template
     *
     * @param string $key The variable name.
     * @param mixed $val The variable value.
     * @return void
     */
    public function __set($key, $val)
    {
        $this->_smarty->assign($key, $val);
    }

    /**
     * Allows testing with empty() and isset() to work
     *
     * @param string $key
     * @return boolean
     */
    public function __isset($key)
    {
        return (null !== $this->_smarty->get_template_vars($key));
    }

    /**
     * Allows unset() on object properties to work
     *
     * @param string $key
     * @return void
     */
    public function __unset($key)
    {
        $this->_smarty->clear_assign($key);
    }
```

```
    /**
     * Assign variables to the template
     *
     * Allows setting a specific key to the specified value, OR passing
     * an array of key => value pairs to set en masse.
     *
     * @see __set()
     * @param string|array $spec The assignment strategy to use (key or
     * array of key => value pairs)
     * @param mixed $value (Optional) If assigning a named variable,
     * use this as the value.
     * @return void
     */
    public function assign($spec, $value = null)
    {
        if (is_array($spec)) {
            $this->_smarty->assign($spec);
            return;
        }

        $this->_smarty->assign($spec, $value);
    }

    /**
     * Clear all assigned variables
     *
     * Clears all variables assigned to Zend_View either via
     * {@link assign()} or property overloading
     * ({@link __get()}/{@link __set()}).
     *
     * @return void
     */
    public function clearVars()
    {
        $this->_smarty->clear_all_assign();
    }

    /**
     * Processes a template and returns the output.
     *
     * @param string $name The template to process.
     * @return string The output.
     */
    public function render($name)
    {
        return $this->_smarty->fetch($name);
    }
}
```

In this example, you would instantiate the `Zend_View_Smarty` class instead of `Zend_View`, and then use it in roughly the same fashion as `Zend_View`:

```
//Example 1. In initView() of initializer.
$view = new Zend_View_Smarty('/path/to/templates');
$viewRenderer =
    Zend_Controller_Action_HelperBroker::getStaticHelper('ViewRenderer');
$viewRenderer->setView($view)
             ->setViewBasePathSpec($view->_smarty->template_dir)
             ->setViewScriptPathSpec(':controller/:action.:suffix')
```

```
            ->setViewScriptPathNoControllerSpec(':action.:suffix')
            ->setViewSuffix('tpl');

//Example 2. Usage in action controller remains the same...
class FooController extends Zend_Controller_Action
{
    public function barAction()
    {
        $this->view->book   = 'Zend PHP 5 Certification Study Guide';
        $this->view->author = 'Davey Shafik and Ben Ramsey'
    }
}

//Example 3. Initializing view in action controller
class FooController extends Zend_Controller_Action
{
    public function init()
    {
        $this->view   = new Zend_View_Smarty('/path/to/templates');
        $viewRenderer = $this->_helper->getHelper('viewRenderer');
        $viewRenderer->setView($this->view)
                     ->setViewBasePathSpec($view->_smarty->template_dir)
                     ->setViewScriptPathSpec(':controller/:action.:suffix')
                     ->setViewScriptPathNoControllerSpec(':action.:suffix')
                     ->setViewSuffix('tpl');
    }
```

4. View Helpers

In your view scripts, often it is necessary to perform certain complex functions over and over: e.g., formatting a date, generating form elements, or displaying action links. You can use helper classes to perform these behaviors for you.

A helper is simply a class. Let's say we want a helper named 'fooBar'. By default, the class is prefixed with 'Zend_View_Helper_' (you can specify a custom prefix when setting a helper path), and the last segment of the class name is the helper name; this segment should be TitleCapped; the full class name is then: `Zend_View_Helper_FooBar`. This class should contain at the minimum a single method, named after the helper, and camelCased: `fooBar()`.

Watch the Case

Helper names are always camelCased, i.e., they never begin with an uppercase character. The class name itself is MixedCased, but the method that is actually executed is camelCased.

Default Helper Path

The default helper path always points to the Zend Framework view helpers, i.e., 'Zend/View/Helper/'. Even if you call `setHelperPath()` to overwrite the existing paths, this path will be set to ensure the default helpers work.

To use a helper in your view script, call it using **$this->helperName()**. Behind the scenes, `Zend_View` will load the `Zend_View_Helper_HelperName` class, create an object instance of it, and call its `helperName()` method. The object instance is persistent within the `Zend_View` instance, and is reused for all future calls to **$this->helperName()**.

4.1. Initial Helpers

`Zend_View` comes with an initial set of helper classes, most of which relate to form element generation and perform the appropriate output escaping automatically. In addition, there are helpers for creating route-based URLs and HTML lists, as well as declaring variables. The currently shipped helpers include:

- `declareVars()`: Primarily for use when using `strictVars()`, this helper can be used to declare template variables that may or may not already be set in the view object, as well as to set default values. Arrays passed as arguments to the method will be used to set default values; otherwise, if the variable does not exist, it is set to an empty string.

- `fieldset($name, $content, $attribs)`: Creates an XHTML fieldset. If `$attribs` contains a 'legend' key, that value will be used for the fieldset legend. The fieldset will surround the `$content` as provided to the helper.

- `form($name, $attribs, $content)`: Generates an XHTML form. All `$attribs` are escaped and rendered as XHTML attributes of the form tag. If `$content` is present and not a boolean FALSE, then that content is rendered within the start and close form tags; if `$content` is a boolean FALSE (the default), only the opening form tag is generated.

- `formButton($name, $value, $attribs)`: Creates an <button /> element.

- `formCheckbox($name, $value, $attribs, $options)`: Creates an <input type="checkbox" /> element.

  By default, when no $value is provided and no $options are present, '0' is assumed to be the unchecked value, and '1' the checked value. If a $value is passed, but no $options are present, the checked value is assumed to be the value passed.

  $options should be an array. If the array is indexed, the first value is the checked value, and the second the unchecked value; all other values are ignored. You may also pass an associative array with the keys 'checked' and 'unChecked'.

  If $options has been passed, if $value matches the checked value, then the element will be marked as checked. You may also mark the element as checked or unchecked by passing a boolean value for the attribute 'checked'.

  The above is probably best summed up with some examples:

```
// '1' and '0' as checked/unchecked options; not checked
echo $this->formCheckbox('foo');

// '1' and '0' as checked/unchecked options; checked
echo $this->formCheckbox('foo', null, array('checked' => true));

// 'bar' and '0' as checked/unchecked options; not checked
echo $this->formCheckbox('foo', 'bar');

// 'bar' and '0' as checked/unchecked options; checked
echo $this->formCheckbox('foo', 'bar', array('checked' => true));

// 'bar' and 'baz' as checked/unchecked options; unchecked
echo $this->formCheckbox('foo', null, null, array('bar', 'baz'));

// 'bar' and 'baz' as checked/unchecked options; unchecked
echo $this->formCheckbox('foo', null, null, array(
```

```
    'checked' => 'bar',
    'unChecked' => 'baz'
));

// 'bar' and 'baz' as checked/unchecked options; checked
echo $this->formCheckbox('foo', 'bar', null, array('bar', 'baz'));
echo $this->formCheckbox('foo',
                         null,
                         array('checked' => true),
                         array('bar', 'baz'));

// 'bar' and 'baz' as checked/unchecked options; unchecked
echo $this->formCheckbox('foo', 'baz', null, array('bar', 'baz'));
echo $this->formCheckbox('foo',
                         null,
                         array('checked' => false),
                         array('bar', 'baz'));
```

In all cases, the markup prepends a hidden element with the unchecked value; this way, if the value is unchecked, you will still get a valid value returned to your form.

- `formErrors($errors, $options)`: Generates an XHTML unordered list to show errors. `$errors` should be a string or an array of strings; `$options` should be any attributes you want placed in the opening list tag.

  You can specify alternate opening, closing, and separator content when rendering the errors by calling several methods on the helper:

  - `setElementStart($string)`; default is '<ul class="errors"%s">
', where %s is replaced with the attributes as specified in `$options`.
  - `setElementSeparator($string)`; default is '
'.
  - `setElementEnd($string)`; default is '
'.

- `formFile($name, $attribs)`: Creates an <input type="file" /> element.
- `formHidden($name, $value, $attribs)`: Creates an <input type="hidden" /> element.
- `formLabel($name, $value, $attribs)`: Creates a <label> element, setting the for attribute to `$name`, and the actual label text to `$value`. If *disable* is passed in attribs, nothing will be returned.
- `formMultiCheckbox($name, $value, $attribs, $options, $listsep)`: Creates a list of checkboxes. `$options` should be an associative array, and may be arbitrarily deep. `$value` may be a single value or an array of selected values that match the keys in the `$options` array. `$listsep` is an HTML break ("<br />") by default. By default, this element is treated as an array; all checkboxes share the same name, and are submitted as an array.
- `formPassword($name, $value, $attribs)`: Creates an <input type="password" /> element.
- `formRadio($name, $value, $attribs, $options)`: Creates a series of <input type="radio" /> elements, one for each of the $options elements. In the $options array, the element key is the radio value, and the element value is the radio label. The $value radio will be preselected for you.
- `formReset($name, $value, $attribs)`: Creates an <input type="reset" /> element.

- `formSelect($name, $value, $attribs, $options)`: Creates a <select>...</select> block, with one <option>one for each of the $options elements. In the $options array, the element key is the option value, and the element value is the option label. The $value option(s) will be preselected for you.
- `formSubmit($name, $value, $attribs)`: Creates an <input type="submit" /> element.
- `formText($name, $value, $attribs)`: Creates an <input type="text" /> element.
- `formTextarea($name, $value, $attribs)`: Creates a <textarea>...</textarea> block.
- `url($urlOptions, $name, $reset)`: Creates a URL string based on a named route. `$urlOptions` should be an associative array of key/value pairs used by the particular route.
- `htmlList($items, $ordered, $attribs, $escape)`: generates unordered and ordered lists based on the `$items` passed to it. If `$items` is a multidimensional array, a nested list will be built. If the `$escape` flag is TRUE (default), individual items will be escaped using the view objects registered escaping mechanisms; pass a FALSE value if you want to allow markup in your lists.

Using these in your view scripts is very easy, here is an example. Note that you all you need to do is call them; they will load and instantiate themselves as they are needed.

```
// inside your view script, $this refers to the Zend_View instance.
//
// say that you have already assigned a series of select options under
// the name $countries as array('us' => 'United States', 'il' =>
// 'Israel', 'de' => 'Germany').
?>
<form action="action.php" method="post">
    <p><label>Your Email:
<?php echo $this->formText('email', 'you@example.com', array('size' => 32)) ?>
    </label></p>
    <p><label>Your Country:
<?php echo $this->formSelect('country', 'us', null, $this->countries) ?>
    </label></p>
    <p><label>Would you like to opt in?
<?php echo $this->formCheckbox('opt_in', 'yes', null, array('yes', 'no')) ?>
    </label></p>
</form>
```

The resulting output from the view script will look something like this:

```
<form action="action.php" method="post">
    <p><label>Your Email:
        <input type="text" name="email" value="you@example.com" size="32" />
    </label></p>
    <p><label>Your Country:
        <select name="country">
            <option value="us" selected="selected">United States</option>
            <option value="il">Israel</option>
            <option value="de">Germany</option>
        </select>
    </label></p>
    <p><label>Would you like to opt in?
        <input type="hidden" name="opt_in" value="no" />
        <input type="checkbox" name="opt_in" value="yes" checked="checked" />
    </label></p>
```

```
</form>
```

4.1.1. Action View Helper

The `Action` view helper enables view scripts to dispatch a given controller action; the result of the response object following the dispatch is then returned. These can be used when a particular action could generate re-usable content or "widget-ized" content.

Actions that result in a `_forward()` or redirect are considered invalid, and will return an empty string.

The API for the `Action` view helper follows that of most MVC components that invoke controller actions: **action($action, $controller, $module = null, array $params = array())**. `$action` and `$controller` are required; if no module is specified, the default module is assumed.

Example 912. Basic Usage of Action View Helper

As an example, you may have a `CommentController` with a `listAction()` method you wish to invoke in order to pull a list of comments for the current request:

```
<div id="sidebar right">
    <div class="item">
        <?php echo $this->action('list',
                                 'comment',
                                 null,
                                 array('count' => 10)); ?>
    </div>
</div>
```

4.1.2. BaseUrl Helper

While most URLs generated by the framework have the base URL prepended automatically, developers will need to prepend the base URL to their own URLs in order for paths to resources to be correct.

Usage of the BaseUrl helper is very straightforward:

```
/*
 * The following assume that the base URL of the page/application is "/mypage".
 */

/*
 * Prints:
 * <base href="/mypage/" />
 */
<base href="<?php echo $this->baseUrl(); ?>" />

/*
 * Prints:
 * <link rel="stylesheet" type="text/css" href="/mypage/css/base.css" />
 */
<link rel="stylesheet" type="text/css"
     href="<?php echo $this->baseUrl('css/base.css'); ?>" />
```

For simplicity's sake, we strip out the entry PHP file (e.g., "`index.php`") from the base URL that was contained in `Zend_Controller`. However,

in some situations this may cause a problem. If one occurs, use `$this->getHelper('BaseUrl')->setBaseUrl()` to set your own BaseUrl.

4.1.3. Currency Helper

Displaying localized currency values is a common task; the `Zend_Currency` view helper is intended to simply this task. See the Zend Currency documentation for specifics on this localization feature. In this section, we will focus simply on usage of the view helper.

There are several ways to initiate the *Currency* view helper:

- Registered, through a previously registered instance in `Zend_Registry`.
- Afterwards, through the fluent interface.
- Directly, through instantiating the class.

A registered instance of `Zend_Currency` is the preferred usage for this helper. Doing so, you can select the currency to be used prior to adding the adapter to the registry.

There are several ways to select the desired currency. First, you may simply provide a currency string; alternately, you may specify a locale. The preferred way is to use a locale as this information is automatically detected and selected via the HTTP client headers provided when a user accesses your application, and ensures the currency provided will match their locale.

We are speaking of "locales" instead of "languages" because a language may vary based on the geographical region in which it is used. For example, English is spoken in different dialects: British English, American English, etc. As a currency always correlates to a country you must give a fully-qualified locale, which means providing both the language *and* region. Therefore, we say "locale" instead of "language."

Example 913. Registered instance

To use a registered instance, simply create an instance of `Zend_Currency` and register it within `Zend_Registry` using `Zend_Currency` as its key.

```
// our example currency
$currency = new Zend_Currency('de_AT');
Zend_Registry::set('Zend_Currency', $currency);

// within your view
echo $this->currency(1234.56);
// this returns '€ 1.234,56'
```

If you are more familiar with the fluent interface, then you can also create an instance within your view and configure the helper afterwards.

Example 914. Within the view

To use the fluent interface, create an instance of `Zend_Currency`, call the helper without a parameter, and call the `setCurrency()` method.

```
// within your view
$currency = new Zend_Currency('de_AT');
$this->currency()->setCurrency($currency)->currency(1234.56);
// this returns '€ 1.234,56'
```

If you are using the helper without `Zend_View` then you can also use it directly.

Example 915. Direct usage

```
// our example currency
$currency = new Zend_Currency('de_AT');

// initiate the helper
$helper = new Zend_View_Helper_Currency($currency);
echo $helper->currency(1234.56); // this returns '€ 1.234,56'
```

As already seen, the `currency()` method is used to return the currency string. Just call it with the value you want to display as a currency. It also accepts some options which may be used to change the behaviour and output of the helper.

Example 916. Direct usage

```
// our example currency
$currency = new Zend_Currency('de_AT');

// initiate the helper
$helper = new Zend_View_Helper_Currency($currency);
echo $helper->currency(1234.56); // this returns '€ 1.234,56'
echo $helper->currency(1234.56, array('precision' => 1));
// this returns '€ 1.234,6'
```

For details about the available options, search for `Zend_Currency`'s `toCurrency()` method.

4.1.4. Cycle Helper

The `Cycle` helper is used to alternate a set of values.

Example 917. Cycle Helper Basic Usage

To add elements to cycle just specify them in constructor or use `assign(array $data)` function

```
<?php foreach ($this->books as $book):?>
  <tr style="background-color:<?php echo $this->cycle(array("#F0F0F0",
                                                            "#FFFFFF"))
                                                 ->next()?>">
  <td><?php echo $this->escape($book['author']) ?></td>
</tr>
<?php endforeach;?>
// Moving in backwards order and assign function
$this->cycle()->assign(array("#F0F0F0","#FFFFFF"));
$this->cycle()->prev();
?>
```

The output

```
<tr style="background-color:'#F0F0F0'">
   <td>First</td>
</tr>
<tr style="background-color:'#FFFFFF'">
   <td>Second</td>
</tr>
```

Example 918. Working with two or more cycles

To use two cycles you have to specify the names of cycles. Just set second parameter in cycle method. **$this->cycle(array("#F0F0F0","#FFFFFF"),'cycle2')**. You can also use setName($name) function.

```
<?php foreach ($this->books as $book):?>
  <tr style="background-color:<?php echo $this->cycle(array("#F0F0F0",
                                                            "#FFFFFF"))
                                                ->next()?>">
  <td><?php echo $this->cycle(array(1,2,3),'number')->next()?></td>
  <td><?php echo $this->escape($book['author'])?></td>
</tr>
<?php endforeach;?>
```

4.1.5. Partial Helper

The `Partial` view helper is used to render a specified template within its own variable scope. The primary use is for reusable template fragments with which you do not need to worry about variable name clashes. Additionally, they allow you to specify partial view scripts from specific modules.

A sibling to the `Partial`, the `PartialLoop` view helper allows you to pass iterable data, and render a partial for each item.

PartialLoop Counter

The `PartialLoop` view helper assigns a variable to the view named *partialCounter* which passes the current position of the array to the view script. This provides an easy way to have alternating colors on table rows for example.

Example 919. Basic Usage of Partials

Basic usage of partials is to render a template fragment in its own view scope. Consider the following partial script:

```
<?php // partial.phtml ?>
<ul>
    <li>From: <?php echo $this->escape($this->from) ?></li>
    <li>Subject: <?php echo $this->escape($this->subject) ?></li>
</ul>
```

You would then call it from your view script using the following:

```
<?php echo $this->partial('partial.phtml', array(
    'from' => 'Team Framework',
    'subject' => 'view partials')); ?>
```

Which would then render:

```
<ul>
    <li>From: Team Framework</li>
    <li>Subject: view partials</li>
</ul>
```

What is a model?

A model used with the `Partial` view helper can be one of the following:

- *Array*. If an array is passed, it should be associative, as its key/value pairs are assigned to the view with keys as view variables.
- *Object implementing toArray() method*. If an object is passed an has a `toArray()` method, the results of `toArray()` will be assigned to the view object as view variables.
- *Standard object*. Any other object will assign the results of `object_get_vars()` (essentially all public properties of the object) to the view object.

If your model is an object, you may want to have it passed *as an object* to the partial script, instead of serializing it to an array of variables. You can do this by setting the 'objectKey' property of the appropriate helper:

```
// Tell partial to pass objects as 'model' variable
$view->partial()->setObjectKey('model');

// Tell partial to pass objects from partialLoop as 'model' variable
// in final partial view script:
$view->partialLoop()->setObjectKey('model');
```

This technique is particularly useful when passing `Zend_Db_Table_Rowsets` to `partialLoop()`, as you then have full access to your row objects within the view scripts, allowing you to call methods on them (such as retrieving values from parent or dependent rows).

Example 920. Using PartialLoop to Render Iterable Models

Typically, you'll want to use partials in a loop, to render the same content fragment many times; this way you can put large blocks of repeated content or complex display logic into a single location. However this has a performance impact, as the partial helper needs to be invoked once for each iteration.

The `PartialLoop` view helper helps solve this issue. It allows you to pass an iterable item (array or object implementing *Iterator*) as the model. It then iterates over this, passing, the items to the partial script as the model. Items in the iterator may be any model the `Partial` view helper allows.

Let's assume the following partial view script:

```
<?php // partialLoop.phtml ?>
    <dt><?php echo $this->key ?></dt>
    <dd><?php echo $this->value ?></dd>
```

And the following "model":

```
$model = array(
    array('key' => 'Mammal', 'value' => 'Camel'),
    array('key' => 'Bird', 'value' => 'Penguin'),
    array('key' => 'Reptile', 'value' => 'Asp'),
    array('key' => 'Fish', 'value' => 'Flounder'),
);
```

In your view script, you could then invoke the `PartialLoop` helper:

```
<dl>
<?php echo $this->partialLoop('partialLoop.phtml', $model) ?>
</dl>
```

```
<dl>
    <dt>Mammal</dt>
    <dd>Camel</dd>

    <dt>Bird</dt>
    <dd>Penguin</dd>

    <dt>Reptile</dt>
    <dd>Asp</dd>

    <dt>Fish</dt>
    <dd>Flounder</dd>
</dl>
```

Example 921. Rendering Partials in Other Modules

Sometime a partial will exist in a different module. If you know the name of the module, you can pass it as the second argument to either `partial()` or `partialLoop()`, moving the `$model` argument to third position.

For instance, if there's a pager partial you wish to use that's in the 'list' module, you could grab it as follows:

```
<?php echo $this->partial('pager.phtml', 'list', $pagerData) ?>
```

In this way, you can re-use partials created specifically for other modules. That said, it's likely a better practice to put re-usable partials in shared view script paths.

4.1.6. Placeholder Helper

The `Placeholder` view helper is used to persist content between view scripts and view instances. It also offers some useful features such as aggregating content, capturing view script content for later use, and adding pre- and post-text to content (and custom separators for aggregated content).

Example 922. Basic Usage of Placeholders

Basic usage of placeholders is to persist view data. Each invocation of the `Placeholder` helper expects a placeholder name; the helper then returns a placeholder container object that you can either manipulate or simply echo out.

```
<?php $this->placeholder('foo')->set("Some text for later") ?>
<?php
    echo $this->placeholder('foo');
    // outputs "Some text for later"
?>
```

Example 923. Using Placeholders to Aggregate Content

Aggregating content via placeholders can be useful at times as well. For instance, your view script may have a variable array from which you wish to retrieve messages to display later; a later view script can then determine how those will be rendered.

The `Placeholder` view helper uses containers that extend `ArrayObject`, providing a rich featureset for manipulating arrays. In addition, it offers a variety of methods for formatting the content stored in the container:

- `setPrefix($prefix)` sets text with which to prefix the content. Use `getPrefix()` at any time to determine what the current setting is.
- `setPostfix($prefix)` sets text with which to append the content. Use `getPostfix()` at any time to determine what the current setting is.
- `setSeparator($prefix)` sets text with which to separate aggregated content. Use `getSeparator()` at any time to determine what the current setting is.
- `setIndent($prefix)` can be used to set an indentation value for content. If an integer is passed, that number of spaces will be used; if a string is passed, the string will be used. Use `getIndent()` at any time to determine what the current setting is.

```
<!-- first view script -->
<?php $this->placeholder('foo')->exchangeArray($this->data) ?>
```

```
<!-- later view script -->
<?php
$this->placeholder('foo')->setPrefix("<ul>\n    <li>")
                         ->setSeparator("</li><li>\n")
                         ->setIndent(4)
                         ->setPostfix("</li></ul>\n");
?>
<?php
    echo $this->placeholder('foo');
    // outputs as unordered list with pretty indentation
?>
```

Because the `Placeholder` container objects extend `ArrayObject`, you can also assign content to a specific key in the container easily, instead of simply pushing it into the container. Keys may be accessed either as object properties or as array keys.

```
<?php $this->placeholder('foo')->bar = $this->data ?>
<?php echo $this->placeholder('foo')->bar ?>
<?php
$foo = $this->placeholder('foo');
echo $foo['bar'];
?>
```

Example 924. Using Placeholders to Capture Content

Occasionally you may have content for a placeholder in a view script that is easiest to template; the `Placeholder` view helper allows you to capture arbitrary content for later rendering using the following API.

- `captureStart($type, $key)` begins capturing content.

  `$type` should be one of the `Placeholder` constants `APPEND` or `SET`. If `APPEND`, captured content is appended to the list of current content in the placeholder; if `SET`, captured content is used as the sole value of the placeholder (potentially replacing any previous content). By default, `$type` is `APPEND`.

  `$key` can be used to specify a specific key in the placeholder container to which you want content captured.

  `captureStart()` locks capturing until `captureEnd()` is called; you cannot nest capturing with the same placeholder container. Doing so will raise an exception.

- `captureEnd()` stops capturing content, and places it in the container object according to how `captureStart()` was called.

```
<!-- Default capture: append -->
<?php $this->placeholder('foo')->captureStart();
foreach ($this->data as $datum): ?>
<div class="foo">
    <h2><?php echo $datum->title ?></h2>
    <p><?php echo $datum->content ?></p>
</div>
<?php endforeach; ?>
<?php $this->placeholder('foo')->captureEnd() ?>
<?php echo $this->placeholder('foo') ?>
```

```
<!-- Capture to key -->
<?php $this->placeholder('foo')->captureStart('SET', 'data');
foreach ($this->data as $datum): ?>
<div class="foo">
    <h2><?php echo $datum->title ?></h2>
    <p><?php echo $datum->content ?></p>
</div>
 <?php endforeach; ?>
<?php $this->placeholder('foo')->captureEnd() ?>
<?php echo $this->placeholder('foo')->data ?>
```

4.1.6.1. Concrete Placeholder Implementations

Zend Framework ships with a number of "concrete" placeholder implementations. These are for commonly used placeholders: doctype, page title, and various <head> elements. In all cases, calling the placeholder with no arguments returns the element itself.

Documentation for each element is covered separately, as linked below:

- Doctype
- HeadLink
- HeadMeta

- HeadScript
- HeadStyle
- HeadTitle
- InlineScript

4.1.7. Doctype Helper

Valid HTML and XHTML documents should include a `DOCTYPE` declaration. Besides being difficult to remember, these can also affect how certain elements in your document should be rendered (for instance, CDATA escaping in <*script*> and <*style*> elements.

The `Doctype` helper allows you to specify one of the following types:

- `XHTML11`
- `XHTML1_STRICT`
- `XHTML1_TRANSITIONAL`
- `XHTML1_FRAMESET`
- `XHTML_BASIC1`
- `HTML4_STRICT`
- `HTML4_LOOSE`
- `HTML4_FRAMESET`
- `HTML5`

You can also specify a custom doctype as long as it is well-formed.

The `Doctype` helper is a concrete implementation of the Placeholder helper.

Example 925. Doctype Helper Basic Usage

You may specify the doctype at any time. However, helpers that depend on the doctype for their output will recognize it only after you have set it, so the easiest approach is to specify it in your bootstrap:

```
$doctypeHelper = new Zend_View_Helper_Doctype();
$doctypeHelper->doctype('XHTML1_STRICT');
```

And then print it out on top of your layout script:

```
<?php echo $this->doctype() ?>
```

Example 926. Retrieving the Doctype

If you need to know the doctype, you can do so by calling `getDoctype()` on the object, which is returned by invoking the helper.

```
$doctype = $view->doctype()->getDoctype();
```

Typically, you'll simply want to know if the doctype is XHTML or not; for this, the `isXhtml()` method will suffice:

```
if ($view->doctype()->isXhtml()) {
    // do something differently
}
```

You can also check if the doctype represents an HTML5 document

```
if ($view->doctype()->isHtml5()) {
    // do something differently
}
```

4.1.8. HeadLink Helper

The HTML *<link>* element is increasingly used for linking a variety of resources for your site: stylesheets, feeds, favicons, trackbacks, and more. The `HeadLink` helper provides a simple interface for creating and aggregating these elements for later retrieval and output in your layout script.

The `HeadLink` helper has special methods for adding stylesheet links to its stack:

- **appendStylesheet($href, $media, $conditionalStylesheet, $extras)**
- **offsetSetStylesheet($index, $href, $media, $conditionalStylesheet, $extras)**
- **prependStylesheet($href, $media, $conditionalStylesheet, $extras)**
- **setStylesheet($href, $media, $conditionalStylesheet, $extras)**

The `$media` value defaults to 'screen', but may be any valid media value. `$conditionalStylesheet` is a string or boolean `FALSE`, and will be used at rendering time to determine if special comments should be included to prevent loading of the stylesheet on certain platforms. `$extras` is an array of any extra values that you want to be added to the tag.

Additionally, the `HeadLink` helper has special methods for adding 'alternate' links to its stack:

- **appendAlternate($href, $type, $title, $extras)**
- **offsetSetAlternate($index, $href, $type, $title, $extras)**
- **prependAlternate($href, $type, $title, $extras)**
- **setAlternate($href, $type, $title, $extras)**

The `headLink()` helper method allows specifying all attributes necessary for a *<link>* element, and allows you to also specify placement -- whether the new element replaces all others, prepends (top of stack), or appends (end of stack).

The `HeadLink` helper is a concrete implementation of the Placeholder helper.

Example 927. HeadLink Helper Basic Usage

You may specify a *headLink* at any time. Typically, you will specify global links in your layout script, and application specific links in your application view scripts. In your layout script, in the <head> section, you will then echo the helper to output it.

```
<?php // setting links in a view script:
$this->headLink()->appendStylesheet('/styles/basic.css')
                 ->headLink(array('rel' => 'favicon',
                                  'href' => '/img/favicon.ico'),
                                  'PREPEND')
                 ->prependStylesheet('/styles/moz.css',
                                     'screen',
                                     true,
                                     array('id' => 'my_stylesheet'));
?>
<?php // rendering the links: ?>
<?php echo $this->headLink() ?>
```

4.1.9. HeadMeta Helper

The HTML <*meta*> element is used to provide meta information about your HTML document -- typically keywords, document character set, caching pragmas, etc. Meta tags may be either of the 'http-equiv' or 'name' types, must contain a 'content' attribute, and can also have either of the 'lang' or 'scheme' modifier attributes.

The `HeadMeta` helper supports the following methods for setting and adding meta tags:

- **appendName($keyValue, $content, $conditionalName)**
- **offsetSetName($index, $keyValue, $content, $conditionalName)**
- **prependName($keyValue, $content, $conditionalName)**
- **setName($keyValue, $content, $modifiers)**
- **appendHttpEquiv($keyValue, $content, $conditionalHttpEquiv)**
- **offsetSetHttpEquiv($index, $keyValue, $content, $conditionalHttpEquiv)**
- **prependHttpEquiv($keyValue, $content, $conditionalHttpEquiv)**
- **setHttpEquiv($keyValue, $content, $modifiers)**
- **setCharset($charset)**

The `$keyValue` item is used to define a value for the 'name' or 'http-equiv' key; `$content` is the value for the 'content' key, and `$modifiers` is an optional associative array that can contain keys for 'lang' and/or 'scheme'.

You may also set meta tags using the `headMeta()` helper method, which has the following signature: **headMeta($content, $keyValue, $keyType = 'name', $modifiers = array(), $placement = 'APPEND')**. `$keyValue` is the content for the key specified in `$keyType`, which

should be either 'name' or 'http-equiv'. `$placement` can be either 'SET' (overwrites all previously stored values), 'APPEND' (added to end of stack), or 'PREPEND' (added to top of stack).

`HeadMeta` overrides each of `append()`, `offsetSet()`, `prepend()`, and `set()` to enforce usage of the special methods as listed above. Internally, it stores each item as a stdClass token, which it later serializes using the `itemToString()` method. This allows you to perform checks on the items in the stack, and optionally modify these items by simply modifying the object returned.

The `HeadMeta` helper is a concrete implementation of the Placeholder helper.

Example 928. HeadMeta Helper Basic Usage

You may specify a new meta tag at any time. Typically, you will specify client-side caching rules or SEO keywords.

For instance, if you wish to specify SEO keywords, you'd be creating a meta name tag with the name 'keywords' and the content the keywords you wish to associate with your page:

```
// setting meta keywords
$this->headMeta()->appendName('keywords', 'framework, PHP, productivity');
```

If you wished to set some client-side caching rules, you'd set http-equiv tags with the rules you wish to enforce:

```
// disabling client-side cache
$this->headMeta()->appendHttpEquiv('expires',
                                   'Wed, 26 Feb 1997 08:21:57 GMT')
                 ->appendHttpEquiv('pragma', 'no-cache')
                 ->appendHttpEquiv('Cache-Control', 'no-cache');
```

Another popular use for meta tags is setting the content type, character set, and language:

```
// setting content type and character set
$this->headMeta()->appendHttpEquiv('Content-Type',
                                   'text/html; charset=UTF-8')
                 ->appendHttpEquiv('Content-Language', 'en-US');
```

If you are serving an HTML5 document, you should provide the character set like this:

```
// setting character set in HTML5
$this->headMeta()->setCharset('UTF-8'); // Will look like <meta charset="UTF-8">
```

As a final example, an easy way to display a transitional message before a redirect is using a "meta refresh":

```
// setting a meta refresh for 3 seconds to a new url:
$this->headMeta()->appendHttpEquiv('Refresh',
                                   '3;URL=http://www.some.org/some.html');
```

When you're ready to place your meta tags in the layout, simply echo the helper:

```
<?php echo $this->headMeta() ?>
```

4.1.10. HeadScript Helper

The HTML <*script*> element is used to either provide inline client-side scripting elements or link to a remote resource containing client-side scripting code. The `HeadScript` helper allows you to manage both.

The `HeadScript` helper supports the following methods for setting and adding scripts:

- **appendFile($src, $type = 'text/javascript', $attrs = array())**
- **offsetSetFile($index, $src, $type = 'text/javascript', $attrs = array())**
- **prependFile($src, $type = 'text/javascript', $attrs = array())**
- **setFile($src, $type = 'text/javascript', $attrs = array())**
- **appendScript($script, $type = 'text/javascript', $attrs = array())**
- **offsetSetScript($index, $script, $type = 'text/javascript', $attrs = array())**
- **prependScript($script, $type = 'text/javascript', $attrs = array())**
- **setScript($script, $type = 'text/javascript', $attrs = array())**

In the case of the `*File()` methods, `$src` is the remote location of the script to load; this is usually in the form of a URL or a path. For the `*Script()` methods, `$script` is the client-side scripting directives you wish to use in the element.

Setting Conditional Comments

`HeadScript` allows you to wrap the script tag in conditional comments, which allows you to hide it from specific browsers. To add the conditional tags, pass the conditional value as part of the `$attrs` parameter in the method calls.

Example 929. Headscript With Conditional Comments

```
// adding scripts
$this->headScript()->appendFile(
    '/js/prototype.js',
    'text/javascript',
    array('conditional' => 'lt IE 7')
);
```

`HeadScript` also allows capturing scripts; this can be useful if you want to create the client-side script programmatically, and then place it elsewhere. The usage for this will be showed in an example below.

Finally, you can also use the `headScript()` method to quickly add script elements; the signature for this is `headScript($mode = 'FILE', $spec, $placement = 'APPEND')`. The `$mode` is either 'FILE' or 'SCRIPT', depending on if you're linking a script or defining one. `$spec` is either the script file to link or the script source itself. `$placement` should be either 'APPEND', 'PREPEND', or 'SET'.

`HeadScript` overrides each of `append()`, `offsetSet()`, `prepend()`, and `set()` to enforce usage of the special methods as listed above. Internally, it stores each item as a stdClass token,

which it later serializes using the `itemToString()` method. This allows you to perform checks on the items in the stack, and optionally modify these items by simply modifying the object returned.

The `HeadScript` helper is a concrete implementation of the Placeholder helper.

Use InlineScript for HTML Body Scripts

`HeadScript`'s sibling helper, InlineScript, should be used when you wish to include scripts inline in the HTML *body*. Placing scripts at the end of your document is a good practice for speeding up delivery of your page, particularly when using 3rd party analytics scripts.

Arbitrary Attributes are Disabled by Default

By default, `HeadScript` only will render <*script*> attributes that are blessed by the W3C. These include 'type', 'charset', 'defer', 'language', and 'src'. However, some javascript frameworks, notably Dojo, utilize custom attributes in order to modify behavior. To allow such attributes, you can enable them via the `setAllowArbitraryAttributes()` method:

```
$this->headScript()->setAllowArbitraryAttributes(true);
```

Example 930. HeadScript Helper Basic Usage

You may specify a new script tag at any time. As noted above, these may be links to outside resource files or scripts themselves.

