Starting “Ethna”

-Web Application Framework-

BoBpp

< *bobpp@as-roma.com* >

前提条件

- PHP 4? 5?
  - PEAR::DB
  - Smarty
- MySQL 4.1
- XHTML1.0 (Strict) + CSS2.1

Agenda

- フレームワークとは
- 各種Web Application Pattern
  - MVC2
  - Front Controller Pattern
- Ethnaとは
- Ethnaチュートリアル
- Q and A

フレームワーク

【名-1】骨組み{ほねぐみ}、フレームワーク、枠組み{わくぐみ}、下部構造{かぶ こうぞう}、骨格{こっかく}

【名-2】構成{こうせい}、体制{たいせい}、組織{そしき}、構造{こうぞう}

What's Framework?

- ビジネスロジックではない、「毎回書くようなこと」の処理
  - 画面遷移
  - フォーム値の精査
  - 認証
  - etc…

これらを担当し、ビジネスロジックの開発に集中させる

Why Need Framework?

- 共同開発による分担の明確化
- コーディング規約による可読性向上
- 共通部分の軽減
- 先人のテクニックを利用

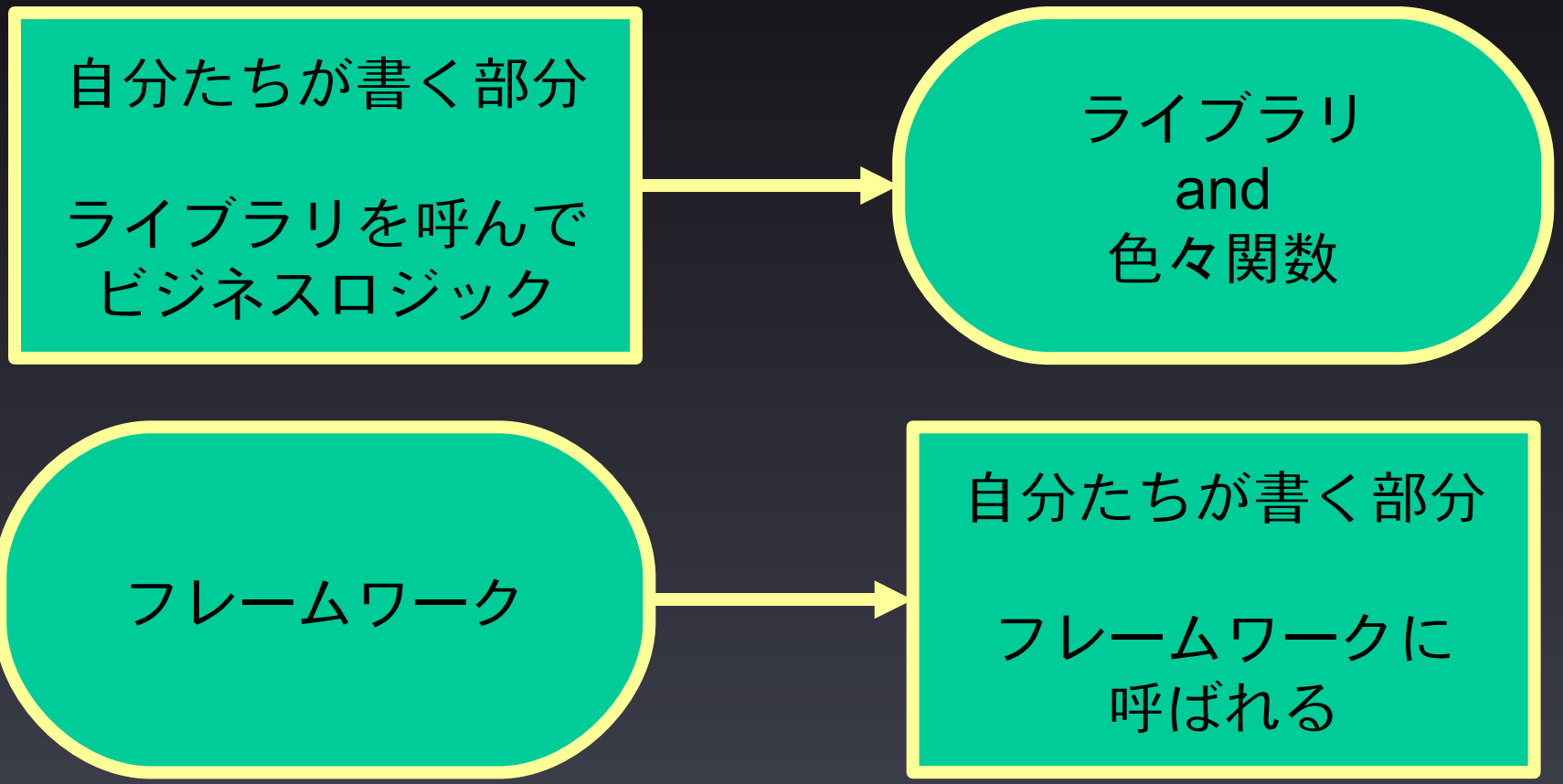
そのかわり

- ラーニングコスト, パフォーマンス

書く内容が変わる

- フレームワーク以前は、呼ぶ部分を書き、自分で出力する。
- フレームワークによるプログラムは、呼ばれる部分を書き、フレームワークに対して何かを返す。

呼ぶ・呼ばれる


自分たちが書く部分
ライブラリを呼んで
ビジネスロジック
ライブラリ
and
色々関数
フレームワーク
自分たちが書く部分
フレームワークに
呼ばれる

ウェブアプリ パターン

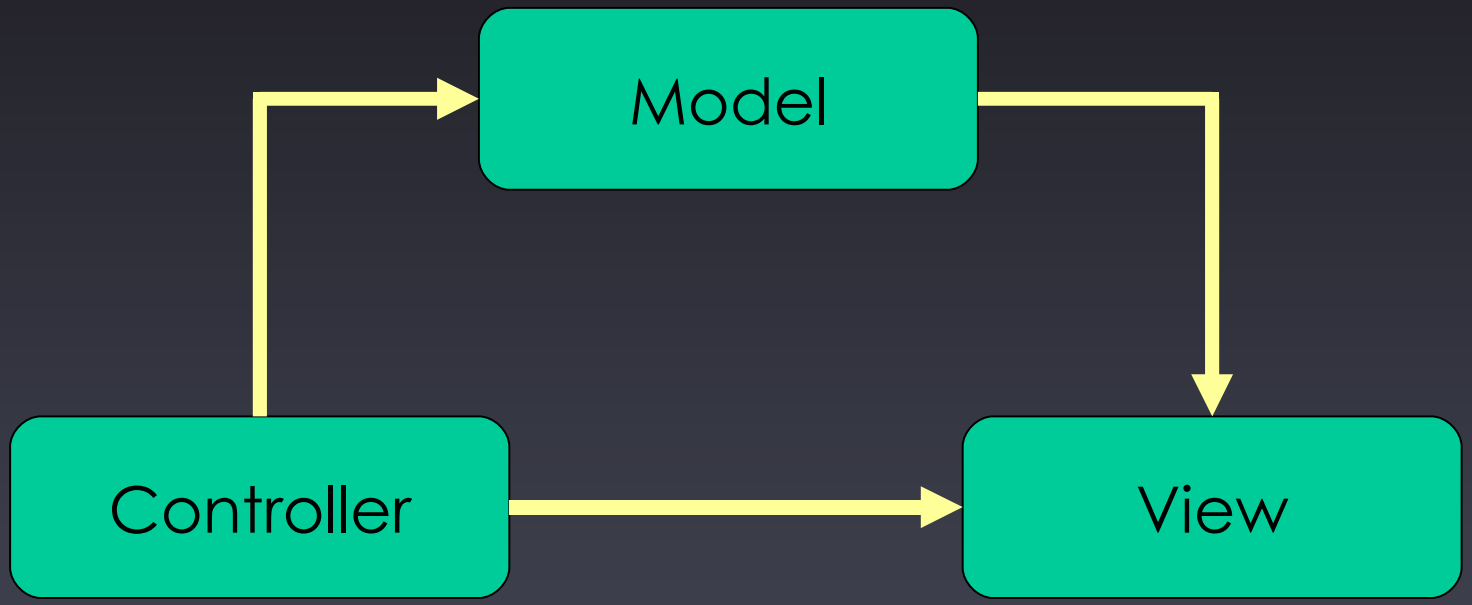
様々な先人の知恵

MVC2

モデルとビューを
コントロールなんだよ
まるでヒントじゃない

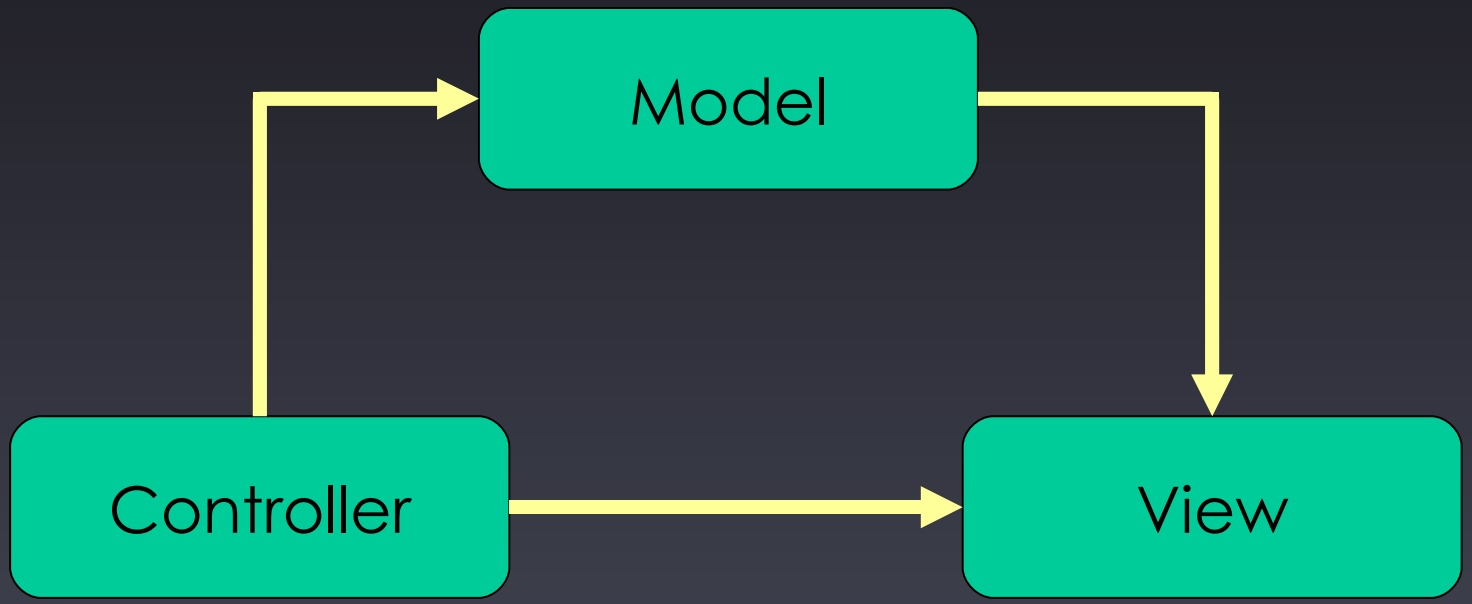
MVC Architecture 2

- アプリケーションを [Model], [View], [Controller] に分ける手法
- 各所の変更に強くなる
- 分担作業が可能になる



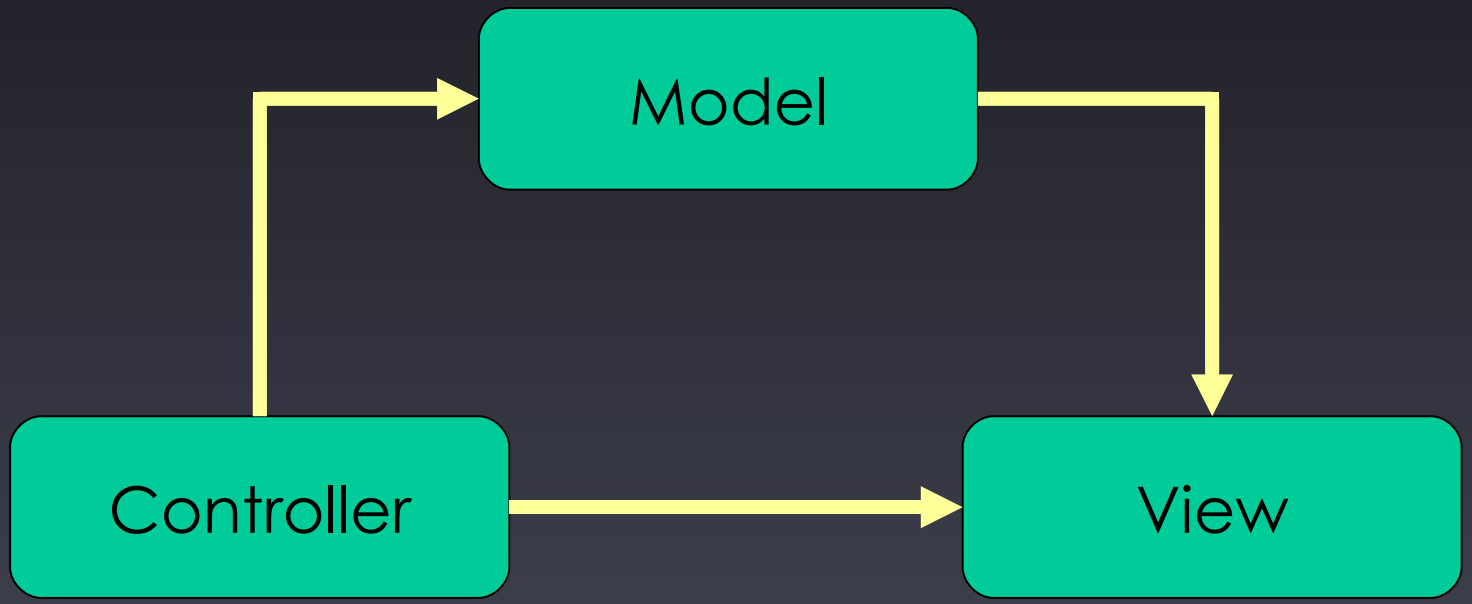
MVC: Controller

- Model, Viewの制御を行う
- 制御のみしか行わない



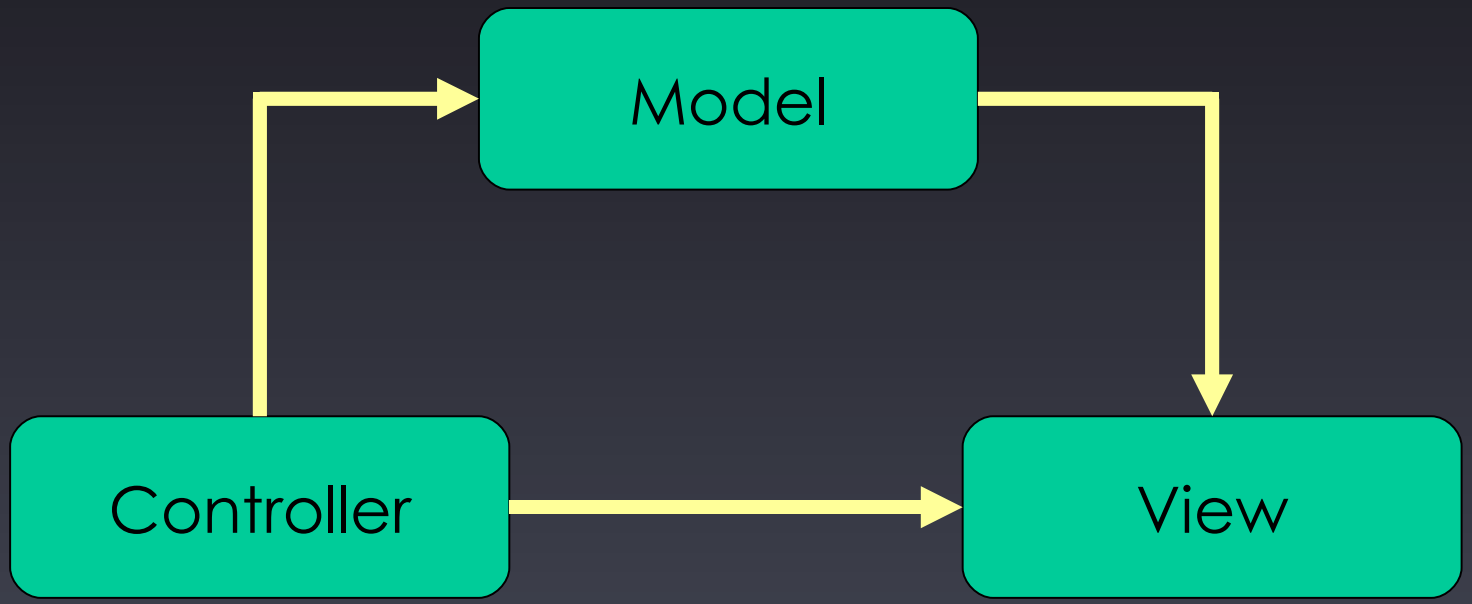
MVC: View

- 表示などの見た目を担当する



MVC: Model

- ビジネスロジックを主に担当する
  DBアクセスもこの部分が中心



Front Controller Pattern

/〜/public_html/index.php
がすべての受け口

index.php

- すべてのリクエストについて index.php　などのファイルを使ってアクセスする
- すべてに必要な処理を一箇所書くだけで済むから便利

Ethna

GREEで使ってるあたりが
アツい！

Ethna…?

- Homepage
  - http://ethna.jp/
- Now Version
  - Stable = 2.1.2
  - Development = 2.3.x
- Developer
  - greeのふじもとさん など