```
// adding scripts
$this->headScript()->appendFile('/js/prototype.js')
                   ->appendScript($onloadScript);
```

Order is often important with client-side scripting; you may need to ensure that libraries are loaded in a specific order due to dependencies each have; use the various append, prepend, and offsetSet directives to aid in this task:

```
// Putting scripts in order

// place at a particular offset to ensure loaded last
$this->headScript()->offsetSetFile(100, '/js/myfuncs.js');

// use scriptaculous effects (append uses next index, 101)
$this->headScript()->appendFile('/js/scriptaculous.js');

// but always have base prototype script load first:
$this->headScript()->prependFile('/js/prototype.js');
```

When you're finally ready to output all scripts in your layout script, simply echo the helper:

```
<?php echo $this->headScript() ?>
```

Example 931. Capturing Scripts Using the HeadScript Helper

Sometimes you need to generate client-side scripts programmatically. While you could use string concatenation, heredocs, and the like, often it's easier just to do so by creating the script and sprinkling in PHP tags. `HeadScript` lets you do just that, capturing it to the stack:

```
<?php $this->headScript()->captureStart() ?>
var action = '<?php echo $this->baseUrl ?>';
$('foo_form').action = action;
<?php $this->headScript()->captureEnd() ?>
```

The following assumptions are made:

- The script will be appended to the stack. If you wish for it to replace the stack or be added to the top, you will need to pass 'SET' or 'PREPEND', respectively, as the first argument to `captureStart()`.
- The script MIME type is assumed to be 'text/javascript'; if you wish to specify a different type, you will need to pass it as the second argument to `captureStart()`.
- If you wish to specify any additional attributes for the <*script*> tag, pass them in an array as the third argument to `captureStart()`.

4.1.11. HeadStyle Helper

The HTML <*style*> element is used to include CSS stylesheets inline in the HTML <*head*> element.

Use HeadLink to link CSS files

HeadLink should be used to create <*link*> elements for including external stylesheets. `HeadStyle` is used when you wish to define your stylesheets inline.

The `HeadStyle` helper supports the following methods for setting and adding stylesheet declarations:

- **appendStyle($content, $attributes = array())**
- **offsetSetStyle($index, $content, $attributes = array())**
- **prependStyle($content, $attributes = array())**
- **setStyle($content, $attributes = array())**

In all cases, `$content` is the actual CSS declarations. `$attributes` are any additional attributes you wish to provide to the style tag: lang, title, media, or dir are all permissible.

Setting Conditional Comments

`HeadStyle` allows you to wrap the style tag in conditional comments, which allows you to hide it from specific browsers. To add the conditional tags, pass the conditional value as part of the `$attributes` parameter in the method calls.

Example 932. Headstyle With Conditional Comments

```
// adding scripts
$this->headStyle()->appendStyle($styles, array('conditional' => 'lt IE 7'));
```

`HeadStyle` also allows capturing style declarations; this can be useful if you want to create the declarations programmatically, and then place them elsewhere. The usage for this will be showed in an example below.

Finally, you can also use the `headStyle()` method to quickly add declarations elements; the signature for this is `headStyle($content$placement = 'APPEND', $attributes = array())`. `$placement` should be either 'APPEND', 'PREPEND', or 'SET'.

`HeadStyle` overrides each of `append()`, `offsetSet()`, `prepend()`, and `set()` to enforce usage of the special methods as listed above. Internally, it stores each item as a stdClass token, which it later serializes using the `itemToString()` method. This allows you to perform checks on the items in the stack, and optionally modify these items by simply modifying the object returned.

The `HeadStyle` helper is a concrete implementation of the Placeholder helper.

UTF-8 encoding used by default

By default, Zend Framework uses UTF-8 as its default encoding, and, specific to this case, `Zend_View` does as well. Character encoding can be set differently on the view object itself using the `setEncoding()` method (or the the `encoding` instantiation parameter). However, since `Zend_View_Interface` does not define accessors for encoding, it's possible that if you are using a custom view implementation with this view helper, you will not have a `getEncoding()` method, which is what the view helper uses internally for determining the character set in which to encode.

If you do not want to utilize UTF-8 in such a situation, you will need to implement a `getEncoding()` method in your custom view implementation.

Example 933. HeadStyle Helper Basic Usage

You may specify a new style tag at any time:

```
// adding styles
$this->headStyle()->appendStyle($styles);
```

Order is very important with CSS; you may need to ensure that declarations are loaded in a specific order due to the order of the cascade; use the various append, prepend, and offsetSet directives to aid in this task:

```
// Putting styles in order

// place at a particular offset:
$this->headStyle()->offsetSetStyle(100, $customStyles);

// place at end:
$this->headStyle()->appendStyle($finalStyles);

// place at beginning
$this->headStyle()->prependStyle($firstStyles);
```

When you're finally ready to output all style declarations in your layout script, simply echo the helper:

```
<?php echo $this->headStyle() ?>
```

Example 934. Capturing Style Declarations Using the HeadStyle Helper

Sometimes you need to generate CSS style declarations programmatically. While you could use string concatenation, heredocs, and the like, often it's easier just to do so by creating the styles and sprinkling in PHP tags. `HeadStyle` lets you do just that, capturing it to the stack:

```
<?php $this->headStyle()->captureStart() ?>
body {
    background-color: <?php echo $this->bgColor ?>;
}
<?php $this->headStyle()->captureEnd() ?>
```

The following assumptions are made:

- The style declarations will be appended to the stack. If you wish for them to replace the stack or be added to the top, you will need to pass 'SET' or 'PREPEND', respectively, as the first argument to `captureStart()`.
- If you wish to specify any additional attributes for the *<style>* tag, pass them in an array as the second argument to `captureStart()`.

4.1.12. HeadTitle Helper

The HTML *<title>* element is used to provide a title for an HTML document. The `HeadTitle` helper allows you to programmatically create and store the title for later retrieval and output.

The `HeadTitle` helper is a concrete implementation of the Placeholder helper. It overrides the `toString()` method to enforce generating a *<title>* element, and adds a `headTitle()` method for quick and easy setting and aggregation of title elements. The signature for that method is `headTitle($title, $setType = 'APPEND')`; by default, the value is appended to the stack (aggregating title segments), but you may also specify either 'PREPEND' (place at top of stack) or 'SET' (overwrite stack).

Example 935. HeadTitle Helper Basic Usage

You may specify a title tag at any time. A typical usage would have you setting title segments for each level of depth in your application: site, controller, action, and potentially resource.

```
// setting the controller and action name as title segments:
$request = Zend_Controller_Front::getInstance()->getRequest();
$this->headTitle($request->getActionName())
     ->headTitle($request->getControllerName());

// setting the site in the title; possibly in the layout script:
$this->headTitle('Zend Framework');

// setting a separator string for segments:
$this->headTitle()->setSeparator(' / ');
```

When you're finally ready to render the title in your layout script, simply echo the helper:

```
<!-- renders <action> / <controller> / Zend Framework -->
<?php echo $this->headTitle() ?>
```

4.1.13. HTML Object Helpers

The HTML *<object>* element is used for embedding media like Flash or QuickTime in web pages. The object view helpers take care of embedding media with minimum effort.

There are four initial Object helpers:

- `htmlFlash()` Generates markup for embedding Flash files.
- `htmlObject()` Generates markup for embedding a custom Object.
- `htmlPage()` Generates markup for embedding other (X)HTML pages.
- `htmlQuicktime()` Generates markup for embedding QuickTime files.

All of these helpers share a similar interface. For this reason, this documentation will only contain examples of two of these helpers.

Example 936. Flash helper

Embedding Flash in your page using the helper is pretty straight-forward. The only required argument is the resource URI.

```
<?php echo $this->htmlFlash('/path/to/flash.swf'); ?>
```

This outputs the following HTML:

```
<object data="/path/to/flash.swf"
        type="application/x-shockwave-flash"
        classid="clsid:D27CDB6E-AE6D-11cf-96B8-444553540000"
        codebase="http://download.macromedia.com/pub/shockwave/cabs/flash/swflash.cab">
</object>
```

Additionally you can specify attributes, parameters and content that can be rendered along with the *<object>*. This will be demonstrated using the `htmlObject()` helper.

Example 937. Customizing the object by passing additional arguments

The first argument in the object helpers is always required. It is the URI to the resource you want to embed. The second argument is only required in the `htmlObject()` helper. The other helpers already contain the correct value for this argument. The third argument is used for passing along attributes to the object element. It only accepts an array with key-value pairs. classid and codebase are examples of such attributes. The fourth argument also only takes a key-value array and uses them to create *<param>* elements. You will see an example of this shortly. Lastly, there is the option of providing additional content to the object. Now for an example which utilizes all arguments.

```
echo $this->htmlObject(
    '/path/to/file.ext',
    'mime/type',
    array(
        'attr1' => 'aval1',
        'attr2' => 'aval2'
    ),
    array(
        'param1' => 'pval1',
        'param2' => 'pval2'
    ),
    'some content'
);

/*
This would output:

<object data="/path/to/file.ext" type="mime/type"
    attr1="aval1" attr2="aval2">
    <param name="param1" value="pval1" />
    <param name="param2" value="pval2" />
    some content
</object>
*/
```

4.1.14. InlineScript Helper

The HTML *<script>* element is used to either provide inline client-side scripting elements or link to a remote resource containing client-side scripting code. The `InlineScript` helper allows you to manage both. It is derived from HeadScript, and any method of that helper is available; however, use the `inlineScript()` method in place of `headScript()`.

Use InlineScript for HTML Body Scripts

`InlineScript`, should be used when you wish to include scripts inline in the HTML *body*. Placing scripts at the end of your document is a good practice for speeding up delivery of your page, particularly when using 3rd party analytics scripts.

Some JS libraries need to be included in the HTML *head*; use HeadScript for those scripts.

4.1.15. JSON Helper

When creating views that return JSON, it's important to also set the appropriate response header. The JSON view helper does exactly that. In addition, by default, it disables layouts (if currently enabled), as layouts generally aren't used with JSON responses.

The JSON helper sets the following header:

```
Content-Type: application/json
```

Most AJAX libraries look for this header when parsing responses to determine how to handle the content.

Usage of the JSON helper is very straightforward:

```
<?php echo $this->json($this->data) ?>
```

Keeping layouts and enabling encoding using Zend_Json_Expr

Each method in the JSON helper accepts a second, optional argument. This second argument can be a boolean flag to enable or disable layouts, or an array of options that will be passed to `Zend_Json::encode()` and used internally to encode data.

To keep layouts, the second parameter needs to be boolean `TRUE`. When the second parameter is an array, keeping layouts can be achieved by including a keepLayouts key with a value of a boolean `TRUE`.

```
// Boolean true as second argument enables layouts:
echo $this->json($this->data, true);

// Or boolean true as "keepLayouts" key:
echo $this->json($this->data, array('keepLayouts' => true));
```

`Zend_Json::encode` allows the encoding of native JSON expressions using `Zend_Json_Expr` objects. This option is disabled by default. To enable this option, pass a boolean `TRUE` to the enableJsonExprFinder key of the options array:

```
<?php echo $this->json($this->data, array(
    'enableJsonExprFinder' => true,
    'keepLayouts'          => true,
)) ?>
```

4.1.16. Navigation Helpers

The navigation helpers are used for rendering navigational elements from Zend_Navigation_Container instances.

There are 5 built-in helpers:

- Breadcrumbs, used for rendering the path to the currently active page.
- Links, used for rendering navigational head links (e.g. **<link rel="next" href="..." />**)

- Menu, used for rendering menus.
- Sitemap, used for rendering sitemaps conforming to the Sitemaps XML format.
- Navigation, used for proxying calls to other navigational helpers.

All built-in helpers extend `Zend_View_Helper_Navigation_HelperAbstract`, which adds integration with ACL and translation. The abstract class implements the interface `Zend_View_Helper_Navigation_Helper`, which defines the following methods:

- `getContainer()` and `setContainer()` gets and sets the navigation container the helper should operate on by default, and `hasContainer()` checks if the helper has container registered.
- `getTranslator()` and `setTranslator()` gets and sets the translator used for translating labels and titles. `getUseTranslator()` and `setUseTranslator()` controls whether the translator should be enabled. The method `hasTranslator()` checks if the helper has a translator registered.
- `getAcl()`, `setAcl()`, `getRole()` and `setRole()`, gets and sets ACL (`Zend_Acl`) instance and role (String or `Zend_Acl_Role_Interface`) used for filtering out pages when rendering. `getUseAcl()` and `setUseAcl()` controls whether ACL should be enabled. The methods `hasAcl()` and `hasRole()` checks if the helper has an ACL instance or a role registered.
- `__toString()`, magic method to ensure that helpers can be rendered by echoing the helper instance directly.
- `render()`, must be implemented by concrete helpers to do the actual rendering.

In addition to the method stubs from the interface, the abstract class also implements the following methods:

- `getIndent()` and `setIndent()` gets and sets indentation. The setter accepts a String or an Integer. In the case of an Integer, the helper will use the given number of spaces for indentation. I.e., `setIndent(4)` means 4 initial spaces of indentation. Indentation can be specified for all helpers except the Sitemap helper.
- `getMinDepth()` and `setMinDepth()` gets and sets the minimum depth a page must have to be included by the helper. Setting `NULL` means no minimum depth.
- `getMaxDepth()` and `setMaxDepth()` gets and sets the maximum depth a page can have to be included by the helper. Setting `NULL` means no maximum depth.
- `getRenderInvisible()` and `setRenderInvisible()` gets and sets whether to render items that have been marked as invisible or not.
- `__call()` is used for proxying calls to the container registered in the helper, which means you can call methods on a helper as if it was a container. See example below.
- `findActive($container, $minDepth, $maxDepth)` is used for finding the deepest active page in the given container. If depths are not given, the method will use the values retrieved from `getMinDepth()` and `getMaxDepth()`. The deepest active page must be between `$minDepth` and `$maxDepth` inclusively. Returns an array containing a reference to the found page instance and the depth at which the page was found.

- `htmlify()` renders an *'a'* HTML element from a `Zend_Navigation_Page` instance.
- `accept()` is used for determining if a page should be accepted when iterating containers. This method checks for page visibility and verifies that the helper's role is allowed access to the page's resource and privilege.
- The static method `setDefaultAcl()` is used for setting a default ACL object that will be used by helpers.
- The static method `setDefaultRole()` is used for setting a default ACL that will be used by helpers

If a navigation container is not explicitly set in a helper using **$helper->setContainer($nav)**, the helper will look for a container instance with the key `Zend_Navigation` in the registry. If a container is not explicitly set or found in the registry, the helper will create an empty `Zend_Navigation` container when calling **$helper->getContainer()**.

Example 938. Proxying calls to the navigation container

Navigation view helpers use the magic method `__call()` to proxy method calls to the navigation container that is registered in the view helper.

```
$this->navigation()->addPage(array(
    'type' => 'uri',
    'label' => 'New page'));
```

The call above will add a page to the container in the `Navigation` helper.

4.1.16.1. Translation of labels and titles

The navigation helpers support translation of page labels and titles. You can set a translator of type `Zend_Translate` or `Zend_Translate_Adapter` in the helper using **$helper->setTranslator($translator)**, or like with other I18n-enabled components; by adding the translator to the registry by using the key `Zend_Translate`.

If you want to disable translation, use **$helper->setUseTranslator(false)**.

The proxy helper will inject its own translator to the helper it proxies to if the proxied helper doesn't already have a translator.

There is no translation in the sitemap helper, since there are no page labels or titles involved in an XML sitemap.

4.1.16.2. Integration with ACL

All navigational view helpers support ACL inherently from the class `Zend_View_Helper_Navigation_HelperAbstract`. A `Zend_Acl` object can be assigned to a helper instance with `$helper->setAcl($acl)`, and role with `$helper->setRole('member')` or `$helper->setRole(new Zend_Acl_Role('member'))` . If ACL is used in the helper, the role in the helper must be allowed by the ACL to access a page's `resource` and/or have the page's `privilege` for the page to be included when rendering.

If a page is not accepted by ACL, any descendant page will also be excluded from rendering.

The proxy helper will inject its own ACL and role to the helper it proxies to if the proxied helper doesn't already have any.

The examples below all show how ACL affects rendering.

4.1.16.3. Navigation setup used in examples

This example shows the setup of a navigation container for a fictional software company.

Notes on the setup:

- The domain for the site is `www.example.com`.
- Interesting page properties are marked with a comment.
- Unless otherwise is stated in other examples, the user is requesting the URL `http://www.example.com/products/server/faq/`, which translates to the page labeled `FAQ` under `Foo Server`.
- The assumed ACL and router setup is shown below the container setup.

```
/*
 * Navigation container (config/array)

 * Each element in the array will be passed to
 * Zend_Navigation_Page::factory() when constructing
 * the navigation container below.
 */
$pages = array(
    array(
        'label'      => 'Home',
        'title'      => 'Go Home',
        'module'     => 'default',
        'controller' => 'index',
        'action'     => 'index',
        'order'      => -100 // make sure home is the first page
    ),
    array(
        'label'      => 'Special offer this week only!',
        'module'     => 'store',
        'controller' => 'offer',
        'action'     => 'amazing',
        'visible'    => false // not visible
    ),
    array(
        'label'      => 'Products',
        'module'     => 'products',
        'controller' => 'index',
        'action'     => 'index',
        'pages'      => array(
            array(
                'label'      => 'Foo Server',
                'module'     => 'products',
                'controller' => 'server',
                'action'     => 'index',
                'pages'      => array(
                    array(
                        'label'      => 'FAQ',
```

```
                        'module'     => 'products',
                        'controller' => 'server',
                        'action'     => 'faq',
                        'rel'        => array(
                            'canonical' => 'http://www.example.com/?page=faq',
                            'alternate' => array(
                                'module'     => 'products',
                                'controller' => 'server',
                                'action'     => 'faq',
                                'params'     => array('format' => 'xml')
                            )
                        )
                    ),
                    array(
                        'label'      => 'Editions',
                        'module'     => 'products',
                        'controller' => 'server',
                        'action'     => 'editions'
                    ),
                    array(
                        'label'      => 'System Requirements',
                        'module'     => 'products',
                        'controller' => 'server',
                        'action'     => 'requirements'
                    )
                )
            ),
            array(
                'label'      => 'Foo Studio',
                'module'     => 'products',
                'controller' => 'studio',
                'action'     => 'index',
                'pages'      => array(
                    array(
                        'label'      => 'Customer Stories',
                        'module'     => 'products',
                        'controller' => 'studio',
                        'action'     => 'customers'
                    ),
                    array(
                        'label'      => 'Support',
                        'module'     => 'prodcts',
                        'controller' => 'studio',
                        'action'     => 'support'
                    )
                )
            )
        )
    ),
    array(
        'label'      => 'Company',
        'title'      => 'About us',
        'module'     => 'company',
        'controller' => 'about',
        'action'     => 'index',
        'pages'      => array(
            array(
                'label'      => 'Investor Relations',
                'module'     => 'company',
                'controller' => 'about',
```

```
                'action'     => 'investors'
            ),
            array(
                'label'      => 'News',
                'class'      => 'rss', // class
                'module'     => 'company',
                'controller' => 'news',
                'action'     => 'index',
                'pages'      => array(
                    array(
                        'label'      => 'Press Releases',
                        'module'     => 'company',
                        'controller' => 'news',
                        'action'     => 'press'
                    ),
                    array(
                        'label'      => 'Archive',
                        'route'      => 'archive', // route
                        'module'     => 'company',
                        'controller' => 'news',
                        'action'     => 'archive'
                    )
                )
            )
        )
    ),
    array(
        'label'      => 'Community',
        'module'     => 'community',
        'controller' => 'index',
        'action'     => 'index',
        'pages'      => array(
            array(
                'label'      => 'My Account',
                'module'     => 'community',
                'controller' => 'account',
                'action'     => 'index',
                'resource'   => 'mvc:community.account' // resource
            ),
            array(
                'label' => 'Forums',
                'uri'   => 'http://forums.example.com/',
                'class' => 'external' // class
            )
        )
    ),
    array(
        'label'      => 'Administration',
        'module'     => 'admin',
        'controller' => 'index',
        'action'     => 'index',
        'resource'   => 'mvc:admin', // resource
        'pages'      => array(
            array(
                'label'      => 'Write new article',
                'module'     => 'admin',
                'controller' => 'post',
                'aciton'     => 'write'
            )
        )
```

```
    )
);

// Create container from array
$container = new Zend_Navigation($pages);

// Store the container in the proxy helper:
$view->getHelper('navigation')->setContainer($container);

// ...or simply:
$view->navigation($container);

// ...or store it in the reigstry:
Zend_Registry::set('Zend_Navigation', $container);
```

In addition to the container above, the following setup is assumed:

```
// Setup router (default routes and 'archive' route):
$front = Zend_Controller_Front::getInstance();
$router = $front->getRouter();
$router->addDefaultRoutes();
$router->addRoute(
    'archive',
    new Zend_Controller_Router_Route(
        '/archive/:year',
        array(
            'module'     => 'company',
            'controller' => 'news',
            'action'     => 'archive',
            'year'       => (int) date('Y') - 1
        ),
        array('year' => '\d+')
    )
);

// Setup ACL:
$acl = new Zend_Acl();
$acl->addRole(new Zend_Acl_Role('member'));
$acl->addRole(new Zend_Acl_Role('admin'));
$acl->add(new Zend_Acl_Resource('mvc:admin'));
$acl->add(new Zend_Acl_Resource('mvc:community.account'));
$acl->allow('member', 'mvc:community.account');
$acl->allow('admin', null);

// Store ACL and role in the proxy helper:
$view->navigation()->setAcl($acl)->setRole('member');

// ...or set default ACL and role statically:
Zend_View_Helper_Navigation_HelperAbstract::setDefaultAcl($acl);
Zend_View_Helper_Navigation_HelperAbstract::setDefaultRole('member');
```

4.1.16.4. Breadcrumbs Helper

Breadcrumbs are used for indicating where in a sitemap a user is currently browsing, and are typically rendered like this: "You are here: Home > Products > FantasticProduct 1.0". The breadcrumbs helper follows the guidelines from Breadcrumbs Pattern - Yahoo! Design Pattern Library, and allows simple customization (minimum/maximum depth, indentation, separator, and whether the last element should be linked), or rendering using a partial view script.

The Breadcrumbs helper works like this; it finds the deepest active page in a navigation container, and renders an upwards path to the root. For MVC pages, the "activeness" of a page is determined by inspecting the request object, as stated in the section on Zend_Navigation_Page_Mvc.

The helper sets the `minDepth` property to 1 by default, meaning breadcrumbs will not be rendered if the deepest active page is a root page. If `maxDepth` is specified, the helper will stop rendering when at the specified depth (e.g. stop at level 2 even if the deepest active page is on level 3).

Methods in the breadcrumbs helper:

- `{get|set}Separator()` gets/sets separator string that is used between breadcrumbs. Defualt is `' > '`.
- `{get|set}LinkLast()` gets/sets whether the last breadcrumb should be rendered as an anchor or not. Default is FALSE.
- `{get|set}Partial()` gets/sets a partial view script that should be used for rendering breadcrumbs. If a partial view script is set, the helper's `render()` method will use the `renderPartial()` method. If no partial is set, the `renderStraight()` method is used. The helper expects the partial to be a String or an Array with two elements. If the partial is a String, it denotes the name of the partial script to use. If it is an Array, the first element will be used as the name of the partial view script, and the second element is the module where the script is found.
- `renderStraight()` is the default render method.
- `renderPartial()` is used for rendering using a partial view script.

Example 939. Rendering breadcrumbs

This example shows how to render breadcrumbs with default settings.

```
In a view script or layout:
<?php echo $this->navigation()->breadcrumbs(); ?>
The two calls above take advantage of the magic __toString() method,
and are equivalent to:
<?php echo $this->navigation()->breadcrumbs()->render(); ?>
Output:
<a href="/products">Products</a>  <a href="/products/server">Foo Server</a>  FAQ
```

Example 940. Specifying indentation

This example shows how to render breadcrumbs with initial indentation.

```
Rendering with 8 spaces indentation:
<?php echo $this->navigation()->breadcrumbs()->setIndent(8);?>
Output:
        <a href="/products">Products</a>  <a href="/products/server">Foo Server</a>  FAQ
```

Example 941. Customize breadcrumbs output

This example shows how to customze breadcrumbs output by specifying various options.

```
In a view script or layout:

<?php
echo $this->navigation()
          ->breadcrumbs()
          ->setLinkLast(true)                  // link last page
          ->setMaxDepth(1)                     // stop at level 1
          ->setSeparator(' &#9654;' . PHP_EOL); // cool separator with newline
?>
Output:
<a href="/products">Products</a> &amp;#9654;
<a href="/products/server">Foo Server</a>

////////////////////////////////////////////////

Setting minimum depth required to render breadcrumbs:

<?php
$this->navigation()->breadcrumbs()->setMinDepth(10);
echo $this->navigation()->breadcrumbs();
?>
Output:
Nothing, because the deepest active page is not at level 10 or deeper.
```

Example 942. Rendering breadcrumbs using a partial view script

This example shows how to render customized breadcrumbs using a partial vew script. By calling `setPartial()`, you can specify a partial view script that will be used when calling `render()`. When a partial is specified, the `renderPartial()` method will be called. This method will find the deepest active page and pass an array of pages that leads to the active page to the partial view script.

In a layout:

```
$partial = ;
echo $this->navigation()->breadcrumbs()
                        ->setPartial(array('breadcrumbs.phtml', 'default'));
```

Contents of `application/modules/default/views/breadcrumbs.phtml`:

```
echo implode(', ', array_map(
        create_function('$a', 'return $a->getLabel();'),
        $this->pages));
```

Output:

```
Products, Foo Server, FAQ
```

4.1.16.5. Links Helper

The links helper is used for rendering HTML `LINK` elements. Links are used for describing document relationships of the currently active page. Read more about links and link types at

Document relationships: the LINK element (HTML4 W3C Rec.) and Link types (HTML4 W3C Rec.) in the HTML4 W3C Recommendation.

There are two types of relations; forward and reverse, indicated by the keyords `'rel'` and `'rev'`. Most methods in the helper will take a `$rel` param, which must be either `'rel'` or `'rev'`. Most methods also take a `$type` param, which is used for specifying the link type (e.g. alternate, start, next, prev, chapter, etc).

Relationships can be added to page objects manually, or found by traversing the container registered in the helper. The method `findRelation($page, $rel, $type)` will first try to find the given `$rel` of `$type` from the `$page` by calling `$page->findRel($type)` or `$page->findRel($type)`. If the `$page` has a relation that can be converted to a page instance, that relation will be used. If the `$page` instance doesn't have the specified `$type`, the helper will look for a method in the helper named `search$rel$type` (e.g. `searchRelNext()` or `searchRevAlternate()`). If such a method exists, it will be used for determining the `$page`'s relation by traversing the container.

Not all relations can be determined by traversing the container. These are the relations that will be found by searching:

- `searchRelStart()`, forward 'start' relation: the first page in the container.
- `searchRelNext()`, forward 'next' relation; finds the next page in the container, i.e. the page after the active page.
- `searchRelPrev()`, forward 'prev' relation; finds the previous page, i.e. the page before the active page.
- `searchRelChapter()`, forward 'chapter' relations; finds all pages on level 0 except the 'start' relation or the active page if it's on level 0.
- `searchRelSection()`, forward 'section' relations; finds all child pages of the active page if the active page is on level 0 (a 'chapter').
- `searchRelSubsection()`, forward 'subsection' relations; finds all child pages of the active page if the active pages is on level 1 (a 'section').
- `searchRevSection()`, reverse 'section' relation; finds the parent of the active page if the active page is on level 1 (a 'section').
- `searchRevSubsection()`, reverse 'subsection' relation; finds the parent of the active page if the active page is on level 2 (a 'subsection').

When looking for relations in the page instance (`$page->getRel($type)` or `$page->getRev($type)`), the helper accepts the values of type String, Array, `Zend_Config`, or `Zend_Navigation_Page`. If a string is found, it will be converted to a `Zend_Navigation_Page_Uri`. If an array or a config is found, it will be converted to one or several page instances. If the first key of the array/config is numeric, it will be considered to contain several pages, and each element will be passed to the page factory. If the first key is not numeric, the array/config will be passed to the page factory directly, and a single page will be returned.

The helper also supports magic methods for finding relations. E.g. to find forward alternate relations, call `$helper->findRelAlternate($page)`, and to find reverse

section relations, call `$helper->findRevSection($page)`. Those calls correspond to `$helper->findRelation($page, 'rel', 'alternate');` and `$helper->findRelation($page, 'rev', 'section');` respectively.

To customize which relations should be rendered, the helper uses a render flag. The render flag is an integer value, and will be used in a bitwse `and` (`&`) operation against the helper's render constants to determine if the relation that belongs to the render constant should be rendered.

See the example below for more information.

- `Zend_View_Helper_Navigation_Link::RENDER_ALTERNATE`
- `Zend_View_Helper_Navigation_Link::RENDER_STYLESHEET`
- `Zend_View_Helper_Navigation_Link::RENDER_START`
- `Zend_View_Helper_Navigation_Link::RENDER_NEXT`
- `Zend_View_Helper_Navigation_Link::RENDER_PREV`
- `Zend_View_Helper_Navigation_Link::RENDER_CONTENTS`
- `Zend_View_Helper_Navigation_Link::RENDER_INDEX`
- `Zend_View_Helper_Navigation_Link::RENDER_GLOSSARY`
- `Zend_View_Helper_Navigation_Link::RENDER_COPYRIGHT`
- `Zend_View_Helper_Navigation_Link::RENDER_CHAPTER`
- `Zend_View_Helper_Navigation_Link::RENDER_SECTION`
- `Zend_View_Helper_Navigation_Link::RENDER_SUBSECTION`
- `Zend_View_Helper_Navigation_Link::RENDER_APPENDIX`
- `Zend_View_Helper_Navigation_Link::RENDER_HELP`
- `Zend_View_Helper_Navigation_Link::RENDER_BOOKMARK`
- `Zend_View_Helper_Navigation_Link::RENDER_CUSTOM`
- `Zend_View_Helper_Navigation_Link::RENDER_ALL`

The constants from `RENDER_ALTERNATE` to `RENDER_BOOKMARK` denote standard HTML link types. `RENDER_CUSTOM` denotes non-standard relations that specified in pages. `RENDER_ALL` denotes standard and non-standard relations.

Methods in the links helper:

- `{get|set}RenderFlag()` gets/sets the render flag. Default is `RENDER_ALL`. See examples below on how to set the render flag.
- `findAllRelations()` finds all relations of all types for a given page.

- `findRelation()` finds all relations of a given type from a given page.
- `searchRel{Start|Next|Prev|Chapter|Section|Subsection}()` traverses a container to find forward relations to the start page, the next page, the previous page, chapters, sections, and subsections.
- `searchRev{Section|Subsection}()` traverses a container to find reverse relations to sections or subsections.
- `renderLink()` renders a single `link` element.

Example 943. Specify relations in pages

This example shows how to specify relations in pages.

```
$container = new Zend_Navigation(array(
    array(
        'label' => 'Relations using strings',
        'rel'   => array(
            'alternate' => 'http://www.example.org/'
        ),
        'rev'   => array(
            'alternate' => 'http://www.example.net/'
        )
    ),
    array(
        'label' => 'Relations using arrays',
        'rel'   => array(
            'alternate' => array(
                'label' => 'Example.org',
                'uri'   => 'http://www.example.org/'
            )
        )
    ),
    array(
        'label' => 'Relations using configs',
        'rel'   => array(
            'alternate' => new Zend_Config(array(
                'label' => 'Example.org',
                'uri'   => 'http://www.example.org/'
            ))
        )
    ),
    array(
        'label' => 'Relations using pages instance',
        'rel'   => array(
            'alternate' => Zend_Navigation_Page::factory(array(
                'label' => 'Example.org',
                'uri'   => 'http://www.example.org/'
            ))
        )
    )
));
```

Example 944. Default rendering of links

This example shows how to render a menu from a container registered/found in the view helper.

```
In a view script or layout:
<?php echo $this->view->navigation()->links(); ?>
Output:
<link rel="alternate" href="/products/server/faq/format/xml">
<link rel="start" href="/" title="Home">
<link rel="next" href="/products/server/editions" title="Editions">
<link rel="prev" href="/products/server" title="Foo Server">
<link rel="chapter" href="/products" title="Products">
<link rel="chapter" href="/company/about" title="Company">
<link rel="chapter" href="/community" title="Community">
<link rel="canonical" href="http://www.example.com/?page=server-faq">
<link rev="subsection" href="/products/server" title="Foo Server">
```

Example 945. Specify which relations to render

This example shows how to specify which relations to find and render.

```
Render only start, next, and prev:
$helper->setRenderFlag(Zend_View_Helper_Navigation_Links::RENDER_START |
                       Zend_View_Helper_Navigation_Links::RENDER_NEXT |
                       Zend_View_Helper_Navigation_Links::RENDER_PREV);

Output:
<link rel="start" href="/" title="Home">
<link rel="next" href="/products/server/editions" title="Editions">
<link rel="prev" href="/products/server" title="Foo Server">
```

```
Render only native link types:
$helper->setRenderFlag(Zend_View_Helper_Navigation_Links::RENDER_ALL ^
                       Zend_View_Helper_Navigation_Links::RENDER_CUSTOM);

Output:
<link rel="alternate" href="/products/server/faq/format/xml">
<link rel="start" href="/" title="Home">
<link rel="next" href="/products/server/editions" title="Editions">
<link rel="prev" href="/products/server" title="Foo Server">
<link rel="chapter" href="/products" title="Products">
<link rel="chapter" href="/company/about" title="Company">
<link rel="chapter" href="/community" title="Community">
<link rev="subsection" href="/products/server" title="Foo Server">
```

```
Render all but chapter:
$helper->setRenderFlag(Zend_View_Helper_Navigation_Links::RENDER_ALL ^
                       Zend_View_Helper_Navigation_Links::RENDER_CHAPTER);

Output:
<link rel="alternate" href="/products/server/faq/format/xml">
<link rel="start" href="/" title="Home">
<link rel="next" href="/products/server/editions" title="Editions">
<link rel="prev" href="/products/server" title="Foo Server">
<link rel="canonical" href="http://www.example.com/?page=server-faq">
<link rev="subsection" href="/products/server" title="Foo Server">
```

4.1.16.6. Menu Helper

The Menu helper is used for rendering menus from navigation containers. By default, the menu will be rendered using HTML `UL` and `LI` tags, but the helper also allows using a partial view script.

Methods in the Menu helper:

- `{get|set}UlClass()` gets/sets the CSS class used in `renderMenu()`.
- `{get|set}OnlyActiveBranch()` gets/sets a flag specifying whether only the active branch of a container should be rendered.
- `{get|set}RenderParents()` gets/sets a flag specifying whether parents should be rendered when only rendering active branch of a container. If set to `FALSE`, only the deepest active menu will be rendered.
- `{get|set}Partial()` gets/sets a partial view script that should be used for rendering menu. If a partial view script is set, the helper's `render()` method will use the `renderPartial()` method. If no partial is set, the `renderMenu()` method is used. The helper expects the partial to be a String or an Array with two elements. If the partial is a String, it denotes the name of the partial script to use. If it is an Array, the first element will be used as the name of the partial view script, and the second element is the module where the script is found.
- `htmlify()` overrides the method from the abstract class to return `span` elements if the page has no `href`.
- `renderMenu($container = null, $options = array())` is the default render method, and will render a container as a HTML `UL` list.

  If `$container` is not given, the container registered in the helper will be rendered.

  `$options` is used for overriding options specified temporarily without rsetting the values in the helper instance. It is an associative array where each key corresponds to an option in the helper.

  Recognized options:

  - `indent`; indentation. Expects a String or an `int` value.
  - `minDepth`; minimum depth. Expcects an `int` or `NULL` (no minimum depth).
  - `maxDepth`; maximum depth. Expcects an `int` or `NULL` (no maximum depth).
  - `ulClass`; CSS class for `ul` element. Expects a String.
  - `onlyActiveBranch`; whether only active branch should be rendered. Expects a Boolean value.
  - `renderParents`; whether parents should be rendered if only rendering active branch. Expects a Boolean value.

  If an option is not given, the value set in the helper will be used.
- `renderPartial()` is used for rendering the menu using a partial view script.
- `renderSubMenu()` renders the deepest menu level of a container's active branch.

```
Output:
<ul class="navigation">
    <li>
        <a title="Go Home" href="/">Home</a>
    </li>
    <li class="active">
        <a href="/products">Products</a>
        <ul>
            <li class="active">
                <a href="/products/server">Foo Server</a>
                <ul>
                    <li class="active">
                        <a href="/products/server/faq">FAQ</a>
                    </li>
                    <li>
                        <a href="/products/server/editions">Editions</a>
                    </li>
                    <li>
                        <a href="/products/server/requirements">System Requirements</a>
                    </li>
                </ul>
            </li>
            <li>
                <a href="/products/studio">Foo Studio</a>
                <ul>
                    <li>
                        <a href="/products/studio/customers">Customer Stories</a>
                    </li>
                    <li>
                        <a href="/prodcts/studio/support">Support</a>
                    </li>
                </ul>
            </li>
        </ul>
    </li>
    <li>
        <a title="About us" href="/company/about">Company</a>
        <ul>
            <li>
                <a href="/company/about/investors">Investor Relations</a>
            </li>
            <li>
                <a class="rss" href="/company/news">News</a>
                <ul>
                    <li>
                        <a href="/company/news/press">Press Releases</a>
                    </li>
                    <li>
                        <a href="/archive">Archive</a>
                    </li>
                </ul>
            </li>
        </ul>
    </li>
    <li>
        <a href="/community">Community</a>
        <ul>
            <li>
                <a href="/community/account">My Account</a>
            </li>
            <li>
                <a class="external" href="http://forums.example.com/">Forums</a>
            </li>
        </ul>
    </li>
</ul>
```

Example 947. Calling renderMenu() directly

This example shows how to render a menu that is not registered in the view helper by calling the `renderMenu()` directly and specifying a few options.

```
<?php
// render only the 'Community' menu
$community = $this->navigation()->findOneByLabel('Community');
$options = array(
    'indent'  => 16,
    'ulClass' => 'community'
);
echo $this->navigation()
          ->menu()
          ->renderMenu($community, $options);
?>
Output:
                <ul class="community">
                    <li>
                        <a href="/community/account">My Account</a>
                    </li>
                    <li>
                        <a class="external" href="http://forums.example.com/">Forums</a>
                    </li>
                </ul>
```

Example 948. Rendering the deepest active menu

This example shows how the `renderSubMenu()` will render the deepest sub menu of the active branch.

Calling `renderSubMenu($container, $ulClass, $indent)` is equivalent to calling `renderMenu($container, $options)` with the following options:

```
array(
    'ulClass'          => $ulClass,
    'indent'           => $indent,
    'minDepth'         => null,
    'maxDepth'         => null,
    'onlyActiveBranch' => true,
    'renderParents'    => false
);
```

```
<?php
echo $this->navigation()
          ->menu()
          ->renderSubMenu(null, 'sidebar', 4);
?>
The output will be the same if 'FAQ' or 'Foo Server' is active:
    <ul class="sidebar">
        <li class="active">
            <a href="/products/server/faq">FAQ</a>
        </li>
        <li>
            <a href="/products/server/editions">Editions</a>
        </li>
        <li>
            <a href="/products/server/requirements">System Requirements</a>
        </li>
    </ul>
```

Example 949. Rendering a menu with maximum depth

```
<?php
echo $this->navigation()
          ->menu()
          ->setMaxDepth(1);
?>
Output:
<ul class="navigation">
    <li>
        <a title="Go Home" href="/">Home</a>
    </li>
    <li class="active">
        <a href="/products">Products</a>
        <ul>
            <li class="active">
                <a href="/products/server">Foo Server</a>
            </li>
            <li>
                <a href="/products/studio">Foo Studio</a>
            </li>
        </ul>
    </li>
    <li>
        <a title="About us" href="/company/about">Company</a>
        <ul>
            <li>
                <a href="/company/about/investors">Investor Relations</a>
            </li>
            <li>
                <a class="rss" href="/company/news">News</a>
            </li>
        </ul>
    </li>
    <li>
        <a href="/community">Community</a>
        <ul>
            <li>
                <a href="/community/account">My Account</a>
            </li>
            <li>
                <a class="external" href="http://forums.example.com/">Forums</a>
            </li>
        </ul>
    </li>
</ul>
```

Example 950. Rendering a menu with minimum depth

```
<?php
echo $this->navigation()
          ->menu()
          ->setMinDepth(1);
?>
Output:
<ul class="navigation">
    <li class="active">
        <a href="/products/server">Foo Server</a>
        <ul>
            <li class="active">
                <a href="/products/server/faq">FAQ</a>
            </li>
            <li>
                <a href="/products/server/editions">Editions</a>
            </li>
            <li>
                <a href="/products/server/requirements">System Requirements</a>
            </li>
        </ul>
    </li>
    <li>
        <a href="/products/studio">Foo Studio</a>
        <ul>
            <li>
                <a href="/products/studio/customers">Customer Stories</a>
            </li>
            <li>
                <a href="/prodcts/studio/support">Support</a>
            </li>
        </ul>
    </li>
    <li>
        <a href="/company/about/investors">Investor Relations</a>
    </li>
    <li>
        <a class="rss" href="/company/news">News</a>
        <ul>
            <li>
                <a href="/company/news/press">Press Releases</a>
            </li>
            <li>
                <a href="/archive">Archive</a>
            </li>
        </ul>
    </li>
    <li>
        <a href="/community/account">My Account</a>
    </li>
    <li>
        <a class="external" href="http://forums.example.com/">Forums</a>
    </li>
</ul>
```

Example 951. Rendering only the active branch of a menu

```
<?php
echo $this->navigation()
          ->menu()
          ->setOnlyActiveBranch(true);
?>
Output:
<ul class="navigation">
    <li class="active">
        <a href="/products">Products</a>
        <ul>
            <li class="active">
                <a href="/products/server">Foo Server</a>
                <ul>
                    <li class="active">
                        <a href="/products/server/faq">FAQ</a>
                    </li>
                    <li>
                        <a href="/products/server/editions">Editions</a>
                    </li>
                    <li>
                        <a href="/products/server/requirements">System Requirements</a>
                    </li>
                </ul>
            </li>
        </ul>
    </li>
</ul>
```

Example 952. Rendering only the active branch of a menu with minimum depth

```
<?php
echo $this->navigation()
          ->menu()
          ->setOnlyActiveBranch(true)
          ->setMinDepth(1);
?>
Output:
<ul class="navigation">
    <li class="active">
        <a href="/products/server">Foo Server</a>
        <ul>
            <li class="active">
                <a href="/products/server/faq">FAQ</a>
            </li>
            <li>
                <a href="/products/server/editions">Editions</a>
            </li>
            <li>
                <a href="/products/server/requirements">System Requirements</a>
            </li>
        </ul>
    </li>
</ul>
```

Example 953. Rendering only the active branch of a menu with maximum depth

```
<?php
echo $this->navigation()
          ->menu()
          ->setOnlyActiveBranch(true)
          ->setMaxDepth(1);
?>
Output:
<ul class="navigation">
    <li class="active">
        <a href="/products">Products</a>
        <ul>
            <li class="active">
                <a href="/products/server">Foo Server</a>
            </li>
            <li>
                <a href="/products/studio">Foo Studio</a>
            </li>
        </ul>
    </li>
</ul>
```

Example 954. Rendering only the active branch of a menu with maximum depth and no parents

```
<?php
echo $this->navigation()
          ->menu()
          ->setOnlyActiveBranch(true)
          ->setRenderParents(false)
          ->setMaxDepth(1);
?>
Output:
<ul class="navigation">
    <li class="active">
        <a href="/products/server">Foo Server</a>
    </li>
    <li>
        <a href="/products/studio">Foo Studio</a>
    </li>
</ul>
```

Example 955. Rendering a custom menu using a partial view script

This example shows how to render a custom menu using a partial vew script. By calling `setPartial()`, you can specify a partial view script that will be used when calling `render()`. When a partial is specified, the `renderPartial()` method will be called. This method will assign the container to the view with the key `container`.

In a layout:

```
$partial = array('menu.phtml', 'default');
$this->navigation()->menu()->setPartial($partial);
echo $this->navigation()->menu()->render();
```

In application/modules/default/views/menu.phtml:

```
foreach ($this->container as $page) {
    echo $this->navigation()->menu()->htmlify($page), PHP_EOL;
}
```

Output:

```
<a title="Go Home" href="/">Home</a>
<a href="/products">Products</a>
<a title="About us" href="/company/about">Company</a>
<a href="/community">Community</a>
```

4.1.16.7. Sitemap Helper

The Sitemap helper is used for generating XML sitemaps, as defined by the Sitemaps XML format. Read more about Sitemaps on Wikpedia.

By default, the sitemap helper uses sitemap validators to validate each element that is rendered. This can be disabled by calling `$helper->setUseSitemapValidators(false)`.

If you disable sitemap validators, the custom properties (see table) are not validated at all.

The sitemap helper also supports Sitemap XSD Schema validation of the generated sitemap. This is disabled by default, since it will require a request to the Schema file. It can be enabled with `$helper->setUseSchemaValidation(true)`.

Table 154. Sitemap XML elements

Element	Description
`loc`	Absolute URL to page. An absolute URL will be generated by the helper.
`lastmod`	The date of last modification of the file, in W3C Datetime format. This time portion can be omitted if desired, and only use YYYY-MM-DD. The helper will try to retrieve the `lastmod` value from the page's custom property `lastmod` if it is set in the page. If the value is not a valid date, it is ignored.

Element	Description
changefreq	How frequently the page is likely to change. This value provides general information to search engines and may not correlate exactly to how often they crawl the page. Valid values are: • always • hourly • daily • weekly • monthly • yearly • never The helper will try to retrieve the changefreq value from the page's custom property changefreq if it is set in the page. If the value is not valid, it is ignored.
priority	The priority of this URL relative to other URLs on your site. Valid values range from 0.0 to 1.0. The helper will try to retrieve the priority value from the page's custom property priority if it is set in the page. If the value is not valid, it is ignored.

Methods in the sitemap helper:

- {get|set}FormatOutput() gets/sets a flag indicating whether XML output should be formatted. This corresponds to the formatOutput property of the native DOMDocument class. Read more at PHP: DOMDocument - Manual. Default is FALSE.
- {get|set}UseXmlDeclaration() gets/sets a flag indicating whether the XML declaration should be included when rendering. Default is TRUE.
- {get|set}UseSitemapValidators() gets/sets a flag indicating whether sitemap validators should be used when generating the DOM sitemap. Default is TRUE.
- {get|set}UseSchemaValidation() gets/sets a flag indicating whether the helper should use XML Schema validation when generating the DOM sitemap. Default is FALSE. If TRUE.
- {get|set}ServerUrl() gets/sets server URL that will be prepended to non-absolute URLs in the url() method. If no server URL is specified, it will be determined by the helper.
- url() is used to generate absolute URLs to pages.
- getDomSitemap() generates a DOMDocument from a given container.

```
  </url>
  <url>
    <loc>http://www.example.com/products/studio</loc>
  </url>
  <url>
    <loc>http://www.example.com/products/studio/customers</loc>
  </url>
  <url>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<urlset xmlns="http://www.sitemaps.org/schemas/sitemap/0.9">
  <url>
    <loc>http://www.example.com/</loc>
  </url>
  <url>
    <loc>http://www.example.com/products</loc>
  </url>
  <url>
    <loc>http://www.example.com/products/server</loc>
  </url>
  <url>
    <loc>http://www.example.com/products/server/faq</loc>
  </url>
  <url>
    <loc>http://www.example.com/products/server/editions</loc>
  </url>
  <url>
    <loc>http://www.example.com/products/server/requirements</loc>
  </url>
  <url>
    <loc>http://www.example.com/products/studio</loc>
  </url>
  <url>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<urlset xmlns="http://www.sitemaps.org/schemas/sitemap/0.9">
  <url>
    <loc>http://www.example.com/</loc>
  </url>
  <url>
    <loc>http://www.example.com/products</loc>
  </url>
  <url>
    <loc>http://www.example.com/products/server</loc>
  </url>
  <url>
    <loc>http://www.example.com/products/studio</loc>
  </url>
  <url>
    <loc>http://www.example.com/company/about</loc>
  </url>
  <url>
    <loc>http://www.example.com/company/about/investors</loc>
  </url>
  <url>
    <loc>http://www.example.com/company/news</loc>
  </url>
  <url>
    <loc>http://www.example.com/community</loc>
  </url>
  <url>
    <loc>http://www.example.com/community/account</loc>
  </url>
  <url>
    <loc>http://forums.example.com/</loc>
  </url>
</urlset>
```

UTF-8 encoding used by default

By default, Zend Framework uses UTF-8 as its default encoding, and, specific to this case, `Zend_View` does as well. Character encoding can be set differently on the view object itself using the `setEncoding()` method (or the the `encoding` instantiation parameter). However, since `Zend_View_Interface` does not define accessors for encoding, it's possible that if you are using a custom view implementation with the Dojo view helper, you will not have a `getEncoding()` method, which is what the view helper uses internally for determining the character set in which to encode.

If you do not want to utilize UTF-8 in such a situation, you will need to implement a `getEncoding()` method in your custom view implementation.

4.1.16.8. Navigation Helper

The Navigation helper is a proxy helper that relays calls to other navigational helpers. It can be considered an entry point to all navigation-related view tasks. The aforementioned navigational helpers are in the namespace `Zend_View_Helper_Navigation`, and would thus require the path `Zend/View/Helper/Navigation` to be added as a helper path to the view. With the proxy helper residing in the `Zend_View_Helper` namespace, it will always be available, without the need to add any helper paths to the view.

The Navigation helper finds other helpers that implement the `Zend_View_Helper_Navigation_Helper` interface, which means custom view helpers can also be proxied. This would, however, require that the custom helper path is added to the view.

When proxying to other helpers, the Navigation helper can inject its container, ACL/role, and translator. This means that you won't have to explicitly set all three in all navigational helpers, nor resort to injecting by means of `Zend_Registry` or static methods.

- `findHelper()` finds the given helper, verifies that it is a navigational helper, and injects container, ACL/role and translator.
- `{get|set}InjectContainer()` gets/sets a flag indicating whether the container should be injected to proxied helpers. Default is TRUE.
- `{get|set}InjectAcl()` gets/sets a flag indicating whether the ACL/role should be injected to proxied helpers. Default is TRUE.
- `{get|set}InjectTranslator()` gets/sets a flag indicating whether the translator should be injected to proxied helpers. Default is TRUE.
- `{get|set}DefaultProxy()` gets/sets the default proxy. Default is `'menu'`.
- `render()` proxies to the render method of the default proxy.

4.1.17. Translate Helper

Often web sites are available in several languages. To translate the content of a site you should simply use Zend Translate and to integrate `Zend Translate` within your view you should use the *Translate* View Helper.

In all following examples we are using the simple Array Translation Adapter. Of course you can also use any instance of `Zend_Translate` and also any subclasses of `Zend_Translate_Adapter`. There are several ways to initiate the *Translate* View Helper:

- Registered, through a previously registered instance in `Zend_Registry`
- Afterwards, through the fluent interface
- Directly, through initiating the class

A registered instance of `Zend_Translate` is the preferred usage for this helper. You can also select the locale to be used simply before you add the adapter to the registry.

We are speaking of locales instead of languages because a language also may contain a region. For example English is spoken in different dialects. There may be a translation for British and one for American English. Therefore, we say "locale" instead of "language."

Example 957. Registered instance

To use a registered instance just create an instance of `Zend_Translate` or `Zend_Translate_Adapter` and register it within `Zend_Registry` using `Zend_Translate` as its key.

```
// our example adapter
$adapter = new Zend_Translate('array', array('simple' => 'einfach'), 'de');
Zend_Registry::set('Zend_Translate', $adapter);

// within your view
echo $this->translate('simple');
// this returns 'einfach'
```

If you are more familiar with the fluent interface, then you can also create an instance within your view and initiate the helper afterwards.

Example 958. Within the view

To use the fluent interface, create an instance of `Zend_Translate` or `Zend_Translate_Adapter`, call the helper without a parameter, and call the `setTranslator()` method.

```
// within your view
$adapter = new Zend_Translate('array', array('simple' => 'einfach'), 'de');
$this->translate()->setTranslator($adapter)->translate('simple');
// this returns 'einfach'
```

If you are using the helper without `Zend_View` then you can also use it directly.

Example 959. Direct usage

```
// our example adapter
$adapter = new Zend_Translate('array', array('simple' => 'einfach'), 'de');

// initiate the adapter
$translate = new Zend_View_Helper_Translate($adapter);
print $translate->translate('simple'); // this returns 'einfach'
```

You would use this way if you are not working with `Zend_View` and need to create translated output.

As already seen, the `translate()` method is used to return the translation. Just call it with the needed messageid of your translation adapter. But it can also replace parameters within the translation string. Therefore, it accepts variable parameters in two ways: either as a list of parameters, or as an array of parameters. As examples:

Example 960. Single parameter

To use a single parameter just add it to the method.

```
// within your view
$date = "Monday";
$this->translate("Today is %1\$s", $date);
// could return 'Heute ist Monday'
```

Keep in mind that if you are using parameters which are also text, you may also need to translate these parameters.

Example 961. List of parameters

Or use a list of parameters and add it to the method.

```
// within your view
$date = "Monday";
$month = "April";
$time = "11:20:55";
$this->translate("Today is %1\$s in %2\$s. Actual time: %3\$s",
                 $date,
                 $month,
                 $time);
// Could return 'Heute ist Monday in April. Aktuelle Zeit: 11:20:55'
```

Example 962. Array of parameters

Or use an array of parameters and add it to the method.

```
// within your view
$date = array("Monday", "April", "11:20:55");
$this->translate("Today is %1\$s in %2\$s. Actual time: %3\$s", $date);
// Could return 'Heute ist Monday in April. Aktuelle Zeit: 11:20:55'
```

Sometimes it is necessary to change the locale of the translation. This can be done either dynamically per translation or statically for all following translations. And you can use it with both a parameter list and an array of parameters. In both cases the locale must be given as the last single parameter.

Example 963. Change locale dynamically

```
// within your view
$date = array("Monday", "April", "11:20:55");
$this->translate("Today is %1\$s in %2\$s. Actual time: %3\$s", $date, 'it');
```

This example returns the Italian translation for the messageid. But it will only be used once. The next translation will use the locale from the adapter. Normally you will set the desired locale within the translation adapter before you add it to the registry. But you can also set the locale from within the helper:

Example 964. Change locale statically

```
// within your view
$date = array("Monday", "April", "11:20:55");
$this->translate()->setLocale('it');
$this->translate("Today is %1\$s in %2\$s. Actual time: %3\$s", $date);
```

The above example sets *'it'* as the new default locale which will be used for all further translations.

Of course there is also a `getLocale()` method to get the currently set locale.

Example 965. Get the currently set locale

```
// within your view
$date = array("Monday", "April", "11:20:55");

// returns 'de' as set default locale from our above examples
$this->translate()->getLocale();

$this->translate()->setLocale('it');
$this->translate("Today is %1\$s in %2\$s. Actual time: %3\$s", $date);

// returns 'it' as new set default locale
$this->translate()->getLocale();
```

4.2. Helper Paths

As with view scripts, your controller can specify a stack of paths for `Zend_View` to search for helper classes. By default, `Zend_View` looks in "Zend/View/Helper/*" for helper classes. You can tell `Zend_View` to look in other locations using the `setHelperPath()` and `addHelperPath()` methods. Additionally, you can indicate a class prefix to use for helpers in the path provided, to allow namespacing your helper classes. By default, if no class prefix is provided, 'Zend_View_Helper_' is assumed.

```
$view = new Zend_View();

// Set path to /path/to/more/helpers, with prefix 'My_View_Helper'
$view->setHelperPath('/path/to/more/helpers', 'My_View_Helper');
```

In fact, you can "stack" paths using the `addHelperPath()` method. As you add paths to the stack, `Zend_View` will look at the most-recently-added path for the requested helper class. This allows you to add to (or even override) the initial distribution of helpers with your own custom helpers.

```
$view = new Zend_View();
// Add /path/to/some/helpers with class prefix 'My_View_Helper'
$view->addHelperPath('/path/to/some/helpers', 'My_View_Helper');
// Add /other/path/to/helpers with class prefix 'Your_View_Helper'
$view->addHelperPath('/other/path/to/helpers', 'Your_View_Helper');

// now when you call $this->helperName(), Zend_View will look first for
// "/path/to/some/helpers/HelperName" using class name
// "Your_View_Helper_HelperName", then for
// "/other/path/to/helpers/HelperName.php" using class name
// "My_View_Helper_HelperName", and finally for
// "Zend/View/Helper/HelperName.php" using class name
// "Zend_View_Helper_HelperName".
```

4.3. Writing Custom Helpers

Writing custom helpers is easy; just follow these rules:

- While not strictly necessary, we recommend either implementing `Zend_View_Helper_Interface` or extending `Zend_View_Helper_Abstract` when creating your helpers. Introduced in 1.6.0, these simply define a `setView()` method; however, in upcoming releases, we plan to implement a strategy pattern that will simplify much of the naming schema detailed below. Building off these now will help you future-proof your code.

- The class name must, at the very minimum, end with the helper name itself, using MixedCaps. E.g., if you were writing a helper called "specialPurpose", the class name would minimally need to be "SpecialPurpose". You may, and should, give the class name a prefix, and it is recommended that you use 'View_Helper' as part of that prefix: "My_View_Helper_SpecialPurpose". (You will need to pass in the prefix, with or without the trailing underscore, to `addHelperPath()` or `setHelperPath()`).

- The class must have a public method that matches the helper name; this is the method that will be called when your template calls "$this->specialPurpose()". In our "specialPurpose" helper example, the required method declaration would be "public function specialPurpose()".

- In general, the class should not echo or print or otherwise generate output. Instead, it should return values to be printed or echoed. The returned values should be escaped appropriately.

- The class must be in a file named after the helper class. Again using our "specialPurpose" helper example, the file has to be named "SpecialPurpose.php".

Place the helper class file somewhere in your helper path stack, and `Zend_View` will automatically load, instantiate, persist, and execute it for you.

Here is an example of our `SpecialPurpose` helper code:

```
class My_View_Helper_SpecialPurpose extends Zend_View_Helper_Abstract
{
    protected $_count = 0;
    public function specialPurpose()
    {
        $this->_count++;
        $output = "I have seen 'The Jerk' {$this->_count} time(s).";
        return htmlspecialchars($output);
    }
}
```

Then in a view script, you can call the `SpecialPurpose` helper as many times as you like; it will be instantiated once, and then it persists for the life of that `Zend_View` instance.

```
// remember, in a view script, $this refers to the Zend_View instance.
echo $this->specialPurpose();
echo $this->specialPurpose();
echo $this->specialPurpose();
```

The output would look something like this:

```
I have seen 'The Jerk' 1 time(s).
I have seen 'The Jerk' 2 time(s).
I have seen 'The Jerk' 3 time(s).
```

Sometimes you will need access to the calling `Zend_View` object -- for instance, if you need to use the registered encoding, or want to render another view script as part of your helper. To get access to the view object, your helper class should have a `setView($view)` method, like the following:

```
class My_View_Helper_ScriptPath
{
    public $view;

    public function setView(Zend_View_Interface $view)
    {
        $this->view = $view;
    }

    public function scriptPath($script)
    {
        return $this->view->getScriptPath($script);
    }
}
```

If your helper class has a `setView()` method, it will be called when the helper class is first instantiated, and passed the current view object. It is up to you to persist the object in your class, as well as determine how it should be accessed.

If you are extending `Zend_View_Helper_Abstract`, you do not need to define this method, as it is defined for you.

4.4. Registering Concrete Helpers

Sometimes it is convenient to instantiate a view helper, and then register it with the view. As of version 1.10.0, this is now possible using the `registerHelper()` method, which expects two arguments: the helper object, and the name by which it will be registered.

```
$helper = new My_Helper_Foo();
// ...do some configuration or dependency injection...

$view->registerHelper($helper, 'foo');
```

If the helper has a `setView()` method, the view object will call this and inject itself into the helper on registration.

Helper name should match a method

The second argument to `registerHelper()` is the name of the helper. A corresponding method name should exist in the helper; otherwise, `Zend_View` will call a non-existent method when invoking the helper, raising a fatal PHP error.

5. Zend_View_Abstract

`Zend_View_Abstract` is the base class on which `Zend_View` is built; `Zend_View` itself simply extends it and declares a concrete implementation of the `_run()` method (which is invoked by `render()`).

Many developers find that they want to extend `Zend_View_Abstract` to add custom functionality, and inevitably run into issues with its design, which includes a number of private members. This document aims to explain the decision behind the design.

`Zend_View` is something of an anti-templating engine in that it uses PHP natively for its templating. As a result, all of PHP is available, and view scripts inherit the scope of their calling object.

It is this latter point that is salient to the design decisions. Internally, `Zend_View::_run()` does the following:

```
protected function _run()
{
    include func_get_arg(0);
}
```

As such, the view scripts have access to the current object (`$this`), *and any methods or members of that object*. Since many operations depend on members with limited visibility, this poses a problem: the view scripts could potentially make calls to such methods or modify critical properties directly. Imagine a script overwriting `$_path` or `$_file` inadvertently -- any further calls to `render()` or view helpers would break!

Fortunately, PHP 5 has an answer to this with its visibility declarations: private members are not accessible by objects extending a given class. This led to the current design: since `Zend_View` *extends* `Zend_View_Abstract`, view scripts are thus limited to only protected or public methods and members of `Zend_View_Abstract` -- effectively limiting the actions it can perform, and allowing us to secure critical areas from abuse by view scripts.

Zend_Wildfire

1. Zend_Wildfire

`Zend_Wildfire` is a component that facilitates communication between PHP code and Wildfire client components.

The purpose of the Wildfire Project is to develop standardized communication channels between a large variety of components and a dynamic and scriptable plugin architecture. At this time, the primary focus is to provide a system that allows server-side PHP code to inject logging messages into the Firebug Console.

The `Zend_Log_Writer_Firebug` component is provided for the purpose of logging to Firebug, and a communication protocol has been developed that uses HTTP request and response headers to send data between the server and client components. It is great for logging intelligence data to the browser that is generated during script execution, without interfering with the page content. With this approach, it is possible to debug AJAX requests that require clean JSON and XML responses.

There is also a `Zend_Db_Profiler_Firebug` component to log database profiling information to Firebug.

Zend_XmlRpc

1. Introduction

From its home page, XML-RPC is described as a "...remote procedure calling using HTTP as the transport and XML as the encoding. XML-RPC is designed to be as simple as possible, while allowing complex data structures to be transmitted, processed and returned."

Zend Framework provides support for both consuming remote XML-RPC services and building new XML-RPC servers.

2. Zend_XmlRpc_Client

2.1. Introduction

Zend Framework provides support for consuming remote XML-RPC services as a client in the `Zend_XmlRpc_Client` package. Its major features include automatic type conversion between PHP and XML-RPC, a server proxy object, and access to server introspection capabilities.

2.2. Method Calls

The constructor of `Zend_XmlRpc_Client` receives the URL of the remote XML-RPC server endpoint as its first parameter. The new instance returned may be used to call any number of remote methods at that endpoint.

To call a remote method with the XML-RPC client, instantiate it and use the `call()` instance method. The code sample below uses a demonstration XML-RPC server on the Zend Framework website. You can use it for testing or exploring the `Zend_XmlRpc` components.

Example 966. XML-RPC Method Call

```
$client = new Zend_XmlRpc_Client('http://framework.zend.com/xmlrpc');

echo $client->call('test.sayHello');

// hello
```

The XML-RPC value returned from the remote method call will be automatically unmarshaled and cast to the equivalent PHP native type. In the example above, a PHP String is returned and is immediately ready to be used.

The first parameter of the `call()` method receives the name of the remote method to call. If the remote method requires any parameters, these can be sent by supplying a second, optional parameter to `call()` with an Array of values to pass to the remote method:

Example 967. XML-RPC Method Call with Parameters

```
$client = new Zend_XmlRpc_Client('http://framework.zend.com/xmlrpc');

$arg1 = 1.1;
$arg2 = 'foo';

$result = $client->call('test.sayHello', array($arg1, $arg2));

// $result is a native PHP type
```

If the remote method doesn't require parameters, this optional parameter may either be left out or an empty `array()` passed to it. The array of parameters for the remote method can contain native PHP types, `Zend_XmlRpc_Value` objects, or a mix of each.

The `call()` method will automatically convert the XML-RPC response and return its equivalent PHP native type. A `Zend_XmlRpc_Response` object for the return value will also be available by calling the `getLastResponse()` method after the call.

2.3. Types and Conversions

Some remote method calls require parameters. These are given to the `call()` method of `Zend_XmlRpc_Client` as an array in the second parameter. Each parameter may be given as either a native PHP type which will be automatically converted, or as an object representing a specific XML-RPC type (one of the `Zend_XmlRpc_Value` objects).

2.3.1. PHP Native Types as Parameters

Parameters may be passed to `call()` as native PHP variables, meaning as a String, Integer, Float, Boolean, Array, or an Object. In this case, each PHP native type will be auto-detected and converted into one of the XML-RPC types according to this table:

Table 155. PHP and XML-RPC Type Conversions

PHP Native Type	XML-RPC Type
integer	int
Zend_Crypt_Math_BigInteger	i8
double	double
boolean	boolean
string	string
null	nil
array	array
associative array	struct
object	array
Zend_Date	dateTime.iso8601
DateTime	dateTime.iso8601

What type do empty arrays get cast to?

Passing an empty array to an XML-RPC method is problematic, as it could represent either an array or a struct. `Zend_XmlRpc_Client` detects such conditions and makes a request to the server's **system.methodSignature** method to determine the appropriate XML-RPC type to cast to.

However, this in itself can lead to issues. First off, servers that do not support **system.methodSignature** will log failed requests, and `Zend_XmlRpc_Client` will resort to casting the value to an XML-RPC array type. Additionally, this means that any call with array arguments will result in an additional call to the remote server.

To disable the lookup entirely, you can call the `setSkipSystemLookup()` method prior to making your XML-RPC call:

```
$client->setSkipSystemLookup(true);
$result = $client->call('foo.bar', array(array()));
```

2.3.2. Zend_XmlRpc_Value Objects as Parameters

Parameters may also be created as `Zend_XmlRpc_Value` instances to specify an exact XML-RPC type. The primary reasons for doing this are:

- When you want to make sure the correct parameter type is passed to the procedure (i.e. the procedure requires an integer and you may get it from a database as a string)
- When the procedure requires base64 or dateTime.iso8601 type (which doesn't exists as a PHP native type)
- When auto-conversion may fail (i.e. you want to pass an empty XML-RPC struct as a parameter. Empty structs are represented as empty arrays in PHP but, if you give an empty array as a parameter it will be auto-converted to an XML-RPC array since it's not an associative array)

There are two ways to create a `Zend_XmlRpc_Value` object: instantiate one of the `Zend_XmlRpc_Value` subclasses directly, or use the static factory method `Zend_XmlRpc_Value::getXmlRpcValue()`.

Table 156. Zend_XmlRpc_Value Objects for XML-RPC Types

XML-RPC Type	Zend_XmlRpc_Value Constant	Zend_XmlRpc_Value Object
int	`Zend_XmlRpc_Value::XMLRPC_TYPE_INTEGER`	`Zend_XmlRpc_Value_Integer`
i8	`Zend_XmlRpc_Value::XMLRPC_TYPE_I8`	`Zend_XmlRpc_Value_BigInteger`
ex:i8	`Zend_XmlRpc_Value::XMLRPC_TYPE_APACHE_I8`	`Zend_XmlRpc_Value_BigInteger`
double	`Zend_XmlRpc_Value::XMLRPC_TYPE_DOUBLE`	`Zend_XmlRpc_Value_Double`
boolean	`Zend_XmlRpc_Value::XMLRPC_TYPE_BOOLEAN`	`Zend_XmlRpc_Value_Boolean`
string	`Zend_XmlRpc_Value::XMLRPC_TYPE_STRING`	`Zend_XmlRpc_Value_String`
nil	`Zend_XmlRpc_Value::XMLRPC_TYPE_NIL`	`Zend_XmlRpc_Value_Nil`
ex:nil	`Zend_XmlRpc_Value::XMLRPC_TYPE_APACHE_NIL`	`Zend_XmlRpc_Value_Nil`
base64	`Zend_XmlRpc_Value::XMLRPC_TYPE_BASE64`	`Zend_XmlRpc_Value_Base64`
dateTime.iso8601	`Zend_XmlRpc_Value::XMLRPC_TYPE_DATETIME`	`Zend_XmlRpc_Value_DateTime`
array	`Zend_XmlRpc_Value::XMLRPC_TYPE_ARRAY`	`Zend_XmlRpc_Value_Array`
struct	`Zend_XmlRpc_Value::XMLRPC_TYPE_STRUCT`	`Zend_XmlRpc_Value_Struct`

Automatic Conversion

When building a new `Zend_XmlRpc_Value` object, its value is set by a PHP type. The PHP type will be converted to the specified type using PHP casting. For example, if a string is given as a value to the `Zend_XmlRpc_Value_Integer` object, it will be converted using **(int)$value**.

2.4. Server Proxy Object

Another way to call remote methods with the XML-RPC client is to use the server proxy. This is a PHP object that proxies a remote XML-RPC namespace, making it work as close to a native PHP object as possible.

To instantiate a server proxy, call the `getProxy()` instance method of `Zend_XmlRpc_Client`. This will return an instance of `Zend_XmlRpc_Client_ServerProxy`. Any method call on the server proxy object will be forwarded to the remote, and parameters may be passed like any other PHP method.

Example 968. Proxy the Default Namespace

```
$client = new Zend_XmlRpc_Client('http://framework.zend.com/xmlrpc');

$service = $client->getProxy();           // Proxy the default namespace

$hello = $service->test->sayHello(1, 2);  // test.Hello(1, 2) returns "hello"
```

The `getProxy()` method receives an optional argument specifying which namespace of the remote server to proxy. If it does not receive a namespace, the default namespace will be proxied. In the next example, the 'test' namespace will be proxied:

Example 969. Proxy Any Namespace

```
$client = new Zend_XmlRpc_Client('http://framework.zend.com/xmlrpc');

$test  = $client->getProxy('test');     // Proxy the "test" namespace

$hello = $test->sayHello(1, 2);         // test.Hello(1,2) returns "hello"
```

If the remote server supports nested namespaces of any depth, these can also be used through the server proxy. For example, if the server in the example above had a method **test.foo.bar()**, it could be called as **$test->foo->bar()**.

2.5. Error Handling

Two kinds of errors can occur during an XML-RPC method call: HTTP errors and XML-RPC faults. The `Zend_XmlRpc_Client` recognizes each and provides the ability to detect and trap them independently.

2.5.1. HTTP Errors

If any HTTP error occurs, such as the remote HTTP server returns a *404 Not Found*, a `Zend_XmlRpc_Client_HttpException` will be thrown.

Example 970. Handling HTTP Errors

```
$client = new Zend_XmlRpc_Client('http://foo/404');

try {

    $client->call('bar', array($arg1, $arg2));

} catch (Zend_XmlRpc_Client_HttpException $e) {

    // $e->getCode() returns 404
    // $e->getMessage() returns "Not Found"

}
```

Regardless of how the XML-RPC client is used, the `Zend_XmlRpc_Client_HttpException` will be thrown whenever an HTTP error occurs.

2.5.2. XML-RPC Faults

An XML-RPC fault is analogous to a PHP exception. It is a special type returned from an XML-RPC method call that has both an error code and an error message. XML-RPC faults are handled differently depending on the context of how the `Zend_XmlRpc_Client` is used.

When the `call()` method or the server proxy object is used, an XML-RPC fault will result in a `Zend_XmlRpc_Client_FaultException` being thrown. The code and message of the exception will map directly to their respective values in the original XML-RPC fault response.

Example 971. Handling XML-RPC Faults

```
$client = new Zend_XmlRpc_Client('http://framework.zend.com/xmlrpc');

try {

    $client->call('badMethod');

} catch (Zend_XmlRpc_Client_FaultException $e) {

    // $e->getCode() returns 1
    // $e->getMessage() returns "Unknown method"

}
```

When the `call()` method is used to make the request, the `Zend_XmlRpc_Client_FaultException` will be thrown on fault. A `Zend_XmlRpc_Response` object containing the fault will also be available by calling `getLastResponse()`.

When the `doRequest()` method is used to make the request, it will not throw the exception. Instead, it will return a `Zend_XmlRpc_Response` object returned will containing the fault. This can be checked with `isFault()` instance method of `Zend_XmlRpc_Response`.

2.6. Server Introspection

Some XML-RPC servers support the de facto introspection methods under the XML-RPC *system.* namespace. `Zend_XmlRpc_Client` provides special support for servers with these capabilities.

A `Zend_XmlRpc_Client_ServerIntrospection` instance may be retrieved by calling the `getIntrospector()` method of `Zend_XmlRpcClient`. It can then be used to perform introspection operations on the server.

2.7. From Request to Response

Under the hood, the `call()` instance method of `Zend_XmlRpc_Client` builds a request object (`Zend_XmlRpc_Request`) and sends it to another method, `doRequest()`, that returns a response object (`Zend_XmlRpc_Response`).

The `doRequest()` method is also available for use directly:

Example 972. Processing Request to Response