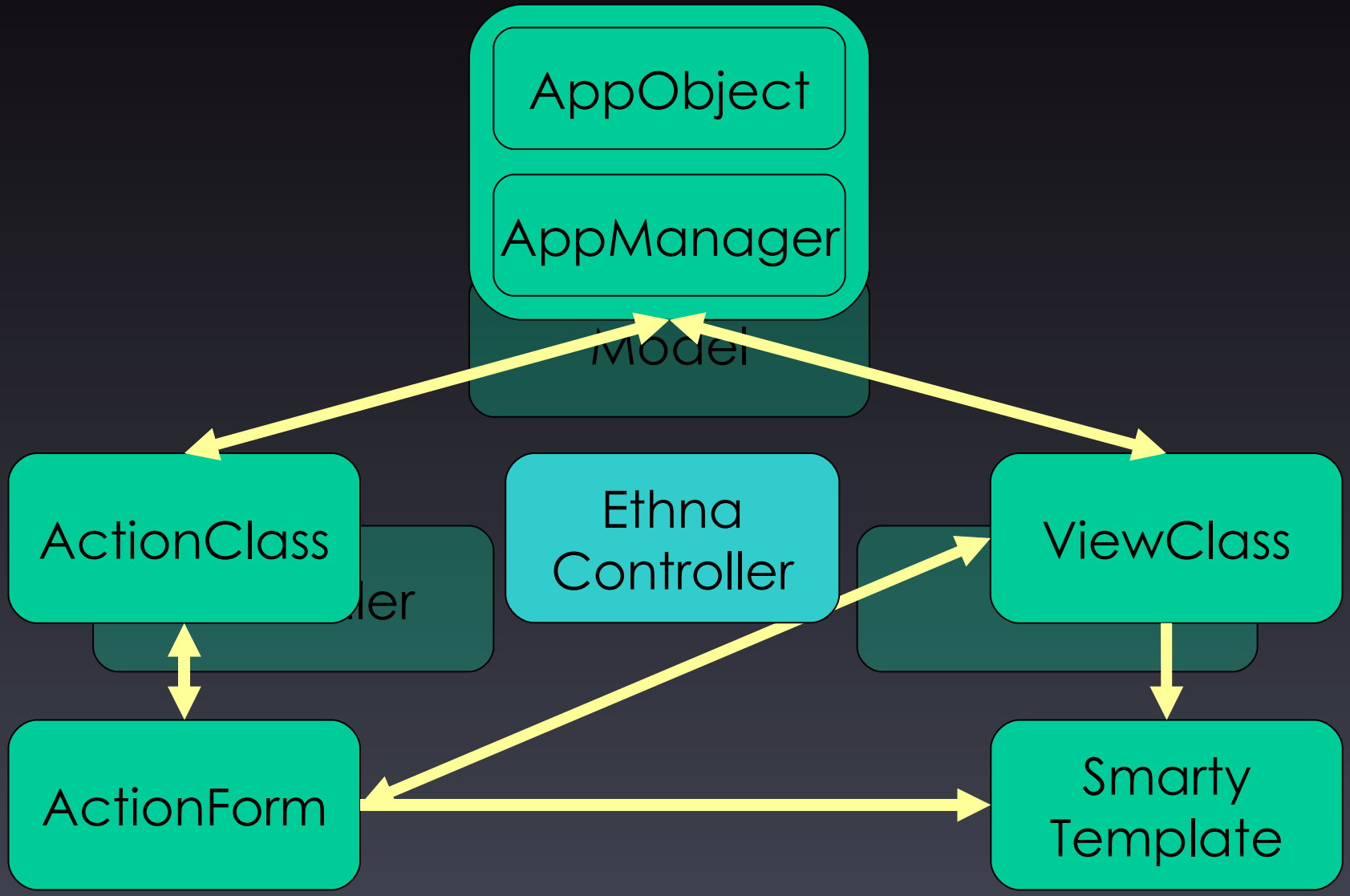
Ethnaの特徴

- フォーム処理
  ActionForm >> QuickForm
- O/R Mapping は ***AppObject***
- AppObjectの管理は ***AppManager***
- これらが連携
- Railsっぽい自動生成機能もあるよ

いくつかの制限

- DB
  - MySQL 4.0系 + MyISAM が基本
  - MySQL 4.1系の文字コード対応は自分で
- PEAR, PEAR::DB, Smarty
  - これらを用いることが一応決まっています
    - DBはADOdb, Creoleを用いることもあるが
      - そろそろ, MDB2とかPDOとかもいくんじゃね？
    - SmartyのViewもRendererで改善されそうだが

MVC on Ethna


AppObject
AppManager
Model
ActionClass
Ethna
Controller
ViewClass
ActionForm
Smarty
Template

Ethna's Flow

Actionをフォーム値から決定する

Actionを処理する

•フォーム値の精査 by ActionForm

•入力値の保存等(ActionForm, AppManager, AppObject利用)

•その結果や入力値による遷移先の画面を決定

View+Templateを処理する

•画面に必要なデータ呼出 (ActionForm, AppManager, AppObject 利用)

•フォーム値, テンプレート変数のセット

•画面表示

Action?

- ユーザの要望
  - これを閲覧したい
  - データの書き込みたい
  - POSTしたのでDBに書き込んでほしい

  これらの単位を***Action***
- コントローラはどのActionを実行すべきか判断している

Action = HTTP Request

- 実装レベルに話を落とすと、
  - HTTP Requestがおきる要因
    - <a> なリンク
    - <form> なサブミット (POST, GET)

  などを***Action***と扱います
- どういう意図のリクエストかを考慮

View?