```
$client = new Zend_XmlRpc_Client('http://framework.zend.com/xmlrpc');

$request = new Zend_XmlRpc_Request();
$request->setMethod('test.sayHello');
$request->setParams(array('foo', 'bar'));

$client->doRequest($request);

// $client->getLastRequest() returns instanceof Zend_XmlRpc_Request
// $client->getLastResponse() returns instanceof Zend_XmlRpc_Response
```

Whenever an XML-RPC method call is made by the client through any means, either the `call()` method, `doRequest()` method, or server proxy, the last request object and its resultant response object will always be available through the methods `getLastRequest()` and `getLastResponse()` respectively.

2.8. HTTP Client and Testing

In all of the prior examples, an HTTP client was never specified. When this is the case, a new instance of `Zend_Http_Client` will be created with its default options and used by `Zend_XmlRpc_Client` automatically.

The HTTP client can be retrieved at any time with the `getHttpClient()` method. For most cases, the default HTTP client will be sufficient. However, the `setHttpClient()` method allows for a different HTTP client instance to be injected.

The `setHttpClient()` is particularly useful for unit testing. When combined with the `Zend_Http_Client_Adapter_Test`, remote services can be mocked out for testing. See the unit tests for `Zend_XmlRpc_Client` for examples of how to do this.

3. Zend_XmlRpc_Server

3.1. Introduction

`Zend_XmlRpc_Server` is intended as a fully-featured XML-RPC server, following the specifications outlined at www.xmlrpc.com. Additionally, it implements the **system.multicall()** method, allowing boxcarring of requests.

3.2. Basic Usage

An example of the most basic use case:

```
$server = new Zend_XmlRpc_Server();
$server->setClass('My_Service_Class');
echo $server->handle();
```

3.3. Server Structure

`Zend_XmlRpc_Server` is composed of a variety of components, ranging from the server itself to request, response, and fault objects.

To bootstrap `Zend_XmlRpc_Server`, the developer must attach one or more classes or functions to the server, via the `setClass()` and `addFunction()` methods.

Once done, you may either pass a `Zend_XmlRpc_Request` object to `Zend_XmlRpc_Server::handle()`, or it will instantiate a `Zend_XmlRpc_Request_Http` object if none is provided -- thus grabbing the request from `php://input`.

`Zend_XmlRpc_Server::handle()` then attempts to dispatch to the appropriate handler based on the method requested. It then returns either a `Zend_XmlRpc_Response`-based object or a `Zend_XmlRpc_Server_Fault`object. These objects both have `__toString()` methods that create valid XML-RPC XML responses, allowing them to be directly echoed.

3.4. Anatomy of a webservice

3.4.1. General considerations

For maximum performance it is recommended to use a simple bootstrap file for the server component. Using `Zend_XmlRpc_Server` inside a `Zend_Controller` is strongly discouraged to avoid the overhead.

Services change over time and while webservices are generally less change intense as code-native APIs, it is recommended to version your service. Do so to lay grounds to provide compatibility for clients using older versions of your service and manage your service lifecycle including deprecation timeframes.To do so just include a version number into your URI. It is also recommended to include the remote protocol name in the URI to allow easy integration of upcoming remoting technologies. http://myservice.ws/*1.0/XMLRPC/*.

3.4.2. What to expose?

Most of the time it is not sensible to expose business objects directly. Business objects are usually small and under heavy change, because change is cheap in this layer of your application. Once deployed and adopted, web services are hard to change. Another concern is I/O and latency: the best webservice calls are those not happening. Therefore service calls need to be more coarse-grained than usual business logic is. Often an additional layer in front of your business objects makes sense. This layer is sometimes referred to as Remote Facade. Such a service layer adds a coarse grained interface on top of your business logic and groups verbose operations into smaller ones.

3.5. Conventions

`Zend_XmlRpc_Server` allows the developer to attach functions and class method calls as dispatchable XML-RPC methods. Via `Zend_Server_Reflection`, it does introspection on all attached methods, using the function and method docblocks to determine the method help text and method signatures.

XML-RPC types do not necessarily map one-to-one to PHP types. However, the code will do its best to guess the appropriate type based on the values listed in @param and @return lines. Some XML-RPC types have no immediate PHP equivalent, however, and should be hinted using the XML-RPC type in the PHPDoc. These include:

- *dateTime.iso8601*, a string formatted as '**YYYYMMDDTHH:mm:ss**'
- *base64*, base64 encoded data
- *struct*, any associative array

An example of how to hint follows:

```
/**
* This is a sample function
*
* @param base64 $val1 Base64-encoded data
* @param dateTime.iso8601 $val2 An ISO date
* @param struct $val3 An associative array
* @return struct
*/
function myFunc($val1, $val2, $val3)
{
}
```

PhpDocumentor does no validation of the types specified for params or return values, so this will have no impact on your API documentation. Providing the hinting is necessary, however, when the server is validating the parameters provided to the method call.

It is perfectly valid to specify multiple types for both params and return values; the XML-RPC specification even suggests that system.methodSignature should return an array of all possible method signatures (i.e., all possible combinations of param and return values). You may do so just as you normally would with PhpDocumentor, using the '|' operator:

```
/**
* This is a sample function
*
* @param string|base64 $val1 String or base64-encoded data
* @param string|dateTime.iso8601 $val2 String or an ISO date
* @param array|struct $val3 Normal indexed array or an associative array
* @return boolean|struct
*/
function myFunc($val1, $val2, $val3)
{
}
```

Allowing multiple signatures can lead to confusion for developers using the services; to keep things simple, a XML-RPC service method should only have a single signature.

3.6. Utilizing Namespaces

XML-RPC has a concept of namespacing; basically, it allows grouping XML-RPC methods by dot-delimited namespaces. This helps prevent naming collisions between methods served by

different classes. As an example, the XML-RPC server is expected to server several methods in the 'system' namespace:

- system.listMethods
- system.methodHelp
- system.methodSignature

Internally, these map to the methods of the same name in `Zend_XmlRpc_Server`.

If you want to add namespaces to the methods you serve, simply provide a namespace to the appropriate method when attaching a function or class:

```
// All public methods in My_Service_Class will be accessible as
// myservice.METHODNAME
$server->setClass('My_Service_Class', 'myservice');

// Function 'somefunc' will be accessible as funcs.somefunc
$server->addFunction('somefunc', 'funcs');
```

3.7. Custom Request Objects

Most of the time, you'll simply use the default request type included with `Zend_XmlRpc_Server`, `Zend_XmlRpc_Request_Http`. However, there may be times when you need XML-RPC to be available via the CLI, a GUI, or other environment, or want to log incoming requests. To do so, you may create a custom request object that extends `Zend_XmlRpc_Request`. The most important thing to remember is to ensure that the `getMethod()` and `getParams()` methods are implemented so that the XML-RPC server can retrieve that information in order to dispatch the request.

3.8. Custom Responses

Similar to request objects, `Zend_XmlRpc_Server` can return custom response objects; by default, a `Zend_XmlRpc_Response_Http` object is returned, which sends an appropriate Content-Type HTTP header for use with XML-RPC. Possible uses of a custom object would be to log responses, or to send responses back to `STDOUT`.

To use a custom response class, use `Zend_XmlRpc_Server::setResponseClass()` prior to calling `handle()`.

3.9. Handling Exceptions via Faults

`Zend_XmlRpc_Server` catches Exceptions generated by a dispatched method, and generates an XML-RPC fault response when such an exception is caught. By default, however, the exception messages and codes are not used in a fault response. This is an intentional decision to protect your code; many exceptions expose more information about the code or environment than a developer would necessarily intend (a prime example includes database abstraction or access layer exceptions).

Exception classes can be whitelisted to be used as fault responses, however. To do so, simply utilize `Zend_XmlRpc_Server_Fault::attachFaultException()` to pass an exception class to whitelist:

```
Zend_XmlRpc_Server_Fault::attachFaultException('My_Project_Exception');
```

If you utilize an exception class that your other project exceptions inherit, you can then whitelist a whole family of exceptions at a time. `Zend_XmlRpc_Server_Exceptions` are always whitelisted, to allow reporting specific internal errors (undefined methods, etc.).

Any exception not specifically whitelisted will generate a fault response with a code of '404' and a message of 'Unknown error'.

3.10. Caching Server Definitions Between Requests

Attaching many classes to an XML-RPC server instance can utilize a lot of resources; each class must introspect using the Reflection API (via `Zend_Server_Reflection`), which in turn generates a list of all possible method signatures to provide to the server class.

To reduce this performance hit somewhat, `Zend_XmlRpc_Server_Cache` can be used to cache the server definition between requests. When combined with `__autoload()`, this can greatly increase performance.

An sample usage follows:

```
function __autoload($class)
{
    Zend_Loader::loadClass($class);
}

$cacheFile = dirname(__FILE__) . '/xmlrpc.cache';
$server = new Zend_XmlRpc_Server();

if (!Zend_XmlRpc_Server_Cache::get($cacheFile, $server)) {
    require_once 'My/Services/Glue.php';
    require_once 'My/Services/Paste.php';
    require_once 'My/Services/Tape.php';

    $server->setClass('My_Services_Glue', 'glue');   // glue. namespace
    $server->setClass('My_Services_Paste', 'paste'); // paste. namespace
    $server->setClass('My_Services_Tape', 'tape');   // tape. namespace

    Zend_XmlRpc_Server_Cache::save($cacheFile, $server);
}

echo $server->handle();
```

The above example attempts to retrieve a server definition from xmlrpc.cache in the same directory as the script. If unsuccessful, it loads the service classes it needs, attaches them to the server instance, and then attempts to create a new cache file with the server definition.

3.11. Usage Examples

Below are several usage examples, showing the full spectrum of options available to developers. Usage examples will each build on the previous example provided.

Example 973. Basic Usage

The example below attaches a function as a dispatchable XML-RPC method and handles incoming calls.

```
/**
 * Return the MD5 sum of a value
 *
 * @param string $value Value to md5sum
 * @return string MD5 sum of value
 */
function md5Value($value)
{
    return md5($value);
}

$server = new Zend_XmlRpc_Server();
$server->addFunction('md5Value');
echo $server->handle();
```

Example 974. Attaching a class

The example below illustrates attaching a class' public methods as dispatchable XML-RPC methods.

```
require_once 'Services/Comb.php';

$server = new Zend_XmlRpc_Server();
$server->setClass('Services_Comb');
echo $server->handle();
```

Example 975. Attaching a class with arguments

The following example illustrates how to attach a class' public methods and passing arguments to its methods. This can be used to specify certain defaults when registering service classes.

```
class Services_PricingService
{
    /**
     * Calculate current price of product with $productId
     *
     * @param ProductRepository $productRepository
     * @param PurchaseRepository $purchaseRepository
     * @param integer $productId
     */
    public function calculate(ProductRepository $productRepository,
                              PurchaseRepository $purchaseRepository,
                              $productId)
    {
        ...
    }
}

$server = new Zend_XmlRpc_Server();
$server->setClass('Services_PricingService',
                  'pricing',
                  new ProductRepository(),
                  new PurchaseRepository());
```

The arguments passed at `setClass()` at server construction time are injected into the method call **pricing.calculate()** on remote invokation. In the example above, only the argument `$purchaseId` is expected from the client.

Example 976. Passing arguments only to constructor

`Zend_XmlRpc_Server` allows to restrict argument passing to constructors only. This can be used for constructor dependency injection. To limit injection to constructors, call `sendArgumentsToAllMethods` and pass `FALSE` as an argument. This disables the default behavior of all arguments being injected into the remote method. In the example below the instance of `ProductRepository` and `PurchaseRepository` is only injected into the constructor of `Services_PricingService2`.

```
class Services_PricingService2
{
    /**
     * @param ProductRepository $productRepository
     * @param PurchaseRepository $purchaseRepository
     */
    public function __construct(ProductRepository $productRepository,
                                PurchaseRepository $purchaseRepository)
    {
        ...
    }

    /**
     * Calculate current price of product with $productId
     *
     * @param integer $productId
     * @return double
     */
    public function calculate($productId)
    {
        ...
    }
}

$server = new Zend_XmlRpc_Server();
$server->sendArgumentsToAllMethods(false);
$server->setClass('Services_PricingService2',
                  'pricing',
                  new ProductRepository(),
                  new PurchaseRepository());
```

Example 977. Attaching a class instance

`setClass()` allows to register a previously instantiated object at the server. Just pass an instance instead of the class name. Obviously passing arguments to the constructor is not possible with pre-instantiated objects.

Example 978. Attaching several classes using namespaces

The example below illustrates attaching several classes, each with their own namespace.

```
require_once 'Services/Comb.php';
require_once 'Services/Brush.php';
require_once 'Services/Pick.php';

$server = new Zend_XmlRpc_Server();
$server->setClass('Services_Comb', 'comb');   // methods called as comb.*
$server->setClass('Services_Brush', 'brush'); // methods called as brush.*
$server->setClass('Services_Pick', 'pick');   // methods called as pick.*
echo $server->handle();
```

Example 979. Specifying exceptions to use as valid fault responses

The example below allows any `Services_Exception`-derived class to report its code and message in the fault response.

```
require_once 'Services/Exception.php';
require_once 'Services/Comb.php';
require_once 'Services/Brush.php';
require_once 'Services/Pick.php';

// Allow Services_Exceptions to report as fault responses
Zend_XmlRpc_Server_Fault::attachFaultException('Services_Exception');

$server = new Zend_XmlRpc_Server();
$server->setClass('Services_Comb', 'comb');   // methods called as comb.*
$server->setClass('Services_Brush', 'brush'); // methods called as brush.*
$server->setClass('Services_Pick', 'pick');   // methods called as pick.*
echo $server->handle();
```

Example 980. Utilizing custom request and response objects

Some use cases require to utilize a custom request object. For example, XML/RPC is not bound to HTTP as a transfer protocol. It is possible to use other transfer protocols like SSH or telnet to send the request and response data over the wire. Another use case is authentication and authorization. In case of a different transfer protocol, one need to change the implementation to read request data.

The example below instantiates a custom request object and passes it to the server to handle.

```
require_once 'Services/Request.php';
require_once 'Services/Exception.php';
require_once 'Services/Comb.php';
require_once 'Services/Brush.php';
require_once 'Services/Pick.php';

// Allow Services_Exceptions to report as fault responses
Zend_XmlRpc_Server_Fault::attachFaultException('Services_Exception');

$server = new Zend_XmlRpc_Server();
$server->setClass('Services_Comb', 'comb');   // methods called as comb.*
$server->setClass('Services_Brush', 'brush'); // methods called as brush.*
$server->setClass('Services_Pick', 'pick');   // methods called as pick.*

// Create a request object
$request = new Services_Request();

echo $server->handle($request);
```

Example 981. Specifying a custom response class

The example below illustrates specifying a custom response class for the returned response.

```
require_once 'Services/Request.php';
require_once 'Services/Response.php';
require_once 'Services/Exception.php';
require_once 'Services/Comb.php';
require_once 'Services/Brush.php';
require_once 'Services/Pick.php';

// Allow Services_Exceptions to report as fault responses
Zend_XmlRpc_Server_Fault::attachFaultException('Services_Exception');

$server = new Zend_XmlRpc_Server();
$server->setClass('Services_Comb', 'comb');   // methods called as comb.*
$server->setClass('Services_Brush', 'brush'); // methods called as brush.*
$server->setClass('Services_Pick', 'pick');   // methods called as pick.*

// Create a request object
$request = new Services_Request();

// Utilize a custom response
$server->setResponseClass('Services_Response');

echo $server->handle($request);
```

3.12. Performance optimization

Example 982. Cache server definitions between requests

The example below illustrates caching server definitions between requests.

```
// Specify a cache file
$cacheFile = dirname(__FILE__) . '/xmlrpc.cache';

// Allow Services_Exceptions to report as fault responses
Zend_XmlRpc_Server_Fault::attachFaultException('Services_Exception');

$server = new Zend_XmlRpc_Server();

// Attempt to retrieve server definition from cache
if (!Zend_XmlRpc_Server_Cache::get($cacheFile, $server)) {
    $server->setClass('Services_Comb', 'comb');   // methods called as comb.*
    $server->setClass('Services_Brush', 'brush'); // methods called as brush.*
    $server->setClass('Services_Pick', 'pick');   // methods called as pick.*

    // Save cache
    Zend_XmlRpc_Server_Cache::save($cacheFile, $server);
}

// Create a request object
$request = new Services_Request();

// Utilize a custom response
$server->setResponseClass('Services_Response');

echo $server->handle($request);
```

The server cache file should be located outside the document root.

Example 983. Optimizing XML generation

`Zend_XmlRpc_Server` uses `DOMDocument` of PHP extension `ext/dom` to generate it's XML output. While `ext/dom` is available on a lot of hosts it is not exactly the fastest. Benchmarks have shown, that `XMLWriter` from `ext/xmlwriter` performs better.

If `ext/xmlwriter` is available on your host, you can select a the `XMLWriter`-based generator to leaverage the performance differences.

```
require_once 'Zend/XmlRpc/Server.php';
require_once 'Zend/XmlRpc/Generator/XMLWriter.php';

Zend_XmlRpc_Value::setGenerator(new Zend_XmlRpc_Generator_XMLWriter());

$server = new Zend_XmlRpc_Server();
...
```

Benchmark your application

Performance is determined by a lot of parameters and benchmarks only apply for the specific test case. Differences come from PHP version, installed extensions, webserver and operating system just to name a few. Please make sure to benchmark your application on your own and decide which generator to use based on *your* numbers.

Benchmark your client

This optimization makes sense for the client side too. Just select the alternate XML generator before doing any work with `Zend_XmlRpc_Client`.

ZendX_Console_Process_Unix

1. ZendX_Console_Process_Unix

1.1. Introduction

`ZendX_Console_Process_Unix` allows developers to spawn an object as a new process, and so do multiple tasks in parallel on console environments. Through its specific nature, it is only working on *nix based systems like Linux, Solaris, Mac/OSx and such. Additionally, the `shmop_*`, `pcntl_*` and `posix_*` modules are required for this component to run. If one of the requirements is not met, it will throw an exception after instantiating the component.

1.2. Basic usage of ZendX_Console_Process_Unix

`ZendX_Console_Process_Unix` is an abstract class, which requires the user to extend it. It has a single abstract method called `_run()` which has to be implemented to create a working process. It also comes with multiple methods for checking the alive status and share variables between the parent and the child process.

The `_run()` method and every method which is called by it is executed by the child process. Every other method which is called directly by the parent is executed by the parent process.

`setVariable()` and `getVariable()` can be used from both the parent- and the child process to share variables. To observe the alive status, the child process should call `_setAlive()` in a frequent interval, so that the parent process can check the last alive time via `getLastAlive()`. To get the PID of the child process, the parent can call `getPid()`.

Example 984. Basic example for processing

This example illustrates a basic child process

```
class MyProcess extends ZendX_Console_Process_Unix
{
    protected function _run()
    {
        for ($i = 0; $i < 10; $i++) {
            // Doing something really important which can't wait: sleeping
            sleep(1);
        }
    }
}

// This part should last about 10 seconds, not 20.
$process1 = new MyProcess();
$process1->start();

$process2 = new MyProcess();
$process2->start();

while ($process1->isRunning() && $process2->isRunning()) {
    sleep(1);
}

echo 'All processes completed';
```

In this example a process is forked twice and executed. As every process runs 10 seconds, the parent process will be finished after 10 seconds (and not 20).

ZendX_JQuery

1. Introduction

As of version 1.7, Zend Framework integrates jQuery view and form helpers through its extras library. The jQuery support is meant as an alternative to the already existing Dojo library integration. Currently jQuery can be integrated into your Zend Framework applications in the following ways:

- View helper to help setup the jQuery (Core and UI) environment
- jQuery UI specific Zend_View helpers
- jQuery UI specific Zend_Form elements and decorators

By default the jQuery javascript dependencies are loaded from the Google Ajax Library Content Distribution Network. The CDN offers both jQuery Core and jQuery UI access points and the view helpers therefore can already offer you most the dependencies out of the box. Currently the Google CDN offers jQuery UI support up to version 1.5.2, but the jQuery view and form helpers already make use of the UI library 1.6 version (AutoComplete, ColorPicker, Spinner, Slider). To make use of these great additions you have to download the release candidate version of the jQuery UI library from its website.

2. ZendX_JQuery View Helpers

Zend Framework provides jQuery related View Helpers through its Extras Library. These can be enabled in two ways, adding jQuery to the view helper path:

```
$view->addHelperPath("ZendX/JQuery/View/Helper", "ZendX_JQuery_View_Helper");
```

Or using the `ZendX_JQuery::enableView(Zend_View_Interface $view)` method that does the same for you.

2.1. jQuery() View Helper

The `jQuery()` view helper simplifies setup of your jQuery environment in your application. It takes care of loading the core and ui library dependencies if necessary and acts as a stack for all the registered onLoad javascript statements. All jQuery view helpers put their javascript code onto this stack. It acts as a collector for everything jQuery in your application with the following responsibilities:

- Handling deployment of CDN or a local path jQuery Core and UI libraries.
- Handling $(document).onLoad() events.
- Specifying additional stylesheet themes to use.

The `jQuery()` view helper implementation, like its `dojo()` pendant, follows the placeholder architecture implementation; the data set in it persists between view objects, and may be directly echo'd from your layout script. Since views specified in a Zend_Layout script file are rendered before the layout itself, the `jQuery()` helper can act as a stack for jQuery statements and render them into the head segment of the html page.

Contrary to Dojo, themes cannot be loaded from a CDN for the jQuery UI widgets and have to be implemented in your pages stylesheet file or loaded from an extra stylesheet file. A default theme called Flora can be obtained from the jQuery UI downloadable file.

Example 985. jQuery() View Helper Example

In this example a jQuery environment using the core and UI libraries will be needed. UI Widgets should be rendered with the Flora thema that is installed in 'public/styles/flora.all.css'. The jQuery libraries are both loaded from local paths.

To register the jQuery functionality inside the view object, you have to add the appropriate helpers to the view helper path. There are many ways of accomplishing this, based on the requirements that the jQuery helpers have. If you need them in one specific view only, you can use the addHelperPath method on initialization of this view, or right before rendering:

```
$view->addHelperPath('ZendX/JQuery/View/Helper/', 'ZendX_JQuery_View_Helper');
```

If you need them throughout your application, you can register them in your bootstrap file using access to the Controller Plugin ViewRenderer:

```
$view = new Zend_View();
$view->addHelperPath('ZendX/JQuery/View/Helper/', 'ZendX_JQuery_View_Helper');

$viewRenderer = new Zend_Controller_Action_Helper_ViewRenderer();
$viewRenderer->setView($view);
Zend_Controller_Action_HelperBroker::addHelper($viewRenderer);
```

Now in the view script we want to display a Date Picker and an Ajax based Link.

```
<?php echo $this->ajaxLink("Show me something",
                    "/hello/world",
                    array('update' => '#content'));?>
<div id="content"></div>

<form method="post" action="/hello/world">
Pick your Date: <?php echo $this->datePicker("dp1",
                                             '',
                                             array(
                                                'defaultDate' =>
                                                    date('Y/m/d', time())));?>
<input type="submit" value="Submit" />
</form>
```

Both helpers now stacked some javascript statements on the jQuery helper and printed a link and a form element respectively. To access the javascript we have to utilize the jQuery() functionality. Both helpers already activated their dependencies that is they have called `jQuery()->enable()` and `jQuery()->uiEnable()`. We only have to print the jQuery() environment, and we choose to do so in the layout script's head segment:

```
<html>
    <head>
        <title>A jQuery View Helper Example</title>
        <?php echo $this->jQuery(); ?>
    </head>

    <body>
        <?php echo $this->layout()->content; ?>
    </body>
</html>
```

Although `$this->layout()->content;` is printed behind the `$this->jQuery()` statement, the content of the view script is rendered before. This way all the javascript onLoad code has already been put on the onLoad stack and can be printed within the head segment of the html document.

2.1.1. jQuery NoConflict Mode

jQuery offers a noConflict mode that allows the library to be run side by side with other javascript libraries that operate in the global namespace, Prototype for example. The Zend Framework jQuery View Helper makes usage of the noConflict mode very easy. If you want to run Prototype and jQuery side by side you can call `ZendX_JQuery_View_Helper_JQuery::enableNoConflictMode();` and all jQuery helpers will operate in the No Conflict Mode.

Example 986. Building your own Helper with No Conflict Mode

To make use of the NoConflict Mode in your own jQuery helper, you only have to use the static method `ZendX_JQuery_View_Helper_JQuery::getJQueryHandler()` method. It returns the variable jQuery is operating in at the moment, either $ or `$j`

```
class MyHelper_SomeHelper extends Zend_View_Helper_Abstract
{
    public function someHelper()
    {
        $jquery = $this->view->jQuery();
        $jquery->enable(); // enable jQuery Core Library

        // get current jQuery handler based on noConflict settings
        $jqHandler = ZendX_JQuery_View_Helper_JQuery::getJQueryHandler();

        $function = '("#element").click(function() '
                  . '{ alert("noConflict Mode Save Helper!"); }'
                  . ')';
        $jquery->addOnload($jqHandler . $function);
        return '';
    }
}
```

2.1.2. jQuery UI Themes

Since there are no online available themes to use out of the box, the implementation of the UI library themes is a bit more complex than with the Dojo helper. The jQuery UI documentation describes for each component what stylesheet information is needed and the Default and Flora Themes from the downloadable archive give hints on the usage of stylesheets. The jQuery helper offers the function `jQuery()->addStylesheet($path);` function to include the dependant stylesheets whenever the helper is enabled and rendered. You can optionally merge the required stylesheet information in your main stylesheet file.

2.1.3. Methods Available

The `jQuery()` view helper always returns an instance of the jQuery placeholder container. That container object has the following methods available:

2.1.3.1. jQuery Core Library methods

- `enable()`: explicitly enable jQuery integration.
- `disable()`: disable jQuery integration.
- `isEnabled()`: determine whether or not jQuery integration is enabled.
- `setVersion()`: set the jQuery version that is used. This also decides on the library loaded from the Google Ajax Library CDN

- `getVersion()`: get the current jQuery that is used. This also decides on the library loaded from the Google Ajax Library CDN
- `useCdn()`: Return true, if CDN usage is currently enabled
- `useLocalPath()`: Return true, if local usage is currently enabled
- `setLocalPath()`: Set the local path to the jQuery Core library
- `getLocalPath()`: If set, return the local path to the jQuery Core library

2.1.3.2. jQuery UI Library methods

- `uiEnable()`: explicitly enable jQuery UI integration.
- `uiDisable()`: disable jQuery UI integration.
- `uiIsEnabled()`: determine whether or not jQuery UI integration is enabled.
- `setUiVersion()`: set the jQuery UI version that is used. This also decides on the library loaded from the Google Ajax Library CDN
- `getUiVersion()`: get the current jQuery UI that is used. This also decides on the library loaded from the Google Ajax Library CDN
- `useUiCdn()`: Return true, if CDN usage is currently enabled for jQuery UI
- `useUiLocal()`: Return true, if local usage is currently enabled for jQuery UI
- `setUiLocalPath()`: Set the local path to the jQuery UI library
- `getUiLocalPath()`: If set, get the local path to the jQuery UI library

2.1.3.3. jQuery Helper Utility methods

- `setView(Zend_View_Interface $view)`: set a view instance in the container.
- `onLoadCaptureStart()`: Start capturing javascript code for jQuery onLoad execution.
- `onLoadCaptureEnd()`: Stop capturing
- `javascriptCaptureStart()`: Start capturing javascript code that has to be rendered after the inclusion of either jQuery Core or UI libraries.
- `javascriptCaptureEnd()`: Stop capturing.
- `addJavascriptFile($path)`: Add javascript file to be included after jQuery Core or UI library.
- `getJavascriptFiles()`: Return all currently registered additional javascript files.
- `clearJavascriptFiles()`: Clear the javascript files
- `addJavascript($statement)`: Add javascript statement to be included after jQuery Core or UI library.
- `getJavascript()`: Return all currently registered additional javascript statements.

- `clearJavascript()`: Clear the javascript statements.
- `addStylesheet($path)`: Add a stylesheet file that is needed for a jQuery view helper to display correctly.
- `getStylesheets()`: Get all currently registered additional stylesheets.
- `addOnLoad($statement)`: Add javascript statement that should be executed on document loading.
- `getOnLoadActions()`: Return all currently registered onLoad statements.
- `setRenderMode($mask)`: Render only a specific subset of the jQuery environment via ZendX_JQuery::RENDER_ constants. Rendering all elements is the default behaviour.
- `getRenderMode()`: Return the current jQuery environment rendering mode.
- `setCdnSsl($bool)`: Set if the CDN Google Ajax Library should be loaded from an SSL or a Non-SSL location.

These are quite a number of methods, but many of them are used for internally by all the additional view helpers and during the printing of the jQuery environment. Unless you want to build your own jQuery helper or have a complex use-case, you will probably only get in contact with a few methods of these.

2.1.4. Refactoring jQuery environment with setRenderMode()

Using the current setup that was described, each page of your website would show a different subset of jQuery code that would be needed to keep the current jQuery related items running. Also different files or stylesheets may be included depending on which helpers you implemented in your application. In production stage you might want to centralize all the javascript your application generated into a single file, or disable stylesheet rendering because you have merged all the stylesheets into a single file and include it statically in your layout. To allow a smooth refactoring you can enable or disable the rendering of certain jQuery environment blocks with help of the following constants and the `jQuery()->setRenderMode($bitmask)` function.

- `ZendX_JQuery::RENDER_LIBRARY`: Renders jQuery Core and UI library
- `ZendX_JQuery::RENDER_SOURCES`: Renders additional javascript files
- `ZendX_JQuery::RENDER_STYLESHEETS`: Renders jQuery related stylesheets
- `ZendX_JQuery::RENDER_JAVASCRIPT`: Render additional javascript statements
- `ZendX_JQuery::RENDER_JQUERY_ON_LOAD`: Render jQuery onLoad statements
- `ZendX_JQuery::RENDER_ALL`: Render all previously mentioned blocks, this is default behaviour.

For an example, if you would have merged jQuery Core and UI libraries as well as other files into a single large file as well as merged stylesheets to keep HTTP requests low on your production application. You could disallow the jQuery helper to render those parts, but render all the other stuff with the following statement in your view:

```
$view->jQuery()
```

```
    ->setRenderMode(ZendX_JQuery::RENDER_JAVASCRIPT |
                    ZendX_JQuery::RENDER_JQUERY_ON_LOAD);
```

This statement makes sure only the required javascript statements and onLoad blocks of the current page are rendered by the jQuery helper.

2.1.5. Migrations

Prior to 1.8 the methods `setCdnVersion()`, `setLocalPath()` `setUiCdnVersion()` and `setUiLocalPath()` all enabled the view helper upon calling, which is considered a bug from the following perspective: If you want to use the any non-default library option, you would have to manually disable the jQuery helper aftwards if you only require it to be loaded in some scenarios. With version 1.8 the jQuery helper does only enable itsself, when `enable()` is called, which all internal jQuery View helpers do upon being called.

2.2. JQuery Helpers

2.2.1. AjaxLink Helper

The AjaxLink helper uses jQuery's ajax capabilities to offer the creation of links that do ajax requests and inject the response into a chosen DOM element. It also offers the possibility to append simple jQuery effects to both the link and the response DOM element. A simple example introduces its functionality:

```
<!-- Inside your View Object -->
<div id="container"></div>
<?php echo $this->view->ajaxLink("Link Name",
                         "url.php",
                         array('update' => '#container')); ?>
```

This example creates a link with the label "Link Name" that fires an ajax request to url.php upon click and renders the response into the div container "#container". The function header for the ajaxLink is as follows: `function ajaxLink($label, $url, $options, $params);` The options array is very powerful and offers you lots of functionality to customize your ajax requests.

Available options are:

Table 157. AjaxLink options

Option	Data Type	Default Value	Description
`update`	`string`	`false`	Container to inject response content into, use jQuery CSS Selector syntax, ie. "#container" or ".box"
`method`	`string`	`Implicit GET or POST`	Request method, is implicitly chosen as GET when no parameters given and POST when parameters given.
`complete`	`string`	`false`	Javascript callback executed, when ajax

Option	Data Type	Default Value	Description
			request is complete. This option allows for shortcut effects, see next section.
beforeSend	string	false	Javascript callback executed right before ajax request is started. This option allows for shortcut effects, see next section.
noscript	boolean	true	If true the link generated will contain a href attribute to the given link for non-javascript enabled browsers. If false href will resolve to "#".
dataType	string	html	What type of data is the Ajax Response of? Possible are Html, Text, Json. Processing Json responses has to be done with custom "complete" callback functions.
attribs	array	null	Additional HTML attributes the ajaxable link should have.
title, id, class	string	false	Convenience shortcuts for HTML Attributes.
inline	boolean	false	Although far from best practice, you can set javascript for this link inline in "onclick" attribute.

To enlighten the usage of this helper it is best to show another bunch of more complex examples. This example assumes that you have only one view object that you want to display and don't care a lot about html best practices, since we have to output the jQuery environment just before the closing body tag.

```
<html>
    <head>
        <title>Zend Framework jQuery AjaxLink Example</title>
        <script language="javascript"
                type="text/javascript"
                src="myCallbackFuncs.js"></script>
    </head>

    <body>
```

```
        <!-- without echoing jQuery this following -->
        <!-- list only prints a list of for links -->
        <ul>
            <li>
                <?php echo $this->ajaxLink("Example 1",
                                           "/ctrl/action1",
                                           array('update' => '#content',
                                                 'noscript' => false,
                                                 'method' => 'POST')); ?>
            </li>
            <li>
                <?php echo $this->ajaxLink("Example 2",
                                           "/ctrl/action2",
                                           array('update' => '#content',
                                                 'class' => 'someLink'),
                                           array('param1' => 'value1',
                                                 'param2' => 'value2')); ?>
            </li>
            <li><?php echo $this->ajaxLink("Example 3",
                                           "/ctrl/action3",
                                           array('dataType' => 'json',
                                                 'complete' =>
                                                    'alert(data)')); ?>
            </li>
            <li><?php echo $this->ajaxLink("Example 4",
                                           "/ctrl/action4",
                                           array('beforeSend' => 'hide',
                                                 'complete' => 'show')); ?>
            </li>
            <li>
                <?php echo $this->ajaxLink("Example 5",
                                           "/ctrl/action5",
                                           array(
                                            'beforeSend' =>
                                             'myBeforeSendCallbackJsFunc();',
                                            'complete' =>
                                             'myCompleteCallbackJsFunc(data);')
                                           ); ?>
            </li>
        </ul>

        <!-- only at this point the javascript is printed to sreen -->
        <?php echo $this->jQuery(); ?>
    </body>
</html>
```

You might have already seen that the 'update', 'complete', and 'beforeSend' options have to be executed in specific order and syntax so that you cannot use those callbacks and override their behaviour completely when you are using `ajaxLink()`. For larger use cases you will probably want to write the request via jQuery on your own. The primary use case for the callbacks is effect usage, other uses may very well become hard to maintain. As shown in Example Link 5, you can also forward the beforeSend/complete Callbacks to your own javascript functions.

2.2.1.1. Shortcut Effects

You can use shortcut effect names to make your links actions more fancy. For example the Container that will contain the ajax response may very well be invisible in the first place. Additionally you can use shortcut effects on the link to hide it after clicking. The following effects can be used for callbacks:

- `complete` callback: 'show', 'showslow', 'shownormal', 'showfast', 'fadein', 'fadeinslow', 'fadeinfast', 'slidedown', 'slidedownslow', 'slidedownfast'. These all correspond to the jQuery effects fadeIn(), show() and slideDown() and will be executed on the container specified in `update`.
- `beforeSend` callback: 'fadeout', 'fadeoutslow', 'fadeoutfast', 'hide', 'hideslow', 'hidefast', 'slideup'. These correspond to the jQuery effects fadeOut(), hide(), slideUp() and are executed on the clicked link.

```
<?php echo $this->ajaxLink("Example 6",
                           "/ctrl/action6",
                           array('beforeSend' => 'hide',
                                 'complete' => 'show')); ?>
```

2.2.2. jQuery UI Library Helpers

The jQuery UI Library offers a range of layout and form specific widgets that are integrated into the Zend Framework via View Helpers. The form-elements are easy to handle and will be described first, whereas the layout specific widgets are a bit more complex to use.

2.2.2.1. jQuery UI Form Helpers

The method signature for all form view helpers closely resembles the Dojo View helpers signature, `helper($id, $value, $params, $attribs)`. A description of the parameters follows:

- `$id`: Will act as the identifier name for the helper element inside a form. If in the attributes no id element is given, this will also become the form element id, that has to be unique across the DOM.
- `$value`: Default value of the element.
- `$params`: Widget specific parameters that customize the look and feel of the widget. These options are unique to each widget and described in the jQuery UI documentation. The data is casted to JSON, so make sure to use the `Zend_Json_Expr` class to mark executable javascript as safe.
- `$attribs`: HTML Attributes of the Form Helper

The following UI widgets are available as form view helpers. Make sure you use the correct version of jQuery UI library to be able to use them. The Google CDN only offers jQuery UI up to version 1.5.2. Some other components are only available from jQuery UI SVN, since they have been removed from the announced 1.6 release.

- `autoComplete($id, $value, $params, $attribs)`: The AutoComplete View helper will be included in a future jQuery UI version (currently only via jQuery SVN) and creates a text field and registeres it to have auto complete functionality. The completion data source has to be given as jQuery related parameters 'url' or 'data' as described in the jQuery UI manual.
- `colorPicker($id, $value, $params, $attribs)`: ColorPicker is currently available in jQuery UI only from SVN and creates a text field that opens up a color picking tool when activated.
- `datePicker($id, $value, $params, $attribs)`: Create a DatePicker inside a text field. This widget is available since jQuery UI 1.5 and can therefore currently be used with the Google CDN. Using the 'handles' option to create multiple handles overwrites the default set value and the jQuery parameter 'startValue' internally inside the view helper.

- `slider($id, $value, $params, $attribs)`: Create a Sliding element that updates its value into a hidden form field. Available since jQuery UI 1.5.

- `spinner($id, $value, $params, $attribs)`: Create a Spinner element that can spin through numeric values in a specified range. This is set on top of a form text field and is available only from jQuery UI SVN

Example 987. Showing jQuery Form View Helper Usage

In this example we want to simulate a fictional web application that offers auctions on travel locations. A user may specify a city to travel, a start and end date, and a maximum amount of money he is willing to pay. Therefore we need an autoComplete field for all the currently known travel locations, a date picker for start and end dates and a spinner to specify the amount.

```
<form method="post" action="bid.php">
    <label for="locaction">Where do you want to travel?</label>
    <?php echo $this->autoComplete("location",
                                   "",
                                   array('data' => array('New York',
                                                         'Mexico City',
                                                         'Sydney',
                                                         'Ruegen',
                                                         'Baden Baden'),
                                         'multiple' => true)); ?>
    <br />

    <label for="startDate">Travel Start Date:</label>
    <?php echo $this->datePicker("startDate", '',
             array(
                'defaultDate' => '+7',
                'minDate' => '+7',
                'onClose' => new Zend_Json_Expr('myJsonFuncCechkingValidity'))); ?>
    <br />

    <label for="startDate">Travel End Date:</label>
    <?php echo $this->datePicker("endDate", '',
             array(
                'defaultDate' => '+14',
                'minDate' => '+7',
                'onClose' => new Zend_Json_Expr('myJsonFuncCechkingValidity'))); ?>
    <br />

    <label for="bid">Your Bid:</label>
    <?php echo $this->spinner("bid",
                              "",
                              array('min' => 1205.50,
                                    'max' => 10000,
                                    'start' => 1205.50,
                                    'currency' => '€')); ?>
    <br />

    <input type="submit" value="Bid!" />
</form>
```

You can see the use of jQuery UI Widget specific parameters. These all correspond to those given in the jQuery UI docs and are converted to JSON and handed through to the view script.

2.2.2.2. Using an Action Helper to Send Data to AutoComplete

The jQuery UI Autocomplete Widget can load data from a remote location rather than from an javascript array, making its usage really useful. Zend Framework currently providers a bunch of server-side AutoComplete Helpers and there is one for jQuery too. You register the helper to the controller helper broker and it takes care of disabling layouts and renders an array of data correctly to be read by the AutoComplete field. To use the Action Helper you have to put this rather long statement into your bootstrap or Controller initialization function:

```
Zend_Controller_Action_HelperBroker::addHelper(
    new ZendX_JQuery_Controller_Action_Helper_AutoComplete()
);
```

You can then directly call the helper to render AutoComplete Output in your Controller

```
class MyIndexController extends Zend_Controller_Action
{
    public function autocompleteAction()
    {
        // The data sent via the ajax call is inside $_GET['q']
        $filter = $_GET['q'];

        // Disable Layout and stuff, just displaying AutoComplete Information.
        $this->_helper->autoComplete(array("New York", "Bonn", "Tokio"));
    }
}
```

2.2.2.3. jQuery UI Layout Helpers

There is a wide range of Layout helpers that the UI library offers. The ones covered by Zend Framework view helpers are Accordion, Dialog, Tabs. Dialog is the most simple one, whereas Accordion and Tab extend a common abstract class and offer a secondary view helper for pane generation. The following view helpers exist in the jQuery view helpers collection, an example accompanies them to show their usage.

- `dialogContainer($id, $content, $params, $attribs)`: Create a Dialog Box that is rendered with the given content.on startup. If the option 'autoOpen' set to false is specified the box will not be displayed on load but can be shown with the additional `dialog("open")` javascript function. See UI docs for details.

- `tabPane($id, $content, $options)`: Add a new pane to a tab container with the given `$id`. The given `$content` is shown in this tab pane. To set the title use `$options['title']`. If `$options['contentUrl']` is set, the content of the tab is requested via ajax on tab activation.

- `tabContainer($id, $params, $attribs)`: Render a tab container with all the currently registered panes. This view helper also offers to add panes with the following syntax: `$this->tabContainer()->addPane($id, $label, $content, $options)`.

- `accordionPane($id, $content, $options)`: Add a new pane to the accordion container with the given `$id`. The given `$content` is shown in this tab pane. To set the title use `$options['title']`.

- `accordionContainer($id, $params, $attribs)`: Render an accordion container with all the currently registered panes. This view helper also offers to add panes with the following

syntax: `$this->accordionContainer()->addPane($id, $label, $content, $options)`.

Example 988. Showing the latest news in a Tab Container

For this example we assume the developer already wrote the controller and model side of the script and assigned an array of news items to the view script. This array contains at most 5 news elements, so we don't have to care about the tab container getting to many tabs.

```
<?php foreach($this->news AS $article): ?>
<?php $this->tabPane("newstab",
                     $article->body,
                     array('title' => $article->title)); ?>
<?php endforeach; ?>
<h2>Latest News</h2>
<?php echo $this->tabContainer("newstab",
                               array(),
                               array('class' => 'flora')); ?>
```

3. ZendX_JQuery Form Elements and Decorators

All View Helpers are pressed into Zend_Form elements or decorators also. They can even be easily integrated into your already existing forms. To enable a Form for Zend_JQuery support you can use two ways: Init your form as `$form = new ZendX_JQuery_Form();` or use the static method `ZendX_JQuery::enableForm($form)` to enable jQuery element support.

3.1. General Elements and Decorator Usage

Both elements and decorators of the Zend jQuery Form set can be initialized with the option key `jQueryParams` to set certain jQuery object related parameters. This jQueryParams array of options matches to the `$params` variable of the corresponding view helpers. For example:

```
$element = new ZendX_JQuery_Form_Element_DatePicker(
                    'dp1',
                    array('jQueryParams' => array('defaultDate' => '2007/10/10'))
                );
// would internally call to:
$view->datePicker("dp1", "", array('defaultDate' => '2007/10/10'), array());
```

Additionally elements jQuery options can be customized by the following methods:

- `setJQueryParam($name, $value)`: Set the jQuery option `$name` to the given value.
- `setJQueryParams($params)`: Set key value pairs of jQuery options and merge them with the already set options.
- `getJQueryParam($name)`: Return the jQuery option with the given name.
- `getJQueryParams()`: Return an array of all currently set jQuery options.

Each jQuery related Decorator also owns a `getJQueryParams()` method, to set options you have to use the `setDecorators()`, `addDecorator()` or `addDecorators()` functionality of a form element and set the jQueryParams key as option:

```
$form->setDecorators(array(
```

```
    'FormElements',
    array('AccordionContainer', array(
        'id'          => 'tabContainer',
        'style'       => 'width: 600px;',
        'jQueryParams' => array(
            'alwaysOpen' => false,
            'animated'   => "easeslide"
        ),
    )),
    'Form'
));
```

3.2. Form Elements

The Zend Framework jQuery Extras Extension comes with the following Form Elements:

- `ZendX_JQuery_Form_Element_AutoComplete`: Proxy to AutoComplete View Helper
- `ZendX_JQuery_Form_Element_ColorPicker`: Proxy to ColorPicker View Helper
- `ZendX_JQuery_Form_Element_DatePicker`: Proxy to DatePicker View Helper
- `ZendX_JQuery_Form_Element_Slider`: Proxy to Slider View Helper
- `ZendX_JQuery_Form_Element_Spinner`: Proxy to Spinner View Helper

jQuery Decorators: Beware the Marker Interface for UiWidgetElements

By default all the jQuery Form elements use the `ZendX_JQuery_Form_Decorator_UiWidgetElement` decorator for rendering the jQuery element with its specific view helper. This decorator is inheritly different from the ViewHelper decorator that is used for most of the default form elements in Zend_Form. To ensure that rendering works correctly for jQuery form elements at least one decorator has to implement the `ZendX_JQuery_Form_Decorator_UiWidgetElementMarker` interface, which the default decorator does. If no marker interface is found an exception is thrown. Use the marker interface if you want to implement your own decorator for the jQuery form element specific rendering.

3.3. Form Decorators

The following Decorators come with the Zend Framework jQuery Extension:

- `ZendX_JQuery_Form_Decorator_AccordionContainer`: Proxy to AccordionContainer View Helper
- `ZendX_JQuery_Form_Decorator_AccordionPane`: Proxy to AccordionPane View Helper
- `ZendX_JQuery_Form_Decorator_DialogContainer`: Proxy to DialogContainer View Helper
- `ZendX_JQuery_Form_Decorator_TabContainer`: Proxy to TabContainer View Helper

- `ZendX_JQuery_Form_Decorator_TabPane`: Proxy to TabPane View Helper
- `ZendX_JQuery_Form_Decorator_UiWidgetElement`: Decorator to Display jQuery Form Elements

Utilizing the Container elements is a bit more complicated, the following example builds a Form with 2 SubForms in a TabContainer:

```
$form = new ZendX_JQuery_Form();
$form->setAction('formdemo.php');
$form->setAttrib('id', 'mainForm');
$form->setAttrib('class', 'flora');

$form->setDecorators(array(
    'FormElements',
    array('TabContainer', array(
        'id'          => 'tabContainer',
        'style'       => 'width: 600px;',
```

Setting the FormId (in this case to 'mainForm') is an important step for the TabContainer. It

```
$subForm1 = new ZendX_JQuery_Form();
$subForm1->setDecorators(array(
    'FormElements',
    array('HtmlTag',
          array('tag' => 'dl')),
    array('TabPane',
          array('jQueryParams' => array('containerId' => 'mainForm',
                                        'title' => 'DatePicker and Slider')))
));

$subForm2 = new ZendX_JQuery_Form();
$subForm2->setDecorators(array(
   'FormElements',
   array('HtmlTag',
         array('tag' => 'dl')),
   array('TabPane',
         array('jQueryParams' => array('containerId' => 'mainForm',
                                       'title' => 'AutoComplete and Spinner')))
));
```

In this stage it is important that the 'containerId' option is set in each SubForm TabPane, or the subforms cannot relate back to the TabContainer.

```
// Add Element Date Picker
$elem = new ZendX_JQuery_Form_Element_DatePicker(
                "datePicker1", array("label" => "Date Picker:")
            );
$elem->setJQueryParam('dateFormat', 'dd.mm.yy');
$subForm1->addElement($elem);

// Add Element Spinner
$elem = new ZendX_JQuery_Form_Element_Spinner(
                "spinner1", array('label' => 'Spinner:')
            );
$elem->setJQueryParams(array('min' => 0, 'max' => 1000, 'start' => 100));
$subForm1->addElement($elem);

// Add Slider Element
$elem = new ZendX_JQuery_Form_Element_Slider(
                "slider1", array('label' => 'Slider:')
            );
$elem->setJQueryParams(array('defaultValue' => '75'));
$subForm2->addElement($elem);

// Add Autocomplete Element
$elem = new ZendX_JQuery_Form_Element_AutoComplete(
                "ac1", array('label' => 'Autocomplete:')
            );
$elem->setJQueryParams(array('data' => array('New York',
                                             'Berlin',
                                             'Bern',
                                             'Boston')));
```

```
$form->addSubForm($subForm1, 'subform1');
$form->addSubForm($subForm2, 'subform2');

$formString = $form->render($view);
```

Appendix A. Zend Framework Requirements

Table of Contents

A.1. Introduction

Zend Framework requires a PHP 5 interpreter with a web server configured to handle PHP scripts correctly. Some features require additional extensions or web server features; in most cases the framework can be used without them, although performance may suffer or ancillary features may not be fully functional. An example of such a dependency is mod_rewrite in an Apache environment, which can be used to implement "pretty URL's" like "`http://www.example.com/user/edit`". If mod_rewrite is not enabled, Zend Framework can be configured to support URL's such as "**http://www.example.com?controller=user&action=edit**". Pretty URL's may be used to shorten URL's for textual representation or search engine optimization (SEO), but they do not directly affect the functionality of the application.

A.1.1. PHP Version

Zend recommends the most current release of PHP for critical security and performance enhancements, and currently supports PHP 5.2.4 or later.

Zend Framework has an extensive collection of unit tests, which you can run using PHPUnit 3.3.0 or later.

A.1.2. PHP Extensions

You will find a table listing all extensions typically found in PHP and how they are used in Zend Framework below. You should verify that the extensions on which Zend Framework components you'll be using in your application are available in your PHP environments. Many applications will not require every extension listed below.

A dependency of type "hard" indicates that the components or classes cannot function properly if the respective extension is not available, while a dependency of type "soft" indicates that the component may use the extension if it is available but will function properly if it is not. Many components will automatically use certain extensions if they are available to optimize performance but will execute code with similar functionality in the component itself if the extensions are unavailable.

Table A.1. PHP Extensions Used in Zend Framework by Component

PHP Extension	Dependency Type	Used by Zend Framework Components
apc	Hard	`Zend_Cache_Backend_Apc`

PHP Extension	Dependency Type	Used by Zend Framework Components
	Soft	Zend_File_Transfer
bcmath	Soft	Zend_Locale
bitset	Soft	Zend_Search_Lucene
bz2	---	---
calendar	---	---
com_dotnet	---	---
ctype	Hard	Zend_Auth_Adapter_Http
		Zend_Gdata
		Zend_Http_Client
		Zend_Pdf
		Zend_Rest_Client
		Zend_Rest_Server
		Zend_Search_Lucene
		Zend_Uri
		Zend_Validate
curl	Hard	Zend_Http_Client_Adapter_Curl
date	Soft	Zend_Amf
dba	---	---
dbase	---	---
dom	Hard	Zend_Amf
		Zend_Dom
		Zend_Feed
		Zend_Gdata
		Zend_Log_Formatter_Xml
		Zend_Rest_Server
		Zend_Soap
		Zend_Search_Lucene
		Zend_Service_Amazon
		Zend_Service_Delicious
		Zend_Service_Flickr
		Zend_Service_Simpy
		Zend_Service_Yahoo
		Zend_XmlRpc
exif	---	---
fbsql	---	---
fdf	---	---
filter	---	---

PHP Extension	Dependency Type	Used by Zend Framework Components
ftp	---	---
gd	Hard	`Zend_Captcha`
		`Zend_Pdf`
gettext	---	---
gmp	---	---
hash	Hard	`Zend_Auth_Adapter_Http`
ibm_db2	Hard	`Zend_Db_Adapter_Db2`
iconv	Hard	`Zend_Currency`
		`Zend_Locale_Format`
		`Zend_Mime`
		`Zend_Pdf`
		`Zend_Search_Lucene`
		`Zend_Service_Audioscrobbler`
		`Zend_Service_Flickr`
		`Zend_XmlRpc_Client`
igbinary	Hard	`Zend_Serializer_Adapter_Igbinary`
imap	---	---
informix	---	---
interbase	Hard	`Zend_Db_Adapter_Firebird`
json	Soft	*`Zend_Json`*
		`Zend_Serializer_Adapter_Json`
ldap	Hard	`Zend_Ldap`
libxml	---	---
mbstring	Hard	`Zend_Feed`
mcrypt	Hard	`Zend_Service_ReCaptcha`
memcache	Hard	`Zend_Cache_Backend_Memcached`
mhash	---	---
mime_magic	Hard	`Zend_Http_Client`
ming	---	---
msql	---	---
mssql	---	---
mysql	---	---
mysqli	Hard	`Zend_Db_Adapter_Mysqli`
ncurses	---	---
oci8	Hard	`Zend_Db_Adapter_Oracle`
odbc	---	---
openssl	---	---

PHP Extension	Dependency Type	Used by Zend Framework Components
pcntl	---	---
pcre	Hard	Virtually all components
pdo	Hard	All PDO database adapters
pdo_dblib	---	---
pdo_firebird	---	---
pdo_mssql	Hard	`Zend_Db_Adapter_Pdo_Mssql`
pdo_mysql	Hard	`Zend_Db_Adapter_Pdo_Mysql`
pdo_oci	Hard	`Zend_Db_Adapter_Pdo_Oci`
pdo_pgsql	Hard	`Zend_Db_Adapter_Pdo_Pgsql`
pdo_sqlite	Hard	`Zend_Db_Adapter_Pdo_Sqlite`
pgsql	---	---
posix	Soft	`Zend_Mail`
pspell	---	---
readline	---	---
recode	---	---
Reflection	Hard	`Zend_Controller`
		`Zend_Filter`
		`Zend_Filter_Input`
		`Zend_Json`
		`Zend_Log`
		`Zend_Rest_Server`
		`Zend_Server_Reflection`
		`Zend_Validate`
		`Zend_View`
		`Zend_XmlRpc_Server`
session	Hard	`Zend_Controller_Action_Helper_Redir`
		`Zend_Session`
shmop	---	---
SimpleXML	Hard	`Zend_Config_Xml`
		`Zend_Feed`
		`Zend_Rest_Client`
		`Zend_Serializer_Adapter_Wddx`
		`Zend_Service_Audioscrobbler`
		`Zend_Soap`
		`Zend_XmlRpc`
	Soft	`Zend_Amf`
soap	Hard	`Zend_Service_StrikeIron`

PHP Extension	Dependency Type	Used by Zend Framework Components
		Zend_Soap
sockets	---	---
SPL	Hard	Virtually all components
SQLite	Hard	Zend_Cache_Backend_Sqlite
standard	Hard	Virtually all components
sybase	---	---
sysvmsg	---	---
sysvsem	---	--
sysvshm	---	---
tidy	---	---
tokenizer	---	---
wddx	Hard	Zend_Serializer_Adapter_Wddx
xml	Hard	Zend_Translate_Adapter_Qt
		Zend_Translate_Adapter_Tmx
		Zend_Translate_Adapter_Xliff
XMLReader	---	---
xmlrpc	---	---
XMLWriter	---	---
xsl	---	---
zip	---	---
zlib	Hard	Zend_Pdf
		Zend_Filter_Compress

A.1.3. Zend Framework Components

Below is a table that lists all available Zend Framework Components and which PHP extension they need. This can help guide you to know which extensions are required for your application. Not all extensions used by Zend Framework are required for every application.

A dependency of type "hard" indicates that the components or classes cannot function properly if the respective extension is not available, while a dependency of type "soft" indicates that the component may use the extension if it is available but will function properly if it is not. Many components will automatically use certain extensions if they are available to optimize performance but will execute code with similar functionality in the component itself if the extensions are unavailable.

Table A.2. Zend Framework Components and the PHP Extensions they use

Zend Framework Components and the PHP Extensions they use	Dependency Type	PHP Extension
All Components	Hard	pcre
		SPL

Zend Framework Components and the PHP Extensions they use	Dependency Type	PHP Extension
		standard
Zend_Acl	---	---
Zend_Amf	Hard	date
	Soft	dom
		SimpleXML
Zend_Auth	Hard	ctype
		hash
Zend_Cache	Hard	apc
		memcache
		sqlite
		zlib
Zend_Captcha	Hard	gd
Zend_Config	Hard	libxml
		SimpleXML
Zend_Console_Getopt	---	---
Zend_Controller	Hard	Reflection
		session
Zend_Currency	Hard	iconv
Zend_Date	---	---
Zend_Db	Hard	ibm_db2
		mysqli
		oci8
		pdo
		pdo_mssql
		pdo_mysql
		pdo_oci
		pdo_pgsql
		pdo_sqlite
Zend_Debug	---	---
Zend_Dojo	---	---
Zend_Dom	Hard	dom
Zend_Exception	---	---
Zend_Feed	Hard	dom
		libxml
		mbstring
		SimpleXML

Zend Framework Components and the PHP Extensions they use	Dependency Type	PHP Extension
Zend_File_Transfer	Soft	apc
		upload_extension
Zend_Filter	Hard	Reflection
	Soft	zlib
Zend_Form	---	---
Zend_Gdata	Hard	ctype
		dom
		libxml
Zend_Http	Hard	ctype
		curl
		mime_magic
Zend_InfoCard	---	---
Zend_Json	Soft	json
	Hard	Reflection
Zend_Layout	---	---
Zend_Ldap	---	ldap
Zend_Loader	---	---
Zend_Locale	Soft	bcmath
	Hard	iconv
Zend_Log	Hard	dom
		libxml
		Reflection
Zend_Mail	Soft	posix
Zend_Measure	---	---
Zend_Memory	---	---
Zend_Mime	Hard	iconv
Zend_OpenId	---	---
Zend_Paginator	---	---
Zend_Pdf	Hard	ctype
		gd
		iconv
		zlib
Zend_ProgressBar	---	---
Zend_Registry	---	---
Zend_Request	---	---
Zend_Rest	Hard	ctype

Zend Framework Components and the PHP Extensions they use	Dependency Type	PHP Extension
		dom
		libxml
		Reflection
		SimpleXML
Zend_Search_Lucene	Soft	bitset
	Hard	ctype
		dom
		iconv
		libxml
Zend_Serializer	Hard	wddx
		SimpleXml
		igbinary
	Soft	*json*
Zend_Server_Reflection	Hard	Reflection
Zend_Service_Akismet	---	---
Zend_Service_Amazon	Hard	dom
		libxml
Zend_Service_Audioscrobbler	Hard	iconv
		libxml
		SimpleXML
Zend_Service_Delicious	Hard	dom
		libxml
Zend_Service_Flickr	Hard	dom
		iconv
		libxml
Zend_Service_Nirvanix	---	---
Zend_Service_ReCaptcha	Hard	*mcrypt*
Zend_Service_Simpy	Hard	dom
		libxml
Zend_Service_SlideShare	---	---
Zend_Service_StrikeIron	Hard	soap
Zend_Service_Technorati	---	---
Zend_Service_Twitter	---	---
Zend_Service_Yahoo	Hard	dom
		libxml
Zend_Session	Hard	session

Zend Framework Components and the PHP Extensions they use	Dependency Type	PHP Extension
Zend_Soap	Hard	*dom*
		SimpleXML
		soap
Zend_Test	---	---
Zend_Text	---	---
Zend_TimeSync	---	---
Zend_Translate	Hard	xml
Zend_Uri	Hard	ctype
Zend_Validate	Hard	ctype
		Reflection
Zend_Version	---	---
Zend_Validate	Hard	Reflection
Zend_Wildfire	---	---
Zend_XmlRpc	Hard	dom
		iconv
		libxml
		Reflection
		SimpleXML

A.1.4. Zend Framework Dependencies

Below you can find a table listing Zend Framework Components and their dependencies to other Zend Framework Components. This can help you if you need to have only single components instead of the complete Zend Framework.

A dependency of type "hard" indicates that the components or classes cannot function properly if the respective dependent component is not available, while a dependency of type "soft" indicates that the component may need the dependent component in special situations or with special adapters. At last a dependency of type "fix" indicated that these components or classes are in any case used by subcomponents, and a dependency of type "sub" indicates that these components can be used by subcomponents in special situations or with special adapters.

Even if it's possible to separate single components for usage from the complete Zend Framework you should keep in mind that this can lead to problems when files are missed or components are used dynamically.

Table A.3. Zend Framework Components and their dependency to other Zend Framework Components

Zend Framework Component	Dependency Type	Dependent Zend Framework Component
Zend_Acl	Hard	Zend_Exception
Zend_Amf	Hard	Zend_Exception

Zend Framework Component	Dependency Type	Dependent Zend Framework Component
		Zend_Server
	Soft	Zend_Date
		Zend_Loader
	Sub	Zend_Locale
		Zend_Registry
Zend_Auth	Hard	Zend_Exception
	Soft	Zend_Db
		Zend_InfoCard
		Zend_Ldap
		Zend_OpenId
		Zend_Session
	Fix	Zend_Controller
		Zend_Http
		Zend_Loader
		Zend_Locale
		Zend_Uri
		Zend_View
	Sub	Zend_Captcha
		Zend_Config
		Zend_Date
		Zend_Dojo
		Zend_Filter
		Zend_Form
		Zend_Json
		Zend_Layout
		Zend_Registry
		Zend_Server
		Zend_Service_ReCaptcha
		Zend_Text
		Zend_Validate
		Zend_Wildfire
Zend_Cache	Hard	Zend_Exception
	Soft	Zend_Log
	Sub	Zend_Captcha
		Zend_Config
		Zend_Controller
		Zend_Date

Zend Framework Component	Dependency Type	Dependent Zend Framework Component
		Zend_Db
		Zend_Dojo
		Zend_Filter
		Zend_Form
		Zend_Http
		Zend_Json
		Zend_Layout
		Zend_Loader
		Zend_Locale
		Zend_Registry
		Zend_Server
		Zend_Service_ReCaptcha
		Zend_Session
		Zend_Text
		Zend_Uri
		Zend_Validate
		Zend_View
		Zend_Wildfire
Zend_Captcha	Hard	Zend_Exception
		Zend_Service_ReCaptcha
		Zend_Text
		Zend_Validate
	Fix	Zend_Http
		Zend_Json
		Zend_Loader
		Zend_Locale
		Zend_Server
		Zend_Uri
	Sub	Zend_Date
		Zend_Filter
		Zend_ReLoader
Zend_Config	Hard	Zend_Exception
Zend_Console_Getopt	Hard	Zend_Exception
	Soft	Zend_Json
	Sub	Zend_Loader
		Zend_Server
Zend_Controller	Hard	Zend_Config

Zend Framework Component	Dependency Type	Dependent Zend Framework Component
		Zend_Exception
		Zend_Loader
		Zend_Registry
		Zend_Uri
		Zend_View
	Soft	Zend_Dojo
		Zend_Filter
		Zend_Json
		Zend_Layout
	Fix	Zend_Locale
		Zend_Validate
	Sub	Zend_Captcha
		Zend_Date
		Zend_Db
		Zend_Form
		Zend_Http
		Zend_Server
		Zend_Service_ReCaptcha
		Zend_Session
		Zend_Text
		Zend_Wildfire
Zend_Currency	Hard	Zend_Exception
		Zend_Locale
	Sub	Zend_Loader
		Zend_Registry
Zend_Date	Hard	Zend_Exception
		Zend_Locale
	Sub	Zend_Loader
		Zend_Registry
Zend_Db	Hard	Zend_Exception
		Zend_Loader
	Soft	Zend_Registry
		Zend_Wildfire
	Sub	Zend_Captcha
		Zend_Config
		Zend_Controller
		Zend_Date

Zend Framework Component	Dependency Type	Dependent Zend Framework Component
		Zend_Db
		Zend_Dojo
		Zend_Filter
		Zend_Form
		Zend_Http
		Zend_Json
		Zend_Layout
		Zend_Server
		Zend_Service_ReCaptcha
		Zend_Session
		Zend_Text
		Zend_Uri
		Zend_Validate
		Zend_View
Zend_Debug	---	---
Zend_Dojo	Hard	Zend_Exception
		Zend_Form
		Zend_Json
		Zend_Registry
		Zend_View
	Soft	Zend_Filter
	Fix	Zend_Config
		Zend_Loader
		Zend_Locale
		Zend_Uri
		Zend_Validate
	Sub	Zend_Captcha
		Zend_Controller
		Zend_Date
		Zend_Db
		Zend_Dojo
		Zend_Http
		Zend_Layout
		Zend_Server
		Zend_Service_ReCaptcha
		Zend_Session
		Zend_Text

Zend Framework Component	Dependency Type	Dependent Zend Framework Component
		Zend_Wildfire
Zend_Dom	Hard	Zend_Exception
Zend_Exception	---	---
Zend_Feed	Hard	Zend_Exception
		Zend_Loader
		Zend_Uri
	Fix	Zend_Locale
		Zend_Validate
	Sub	Zend_Date
		Zend_Filter
		Zend_Http
		Zend_Registry
Zend_File_Transfer	Hard	Zend_Exception
	Soft	Zend_Loader
Zend_Filter	Hard	Zend_Exception
		Zend_Loader
		Zend_Validate
	Soft	Zend_Locale
	Sub	Zend_Date
		Zend_Registry
Zend_Form	Hard	Zend_Exception
		Zend_Filter
		Zend_Validate
	Soft	Zend_Captcha
		Zend_Controller
		Zend_Json
		Zend_Loader
		Zend_Registry
		Zend_Session
	Fix	Zend_Config
		Zend_Http
		Zend_Locale
		Zend_Server
		Zend_Service_ReCaptcha
		Zend_Text
		Zend_Uri
		Zend_View

Zend Framework Component	Dependency Type	Dependent Zend Framework Component
	Sub	Zend_Date
		Zend_Db
		Zend_Dojo
		Zend_Form
		Zend_Layout
		Zend_Wildfire
Zend_Gdata	Hard	Zend_Exception
		Zend_Http
		Zend_Mime
		Zend_Version
	Soft	Zend_Loader
	Fix	Zend_Locale
		Zend_Uri
		Zend_Validate
	Sub	Zend_Date
		Zend_Filter
		Zend_Registry
Zend_Http	Hard	Zend_Exception
		Zend_Loader
		Zend_Uri
	Fix	Zend_Locale
		Zend_Validate
	Sub	Zend_Date
		Zend_Filter
		Zend_Registry
Zend_InfoCard	Hard	Zend_Exception
		Zend_Loader
Zend_Json	Hard	Zend_Exception
		Zend_Loader
		Zend_Server
Zend_Layout	Hard	Zend_Exception
	Soft	Zend_Controller
		Zend_Filter
		Zend_Loader
		Zend_View
	Fix	Zend_Config
		Zend_Layout

Zend Framework Component	Dependency Type	Dependent Zend Framework Component
		Zend_Registry
		Zend_Uri
		Zend_Validate
	Sub	Zend_Captcha
		Zend_Date
		Zend_Db
		Zend_Dojo
		Zend_Form
		Zend_Http
		Zend_Json
		Zend_Locale
		Zend_Server
		Zend_Service_ReCaptcha
		Zend_Session
		Zend_Text
		Zend_Wildfire
Zend_Ldap	Hard	Zend_Exception
Zend_Loader	Hard	Zend_Exception
Zend_Locale	Hard	Zend_Exception
	Soft	Zend_Registry
	Sub	Zend_Loader
Zend_Log	Hard	Zend_Exception
	Soft	Zend_Wildfire
	Sub	Zend_Captcha
		Zend_Config
		Zend_Controller
		Zend_Date
		Zend_Db
		Zend_Dojo
		Zend_Filter
		Zend_Form
		Zend_Http
		Zend_Json
		Zend_Layout
		Zend_Loader
		Zend_Registry
		Zend_Server

Zend Framework Component	Dependency Type	Dependent Zend Framework Component
		Zend_Service_ReCaptcha
		Zend_Session
		Zend_Text
		Zend_Uri
		Zend_Validate
		Zend_View
Zend_Mail	Hard	Zend_Exception
		Zend_Loader
		Zend_Mime
		Zend_Validate
	Fix	Zend_Locale
	Sub	Zend_Date
		Zend_Filter
		Zend_Registry
Zend_Measure	Hard	Zend_Exception
		Zend_Locale
		Zend_Registry
	Sub	Zend_Loader
Zend_Memory	Hard	Zend_Cache
		Zend_Exception
	Sub	Zend_Captcha
		Zend_Config
		Zend_Controller
		Zend_Date
		Zend_Db
		Zend_Dojo
		Zend_Filter
		Zend_Form
		Zend_Http
		Zend_Json
		Zend_Layout
		Zend_Loader
		Zend_Locale
		Zend_Log
		Zend_Registry
		Zend_Server
		Zend_Service_ReCaptcha

Zend Framework Component	Dependency Type	Dependent Zend Framework Component
		Zend_Session
		Zend_Text
		Zend_Uri
		Zend_Validate
		Zend_View
		Zend_Wildfire
Zend_Mime	Hard	Zend_Exception
Zend_OpenId	Hard	Zend_Controller
		Zend_Exception
		Zend_Http
		Zend_Session
	Fix	Zend_Config
		Zend_Dojo
		Zend_Loader
		Zend_Locale
		Zend_Registry
		Zend_Uri
		Zend_Validate
		Zend_View
	Sub	Zend_Captcha
		Zend_Date
		Zend_Db
		Zend_Filter
		Zend_Form
		Zend_Json
		Zend_Layout
		Zend_Server
		Zend_Service_ReCaptcha
		Zend_Text
		Zend_Wildfire
Zend_Paginator	Hard	Zend_Exception
		Zend_Json
		Zend_Loader
	Soft	Zend_Controller
		Zend_Db
		Zend_View
	Fix	Zend_Server

Zend Framework Component	Dependency Type	Dependent Zend Framework Component
	Sub	Zend_Captcha
		Zend_Config
		Zend_Date
		Zend_Dojo
		Zend_Filter
		Zend_Form
		Zend_Http
		Zend_Layout
		Zend_Locale
		Zend_Registry
		Zend_Service_ReCaptcha
		Zend_Session
		Zend_Text
		Zend_Uri
		Zend_Validate
		Zend_Wildfire
Zend_Pdf	Hard	Zend_Exception
		Zend_Log
		Zend_Memory
	Fix	Zend_Cache
	Sub	Zend_Captcha
		Zend_Config
		Zend_Controller
		Zend_Date
		Zend_Db
		Zend_Dojo
		Zend_Filter
		Zend_Form
		Zend_Http
		Zend_Json
		Zend_Layout
		Zend_Loader
		Zend_Locale
		Zend_Registry
		Zend_Server
		Zend_Service_ReCaptcha
		Zend_Session

Zend Framework Component	Dependency Type	Dependent Zend Framework Component
		Zend_Text
		Zend_Uri
		Zend_Validate
		Zend_View
		Zend_Wildfire
Zend_Progressbar	Hard	Zend_Config
		Zend_Exception
		Zend_Json
	Soft	Zend_Session
	Fix	Zend_Db
		Zend_Loader
		Zend_Server
	Sub	Zend_Captcha
		Zend_Date
		Zend_Dojo
		Zend_Filter
		Zend_Form
		Zend_Http
		Zend_Layout
		Zend_Registry
		Zend_Service_ReCaptcha
		Zend_Text
		Zend_Uri
		Zend_Validate
		Zend_View
		Zend_Wildfire
Zend_Registry	Hard	Zend_Exception
	Soft	Zend_Loader
Zend_Request	---	---
Zend_Rest	Hard	Zend_Exception
		Zend_Server
		Zend_Service
		Zend_Uri
	Fix	Zend_Http
		Zend_Loader
		Zend_Locale
		Zend_Validate

Zend Framework Component	Dependency Type	Dependent Zend Framework Component
	Sub	Zend_Date
		Zend_Filter
		Zend_Registry
Zend_Search_Lucene	Hard	Zend_Exception
Zend_Serializer	Hard	Zend_Exception
		Zend_Loader
	Soft	Zend_Json
		Zend_Amf
Zend_Server	Hard	Zend_Exception
Zend_Service_Akismet	Hard	Zend_Exception
		Zend_Http
		Zend_Uri
		Zend_Version
	Fix	Zend_Loader
		Zend_Locale
		Zend_Validate
	Sub	Zend_Date
		Zend_Filter
		Zend_Registry
Zend_Service_Amazon	Hard	Zend_Exception
		Zend_Http
		Zend_Rest
	Fix	Zend_Loader
		Zend_Locale
		Zend_Server
		Zend_Service
		Zend_Uri
		Zend_Validate
	Sub	Zend_Date
		Zend_Filter
		Zend_Registry
Zend_Service_Audioscrobbler	Hard	Zend_Exception
		Zend_Http
	Fix	Zend_Loader
		Zend_Locale
		Zend_Uri
		Zend_Validate

Zend Framework Component	Dependency Type	Dependent Zend Framework Component
	Sub	Zend_Date
		Zend_Filter
		Zend_Registry
Zend_Service_Delicious	Hard	Zend_Date
		Zend_Exception
		Zend_Http
		Zend_Json
		Zend_Rest
	Fix	Zend_Loader
		Zend_Locale
		Zend_Server
		Zend_Service
		Zend_Uri
		Zend_Validate
	Sub	Zend_Filter
		Zend_Registry
Zend_Service_Flickr	Hard	Zend_Exception
		Zend_Http
	Soft	Zend_Rest
		Zend_Validate
	Fix	Zend_Loader
		Zend_Locale
		Zend_Server
		Zend_Service
		Zend_Uri
	Sub	Zend_Date
		Zend_Filter
		Zend_Registry
Zend_Service_Nirvanix	Hard	Zend_Exception
		Zend_Http
		Zend_Loader
	Fix	Zend_Locale
		Zend_Uri
		Zend_Validate
	Sub	Zend_Date
		Zend_Filter
		Zend_Registry

Zend Framework Component	Dependency Type	Dependent Zend Framework Component
Zend_Service_ReCaptcha	Hard	Zend_Exception
		Zend_Http
		Zend_Json
	Fix	Zend_Loader
		Zend_Locale
		Zend_Server
		Zend_Uri
		Zend_Validate
	Sub	Zend_Date
		Zend_Filter
		Zend_Registry
Zend_Service_Simpy	Hard	Zend_Exception
		Zend_Http
		Zend_Rest
	Fix	Zend_Loader
		Zend_Locale
		Zend_Server
		Zend_Service
		Zend_Uri
		Zend_Validate
	Sub	Zend_Date
		Zend_Filter
		Zend_Registry
Zend_Service_SlideShare	Hard	Zend_Cache
		Zend_Exception
		Zend_Http
	Fix	Zend_Loader
		Zend_Locale
		Zend_Uri
		Zend_Validate
	Sub	Zend_Captcha
		Zend_Config
		Zend_Controller
		Zend_Date
		Zend_Db
		Zend_Dojo
		Zend_Filter

Zend Framework Component	Dependency Type	Dependent Zend Framework Component
		Zend_Form
		Zend_Json
		Zend_Layout
		Zend_Log
		Zend_Registry
		Zend_Server
		Zend_Service_ReCaptcha
		Zend_Session
		Zend_Text
		Zend_View
		Zend_Wildfire
Zend_Service_StrikeIron	Hard	Zend_Exception
		Zend_Http
		Zend_Loader
	Fix	Zend_Locale
		Zend_Uri
		Zend_Validate
	Fix	Zend_Date
		Zend_Filter
		Zend_Registry
Zend_Service_Technorati	Hard	Zend_Date
		Zend_Exception
		Zend_Http
		Zend_Uri
		Zend_Locale
	Soft	Zend_Rest
	Fix	Zend_Loader
		Zend_Server
		Zend_Service
		Zend_Validate
	Sub	Zend_Filter
		Zend_Registry
Zend_Service_Twitter	Hard	Zend_Exception
		Zend_Feed
		Zend_Http
		Zend_Json
		Zend_Rest

Zend Framework Component	Dependency Type	Dependent Zend Framework Component
		Zend_Uri
	Fix	Zend_Loader
		Zend_Locale
		Zend_Server
		Zend_Service
		Zend_Validate
	Fix	Zend_Date
		Zend_Filter
		Zend_Registry
Zend_Service_Yahoo	Hard	Zend_Exception
		Zend_Http
		Zend_Rest
	Soft	Zend_Validate
	Fix	Zend_Loader
		Zend_Locale
		Zend_Server
		Zend_Service
		Zend_Uri
	Sub	Zend_Date
		Zend_Filter
		Zend_Registry
Zend_Session	Hard	Zend_Exception
	Soft	Zend_Config
		Zend_Db
		Zend_Loader
	Sub	Zend_Captcha
		Zend_Date
		Zend_Dojo
		Zend_Filter
		Zend_Form
		Zend_Http
		Zend_Json
		Zend_Layout
		Zend_Registry
		Zend_Server
		Zend_Service_ReCaptcha
		Zend_Session

Zend Framework Component	Dependency Type	Dependent Zend Framework Component
		Zend_Text
		Zend_Uri
		Zend_Validate
		Zend_View
		Zend_Wildfire
Zend_Soap	Hard	Zend_Exception
		Zend_Server
		Zend_Uri
	Fix	Zend_Loader
		Zend_Locale
		Zend_Validate
	Sub	Zend_Date
		Zend_Filter
		Zend_Registry
Zend_Test	Hard	Zend_Controller
		Zend_Dom
		Zend_Exception
		Zend_Layout
		Zend_Registry
		Zend_Session
	Soft	Zend_Loader
	Fix	Zend_Config
		Zend_Locale
		Zend_Uri
		Zend_Validate
		Zend_View
	Sub	Zend_Captcha
		Zend_Date
		Zend_Db
		Zend_Dojo
		Zend_Filter
		Zend_Form
		Zend_Http
		Zend_Json
		Zend_Server
		Zend_Service_ReCaptcha
		Zend_Text

Zend Framework Component	Dependency Type	Dependent Zend Framework Component
		Zend_Wildfire
Zend_Text	Hard	Zend_Exception
	Soft	Zend_Loader
Zend_TimeSync	Hard	Zend_Date
		Zend_Exception
		Zend_Loader
	Fix	Zend_Locale
	Sub	Zend_Registry
Zend_Translate	Hard	Zend_Exception
		Zend_Loader
		Zend_Locale
	Sub	Zend_Registry
Zend_Uri	Hard	Zend_Exception
		Zend_Loader
		Zend_Locale
		Zend_Validate
	Soft	Zend_Date
		Zend_Filter
		Zend_Registry
Zend_Validate	Hard	Zend_Exception
		Zend_Loader
		Zend_Locale
	Soft	Zend_Date
		Zend_Filter
		Zend_Registry
Zend_Version	---	---
Zend_View	Hard	Zend_Controller
		Zend_Exception
		Zend_Loader
		Zend_Locale
		Zend_Registry
	Soft	Zend_Json
		Zend_Layout
	Fix	Zend_Config
		Zend_Uri
		Zend_Validate
	Sub	Zend_Captcha

Zend Framework Component	Dependency Type	Dependent Zend Framework Component
		Zend_Date
		Zend_Db
		Zend_Dojo
		Zend_Filter
		Zend_Form
		Zend_Http
		Zend_Server
		Zend_Service_ReCaptcha
		Zend_Session
		Zend_Text
		Zend_Wildfire
Zend_Wildfire	Hard	Zend_Controller
		Zend_Exception
		Zend_Json
		Zend_Loader
	Fix	Zend_Config
		Zend_Layout
		Zend_Registry
		Zend_Server
		Zend_Uri
		Zend_Validate
		Zend_View
	Sub	Zend_Captcha
		Zend_Date
		Zend_Db
		Zend_Dojo
		Zend_Filter
		Zend_Form
		Zend_Http
		Zend_Layout
		Zend_Service_ReCaptcha
		Zend_Session
		Zend_Text
Zend_XmlRpc	Hard	Zend_Exception
		Zend_Http
		Zend_Server
	Fix	Zend_Loader

Zend Framework Component	Dependency Type	Dependent Zend Framework Component
		Zend_Uri
		Zend_Validate
		Zend_Locale
	Sub	Zend_Date
		Zend_Filter
		Zend_Registry

Appendix B. Zend Framework Migration Notes

Table of Contents

B.1. Zend Framework 1.10

When upgrading from a previous release to Zend Framework 1.10 or higher you should note the following migration notes.

B.1.1. Zend_Controller_Front

A wrong behaviour was fixed, when there was no module route and no route matched the given request. Previously, the router returned an unmodified request object, so the front controller just displayed the default controller and action. Since Zend Framework 1.10, the router will correctly as noted in the router interface, throw an exception if no route matches. The error plugin will then catch that exception and forward to the error controller. You can then test for that specific error with the constant :