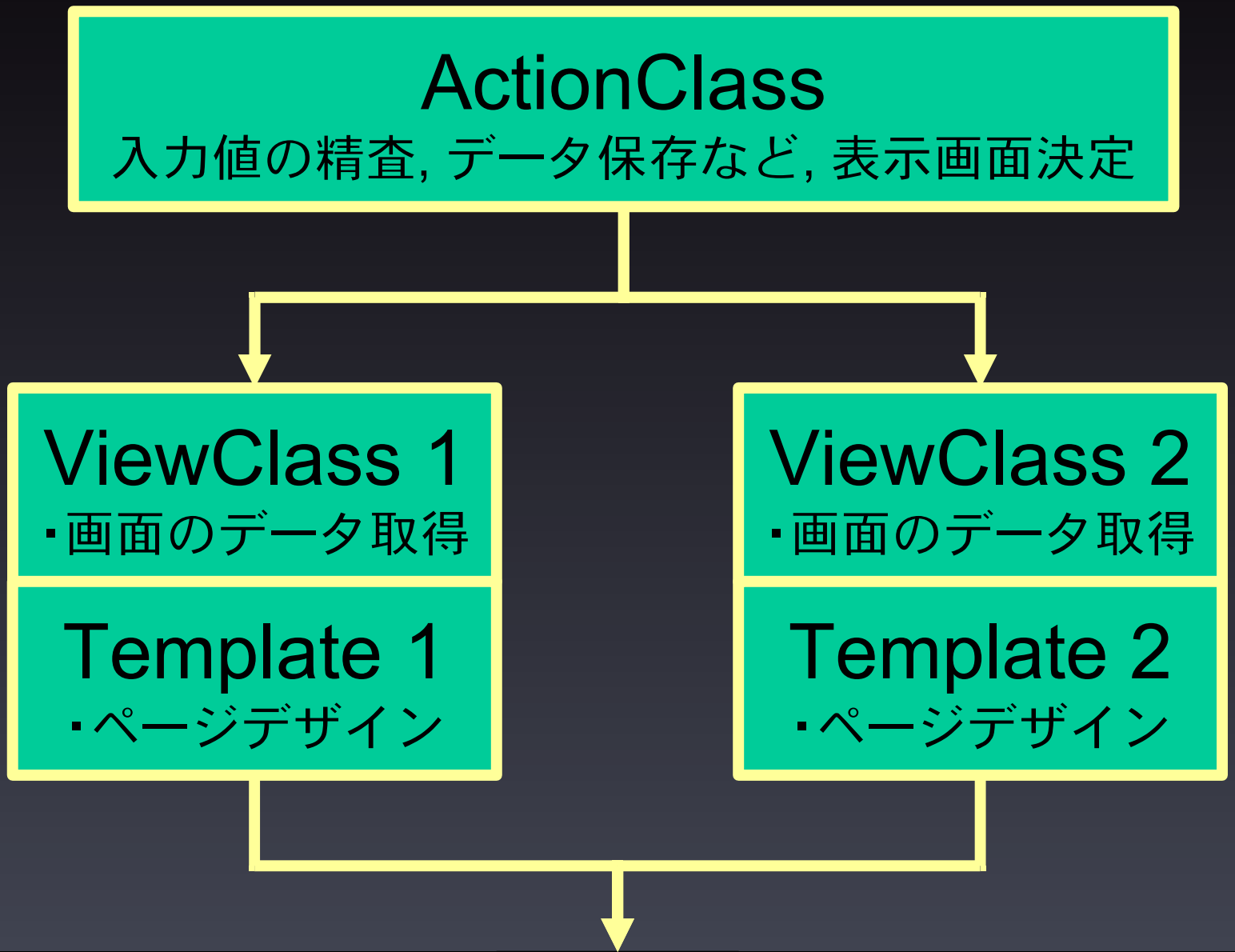
- 要望に対してシステムが出したい画面
  - 閲覧画面
  - 書き込み画面
  - 書き込み完了画面

  これらの単位を***View***
- Viewにデザインを決めるTemplateがあってページが成り立つ

View != HTTP Request

- HTMLファイルの世界は
  HTTP Request →
  ファイルに1:1対応HTML
- 動的な世界は
  HTTP Request →
  どの画面が出るか未定
  - だから画面遷移図とか