```
/**
 * Before 1.10
 */
    public function errorAction()
    {
        $errors = $this->_getParam('error_handler');

        switch ($errors->type) {
            case Zend_Controller_Plugin_ErrorHandler::EXCEPTION_NO_CONTROLLER:
            case Zend_Controller_Plugin_ErrorHandler::EXCEPTION_NO_ACTION:
    // ...

/**
 * With 1.10
 */
    public function errorAction()
    {
        $errors = $this->_getParam('error_handler');

        switch ($errors->type) {
            case Zend_Controller_Plugin_ErrorHandler::EXCEPTION_NO_ROUTE:
            case Zend_Controller_Plugin_ErrorHandler::EXCEPTION_NO_CONTROLLER:
            case Zend_Controller_Plugin_ErrorHandler::EXCEPTION_NO_ACTION:
    // ...
```

B.1.2. Zend_Feed_Reader

With the introduction of Zend Framework 1.10, `Zend_Feed_Reader`'s handling of retrieving Authors and Contributors was changed, introducing a break in backwards compatibility. This change was an effort to harmonise the treatment of such data across the RSS and Atom classes of the component and enable the return of Author and Contributor data in more accessible, usable and detailed form. It also rectifies an error in that it was assumed any author element referred to a name. In RSS this is incorrect as an author element is actually only required to provide an email address. In addition, the original implementation applied its RSS limits to Atom feeds significantly reducing the usefulness of the parser with that format.

The change means that methods like `getAuthors()` and `getContributors` no longer return a simple array of strings parsed from the relevant RSS and Atom elements. Instead, the return value is an `ArrayObject` subclass called `Zend_Feed_Reader_Collection_Author` which simulates an iterable multidimensional array of Authors. Each member of this object will be a simple array with three potential keys (as the source data permits). These include: name, email and uri.

The original behaviour of such methods would have returned a simple array of strings, each string attempting to present a single name, but in reality this was unreliable since there is no rule governing the format of RSS Author strings.

The simplest method of simulating the original behaviour of these methods is to use the `Zend_Feed_Reader_Collection_Author`'s `getValues()` which also returns a simple array of strings representing the "most relevant data", for authors presumed to be their name. Each value in the resulting array is derived from the "name" value attached to each Author (if present). In most cases this simple change is easy to apply as demonstrated below.

```
/**
 * Before 1.10
 */
$feed = Zend_Feed_Reader::import('http://example.com/feed');
$authors = $feed->getAuthors();

/**
 * With 1.10
 */
$feed = Zend_Feed_Reader::import('http://example.com/feed');
$authors = $feed->getAuthors()->getValues();
```

B.1.3. Zend_File_Transfer

B.1.3.1. Security change

For security reasons `Zend_File_Transfer` does no longer store the original mimetype and filesize which is given from the requesting client into its internal storage. Instead the real values will be detected at initiation.

Additionally the original values within `$_FILES` will be overridden within the real values at initiation. This makes also `$_FILES` secure.

When you are in need of the original values you can eighter store them before initiating `Zend_File_Transfer` or use the disableInfos option at initiation. Note that this option is useless when its given after initiation.

B.1.3.2. Count validation

Before release 1.10 the `MimeType` validator used a wrong naming. For consistency the following constants have been changed:

Table B.1. Changed Validation Messages

Old	New	Value	
`TOO_MUCH`	`TOO_MANY`	Too many files, maximum '%max%' are allowed but '%count%' are given	
`TOO_LESS`	`TOO_FEW`	Too few files, minimum '%min%' are expected but '%count%' are given	

When you are translating these messages within your code then use the new constants. As benefit you don't need to translate the original string anymore to get a correct spelling.

B.1.4. Zend_Filter_HtmlEntities

In order to default to a more secure character encoding, `Zend_Filter_HtmlEntities` now defaults to UTF-8 instead of ISO-8859-1.

Additionally, because the actual mechanism is dealing with character encodings and not character sets, two new methods have been added, `setEncoding()` and `getEncoding()`. The previous methods `setCharSet()` and `setCharSet()` are now deprecated and proxy to the new methods. Finally, instead of using the protected members directly within the `filter()` method, these members are retrieved by their explicit accessors. If you were extending the filter in the past, please check your code and unit tests to ensure everything still continues to work.

B.1.5. Zend_Filter_StripTags

`Zend_Filter_StripTags` contains a flag, `commentsAllowed`, that, in previous versions, allowed you to optionally whitelist HTML comments in HTML text filtered by the class. However, this opens code enabling the flag to XSS attacks, particularly in Internet Explorer (which allows specifying conditional functionality via HTML comments). Starting in version 1.9.7 (and backported to versions 1.8.5 and 1.7.9), the `commentsAllowed` flag no longer has any meaning, and all HTML comments, including those containing other HTML tags or nested commments, will be stripped from the final output of the filter.

B.1.6. Zend_Translate

B.1.6.1. Xliff adapter

In past the Xliff adapter used the source string as message Id. According to the Xliff standard the trans-unit Id should be used. This behaviour was corrected with Zend Framework 1.10. Now the trans-unit Id is used as message Id per default.

But you can still get the incorrect and old behaviour by setting the useId option to FALSE.

```
$trans = new Zend_Translate('xliff', '/path/to/source', $locale, array('useId' => false));
```

B.1.7. Zend_Validate

B.1.7.1. Self written validators

When setting returning a error from within a self written validator you have to call the `_error()` method. Before Zend Framework 1.10 you were able to call this method without giving a parameter. It used then the first found message template.

This behaviour is problematic when you have validators with more than one different message to be returned. Also when you extend an existing validator you can get unexpected results. This could lead to the problem that your user get not the message you expected.

```
My_Validator extends Zend_Validate_Abstract
{
    public isValid($value)
    {
        ...
        $this->_error(); // unexpected results between different OS
        ...
    }
}
```

To prevent this problem the `_error()` method is no longer allowed to be called without giving a parameter.

```
My_Validator extends Zend_Validate_Abstract
{
    public isValid($value)
    {
        ...
        $this->_error(self::MY_ERROR); // defined error, no unexpected results
        ...
    }
}
```

B.1.7.2. Simplification in date validator

Before Zend Framework 1.10 2 identical messages were thrown within the date validator. These were `NOT_YYYY_MM_DD` and `FALSEFORMAT`. As of Zend Framework 1.10 only the `FALSEFORMAT` message will be returned when the given date does not match the set format.

B.1.7.3. Fixes in Alpha, Alnum and Barcode validator

Before Zend Framework 1.10 the messages within the 2 barcode adapters, the Alpha and the Alnum validator were identical. This introduced problems when using custom messages, translations or multiple instances of these validators.

As with Zend Framework 1.10 the values of the constants were changed to be unique. When you used the constants as proposed in the manual there is no change for you. But when you used the content of the constants in your code then you will have to change them. The following table shows you the changed values:

Table B.2. Available Validation Messages

Validator	Constant	Value
`Alnum`	`STRING_EMPTY`	alnumStringEmpty
`Alpha`	`STRING_EMPTY`	alphaStringEmpty
`Barcode_Ean13`	`INVALID`	ean13Invalid
`Barcode_Ean13`	`INVALID_LENGTH`	ean13InvalidLength
`Barcode_UpcA`	`INVALID`	upcaInvalid
`Barcode_UpcA`	`INVALID_LENGTH`	upcaInvalidLength
`Digits`	`STRING_EMPTY`	digitsStringEmpty

B.2. Zend Framework 1.9

When upgrading from a release of Zend Framework earlier than 1.9.0 to any 1.9 release, you should note the following migration notes.

B.2.1. Zend_File_Transfer

B.2.1.1. MimeType validation

For security reasons we had to turn off the default fallback mechanism of the `MimeType`, `ExcludeMimeType`, `IsCompressed` and `IsImage` validators. This means, that if the *fileInfo* or *magicMime* extensions can not be found, the validation will always fail.

If you are in need of validation by using the HTTP fields which are provided by the user then you can turn on this feature by using the `enableHeaderCheck()` method.

Security hint

You should note that relying on the HTTP fields, which are provided by your user, is a security risk. They can easily be changed and could allow your user to provide a malcious file.

Example B.1. Allow the usage of the HTTP fields

```
// at initiation
$valid = new Zend_File_Transfer_Adapter_Http(array('headerCheck' => true);

// or afterwards
$valid->enableHeaderCheck();
```

B.2.2. Zend_Filter

Prior to the 1.9 release, `Zend_Filter` allowed the usage of the static `get()` method. As with release 1.9 this method has been renamed to `filterStatic()` to be more descriptive. The old `get()` method is marked as deprecated.

B.2.3. Zend_Http_Client

B.2.3.1. Changes to internal uploaded file information storage

In version 1.9 of Zend Framework, there has been a change in the way `Zend_Http_Client` internally stores information about files to be uploaded, set using the `Zend_Http_Client::setFileUpload()` method.

This change was introduced in order to allow multiple files to be uploaded with the same form name, as an array of files. More information about this issue can be found in this bug report.

Example B.2. Internal storage of uploaded file information

```
// Upload two files with the same form element name, as an array
$client = new Zend_Http_Client();
$client->setFileUpload('file1.txt',
                       'userfile[]',
                       'some raw data',
                       'text/plain');
$client->setFileUpload('file2.txt',
                       'userfile[]',
                       'some other data',
                       'application/octet-stream');

// In Zend Framework 1.8 or older, the value of
// the protected member $client->files is:
// $client->files = array(
//     'userfile[]' => array('file2.txt',
                             'application/octet-stream',
                             'some other data')
// );

// In Zend Framework 1.9 or newer, the value of $client->files is:
// $client->files = array(
//     array(
//         'formname' => 'userfile[]',
//         'filename' => 'file1.txt,
//         'ctype'    => 'text/plain',
//         'data'     => 'some raw data'
//     ),
//     array(
//         'formname' => 'userfile[]',
//         'filename' => 'file2.txt',
//         'formname' => 'application/octet-stream',
//         'formname' => 'some other data'
//     )
// );
```

As you can see, this change permits the usage of the same form element name with more than one file - however, it introduces a subtle backwards-compatibility change and as such should be noted.

B.2.3.2. Deprecation of Zend_Http_Client::_getParametersRecursive()

Starting from version 1.9, the protected method `_getParametersRecursive()` is no longer used by `Zend_Http_Client` and is deprecated. Using it will cause an `E_NOTICE` message to be emitted by PHP.

If you subclass `Zend_Http_Client` and call this method, you should look into using the `Zend_Http_Client::_flattenParametersArray()` static method instead.

Again, since this `_getParametersRecursive()` is a protected method, this change will only affect users who subclass `Zend_Http_Client`.

B.2.4. Zend_Locale

B.2.4.1. Depreciated methods

Some specialized translation methods have been depreciated because they duplicate existing behaviour. Note that the old methods will still work, but a user notice is triggered which describes

the new call. The methods will be erased with 2.0. See the following list for old and new method call.

Table B.3. List of measurement types

Old call	New call
`getLanguageTranslationList($locale)`	`getTranslationList('language', $locale)`
`getScriptTranslationList($locale)`	`getTranslationList('script', $locale)`
`getCountryTranslationList($locale)`	`getTranslationList('territory', $locale, 2)`
`getTerritoryTranslationList($locale)`	`getTranslationList('territory', $locale, 1)`
`getLanguageTranslation($value, $locale)`	`getTranslation($value, 'language', $locale)`
`getScriptTranslation($value, $locale)`	`getTranslation($value, 'script', $locale)`
`getCountryTranslation($value, $locale)`	`getTranslation($value, 'country', $locale)`
`getTerritoryTranslation($value, $locale)`	`getTranslation($value, 'territory', $locale)`

B.2.5. Zend_View_Helper_Navigation

Prior to the 1.9 release, the menu helper (`Zend_View_Helper_Navigation_Menu`) did not render sub menus correctly. When onlyActiveBranch was `TRUE` and the option renderParents `FALSE`, nothing would be rendered if the deepest active page was at a depth lower than the minDepth option.

In simpler words; if minDepth was set to '1' and the active page was at one of the first level pages, nothing would be rendered, as the following example shows.

Consider the following container setup:

```
<?php
$container = new Zend_Navigation(array(
    array(
        'label' => 'Home',
        'uri'   => '#'
    ),
    array(
        'label'  => 'Products',
        'uri'    => '#',
        'active' => true,
        'pages'  => array(
            array(
                'label' => 'Server',
                'uri'   => '#'
            ),
            array(
                'label' => 'Studio',
                'uri'   => '#'
```

```
            )
        )
    ),
    array(
        'label' => 'Solutions',
        'uri'   => '#'
    )
));
```

The following code is used in a view script:

```
<?php echo $this->navigation()->menu()->renderMenu($container, array(
    'minDepth'         => 1,
    'onlyActiveBranch' => true,
    'renderParents'    => false
)); ?>
```

Before release 1.9, the code snippet above would output nothing.

Since release 1.9, the `_renderDeepestMenu()` method in `Zend_View_Helper_Navigation_Menu` will accept active pages at one level below minDepth, as long as the page has children.

The same code snippet will now output the following:

```
<ul class="navigation">
    <li>
        <a href="#">Server</a>
    </li>
    <li>
        <a href="#">Studio</a>
    </li>
</ul>
```

B.2.6. Security fixes as with 1.9.7

Additionally, users of the 1.9 series may be affected by other changes starting in version 1.9.7. These are all security fixes that also have potential backwards compatibility implications.

B.2.6.1. Zend_Dojo_View_Helper_Editor

A slight change was made in the 1.9 series to modify the default usage of the Editor dijit to use div tags instead of a textarea tag; the latter usage has security implications, and usage of div tags is recommended by the Dojo project.

In order to still allow graceful degradation, a new `degrade` option was added to the view helper; this would allow developers to optionally use a textarea instead. However, this opens applications developed with that usage to XSS vectors. In 1.9.7, we have removed this option. Graceful degradation is still supported, however, via a noscript tag that embeds a textarea. This solution addressess all security concerns.

The takeaway is that if you were using the `degrade` flag, it will simply be ignored at this time.

B.2.6.2. Zend_Filter_HtmlEntities

In order to default to a more secure character encoding, `Zend_Filter_HtmlEntities` now defaults to UTF-8 instead of ISO-8859-1.

Additionally, because the actual mechanism is dealing with character encodings and not character sets, two new methods have been added, `setEncoding()` and `getEncoding()`. The previous methods `setCharSet()` and `setCharSet()` are now deprecated and proxy to the new methods. Finally, instead of using the protected members directly within the `filter()` method, these members are retrieved by their explicit accessors. If you were extending the filter in the past, please check your code and unit tests to ensure everything still continues to work.

B.2.6.3. Zend_Filter_StripTags

`Zend_Filter_StripTags` contains a flag, `commentsAllowed`, that, in previous versions, allowed you to optionally whitelist HTML comments in HTML text filtered by the class. However, this opens code enabling the flag to XSS attacks, particularly in Internet Explorer (which allows specifying conditional functionality via HTML comments). Starting in version 1.9.7 (and backported to versions 1.8.5 and 1.7.9), the `commentsAllowed` flag no longer has any meaning, and all HTML comments, including those containing other HTML tags or nested commments, will be stripped from the final output of the filter.

B.3. Zend Framework 1.8

When upgrading from a previous release to Zend Framework 1.8 or higher you should note the following migration notes.

B.3.1. Zend_Controller

B.3.1.1. Standard Route Changes

As translated segments were introduced into the new standard route, the '@' character is now a special character in the beginning of a route segment. To be able to use it in a static segment, you must escape it by prefixing it with second '@' character. The same rule now applies for the ':' character.

B.3.2. Zend_Locale

B.3.2.1. Default caching

As with Zend Framework 1.8 a default caching was added. The reason behind this change was, that most users had performance problems but did not add caching at all. As the I18n core is a bottleneck when no caching is used we decided to add a default caching when no cache has been set to `Zend_Locale`.

Sometimes it is still wanted to prevent caching at all even if this decreases performance. To do so you can simply disable caching by using the `disableCache()` method.

Example B.3. Disabling default caching

```
Zend_Locale::disableCache(true);
```

B.4. Zend Framework 1.7

When upgrading from a previous release to Zend Framework 1.7 or higher you should note the following migration notes.

B.4.1. Zend_Controller

B.4.1.1. Dispatcher Interface Changes

Users brought to our attention the fact that `Zend_Controller_Action_Helper_ViewRenderer` were using a method of the dispatcher abstract class that was not in the dispatcher interface. We have now added the following method to ensure that custom dispatchers will continue to work with the shipped implementations:

- `formatModuleName()`: should be used to take a raw controller name, such as one that would be packaged inside a request object, and reformat it to a proper class name that a class extending `Zend_Controller_Action` would use

B.4.2. Zend_File_Transfer

B.4.2.1. Changes when using filters and validators

As noted by users, the validators from `Zend_File_Transfer` do not work in conjunction with `Zend_Config` due to the fact that they have not used named arrays.

Therefor, all filters and validators for `Zend_File_Transfer` have been reworked. While the old signatures continue to work, they have been marked as deprecated, and will emit a PHP notice asking you to fix them.

The following list shows you the changes you will have to do for proper usage of the parameters.

B.4.2.1.1. Filter: Rename

- Old method API: `Zend_Filter_File_Rename($oldfile, $newfile, $overwrite)`
- New method API: `Zend_Filter_File_Rename($options)` where `$options` accepts the following array keys: *source* equals to `$oldfile`, *target* equals to `$newfile`, *overwrite* equals to `$overwrite`.

Example B.4. Changes for the rename filter from 1.6 to 1.7

```
// Example for 1.6
$upload = new Zend_File_Transfer_Adapter_Http();
$upload->addFilter('Rename',
                   array('/path/to/oldfile', '/path/to/newfile', true));

// Same example for 1.7
$upload = new Zend_File_Transfer_Adapter_Http();
$upload->addFilter('Rename',
                   array('source' => '/path/to/oldfile',
                         'target' => '/path/to/newfile',
                         'overwrite' => true));
```

B.4.2.1.2. Validator: Count

- Old method API: `Zend_Validate_File_Count($min, $max)`
- New method API: `Zend_Validate_File_Count($options)` where `$options` accepts the following array keys: *min* equals to `$min`, *max* equals to `$max`.

Example B.5. Changes for the count validator from 1.6 to 1.7

```
// Example for 1.6
$upload = new Zend_File_Transfer_Adapter_Http();
$upload->addValidator('Count',
                      array(2, 3));

// Same example for 1.7
$upload = new Zend_File_Transfer_Adapter_Http();
$upload->addValidator('Count',
                      false,
                      array('min' => 2,
                            'max' => 3));
```

B.4.2.1.3. Validator:Extension

- Old method API: `Zend_Validate_File_Extension($extension, $case)`
- New method API: `Zend_Validate_File_Extension($options)` where `$options` accepts the following array keys: * equals to `$extension` and can have any other key, *case* equals to `$case`.

Example B.6. Changes for the extension validator from 1.6 to 1.7

```
// Example for 1.6
$upload = new Zend_File_Transfer_Adapter_Http();
$upload->addValidator('Extension',
                      array('jpg,gif,bmp', true));

// Same example for 1.7
$upload = new Zend_File_Transfer_Adapter_Http();
$upload->addValidator('Extension',
                      false,
                      array('extension1' => 'jpg,gif,bmp',
                            'case' => true));
```

B.4.2.1.4. Validator: FilesSize

- Old method API: `Zend_Validate_File_FilesSize($min, $max, $bytestring)`
- New method API: `Zend_Validate_File_FilesSize($options)` where `$options` accepts the following array keys: *min* equals to `$min`, *max* equals to `$max`, *bytestring* equals to `$bytestring`.

Additionally, the `useByteString()` method signature has changed. It can only be used to test if the validator is expecting to use byte strings in generated messages. To set the value of the flag, use the `setUseByteString()` method.

Example B.7. Changes for the filessize validator from 1.6 to 1.7

```
// Example for 1.6
$upload = new Zend_File_Transfer_Adapter_Http();
$upload->addValidator('FilesSize',
                  array(100, 10000, true));

// Same example for 1.7
$upload = new Zend_File_Transfer_Adapter_Http();
$upload->addValidator('FilesSize',
                      false,
                      array('min' => 100,
                            'max' => 10000,
                            'bytestring' => true));

// Example for 1.6
$upload->useByteString(true); // set flag

// Same example for 1.7
$upload->setUseByteSting(true); // set flag
```

B.4.2.1.5. Validator: Hash

- Old method API: `Zend_Validate_File_Hash($hash, $algorithm)`
- New method API: `Zend_Validate_File_Hash($options)` where `$options` accepts the following array keys: * equals to `$hash` and can have any other key, *algorithm* equals to `$algorithm`.

Example B.8. Changes for the hash validator from 1.6 to 1.7

```
// Example for 1.6
$upload = new Zend_File_Transfer_Adapter_Http();
$upload->addValidator('Hash',
                  array('12345', 'md5'));

// Same example for 1.7
$upload = new Zend_File_Transfer_Adapter_Http();
$upload->addValidator('Hash',
                      false,
                      array('hash1' => '12345',
                            'algorithm' => 'md5'));
```

B.4.2.1.6. Validator: ImageSize

- Old method API: `Zend_Validate_File_ImageSize($minwidth, $minheight, $maxwidth, $maxheight)`
- New method API: `Zend_Validate_File_FilesSize($options)` where `$options` accepts the following array keys: *minwidth* equals to `$minwidth`, *maxwidth* equals to `$maxwidth`, *minheight* equals to `$minheight`, *maxheight* equals to `$maxheight`.

Example B.9. Changes for the imagesize validator from 1.6 to 1.7

```
// Example for 1.6
$upload = new Zend_File_Transfer_Adapter_Http();
$upload->addValidator('ImageSize',
                      array(10, 10, 100, 100));

// Same example for 1.7
$upload = new Zend_File_Transfer_Adapter_Http();
$upload->addValidator('ImageSize',
                      false,
                      array('minwidth' => 10,
                            'minheight' => 10,
                            'maxwidth' => 100,
                            'maxheight' => 100));
```

B.4.2.1.7. Validator: Size

- Old method API: `Zend_Validate_File_Size($min, $max, $bytestring)`
- New method API: `Zend_Validate_File_Size($options)` where `$options` accepts the following array keys: *min* equals to `$min`, *max* equals to `$max`, *bytestring* equals to `$bytestring`.

Example B.10. Changes for the size validator from 1.6 to 1.7

```
// Example for 1.6
$upload = new Zend_File_Transfer_Adapter_Http();
$upload->addValidator('Size',
                      array(100, 10000, true));

// Same example for 1.7
$upload = new Zend_File_Transfer_Adapter_Http();
$upload->addValidator('Size',
                      false,
                      array('min' => 100,
                            'max' => 10000,
                            'bytestring' => true));
```

B.4.3. Zend_Locale

B.4.3.1. Changes when using isLocale()

According to the coding standards `isLocale()` had to be changed to return a boolean. In previous releases a string was returned on success. For release 1.7 a compatibility mode has been added which allows to use the old behaviour of a returned string, but it triggers a user warning to mention you to change to the new behaviour. The rerouting which the old behaviour of `isLocale()` could have done is no longer neccessary as all I18n will now process a rerouting themself.

To migrate your scripts to the new API, simply use the method as shown below.

Example B.11. How to change isLocale() from 1.6 to 1.7

```
// Example for 1.6
if ($locale = Zend_Locale::isLocale($locale)) {
    // do something
}

// Same example for 1.7

// You should change the compatiblity mode to prevent user warnings
// But you can do this in your bootstrap
Zend_Locale::$compatibilityMode = false;

if (Zend_Locale::isLocale($locale)) {
}
```

Note that you can use the second parameter to see if the locale is correct without processing a rerouting.

```
// Example for 1.6
if ($locale = Zend_Locale::isLocale($locale, false)) {
    // do something
}

// Same example for 1.7

// You should change the compatiblity mode to prevent user warnings
// But you can do this in your bootstrap
Zend_Locale::$compatibilityMode = false;

if (Zend_Locale::isLocale($locale, false)) {
    if (Zend_Locale::isLocale($locale, true)) {
        // no locale at all
    }

    // original string is no locale but can be rerouted
}
```

B.4.3.2. Changes when using getDefault()

The meaning of the `getDefault()` method has been change due to the fact that we integrated a framework locale which can be set with `setDefault()`. It does no longer return the locale chain but only the set framework locale.

To migrate your scripts to the new API, simply use the method as shown below.

Example B.12. How to change getDefault() from 1.6 to 1.7

```
// Example for 1.6
$locales = $locale->getDefault(Zend_Locale::BROWSER);

// Same example for 1.7

// You should change the compatiblity mode to prevent user warnings
// But you can do this in your bootstrap
Zend_Locale::$compatibilityMode = false;

$locale = Zend_Locale::getOrder(Zend_Locale::BROWSER);
```

Note that the second parameter of the old `getDefault()` implementation is not available anymore, but the returned values are the same.

Per default the old behaviour is still active, but throws a user warning. When you have changed your code to the new behaviour you should also change the compatibility mode to FALSE so that no warning is thrown anymore.

B.4.4. Zend_Translate

B.4.4.1. Setting languages

When using automatic detection of languages, or setting languages manually to `Zend_Translate` you may have mentioned that from time to time a notice is thrown about not added or empty translations. In some previous release also an exception was raised in some cases.

The reason is, that when a user requests a non existing language, you have no simple way to detect what's going wrong. So we added those notices which show up in your log and tell you that the user requested a language which you do not support. Note that the code, even when we trigger such an notice, keeps working without problems.

But when you use a own error or exception handler, like xdebug, you will get all notices returned, even if this was not your intention. This is due to the fact that these handlers override all settings from within PHP.

To get rid of these notices you can simply set the new option 'disableNotices' to TRUE. It defaults to FALSE.

Example B.13. Setting languages without getting notices

Let's assume that we have 'en' available and our user requests 'fr' which is not in our portfolio of translated languages.

```
$language = new Zend_Translate('gettext',
                               '/path/to/translations',
                               'auto');
```

In this case we will get an notice about a not available language 'fr'. Simply add the option and the notices will be disabled.

```
$language = new Zend_Translate('gettext',
                               '/path/to/translations',
                               'auto',
                               array('disableNotices' => true));
```

B.4.5. Zend_View

The API changes within `Zend_View` are only notable for you when you are upgrading to release 1.7.5 or higher.

Prior to the 1.7.5 release, the Zend Framework team was notified of a potential Local File Inclusion (LFI) vulnerability in the `Zend_View::render()` method. Prior to 1.7.5, the method allowed, by default, the ability to specify view scripts that included parent directory notation (e.g., "../" or "..\"). This opens the possibility for an LFI attack if unfiltered user input is passed to the `render()` method:

```
// Where $_GET['foobar'] = '../../../../etc/passwd'
echo $view->render($_GET['foobar']); // LFI inclusion
```

`Zend_View` now by default raises an exception when such a view script is requested.

B.4.5.1. Disabling LFI protection for the render() method

Since a number of developers reported that they were using such notation within their applications that was *not* the result of user input, a special flag was created to allow disabling the default protection. You have two methods for doing so: by passing the 'lfiProtectionOn' key to the constructor options, or by explicitly calling the `setLfiProtection()` method.

```
// Disabling via constructor
$view = new Zend_View(array('lfiProtectionOn' => false));

// Disabling via exlicit method call:
$view = new Zend_View();
$view->setLfiProtection(false);
```

B.5. Zend Framework 1.6

When upgrading from a previous release to Zend Framework 1.6 or higher you should note the following migration notes.

B.5.1. Zend_Controller

B.5.1.1. Dispatcher Interface Changes

Users brought to our attention the fact that `Zend_Controller_Front` and `Zend_Controller_Router_Route_Module` were each using methods of the dispatcher that were not in the dispatcher interface. We have now added the following three methods to ensure that custom dispatchers will continue to work with the shipped implementations:

- `getDefaultModule()`: should return the name of the default module.
- `getDefaultControllerName()`: should return the name of the default controller.
- `getDefaultAction()`: should return the name of the default action.

B.5.2. Zend_File_Transfer

B.5.2.1. Changes when using validators

As noted by users, the validators from `Zend_File_Transfer` do not work the same way like the default ones from `Zend_Form`. `Zend_Form` allows the usage of a `$breakChainOnFailure` parameter which breaks the validation for all further validators when an validation error has occurred.

So we added this parameter also to all existing validators from `Zend_File_Transfer`.

- Old method API: `addValidator($validator, $options, $files)`.
- New method API: `addValidator($validator, $breakChainOnFailure, $options, $files)`.

To migrate your scripts to the new API, simply add a FALSE after defining the wished validator.

Example B.14. How to change your file validators from 1.6.1 to 1.6.2

```
// Example for 1.6.1
$upload = new Zend_File_Transfer_Adapter_Http();
$upload->addValidator('FilesSize', array('1B', '100kB'));

// Same example for 1.6.2 and newer
// Note the added boolean false
$upload = new Zend_File_Transfer_Adapter_Http();
$upload->addValidator('FilesSize', false, array('1B', '100kB'));
```

B.6. Zend Framework 1.5

When upgrading from a previous release to Zend Framework 1.5 or higher you should note the following migration notes.

B.6.1. Zend_Controller

Though most basic functionality remains the same, and all documented functionality remains the same, there is one particular *undocumented* "feature" that has changed.

When writing URLs, the documented way to write camelCased action names is to use a word separator; these are '.' or '-' by default, but may be configured in the dispatcher. The dispatcher internally lowercases the action name, and uses these word separators to re-assemble the action method using camelCasing. However, because PHP functions are not case sensitive, you *could* still write URLs using camelCasing, and the dispatcher would resolve these to the same location. For example, 'camel-cased' would become 'camelCasedAction' by the dispatcher, whereas 'camelCased' would become 'camelcasedAction'; however, due to the case insensitivity of PHP, both will execute the same method.

This causes issues with the ViewRenderer when resolving view scripts. The canonical, documented way is that all word separators are converted to dashes, and the words lowercased. This creates a semantic tie between the actions and view scripts, and the normalization ensures that the scripts can be found. However, if the action 'camelCased' is called and actually resolves, the word separator is no longer present, and the ViewRenderer attempts to resolve to a different location -- `camelcased.phtml` instead of `camel-cased.phtml`.

Some developers relied on this "feature", which was never intended. Several changes in the 1.5.0 tree, however, made it so that the ViewRenderer no longer resolves these paths; the semantic tie is now enforced. First among these, the dispatcher now enforces case sensitivity in action names. What this means is that referring to your actions on the url using camelCasing will no longer resolve to the same method as using word separators (i.e., 'camel-casing'). This leads to the ViewRenderer now only honoring the word-separated actions when resolving view scripts.

If you find that you were relying on this "feature", you have several options:

- Best option: rename your view scripts. Pros: forward compatibility. Cons: if you have many view scripts that relied on the former, unintended behavior, you will have a lot of renaming to do.
- Second best option: The ViewRenderer now delegates view script resolution to `Zend_Filter_Inflector`; you can modify the rules of the inflector to no longer separate the words of an action with a dash:

```
$viewRenderer =
    Zend_Controller_Action_HelperBroker::getStaticHelper('viewRenderer');
$inflector = $viewRenderer->getInflector();
$inflector->setFilterRule(':action', array(
    new Zend_Filter_PregReplace(
        '#[^a-z0-9' . preg_quote(DIRECTORY_SEPARATOR, '#') . ']+#i',
        ''
    ),
    'StringToLower'
));
```

  The above code will modify the inflector to no longer separate the words with dash; you may also want to remove the 'StringToLower' filter if you *do* want the actual view script names camelCased as well.

  If renaming your view scripts would be too tedious or time consuming, this is your best option until you can find the time to do so.

- Least desirable option: You can force the dispatcher to dispatch camelCased action names with a new front controller flag, useCaseSensitiveActions:

```
$front->setParam('useCaseSensitiveActions', true);
```

  This will allow you to use camelCasing on the url and still have it resolve to the same action as when you use word separators. However, this will mean that the original issues will cascade on through; you will likely need to use the second option above in addition to this for things to work at all reliably.

  Note, also, that usage of this flag will raise a notice that this usage is deprecated.

B.7. Zend Framework 1.0

When upgrading from a previous release to Zend Framework 1.0 or higher you should note the following migration notes.

B.7.1. Zend_Controller

The principal changes introduced in 1.0.0RC1 are the introduction of and default enabling of the ErrorHandler plugin and the ViewRenderer action helper. Please read the documentation to each thoroughly to see how they work and what effect they may have on your applications.

The `ErrorHandler` plugin runs during `postDispatch()` checking for exceptions, and forwarding to a specified error handler controller. You should include such a controller in your application. You may disable it by setting the front controller parameter noErrorHandler:

```
$front->setParam('noErrorHandler', true);
```

The `ViewRenderer` action helper automates view injection into action controllers as well as autorendering of view scripts based on the current action. The primary issue you may encounter is if you have actions that do not render view scripts and neither forward or redirect, as the `ViewRenderer` will attempt to render a view script based on the action name.

There are several strategies you can take to update your code. In the short term, you can globally disable the `ViewRenderer` in your front controller bootstrap prior to dispatching:

```
// Assuming $front is an instance of Zend_Controller_Front
$front->setParam('noViewRenderer', true);
```

However, this is not a good long term strategy, as it means most likely you'll be writing more code.

When you're ready to start using the `ViewRenderer` functionality, there are several things to look for in your controller code. First, look at your action methods (the methods ending in 'Action'), and determine what each is doing. If none of the following is happening, you'll need to make changes:

- Calls to **$this->render();**
- Calls to **$this->_forward();**
- Calls to **$this->_redirect();**
- Calls to the `Redirector` action helper

The easiest change is to disable auto-rendering for that method:

```
$this->_helper->viewRenderer->setNoRender();
```

If you find that none of your action methods are rendering, forwarding, or redirecting, you will likely want to put the above line in your `preDispatch()` or `init()` methods:

```
public function preDispatch()
{
    // disable view script autorendering
    $this->_helper->viewRenderer->setNoRender()
    // .. do other things...
}
```

If you are calling `render()`, and you're using the Conventional Modular directory structure, you'll want to change your code to make use of autorendering:

- If you're rendering multiple view scripts in a single action, you don't need to change a thing.
- If you're simply calling `render()` with no arguments, you can remove such lines.
- If you're calling `render()` with arguments, and not doing any processing afterwards or rendering multiple view scripts, you can change these calls to read **$this->_helper->viewRenderer();**.

If you're not using the conventional modular directory structure, there are a variety of methods for setting the view base path and script path specifications so that you can make use of the `ViewRenderer`. Please read the ViewRenderer documentation for information on these methods.

If you're using a view object from the registry, or customizing your view object, or using a different view implementation, you'll want to inject the `ViewRenderer` with this object. This can be done easily at any time.

- Prior to dispatching a front controller instance:

```
// Assuming $view has already been defined
$viewRenderer = new Zend_Controller_Action_Helper_ViewRenderer($view);
Zend_Controller_Action_HelperBroker::addHelper($viewRenderer);
```

- Any time during the bootstrap process:

```
$viewRenderer =
    Zend_Controller_Action_HelperBroker::getStaticHelper('viewRenderer');
$viewRenderer->setView($view);
```

There are many ways to modify the `ViewRenderer`, including setting a different view script to render, specifying replacements for all replaceable elements of a view script path (including the suffix), choosing a response named segment to utilize, and more. If you aren't using the conventional modular directory structure, you can even associate different path specifications with the `ViewRenderer`.

We encourage you to adapt your code to use the `ErrorHandler` and `ViewRenderer` as they are now core functionality.

B.7.2. Zend_Currency

Creating an object of `Zend_Currency` has become simpler. You no longer have to give a script or set it to `NULL`. The optional script parameter is now an option which can be set through the `setFormat()` method.

```
$currency = new Zend_Currency($currency, $locale);
```

The `setFormat()` method takes now an array of options. These options are set permanently and override all previously set values. Also a new option 'precision' has been added. The following options have been refactored:

- *position*: Replacement for the old 'rules' parameter.
- *script*: Replacement for the old 'script' parameter.
- *format*: Replacement for the old 'locale' parameter which does not set new currencies but only the number format.
- *display*: Replacement for the old 'rules' parameter.
- *precision*: New parameter.
- *name*: Replacement for the ole 'rules' parameter. Sets the full currencies name.
- *currency*: New parameter.

- *symbol*: New parameter.

```
$currency->setFormat(array $options);
```

The `toCurrency()` method no longer supports the optional 'script' and 'locale' parameters. Instead it takes an options array which can contain the same keys as for the `setFormat()` method.

```
$currency->toCurrency($value, array $options);
```

The methods `getSymbol()`, `getShortName()`, `getName()`, `getRegionList()` and `getCurrencyList()` are no longer static and can be called from within the object. They return the set values of the object if no parameter has been set.

B.8. Zend Framework 0.9

When upgrading from a previous release to Zend Framework 0.9 or higher you should note the following migration notes.

B.8.1. Zend_Controller

0.9.3 introduces action helpers. As part of this change, the following methods have been removed as they are now encapsulated in the redirector action helper:

- `setRedirectCode()`; use `Zend_Controller_Action_Helper_Redirector::setCode()`.
- `setRedirectPrependBase()`; use `Zend_Controller_Action_Helper_Redirector::setPrependBase()`.
- `setRedirectExit()`; use `Zend_Controller_Action_Helper_Redirector::setExit()`.

Read the action helpers documentation for more information on how to retrieve and manipulate helper objects, and the redirector helper documentation for more information on setting redirect options (as well as alternate methods for redirecting).

B.9. Zend Framework 0.8

When upgrading from a previous release to Zend Framework 0.8 or higher you should note the following migration notes.

B.9.1. Zend_Controller

Per previous changes, the most basic usage of the MVC components remains the same:

```
Zend_Controller_Front::run('/path/to/controllers');
```

However, the directory structure underwent an overhaul, several components were removed, and several others either renamed or added. Changes include:

- `Zend_Controller_Router` was removed in favor of the rewrite router.
- `Zend_Controller_RewriteRouter` was renamed to `Zend_Controller_Router_Rewrite`, and promoted to the standard router shipped with the framework; `Zend_Controller_Front` will use it by default if no other router is supplied.

- A new route class for use with the rewrite router was introduced, `Zend_Controller_Router_Route_Module`; it covers the default route used by the MVC, and has support for controller modules.
- `Zend_Controller_Router_StaticRoute` was renamed to `Zend_Controller_Router_Route_Static`.
- `Zend_Controller_Dispatcher` was renamed `Zend_Controller_Dispatcher_Standard`.
- `Zend_Controller_Action::_forward()`'s arguments have changed. The signature is now:

```
final protected function _forward($action,
                                  $controller = null,
                                  $module = null,
                                  array $params = null);
```

`$action` is always required; if no controller is specified, an action in the current controller is assumed. `$module` is always ignored unless `$controller` is specified. Finally, any `$params` provided will be appended to the request object. If you do not require the controller or module, but still need to pass parameters, simply specify `NULL` for those values.

B.10. Zend Framework 0.6

When upgrading from a previous release to Zend Framework 0.6 or higher you should note the following migration notes.

B.10.1. Zend_Controller

The most basic usage of the MVC components has not changed; you can still do each of the following:

```
Zend_Controller_Front::run('/path/to/controllers');
```

```
/* -- create a router -- */
$router = new Zend_Controller_RewriteRouter();
$router->addRoute('user',
                  'user/:username',
                  array('controller' => 'user', 'action' => 'info')
);

/* -- set it in a controller -- */
$ctrl = Zend_Controller_Front::getInstance();
$ctrl->setRouter($router);

/* -- set controller directory and dispatch -- */
$ctrl->setControllerDirectory('/path/to/controllers');
$ctrl->dispatch();
```

We encourage use of the Response object to aggregate content and headers. This will allow for more flexible output format switching (for instance, JSON or XML instead of XHTML) in your applications. By default, `dispatch()` will render the response, sending both headers and rendering any content. You may also have the front controller return the response using `returnResponse()`, and then render the response using your own logic. A future version of the front controller may enforce use of the response object via output buffering.

There are many additional features that extend the existing API, and these are noted in the documentation.

The main changes you will need to be aware of will be found when subclassing the various components. Key amongst these are:

- `Zend_Controller_Front::dispatch()` by default traps exceptions in the response object, and does not render them, in order to prevent sensitive system information from being rendered. You can override this in several ways:
  - Set `throwExceptions()` in the front controller:

    ```
    $front->throwExceptions(true);
    ```

  - Set `renderExceptions()` in the response object:

    ```
    $response->renderExceptions(true);
    $front->setResponse($response);
    $front->dispatch();

    // or:
    $front->returnResponse(true);
    $response = $front->dispatch();
    $response->renderExceptions(true);
    echo $response;
    ```

- `Zend_Controller_Dispatcher_Interface::dispatch()` now accepts and returns a The Request Object instead of a dispatcher token.
- `Zend_Controller_Router_Interface::route()` now accepts and returns a The Request Object instead of a dispatcher token.
- `Zend_Controller_Action` changes include:
  - The constructor now accepts exactly three arguments, `Zend_Controller_Request_Abstract $request`, `Zend_Controller_Response_Abstract $response`, and Array `$params` (optional). `Zend_Controller_Action::__construct()` uses these to set the request, response, and invokeArgs properties of the object, and if overriding the constructor, you should do so as well. Better yet, use the `init()` method to do any instance configuration, as this method is called as the final action of the constructor.
  - `run()` is no longer defined as final, but is also no longer used by the front controller; its sole purpose is for using the class as a page controller. It now takes two optional arguments, a `Zend_Controller_Request_Abstract $request` and a `Zend_Controller_Response_Abstract $response`.
  - `indexAction()` no longer needs to be defined, but is encouraged as the default action. This allows using the RewriteRouter and action controllers to specify different default action methods.
  - `__call()` should be overridden to handle any undefined actions automatically.
  - `_redirect()` now takes an optional second argument, the HTTP code to return with the redirect, and an optional third argument, `$prependBase`, that can indicate that the base URL registered with the request object should be prepended to the url specified.

- The `$_action` property is no longer set. This property was a `Zend_Controller_Dispatcher_Token`, which no longer exists in the current incarnation. The sole purpose of the token was to provide information about the requested controller, action, and URL parameters. This information is now available in the request object, and can be accessed as follows:

```
// Retrieve the requested controller name
// Access used to be via: $this->_action->getControllerName().
// The example below uses getRequest(), though you may also directly
// access the $_request property; using getRequest() is recommended as
// a parent class may override access to the request object.
$controller = $this->getRequest()->getControllerName();

// Retrieve the requested action name
// Access used to be via: $this->_action->getActionName().
$action = $this->getRequest()->getActionName();

// Retrieve the request parameters
// This hasn't changed; the _getParams() and _getParam() methods simply
// proxy to the request object now.
$params = $this->_getParams();
// request 'foo' parameter, using 'default' as default value if not found
$foo = $this->_getParam('foo', 'default');
```

- `noRouteAction()` has been removed. The appropriate way to handle non-existent action methods should you wish to route them to a default action is using `__call()`:

```
public function __call($method, $args)
{
    // If an unmatched 'Action' method was requested, pass on to the
    // default action method:
    if ('Action' == substr($method, -6)) {
        return $this->defaultAction();
    }

    throw new Zend_Controller_Exception('Invalid method called');
}
```

- `Zend_Controller_RewriteRouter::setRewriteBase()` has been removed. Use `Zend_Controller_Front::setBaseUrl()` instead (or `Zend_Controller_Request_Http::setBaseUrl()`, if using that request class).

- `Zend_Controller_Plugin_Interface` was replaced by `Zend_Controller_Plugin_Abstract`. All methods now accept and return a The Request Object instead of a dispatcher token.

Appendix C. Zend Framework Coding Standard for PHP

Table of Contents

C.1. Overview

C.1.1. Scope

This document provides guidelines for code formatting and documentation to individuals and teams contributing to Zend Framework. Many developers using Zend Framework have also found these coding standards useful because their code's style remains consistent with all Zend Framework code. It is also worth noting that it requires significant effort to fully specify coding standards.

Note: Sometimes developers consider the establishment of a standard more important than what that standard actually suggests at the most detailed level of design. The guidelines in Zend Framework's coding standards capture practices that have worked well on the Zend Framework project. You may modify these standards or use them as is in accordance with the terms of our license.

Topics covered in Zend Framework's coding standards include:

- PHP File Formatting

- Naming Conventions
- Coding Style
- Inline Documentation

C.1.2. Goals

Coding standards are important in any development project, but they are particularly important when many developers are working on the same project. Coding standards help ensure that the code is high quality, has fewer bugs, and can be easily maintained.

C.2. PHP File Formatting

C.2.1. General

For files that contain only PHP code, the closing tag ("?>") is never permitted. It is not required by PHP, and omitting it´ prevents the accidental injection of trailing white space into the response.

Important: Inclusion of arbitrary binary data as permitted by `__HALT_COMPILER()` is prohibited from PHP files in the Zend Framework project or files derived from them. Use of this feature is only permitted for some installation scripts.

C.2.2. Indentation

Indentation should consist of 4 spaces. Tabs are not allowed.

C.2.3. Maximum Line Length

The target line length is 80 characters. That is to say, Zend Framework developers should strive keep each line of their code under 80 characters where possible and practical. However, longer lines are acceptable in some circumstances. The maximum length of any line of PHP code is 120 characters.

C.2.4. Line Termination

Line termination follows the Unix text file convention. Lines must end with a single linefeed (LF) character. Linefeed characters are represented as ordinal 10, or hexadecimal 0x0A.

Note: Do not use carriage returns (CR) as is the convention in Apple OS's (0x0D) or the carriage return - linefeed combination (CRLF) as is standard for the Windows OS (0x0D, 0x0A).

C.3. Naming Conventions

C.3.1. Classes

Zend Framework standardizes on a class naming convention whereby the names of the classes directly map to the directories in which they are stored. The root level directory of Zend Framework's standard library is the "Zend/" directory, whereas the root level directory of Zend Framework's extras library is the "ZendX/" directory. All Zend Framework classes are stored hierarchically under these root directories..

Class names may only contain alphanumeric characters. Numbers are permitted in class names but are discouraged in most cases. Underscores are only permitted in place of the path separator; the filename "`Zend/Db/Table.php`" must map to the class name "`Zend_Db_Table`".

If a class name is comprised of more than one word, the first letter of each new word must be capitalized. Successive capitalized letters are not allowed, e.g. a class "Zend_PDF" is not allowed while "`Zend_Pdf`" is acceptable.

These conventions define a pseudo-namespace mechanism for Zend Framework. Zend Framework will adopt the PHP namespace feature when it becomes available and is feasible for our developers to use in their applications.

See the class names in the standard and extras libraries for examples of this classname convention.

Important: Code that must be deployed alongside Zend Framework libraries but is not part of the standard or extras libraries (e.g. application code or libraries that are not distributed by Zend) must never start with "Zend_" or "ZendX_".

C.3.2. Abstract Classes

In general, abstract classes follow the same conventions as classes, with one additional rule: abstract class names must end in the term, "Abstract", and that term must not be preceded by an underscore. As an example, `Zend_Controller_Plugin_Abstract` is considered an invalid name, but `Zend_Controller_PluginAbstract` or `Zend_Controller_Plugin_PluginAbstract` would be valid names.

This naming convention is new with version 1.9.0 of Zend Framework. Classes that pre-date that version may not follow this rule, but will be renamed in the future in order to comply.

C.3.3. Interfaces

In general, interfaces follow the same conventions as classes, with one additional rule: interface names may optionally end in the term, "Interface", but that term must not be preceded by an underscore. As an example, `Zend_Controller_Plugin_Interface` is considered an invalid name, but `Zend_Controller_PluginInterface` or `Zend_Controller_Plugin_PluginInterface` would be valid names.

While this rule is not required, it is strongly recommended, as it provides a good visual cue to developers as to which files contain interfaces rather than classes.

This naming convention is new with version 1.9.0 of Zend Framework. Classes that pre-date that version may not follow this rule, but will be renamed in the future in order to comply.

C.3.4. Filenames

For all other files, only alphanumeric characters, underscores, and the dash character ("-") are permitted. Spaces are strictly prohibited.

Any file that contains PHP code should end with the extension "`.php`", with the notable exception of view scripts. The following examples show acceptable filenames for Zend Framework classes:

```
Zend/Db.php

Zend/Controller/Front.php

Zend/View/Helper/FormRadio.php
```

File names must map to class names as described above.

C.3.5. Functions and Methods

Function names may only contain alphanumeric characters. Underscores are not permitted. Numbers are permitted in function names but are discouraged in most cases.

Function names must always start with a lowercase letter. When a function name consists of more than one word, the first letter of each new word must be capitalized. This is commonly called "camelCase" formatting.

Verbosity is generally encouraged. Function names should be as verbose as is practical to fully describe their purpose and behavior.

These are examples of acceptable names for functions:

```
filterInput()

getElementById()

widgetFactory()
```

For object-oriented programming, accessors for instance or static variables should always be prefixed with "get" or "set". In implementing design patterns, such as the singleton or factory patterns, the name of the method should contain the pattern name where practical to more thoroughly describe behavior.

For methods on objects that are declared with the "private" or "protected" modifier, the first character of the method name must be an underscore. This is the only acceptable application of an underscore in a method name. Methods declared "public" should never contain an underscore.

Functions in the global scope (a.k.a "floating functions") are permitted but discouraged in most cases. Consider wrapping these functions in a static class.

C.3.6. Variables

Variable names may only contain alphanumeric characters. Underscores are not permitted. Numbers are permitted in variable names but are discouraged in most cases.

For instance variables that are declared with the "private" or "protected" modifier, the first character of the variable name must be a single underscore. This is the only acceptable application of an underscore in a variable name. Member variables declared "public" should never start with an underscore.

As with function names (see section 3.3) variable names must always start with a lowercase letter and follow the "camelCaps" capitalization convention.

Verbosity is generally encouraged. Variables should always be as verbose as practical to describe the data that the developer intends to store in them. Terse variable names such as "`$i`"

and "$n" are discouraged for all but the smallest loop contexts. If a loop contains more than 20 lines of code, the index variables should have more descriptive names.

C.3.7. Constants

Constants may contain both alphanumeric characters and underscores. Numbers are permitted in constant names.

All letters used in a constant name must be capitalized, while all words in a constant name must be separated by underscore characters.

For example, `EMBED_SUPPRESS_EMBED_EXCEPTION` is permitted but `EMBED_SUPPRESSEMBEDEXCEPTION` is not.

Constants must be defined as class members with the "const" modifier. Defining constants in the global scope with the "define" function is permitted but strongly discouraged.

C.4. Coding Style

C.4.1. PHP Code Demarcation

PHP code must always be delimited by the full-form, standard PHP tags:

```
<?php

?>
```

Short tags are never allowed. For files containing only PHP code, the closing tag must always be omitted (See General standards).

C.4.2. Strings

C.4.2.1. String Literals

When a string is literal (contains no variable substitutions), the apostrophe or "single quote" should always be used to demarcate the string:

```
$a = 'Example String';
```

C.4.2.2. String Literals Containing Apostrophes

When a literal string itself contains apostrophes, it is permitted to demarcate the string with quotation marks or "double quotes". This is especially useful for `SQL` statements:

```
$sql = "SELECT `id`, `name` from `people` "
     . "WHERE `name`='Fred' OR `name`='Susan'";
```

This syntax is preferred over escaping apostrophes as it is much easier to read.

C.4.2.3. Variable Substitution

Variable substitution is permitted using either of these forms:

```
$greeting = "Hello $name, welcome back!";
```

```
$greeting = "Hello {$name}, welcome back!";
```

For consistency, this form is not permitted:

```
$greeting = "Hello ${name}, welcome back!";
```

C.4.2.4. String Concatenation

Strings must be concatenated using the "." operator. A space must always be added before and after the "." operator to improve readability:

```
$company = 'Zend' . ' ' . 'Technologies';
```

When concatenating strings with the "." operator, it is encouraged to break the statement into multiple lines to improve readability. In these cases, each successive line should be padded with white space such that the "."; operator is aligned under the "=" operator:

```
$sql = "SELECT `id`, `name` FROM `people` "
     . "WHERE `name` = 'Susan' "
     . "ORDER BY `name` ASC ";
```

C.4.3. Arrays

C.4.3.1. Numerically Indexed Arrays

Negative numbers are not permitted as indices.

An indexed array may start with any non-negative number, however all base indices besides 0 are discouraged.

When declaring indexed arrays with the Array function, a trailing space must be added after each comma delimiter to improve readability:

```
$sampleArray = array(1, 2, 3, 'Zend', 'Studio');
```

It is permitted to declare multi-line indexed arrays using the "array" construct. In this case, each successive line must be padded with spaces such that beginning of each line is aligned:

```
$sampleArray = array(1, 2, 3, 'Zend', 'Studio',
                     $a, $b, $c,
                     56.44, $d, 500);
```

Alternately, the initial array item may begin on the following line. If so, it should be padded at one indentation level greater than the line containing the array declaration, and all successive lines should have the same indentation; the closing paren should be on a line by itself at the same indentation level as the line containing the array declaration:

```
$sampleArray = array(
    1, 2, 3, 'Zend', 'Studio',
    $a, $b, $c,
    56.44, $d, 500,
);
```

When using this latter declaration, we encourage using a trailing comma for the last item in the array; this minimizes the impact of adding new items on successive lines, and helps to ensure no parse errors occur due to a missing comma.

C.4.3.2. Associative Arrays

When declaring associative arrays with the Array construct, breaking the statement into multiple lines is encouraged. In this case, each successive line must be padded with white space such that both the keys and the values are aligned:

```
$sampleArray = array('firstKey'  => 'firstValue',
                     'secondKey' => 'secondValue');
```

Alternately, the initial array item may begin on the following line. If so, it should be padded at one indentation level greater than the line containing the array declaration, and all successive lines should have the same indentation; the closing paren should be on a line by itself at the same indentation level as the line containing the array declaration. For readability, the various "=>" assignment operators should be padded such that they align.

```
$sampleArray = array(
    'firstKey'  => 'firstValue',
    'secondKey' => 'secondValue',
);
```

When using this latter declaration, we encourage using a trailing comma for the last item in the array; this minimizes the impact of adding new items on successive lines, and helps to ensure no parse errors occur due to a missing comma.

C.4.4. Classes

C.4.4.1. Class Declaration

Classes must be named according to Zend Framework's naming conventions.

The brace should always be written on the line underneath the class name.

Every class must have a documentation block that conforms to the PHPDocumentor standard.

All code in a class must be indented with four spaces.

Only one class is permitted in each PHP file.

Placing additional code in class files is permitted but discouraged. In such files, two blank lines must separate the class from any additional PHP code in the class file.

The following is an example of an acceptable class declaration:

```
/**
 * Documentation Block Here
 */
class SampleClass
{
    // all contents of class
    // must be indented four spaces
}
```

Classes that extend other classes or which implement interfaces should declare their dependencies on the same line when possible.

```
class SampleClass extends FooAbstract implements BarInterface
{
}
```

If as a result of such declarations, the line length exceeds the maximum line length, break the line before the "extends" and/or "implements" keywords, and pad those lines by one indentation level.

```
class SampleClass
    extends FooAbstract
    implements BarInterface
{
}
```

If the class implements multiple interfaces and the declaration exceeds the maximum line length, break after each comma separating the interfaces, and indent the interface names such that they align.

```
class SampleClass
    implements BarInterface,
               BazInterface
{
}
```

C.4.4.2. Class Member Variables

Member variables must be named according to Zend Framework's variable naming conventions.

Any variables declared in a class must be listed at the top of the class, above the declaration of any methods.

The *var* construct is not permitted. Member variables always declare their visibility by using one of the private, protected, or public modifiers. Giving access to member variables directly by declaring them as public is permitted but discouraged in favor of accessor methods (set & get).

C.4.5. Functions and Methods

C.4.5.1. Function and Method Declaration

Functions must be named according to Zend Framework's function naming conventions.

Methods inside classes must always declare their visibility by using one of the private, protected, or public modifiers.

As with classes, the brace should always be written on the line underneath the function name. Space between the function name and the opening parenthesis for the arguments is not permitted.

Functions in the global scope are strongly discouraged.

The following is an example of an acceptable function declaration in a class:

```
/**
 * Documentation Block Here
 */
```

```
class Foo
{
    /**
     * Documentation Block Here
     */
    public function bar()
    {
        // all contents of function
        // must be indented four spaces
    }
}
```

In cases where the argument list exceeds the maximum line length, you may introduce line breaks. Additional arguments to the function or method must be indented one additional level beyond the function or method declaration. A line break should then occur before the closing argument paren, which should then be placed on the same line as the opening brace of the function or method with one space separating the two, and at the same indentation level as the function or method declaration. The following is an example of one such situation:

```
/**
 * Documentation Block Here
 */
class Foo
{
    /**
     * Documentation Block Here
     */
    public function bar($arg1, $arg2, $arg3,
        $arg4, $arg5, $arg6
    ) {
        // all contents of function
        // must be indented four spaces
    }
}
```

Note: Pass-by-reference is the only parameter passing mechanism permitted in a method declaration.

```
/**
 * Documentation Block Here
 */
class Foo
{
    /**
     * Documentation Block Here
     */
    public function bar(&$baz)
    {}
}
```

Call-time pass-by-reference is strictly prohibited.

The return value must not be enclosed in parentheses. This can hinder readability, in additional to breaking code if a method is later changed to return by reference.

```
/**
```

```
 * Documentation Block Here
 */
class Foo
{
    /**
     * WRONG
     */
    public function bar()
    {
        return($this->bar);
    }

    /**
     * RIGHT
     */
    public function bar()
    {
        return $this->bar;
    }
}
```

C.4.5.2. Function and Method Usage

Function arguments should be separated by a single trailing space after the comma delimiter. The following is an example of an acceptable invocation of a function that takes three arguments:

```
threeArguments(1, 2, 3);
```

Call-time pass-by-reference is strictly prohibited. See the function declarations section for the proper way to pass function arguments by-reference.

In passing arrays as arguments to a function, the function call may include the "array" hint and may be split into multiple lines to improve readability. In such cases, the normal guidelines for writing arrays still apply:

```
threeArguments(array(1, 2, 3), 2, 3);

threeArguments(array(1, 2, 3, 'Zend', 'Studio',
                     $a, $b, $c,
                     56.44, $d, 500), 2, 3);

threeArguments(array(
    1, 2, 3, 'Zend', 'Studio',
    $a, $b, $c,
    56.44, $d, 500
), 2, 3);
```

C.4.6. Control Statements

C.4.6.1. If/Else/Elseif

Control statements based on the *if* and *elseif* constructs must have a single space before the opening parenthesis of the conditional and a single space after the closing parenthesis.

Within the conditional statements between the parentheses, operators must be separated by spaces for readability. Inner parentheses are encouraged to improve logical grouping for larger conditional expressions.

The opening brace is written on the same line as the conditional statement. The closing brace is always written on its own line. Any content within the braces must be indented using four spaces.

```
if ($a != 2) {
    $a = 2;
}
```

If the conditional statement causes the line length to exceed the maximum line length and has several clauses, you may break the conditional into multiple lines. In such a case, break the line prior to a logic operator, and pad the line such that it aligns under the first character of the conditional clause. The closing paren in the conditional will then be placed on a line with the opening brace, with one space separating the two, at an indentation level equivalent to the opening control statement.

```
if (($a == $b)
    && ($b == $c)
    || (Foo::CONST == $d)
) {
    $a = $d;
}
```

The intention of this latter declaration format is to prevent issues when adding or removing clauses from the conditional during later revisions.

For "if" statements that include "elseif" or "else", the formatting conventions are similar to the "if" construct. The following examples demonstrate proper formatting for "if" statements with "else" and/or "elseif" constructs:

```
if ($a != 2) {
    $a = 2;
} else {
    $a = 7;
}

if ($a != 2) {
    $a = 2;
} elseif ($a == 3) {
    $a = 4;
} else {
    $a = 7;
}

if (($a == $b)
    && ($b == $c)
    || (Foo::CONST == $d)
) {
    $a = $d;
} elseif (($a != $b)
          || ($b != $c)
) {
    $a = $c;
} else {
    $a = $b;
}
```

PHP allows statements to be written without braces in some circumstances. This coding standard makes no differentiation- all "if", "elseif" or "else" statements must use braces.

C.4.6.2. Switch

Control statements written with the "switch" statement must have a single space before the opening parenthesis of the conditional statement and after the closing parenthesis.

All content within the "switch" statement must be indented using four spaces. Content under each "case" statement must be indented using an additional four spaces.

```
switch ($numPeople) {
    case 1:
        break;

    case 2:
        break;

    default:
        break;
}
```

The construct default should never be omitted from a switch statement.

Note: It is sometimes useful to write a case statement which falls through to the next case by not including a break or return within that case. To distinguish these cases from bugs, any case statement where break or return are omitted should contain a comment indicating that the break was intentionally omitted.

C.4.7. Inline Documentation

C.4.7.1. Documentation Format

All documentation blocks ("docblocks") must be compatible with the phpDocumentor format. Describing the phpDocumentor format is beyond the scope of this document. For more information, visit: http://phpdoc.org/

All class files must contain a "file-level" docblock at the top of each file and a "class-level" docblock immediately above each class. Examples of such docblocks can be found below.

C.4.7.2. Files

Every file that contains PHP code must have a docblock at the top of the file that contains these phpDocumentor tags at a minimum:

```
/**
 * Short description for file
 *
 * Long description for file (if any)...
 *
 * LICENSE: Some license information
 *
 * @category   Zend
 * @package    Zend_Magic
 * @subpackage Wand
 * @copyright  Copyright (c) 2005-2010 Zend Technologies USA Inc. (http://www.zend.com)
 * @license    http://framework.zend.com/license   BSD License
 * @version    $Id:$
 * @link       http://framework.zend.com/package/PackageName
```

```
 * @since      File available since Release 1.5.0
*/
```

The @category annotation must have a value of "Zend".

The @package annotation must be assigned, and should be equivalent to the component name of the class contained in the file; typically, this will only have two segments, the "Zend" prefix, and the component name.

The @subpackage annotation is optional. If provided, it should be the subcomponent name, minus the class prefix. In the example above, the assumption is that the class in the file is either "`Zend_Magic_Wand`", or uses that classname as part of its prefix.

C.4.7.3. Classes

Every class must have a docblock that contains these phpDocumentor tags at a minimum:

```
/**
 * Short description for class
 *
 * Long description for class (if any)...
 *
 * @category   Zend
 * @package    Zend_Magic
 * @subpackage Wand
 * @copyright  Copyright (c) 2005-2010 Zend Technologies USA Inc. (http://www.zend.com)
 * @license    http://framework.zend.com/license   BSD License
 * @version    Release: @package_version@
 * @link       http://framework.zend.com/package/PackageName
 * @since      Class available since Release 1.5.0
 * @deprecated Class deprecated in Release 2.0.0
 */
```

The @category annotation must have a value of "Zend".

The @package annotation must be assigned, and should be equivalent to the component to which the class belongs; typically, this will only have two segments, the "Zend" prefix, and the component name.

The @subpackage annotation is optional. If provided, it should be the subcomponent name, minus the class prefix. In the example above, the assumption is that the class described is either "`Zend_Magic_Wand`", or uses that classname as part of its prefix.

C.4.7.4. Functions

Every function, including object methods, must have a docblock that contains at a minimum:

- A description of the function
- All of the arguments
- All of the possible return values

It is not necessary to use the "@access" tag because the access level is already known from the "public", "private", or "protected" modifier used to declare the function.

If a function or method may throw an exception, use @throws for all known exception classes:

```
@throws exceptionclass [description]
```

Appendix D. Zend Framework Documentation Standard

Table of Contents

D.1. Overview

D.1.1. Scope

This document provides guidelines for creation of the end-user documentation found within Zend Framework. It is intended as a guide to Zend Framework contributors, who must write documentation as part of component contributions, as well as to documentation translators. The standards contained herein are intended to ease translation of documentation, minimize visual and stylistic differences between different documentation files, and make finding changes in documentation easier with **diff** tools.

Topics covered in Zend Framework's documentation standards include documentation file formatting and recommendations for documentation quality.

D.2. Documentation File Formatting

D.2.1. XML Tags

Each manual file must include the following XML declarations at the top of the file:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Reviewed: no -->
```

XML files from translated languages must also include a revision tag containing the revision of the corresponding English-language file the translation was based on.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- EN-Revision: 14978 -->
<!-- Reviewed: no -->
```

D.2.2. Maximum Line Length

The maximum line length, including tags, attributes, and indentation, is not to exceed 100 characters. There is only one exception to this rule: attribute and value pairs are allowed to exceed the 100 chars as they are not allowed to be separated.

D.2.3. Indentation

Indentation should consist of 4 spaces. Tabs are not allowed.

Tags which are at the same level must have the same indentation.

```
<sect1>
</sect1>

<sect1>
</sect1>
```

Tags which are one level under the previous tag must be indented with 4 additional spaces.

```
<sect1>
    <sect2>
    </sect2>
</sect1>
```

Multiple block tags within the same line are not allowed; multiple inline tags are allowed, however.

```
<!-- NOT ALLOWED: -->
<sect1><sect2>
</sect2></sect1>

<!-- ALLOWED -->
<para>
    <classname>Zend_Magic</classname> does not exist. <classname>Zend_Acl</classname> does.
</para>
```

D.2.4. Line Termination

Line termination follows the Unix text file convention. Lines must end with a single linefeed (LF) character. Linefeed characters are represented as ordinal 10, or hexadecimal 0x0A.

Note: Do not use carriage returns (CR) as is the convention in Apple OS's (0x0D) or the carriage return - linefeed combination (CRLF) as is standard for the Windows OS (0x0D, 0x0A).

D.2.5. Empty tags

Empty tags are not allowed; all tags must contain text or child tags.

```
<!-- NOT ALLOWED -->
<para>
    Some text. <link></link>
</para>

<para>
</para>
```

D.2.6. Usage of whitespace within documents

D.2.6.1. Whitespace within tags

Opening block tags should have no whitespace immediately following them other than line breaks (and indentation on the following line).

```
<!-- NOT ALLOWED -->
<sect1>WHITESPACE
</sect1>
```

Opening inline tags should have no whitespace immediately following them.

```
<!-- NOT ALLOWED -->
This is the class <classname> Zend_Class</classname>.

<!-- OK -->
This is the class <classname>Zend_Class</classname>.
```

Closing block tags may be preceded by whitespace equivalent to the current indentation level, but no more than that amount.

```
<!-- NOT ALLOWED -->
    <sect1>
     </sect1>

<!-- OK -->
    <sect1>
    </sect1>
```

Closing inline tags must not be preceded by any whitespace.

```
<!-- NOT ALLOWED -->
This is the class <classname>Zend_Class </classname>

<!-- OK -->
This is the class <classname>Zend_Class</classname>
```

D.2.6.2. Multiple line breaks

Multiple line breaks within or between tags are not allowed.

```
<!-- NOT ALLOWED -->
<para>
    Some text...

    ... and more text
</para>
```

```
<para>
    Another paragraph.
</para>

<!-- OK -->
<para>
    Some text...
    ... and more text
</para>

<para>
    Another paragraph.
</para>
```

D.2.6.3. Separation between tags

Tags at the same level must be separated by an empty line to improve readability.

```
<!-- NOT ALLOWED -->
<para>
    Some text...
</para>
<para>
    More text...
</para>

<!-- OK -->
<para>
    Some text...
</para>

<para>
    More text...
</para>
```

The first child tag should open directly below its parent, with no empty line between them; the last child tag should close directly before the closing tag of its parent.

```
<!-- NOT ALLOWED -->
<sect1>

    <sect2>
    </sect2>

    <sect2>
    </sect2>

    <sect2>
    </sect2>

</sect1>

<!-- OK -->
<sect1>
    <sect2>
    </sect2>

    <sect2>
```

```
    </sect2>

    <sect2>
    </sect2>
</sect1>
```

D.2.7. Program Listings

The opening *<programlisting>* tag must indicate the appropriate "language" attribute and be indented at the same level as its sibling blocks.

```
<para>Sibling paragraph.</para>

<programlisting language="php"><![CDATA[
```

CDATA should be used around all program listings.

<programlisting> sections must not add linebreaks or whitespace at the beginning or end of the section, as these are then represented in the final output.

```
<!-- NOT ALLOWED -->
<programlisting language="php"><![CDATA[

$render = "xxx";

]]></programlisting>

<!-- OK -->
<programlisting language="php"><![CDATA[
$render = "xxx";
]]></programlisting>
```

Ending CDATA and *<programlisting>* tags should be on the same line, without any indentation.

```
<!-- NOT ALLOWED -->
    <programlisting language="php"><![CDATA[
$render = "xxx";
]]>
    </programlisting>

<!-- NOT ALLOWED -->
    <programlisting language="php"><![CDATA[
$render = "xxx";
    ]]></programlisting>

<!-- OK -->
    <programlisting language="php"><![CDATA[
$render = "xxx";
]]></programlisting>
```

The *<programlisting>* tag should contain the "language" attribute with a value appropriate to the contents of the program listing. Typical values include "css", "html", "ini", "javascript", "php", "text", and "xml".

```
<!-- PHP -->
<programlisting language="php"><![CDATA[
```

```
<!-- Javascript -->
<programlisting language="javascript"><![CDATA[

<!-- XML -->
<programlisting language="xml"><![CDATA[
```

For program listings containing only PHP code, PHP tags (e.g., "<?php", "?>") are not required, and should not be used. They simply clutter the narrative, and are implied by the use of the *<programlisting>* tag.

```
<!-- NOT ALLOWED -->
<programlisting language="php"<![CDATA[<?php
    // ...
?>]]></programlisting>

<programlisting language="php"<![CDATA[
<?php
    // ...
?>
]]></programlisting>
```

Line lengths within program listings should follow the coding standards recommendations.

Refrain from using `require_once()`, `require()`, `include_once()`, and `include()` calls within PHP listings. They simply clutter the narrative, and are largely obviated when using an autoloader. Use them only when they are essential to the example.

Never use short tags

Short tags (e.g., "<?", "<?=") should never be used within *programlisting* or the narrative of a document.

D.2.8. Notes on specific inline tags

D.2.8.1. classname

The tag *<classname>* must be used each time a class name is represented by itself; it should not be used when combined with a method name, variable name, or constant, and no other content is allowed within the tag.

```
<para>
    The class <classname>Zend_Class</classname>.
</para>
```

D.2.8.2. varname

Variables must be wrapped in the *<varname>* tag. Variables must be written using the "$" sigil. No other content is allowed within this tag, unless a class name is used, which indicates a class variable.

```
<para>
    The variable <varname>$var</varname> and the class variable
    <varname>Zend_Class::$var</varname>.
</para>
```

D.2.8.3. methodname

Methods must be wrapped in the *<methodname>* tag. Methods must either include the full method signature or at the least a pair of closing parentheses (e.g., "()"). No other content is allowed within this tag, unless a class name is used, which indicates a class method.

```
<para>
    The method <methodname>foo()</methodname> and the class method
    <methodname>Zend_Class::foo()</methodname>. A method with a full signature:
    <methodname>foo($bar, $baz)</methodname>
</para>
```

D.2.8.4. constant

Use the *<constant>* tag when denoting constants. Constants must be written in UPPERCASE. No other content is allowed within this tag, unless a class name is used, which indicates a class constant.

```
<para>
    The constant <constant>FOO</constant> and the class constant
    <constant>Zend_Class::FOO</constant>.
</para>
```

D.2.8.5. filename

Filenames and paths must be wrapped in the *<filename>* tag. No other content is allowed in this tag.

```
<para>
    The filename <filename>application/Bootstrap.php</filename>.
</para>
```

D.2.8.6. command

Commands, shell scripts, and program calls must be wrapped in the *<command>* tag. If the command includes arguments, these should also be included within the tag.

```
<para>
    Execute <command>zf.sh create project</command>.
</para>
```

D.2.8.7. code

Usage of the *<code>* tag is discouraged, in favor of the other inline tasks discussed previously.

D.2.9. Notes on specific block tags

D.2.9.1. title

The *<title>* tag is not allowed to hold other tags.

```
<!-- NOT ALLOWED -->
<title>Using <classname>Zend_Class</classname></title>

<!-- OK -->
```

```
<title>Using Zend_Class</title>
```

D.3. Recommendations

D.3.1. Use editors without autoformatting

For editing the documentation, typically you should not use formal XML editors. Such editors normally autoformat existing documents to fit their own standards and/or do not strictly follow the docbook standard. As examples, we have seen them erase the CDATA tags, change 4 space separation to tabs or 2 spaces, etc.

The style guidelines were written in large part to assist translators in recognizing the lines that have changed using normal **diff** tools. Autoformatting makes this process more difficult.

D.3.2. Use Images

Good images and diagrams can improve readability and comprehension. Use them whenever they will assist in these goals. Images should be placed in the `documentation/manual/en/figures/` directory, and be named after the section identifier in which they occur.

D.3.3. Use Case Examples

Look for good use cases submitted by the community, especially those posted in proposal comments or on one of the mailing lists. Examples often illustrate usage far better than the narrative does.

When writing your examples for inclusion in the manual, follow all coding standards and documentation standards.

D.3.4. Avoid Replicating phpdoc Contents

The manual is intended to be a reference guide for end-user usage. Replicating the phpdoc documentation for internal-use components and classes is not wanted, and the narrative should be focussed on usage, not the internal workings. In any case, at this time, we would like the documentation teams to focus on translating the English manual, not the phpdoc comments.

D.3.5. Use Links

Link to other sections of the manual or to external sources instead of recreating documentation.

Linking to other sections of the manual may be done using either the <*xref*> tag (which will substitute the section title for the link text) or the <*link*> tag (to which you must provide link text).

```
<para>
    "Xref" links to a section: <xref
        linkend="doc-standard.recommendations.links" />.
</para>

<para>
    "Link" links to a section, using descriptive text: <link
        linkend="doc-standard.recommendations.links">documentation on
        links</link>.
</para>
```

To link to an external resource, use <*ulink*>:

```
<para>
    The <ulink url="http://framework.zend.com/">Zend Framework site</ulink>.
</para>
```

Appendix E. Recommended Project Structure for Zend Framework MVC Applications

Table of Contents

E.1. Overview

Many developers seek guidance on the best project structure for a Zend Framework project in a relatively flexible environment. A "flexible" environment is one in which the developer can manipulate their file systems and web server configurations as needed to achieve the most ideal project structure to run and secure their application. The default project structure will assume that the developer has such flexibility at their disposal.

The following directory structure is designed to be maximally extensible for complex projects, while providing a simple subset of folder and files for project with simpler requirements. This structure also works without alteration for both modular and non-modular Zend Framework applications. The `.htaccess` files require URL rewrite functionality in the web server as described in the Rewrite Configuration Guide, also included in this appendix.

It is not the intention that this project structure will support all possible Zend Framework project requirements. The default project profile used by `Zend_Tool` reflect this project structure, but applications with requirements not supported by this structure should use a custom project profile.

E.2. Recommended Project Directory Structure

```
<project name>/
    application/
        configs/
            application.ini
        controllers/
            helpers/
        forms/
        layouts/
            filters/
            helpers/
            scripts/
        models/
        modules/
        services/
        views/
            filters/
```

```
            helpers/
            scripts/
        Bootstrap.php
    data/
        cache/
        indexes/
        locales/
        logs/
        sessions/
        uploads/
    docs/
    library/
    public/
        css/
        images/
        js/
        .htaccess
        index.php
    scripts/
        jobs/
        build/
    temp/
    tests/
```

The following describes the use cases for each directory as listed.

- *`application/`*: This directory contains your application. It will house the MVC system, as well as configurations, services used, and your bootstrap file.
    - *`configs/`*: The application-wide configuration directory.
    - *`controllers/`*, *`models/`*, and *`views/`*: These directories serve as the default controller, model or view directories. Having these three directories inside the application directory provides the best layout for starting a simple project as well as starting a modular project that has global `controllers/models/views`.
    - *`controllers/helpers/`*: These directories will contain action helpers. Action helpers will be namespaced either as "`Controller_Helper_`" for the default module or "<Module>_Controller_Helper" in other modules.
    - *`layouts/`*: This layout directory is for MVC-based layouts. Since `Zend_Layout` is capable of MVC- and non-MVC-based layouts, the location of this directory reflects that layouts are not on a 1-to-1 relationship with controllers and are independent of templates within `views/`.
    - *`modules/`*: Modules allow a developer to group a set of related controllers into a logically organized group. The structure under the modules directory would resemble the structure under the application directory.
    - *`services/`*: This directory is for your application specific web-service files that are provided by your application, or for implementing a Service Layer for your models.
    - *`Bootstrap.php`*: This file is the entry point for your application, and should implement `Zend_Application_Bootstrap_Bootstrapper`. The purpose for this file is to bootstrap the application and make components available to the application by initializing them.
- *`data/`*: This directory provides a place to store application data that is volatile and possibly temporary. The disturbance of data in this directory might cause the application to fail. Also, the

information in this directory may or may not be committed to a subversion repository. Examples of things in this directory are session files, cache files, sqlite databases, logs and indexes.

- *`docs/`*: This directory contains documentation, either generated or directly authored.
- *`library/`*: This directory is for common libraries on which the application depends, and should be on the PHP include_path. Developers should place their application's library code under this directory in a unique namespace, following the guidelines established in the PHP manual's Userland Naming Guide, as well as those established by Zend itself. This directory may also include Zend Framework itself; if so, you would house it in `library/Zend/`.
- *`public/`*: This directory contains all public files for your application. `index.php` sets up and invokes `Zend_Application`, which in turn invokes the `application/Bootstrap.php` file, resulting in dispatching the front controller. The web root of your web server would typically be set to this directory.
- *`scripts/`*: This directory contains maintenance and/or build scripts. Such scripts might include command line, cron, or phing build scripts that are not executed at runtime but are part of the correct functioning of the application.
- *`temp/`*: The `temp/` folder is set aside for transient application data. This information would not typically be committed to the applications svn repository. If data under the `temp/` directory were deleted, the application should be able to continue running with a possible decrease in performance until data is once again restored or recached.
- *`tests/`*: This directory contains application tests. These could be hand-written, PHPUnit tests, Selenium-RC based tests or based on some other testing framework. By default, library code can be tested by mimicing the directory structure of your `library/` directory. Additionally, functional tests for your application could be written mimicing the `application/` directory structure (including the application subdirectory).

E.3. Module Structure

The directory structure for modules should mimic that of the `application/` directory in the recommended project structure:

```
<modulename>
    configs/
        application.ini
    controllers/
        helpers/
    forms/
    layouts/
        filters/
        helpers/
        scripts/
    models/
    services/
    views/
        filters/
        helpers/
        scripts/
    Bootstrap.php
```

The purpose of these directories remains exactly the same as for the recommended project directory structure.

E.4. Rewrite Configuration Guide

URL rewriting is a common function of HTTP servers. However, the rules and configuration differ widely between them. Below are some common approaches across a variety of popular web servers available at the time of writing.

E.4.1. Apache HTTP Server

All examples that follow use mod_rewrite, an official module that comes bundled with Apache. To use it, mod_rewrite must either be included at compile time or enabled as a Dynamic Shared Object (DSO). Please consult the Apache documentation for your version for more information.

E.4.1.1. Rewriting inside a VirtualHost

Here is a very basic virtual host definition. These rules direct all requests to `index.php`, except when a matching file is found under the document_root.

```
<VirtualHost my.domain.com:80>
    ServerName   my.domain.com
    DocumentRoot /path/to/server/root/my.domain.com/public

    RewriteEngine off

    <Location />
        RewriteEngine On
        RewriteCond %{REQUEST_FILENAME} -s [OR]
        RewriteCond %{REQUEST_FILENAME} -l [OR]
        RewriteCond %{REQUEST_FILENAME} -d
        RewriteRule ^.*$ - [NC,L]
        RewriteRule ^.*$ /index.php [NC,L]
    </Location>
</VirtualHost>
```

Note the slash ("/") prefixing `index.php`; the rules for `.htaccess` differ in this regard.

E.4.1.2. Rewriting within a .htaccess file

Below is a sample `.htaccess` file that utilizes mod_rewrite. It is similar to the virtual host configuration, except that it specifies only the rewrite rules, and the leading slash is omitted from `index.php`.

```
RewriteEngine On
RewriteCond %{REQUEST_FILENAME} -s [OR]
RewriteCond %{REQUEST_FILENAME} -l [OR]
RewriteCond %{REQUEST_FILENAME} -d
RewriteRule ^.*$ - [NC,L]
RewriteRule ^.*$ index.php [NC,L]
```

There are many ways to configure mod_rewrite; if you would like more information, see Jayson Minard's Blueprint for PHP Applications: Bootstrapping.

E.4.2. Microsoft Internet Information Server

As of version 7.0, IIS now ships with a standard rewrite engine. You may use the following configuration to create the appropriate rewrite rules.

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
    <system.webServer>
        <rewrite>
            <rules>
                <rule name="Imported Rule 1" stopProcessing="true">
                    <match url="^.*$" />
                    <conditions logicalGrouping="MatchAny">
                        <add input="{REQUEST_FILENAME}"
                             matchType="IsFile" pattern=""
                             ignoreCase="false" />
                        <add input="{REQUEST_FILENAME}"
                             matchType="IsDirectory"
                             pattern=""
                             ignoreCase="false" />
                    </conditions>
                    <action type="None" />
                </rule>
                <rule name="Imported Rule 2" stopProcessing="true">
                    <match url="^.*$" />
                    <action type="Rewrite" url="index.php" />
                </rule>
            </rules>
        </rewrite>
    </system.webServer>
</configuration>
```

Appendix F. Zend Framework Performance Guide

Table of Contents

F.1. Introduction

The purpose of this appendix is to provide some concrete strategies for improving the performance of your Zend Framework applications. The guide is presented in a "Question and Answer" format, and broken into areas of concern.

F.2. Class Loading

Anyone who ever performs profiling of a Zend Framework application will immediately recognize that class loading is relatively expensive in Zend Framework. Between the sheer number of class files that need to be loaded for many components, to the use of plugins that do not have a 1:1 relationship between their class name and the file system, the various calls to `include_once()` and `require_once()` can be problematic. This chapter intends to provide some concrete solutions to these issues.

F.2.1. How can I optimize my include_path?

One trivial optimization you can do to increase the speed of class loading is to pay careful attention to your include_path. In particular, you should do four things: use absolute paths (or paths relative to absolute paths), reduce the number of include paths you define, have your Zend Framework include_path as early as possible, and only include the current directory path at the end of your include_path.

F.2.1.1. Use absolute paths

While this may seem a micro-optimization, the fact is that if you don't, you'll get very little benefit from PHP's realpath cache, and as a result, opcode caching will not perform nearly as you may expect.

There are two easy ways to ensure this. First, you can hardcode the paths in your `php.ini`, `httpd.conf`, or `.htaccess`. Second, you can use PHP's `realpath()` function when setting your include_path:

```
$paths = array(
    realpath(dirname(__FILE__) . '/../library'),
    '.',
);
set_include_path(implode(PATH_SEPARATOR, $paths);
```

You *can* use relative paths -- so long as they are relative to an absolute path:

```
define('APPLICATION_PATH', realpath(dirname(__FILE__)));
$paths = array(
    APPLICATION_PATH . '/../library'),
    '.',
);
set_include_path(implode(PATH_SEPARATOR, $paths);
```

However, even so, it's typically a trivial task to simply pass the path to `realpath()`.

F.2.1.2. Reduce the number of include paths you define

Include paths are scanned in the order in which they appear in the include_path. Obviously, this means that you'll get a result faster if the file is found on the first scan rather than the last. Thus, a rather obvious enhancement is to simply reduce the number of paths in your include_path to only what you need. Look through each include_path you've defined, and determine if you actually have any functionality in that path that is used in your application; if not, remove it.

Another optimization is to combine paths. For instance, Zend Framework follows PEAR naming conventions; thus, if you are using PEAR libraries (or libraries from another framework or component library that follows PEAR CS), try to put all of these libraries on the same include_path. This can often be achieved by something as simple as symlinking one or more libraries into a common directory.

F.2.1.3. Define your Zend Framework include_path as early as possible

Continuing from the previous suggestion, another obvious optimization is to define your Zend Framework include_path as early as possible in your include_path. In most cases, it should be the first path in the list. This ensures that files included from Zend Framework are found on the first scan.

F.2.1.4. Define the current directory last, or not at all

Most include_path examples show using the current directory, or '.'. This is convenient for ensuring that scripts in the same directory as the file requiring them can be loaded. However, these same examples typically show this path item as the first item in the include_path -- which means that the current directory tree is always scanned first. In most cases, with Zend Framework applications, this is not desired, and the path may be safely pushed to the last item in the list.

Example F.1. Example: Optimized include_path

Let's put all of these suggestions together. Our assumption will be that you are using one or more PEAR libraries in conjunction with Zend Framework -- perhaps the PHPUnit and `Archive_Tar` libraries -- and that you occasionally need to include files relative to the current file.

First, we'll create a library directory in our project. Inside that directory, we'll symlink our Zend Framework's `library/Zend` directory, as well as the necessary directories from our PEAR installation:

```
library
    Archive/
    PEAR/
    PHPUnit/
    Zend/
```

This allows us to add our own library code if necessary, while keeping shared libraries intact.

Next, we'll opt to create our include_path programmatically within our `public/index.php` file. This allows us to move our code around on the file system, without needing to edit the include_path every time.

We'll borrow ideas from each of the suggestions above: we'll use absolute paths, as determined using `realpath()`; we'll include Zend Framework's include path early; we've already consolidated include_paths; and we'll put the current directory as the last path. In fact, we're doing really well here -- we're going to end up with only two paths.

```
$paths = array(
    realpath(dirname(__FILE__) . '/../library'),
    '.'
);
set_include_path(implode(PATH_SEPARATOR, $paths));
```

F.2.2. How can I eliminate unnecessary require_once statements?

Lazy loading is an optimization technique designed to push the expensive operation of loading a class file until the last possible moment -- i.e., when instantiating an object of that class, calling a static class method, or referencing a class constant or static property. PHP supports this via autoloading, which allows you to define one or more callbacks to execute in order to map a class name to a file.

However, most benefits you may reap from autoloading are negated if your library code is still performing `require_once()` calls -- which is precisely the case with Zend Framework. So, the question is: how can you eliminate those `require_once()` calls in order to maximize autoloader performance?

F.2.2.1. Strip require_once calls with find and sed

An easy way to strip `require_once()` calls is to use the UNIX utilities 'find' and 'sed' in conjunction to comment out each call. Try executing the following statements (where '%' indicates the shell prompt):

```
% cd path/to/ZendFramework/library
% find . -name '*.php' -not -wholename '*/Loader/Autoloader.php' \
```

```
  -not -wholename '*/Application.php' -print0 | \
  xargs -0 sed --regexp-extended --in-place 's/(require_once)/\/\/ \1/g'
```

This one-liner (broken into two lines for readability) iterates through each PHP file and tells it to replace each instance of 'require_once' with '// require_once', effectively commenting out each such statement. (It selectively keeps `require_once()` calls within `Zend_Application` and `Zend_Loader_Autoloader`, as these classes will fail without them.)

This command could be added to an automated build or release process trivially, helping boost performance in your production application. It should be noted, however, that if you use this technique, you *must* utilize autoloading; you can do that from your "`public/index.php`" file with the following code:

```
require_once 'Zend/Loader/Autoloader.php';
Zend_Loader_Autoloader::getInstance();
```

F.2.3. How can I speed up plugin loading?

Many components have plugins, which allow you to create your own classes to utilize with the component, as well as to override existing, standard plugins shipped with Zend Framework. This provides important flexibility to the framework, but at a price: plugin loading is a fairly expensive task.

The plugin loader allows you to register class prefix / path pairs, allowing you to specify class files in non-standard paths. Each prefix can have multiple paths associated with it. Internally, the plugin loader loops through each prefix, and then through each path attached to it, testing to see if the file exists and is readable on that path. It then loads it, and tests to see that the class it is looking for is available. As you might imagine, this can lead to many stat calls on the file system.

Multiply this by the number of components that use the PluginLoader, and you get an idea of the scope of this issue. At the time of this writing, the following components made use of the PluginLoader:

- `Zend_Controller_Action_HelperBroker`: helpers
- `Zend_Dojo`: view helpers, form elements and decorators
- `Zend_File_Transfer`: adapters
- `Zend_Filter_Inflector`: filters (used by the ViewRenderer action helper and `Zend_Layout`)
- `Zend_Filter_Input`: filters and validators
- `Zend_Form`: elements, validators, filters, decorators, captcha and file transfer adapters
- `Zend_Paginator`: adapters
- `Zend_View`: helpers, filters

How can you reduce the number of such calls made?

F.2.3.1. Use the PluginLoader include file cache

Zend Framework 1.7.0 adds an include file cache to the PluginLoader. This functionality writes "`include_once()`" calls to a file, which you can then include in your bootstrap. While this

introduces extra `include_once()` calls to your code, it also ensures that the PluginLoader returns as early as possible.

The PluginLoader documentation includes a complete example of its use.

F.3. Zend_Db Performance

`Zend_Db` is a database abstraction layer, and is intended to provide a common API for SQL operations. `Zend_Db_Table` is a Table Data Gateway, intended to abstract common table-level database operations. Due to their abstract nature and the "magic" they do under the hood to perform their operations, they can sometimes introduce performance overhead.

F.3.1. How can I reduce overhead introduced by Zend_Db_Table for retrieving table metadata?

In order to keep usage as simple as possible, and also to support constantly changing schemas during development, `Zend_Db_Table` does some magic under the hood: on first use, it fetches the table schema and stores it within object members. This operation is typically expensive, regardless of the database -- which can contribute to bottlenecks in production.

Fortunately, there are techniques for improving the situation.

F.3.1.1. Use the metadata cache

`Zend_Db_Table` can optionally utilize `Zend_Cache` to cache table metadata. This is typically faster to access and less expensive than fetching the metadata from the database itself.

The `Zend_Db_Table` documentation includes information on metadata caching.

F.3.1.2. Hardcode your metadata in the table definition

As of 1.7.0, `Zend_Db_Table` also provides support for hardcoding metadata in the table definition. This is an advanced use case, and should only be used when you know the table schema is unlikely to change, or that you're able to keep the definitions up-to-date.

F.3.2. SQL generated with Zend_Db_Select s not hitting my indexes; how can I make it better?

`Zend_Db_Select` is relatively good at its job. However, if you are performing complex queries requiring joins or sub-selects, it can often be fairly naive.

F.3.2.1. Write your own tuned SQL

The only real answer is to write your own SQL; `Zend_Db` does not require the usage of `Zend_Db_Select`, so providing your own, tuned SQL select statements is a perfectly legitimate approach,

Run `EXPLAIN` on your queries, and test a variety of approaches until you can reliably hit your indices in the most performant way -- and then hardcode the SQL as a class property or constant.

If the SQL requires variable arguments, provide placeholders in the SQL, and utilize a combination of `vsprintf()` and `array_walk()` to inject the values into the SQL:

```
// $adapter is the DB adapter. In Zend_Db_Table, retrieve
// it using $this->getAdapter().
$sql = vsprintf(
    self::SELECT_FOO,
    array_walk($values, array($adapter, 'quoteInto'))
);
```

F.4. Internationalization (i18n) and Localization (l10n)

Internationalizing and localizing a site are fantastic ways to expand your audience and ensure that all visitors can get to the information they need. However, it often comes with a performance penalty. Below are some strategies you can employ to reduce the overhead of i18n and l10n.

F.4.1. Which translation adapter should I use?

Not all translation adapters are made equal. Some have more features than others, and some perform better than others. Additionally, you may have business requirements that force you to use a particular adapter. However, if you have a choice, which adapters are fastest?

F.4.1.1. Use non-XML translation adapters for greatest speed

Zend Framework ships with a variety of translation adapters. Fully half of them utilize an XML format, incurring memory and performance overhead. Fortunately, there are several adapters that utilize other formats that can be parsed much more quickly. In order of speed, from fastest to slowest, they are:

- *Array*: this is the fastest, as it is, by definition, parsed into a native PHP format immediately on inclusion.
- *CSV*: uses `fgetcsv()` to parse a CSV file and transform it into a native PHP format.
- *INI*: uses `parse_ini_file()` to parse an INI file and transform it into a native PHP format. This and the CSV adapter are roughly equivalent performance-wise.
- *Gettext*: The gettext adapter from Zend Framework does *not* use the gettext extension as it is not thread safe and does not allow specifying more than one locale per server. As a result, it is slower than using the gettext extension directly, but, because the gettext format is binary, it's faster to parse than XML.

If high performance is one of your concerns, we suggest utilizing one of the above adapters.

F.4.2. How can I make translation and localization even faster?

Maybe, for business reasons, you're limited to an XML-based translation adapter. Or perhaps you'd like to speed things up even more. Or perhaps you want to make l10n operations faster. How can you do this?

F.4.2.1. Use translation and localization caches

Both `Zend_Translate` and `Zend_Locale` implement caching functionality that can greatly affect performance. In the case of each, the major bottleneck is typically reading the files, not the actual lookups; using a cache eliminates the need to read the translation and/or localization files.

You can read about caching of translation and localization strings in the following locations:

- Zend_Translate adapter caching
- Zend_Locale caching

F.5. View Rendering

When using Zend Framework's MVC layer, chances are you will be using Zend_View. Zend_View is performs well compared to other view or templating engines; since view scripts are written in PHP, you do not incur the overhead of compiling custom markup to PHP, nor do you need to worry that the compiled PHP is not optimized. However, Zend_View presents its own issues: extension is done via overloading (view helpers), and a number of view helpers, while carrying out key functionality do so with a performance cost.

F.5.1. How can I speed up resolution of view helpers?

Most Zend_View "methods" are actually provided via overloading to the helper system. This provides important flexibility to Zend_View; instead of needing to extend Zend_View and provide all the helper methods you may utilize in your application, you can define your helper methods in separate classes and consume them at will as if they were direct methods of Zend_View. This keeps the view object itself relatively thin, and ensures that objects are created only when needed.

Internally, Zend_View uses the PluginLoader to look up helper classes. This means that for each helper you call, Zend_View needs to pass the helper name to the PluginLoader, which then needs to determine the class name, load the class file if necessary, and then return the class name so it may be instantiated. Subsequent uses of the helper are much faster, as Zend_View keeps an internal registry of loaded helpers, but if you use many helpers, the calls add up.

The question, then, is: how can you speed up helper resolution?

F.5.1.1. Use the PluginLoader include file cache

The simplest, cheapest solution is the same as for general PluginLoader performance: use the PluginLoader include file cache. Anecdotal evidence has shown this technique to provide a 25-30% performance gain on systems without an opcode cache, and a 40-65% gain on systems with an opcode cache.

F.5.1.2. Extend Zend_View to provide often used helper methods

Another solution for those seeking to tune performance even further is to extend Zend_View to manually add the helper methods they most use in their application. Such helper methods may simply manually instantiate the appropriate helper class and proxy to it, or stuff the full helper implementation into the method.

```
class My_View extends Zend_View
{
    /**
     * @var array Registry of helper classes used
     */
    protected $_localHelperObjects = array();

    /**
     * Proxy to url view helper
     *
     * @param  array $urlOptions Options passed to the assemble method
```

```
     *                            of the Route object.
     * @param  mixed $name The name of a Route to use. If null it will
     *                     use the current Route
     * @param  bool $reset Whether or not to reset the route defaults
     *                     with those provided
     * @return string Url for the link href attribute.
     */
    public function url(array $urlOptions = array(), $name = null,
        $reset = false, $encode = true
    ) {
        if (!array_key_exists('url', $this->_localHelperObjects)) {
            $this->_localHelperObjects['url'] = new Zend_View_Helper_Url();
            $this->_localHelperObjects['url']->setView($this);
        }
        $helper = $this->_localHelperObjects['url'];
        return $helper->url($urlOptions, $name, $reset, $encode);
    }

    /**
     * Echo a message
     *
     * Direct implementation.
     *
     * @param  string $string
     * @return string
     */
    public function message($string)
    {
        return "<h1>" . $this->escape($message) . "</h1>\n";
    }
}
```

Either way, this technique will substantially reduce the overhead of the helper system by avoiding calls to the PluginLoader entirely, and either benefiting from autoloading or bypassing it altogether.

F.5.2. How can I speed up view partials?

Those who use partials heavily and who profile their applications will often immediately notice that the `partial()` view helper incurs a lot of overhead, due to the need to clone the view object. Is it possible to speed this up?

F.5.2.1. Use partial() only when really necessary

The `partial()` view helper accepts three arguments:

- `$name`: the name of the view script to render
- `$module`: the name of the module in which the view script resides; or, if no third argument is provided and this is an array or object, it will be the `$model` argument.
- `$model`: an array or object to pass to the partial representing the clean data to assign to the view.

The power and use of `partial()` come from the second and third arguments. The `$module` argument allows `partial()` to temporarily add a script path for the given module so that the partial view script will resolve to that module; the `$model` argument allows you to explicitly pass

variables for use with the partial view. If you're not passing either argument, *use `render()` instead*!

Basically, unless you are actually passing variables to the partial and need the clean variable scope, or rendering a view script from another MVC module, there is no reason to incur the overhead of `partial()`; instead, use `Zend_View`'s built-in `render()` method to render the view script.

F.5.3. How can I speed up calls to the action() view helper?

Version 1.5.0 introduced the `action()` view helper, which allows you to dispatch an MVC action and capture its rendered content. This provides an important step towards the DRY principle, and promotes code reuse. However, as those who profile their applications will quickly realize, it, too, is an expensive operation. Internally, the `action()` view helper needs to clone new request and response objects, invoke the dispatcher, invoke the requested controller and action, etc.

How can you speed it up?

F.5.3.1. Use the ActionStack when possible

Introduced at the same time as the `action()` view helper, the ActionStack consists of an action helper and a front controller plugin. Together, they allow you to push additional actions to invoke during the dispatch cycle onto a stack. If you are calling `action()` from your layout view scripts, you may want to instead use the ActionStack, and render your views to discrete response segments. As an example, you could write a `dispatchLoopStartup()` plugin like the following to add a login form box to each page:

```
class LoginPlugin extends Zend_Controller_Plugin_Abstract
{
    protected $_stack;

    public function dispatchLoopStartup(
        Zend_Controller_Request_Abstract $request
    ) {
        $stack = $this->getStack();
        $loginRequest = new Zend_Controller_Request_Simple();
        $loginRequest->setControllerName('user')
                     ->setActionName('index')
                     ->setParam('responseSegment', 'login');
        $stack->pushStack($loginRequest);
    }

    public function getStack()
    {
        if (null === $this->_stack) {
            $front = Zend_Controller_Front::getInstance();
            if (!$front->hasPlugin('Zend_Controller_Plugin_ActionStack')) {
                $stack = new Zend_Controller_Plugin_ActionStack();
                $front->registerPlugin($stack);
            } else {
                $stack = $front->getPlugin('ActionStack')
            }
            $this->_stack = $stack;
        }
        return $this->_stack;
    }
}
```

The `UserController::indexAction()` method might then use the `$responseSegment` parameter to indicate which response segment to render to. In the layout script, you would then simply render that response segment:

```
<?php $this->layout()->login ?>
```

While the ActionStack still requires a dispatch cycle, this is still cheaper than the `action()` view helper as it does not need to clone objects and reset internal state. Additionally, it ensures that all pre and post dispatch plugins are invoked, which may be of particular concern if you are using front controller plugins for handling ACL's to particular actions.

F.5.3.2. Favor helpers that query the model over action()

In most cases, using `action()` is simply overkill. If you have most business logic nested in your models and are simply querying the model and passing the results to a view script, it will typically be faster and cleaner to simply write a view helper that pulls the model, queries it, and does something with that information.

As an example, consider the following controller action and view script:

```
class BugController extends Zend_Controller_Action
{
    public function listAction()
    {
        $model = new Bug();
        $this->view->bugs = $model->fetchActive();
    }
}

// bug/list.phtml:
echo "<ul>\n";
foreach ($this->bugs as $bug) {
    printf("<li><b>%s</b>: %s</li>\n",
        $this->escape($bug->id),
        $this->escape($bug->summary)
    );
}
echo "</ul>\n";
```

Using `action()`, you would then invoke it with the following:

```
<?php $this->action('list', 'bug') ?>
```

This could be refactored to a view helper that looks like the following:

```
class My_View_Helper_BugList extends Zend_View_Helper_Abstract
{
    public function bugList()
    {
        $model = new Bug();
        $html  = "<ul>\n";
        foreach ($model->fetchActive() as $bug) {
            $html .= sprintf(
                "<li><b>%s</b>: %s</li>\n",
                $this->view->escape($bug->id),
                $this->view->escape($bug->summary)
            );
```

```
        }
        $html .= "</ul>\n";
        return $html;
    }
}
```

You would then invoke the helper as follows:

```
<?php $this->bugList() ?>
```

This has two benefits: it no longer incurs the overhead of the `action()` view helper, and also presents a more semantically understandable API.

Appendix G. Copyright Information

The following copyrights are applicable to portions of Zend Framework.