Action, View and Template



チュートリアル

作りながら基本をマスター

作るもの

- 今回作るものは、「ブックマーク」です。
- 要求としては
  - ページタイトル
  - URL
  - 内容の詳細

  が記録・閲覧(リスト)・編集・削除できる
- ページャ等は考慮しない

Install

PEAR, PEAR::DB, Smartyは最低限必須

<注意> Smarty等は、以下の形式で呼べるように

```
require_once 'Smarty/Smarty.class.php';
```

Version 2.1.2の場合

$ pear install
http://ethna.jp/pear/Ethna-2.1.2.tgz

Make project

$ **ethna add-project <projName> /project/path/**

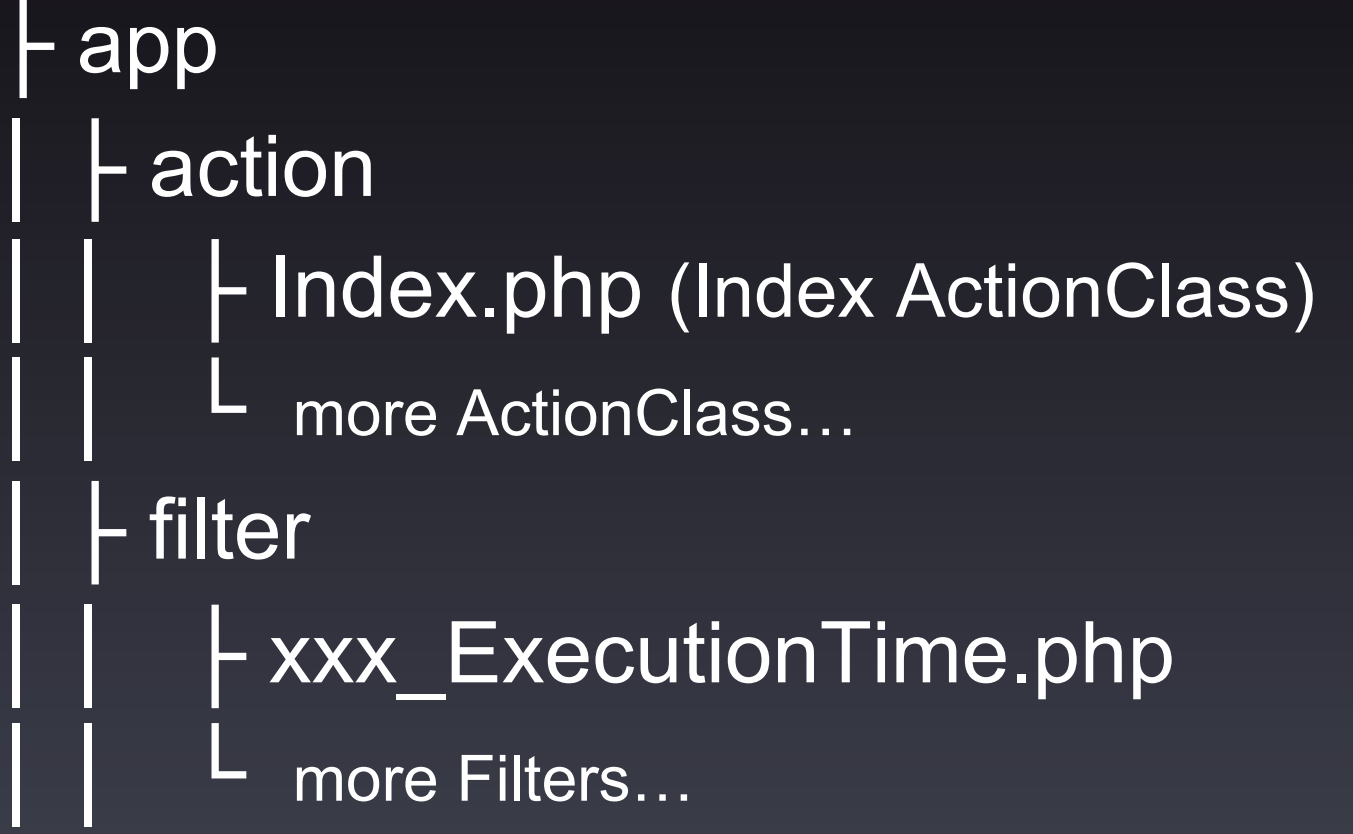
- こうすることで、ファイルがジェネレートされてプロジェクトが生成される

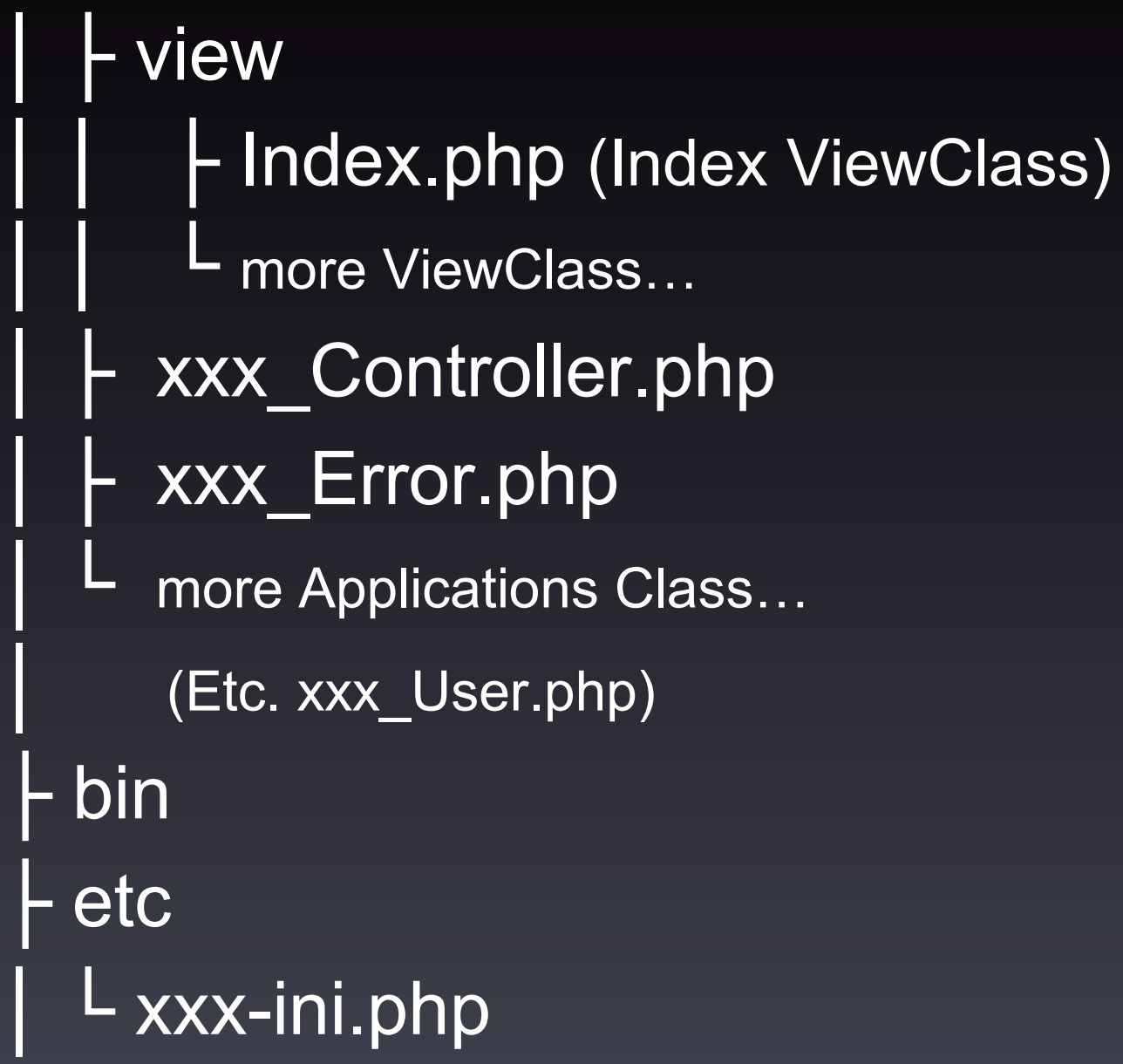
まずは見てみる

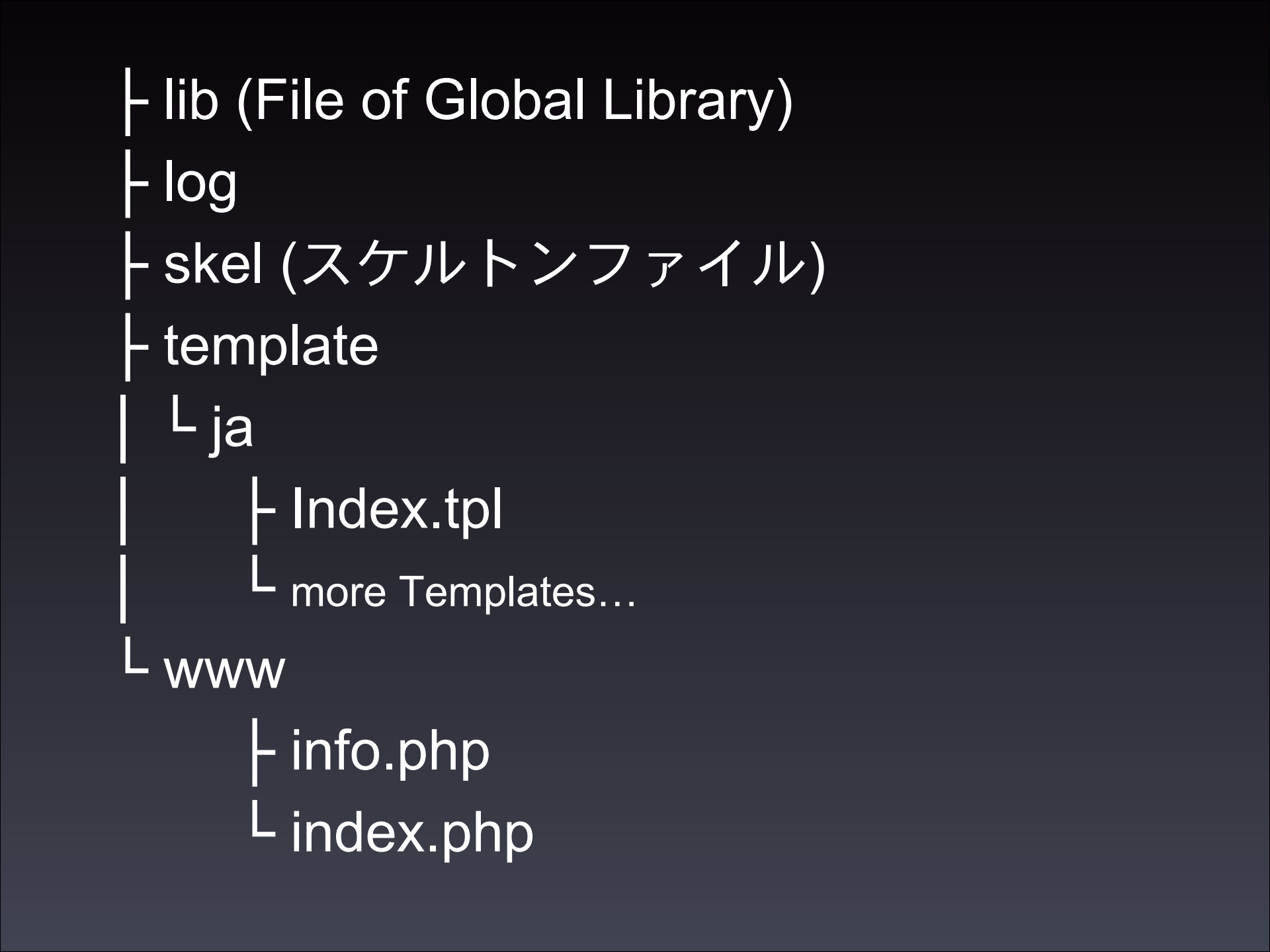
- 今の時点で IndexAction, IndexViewはできていますので見てみましょう

- Apacheの設定で
  alias /bookmark /path/to/project/www/
  として再起動
- http://localhost/bookmark/ にアクセス

Directory Structure




view
Index.php (Index ViewClass)
more ViewClass…
xxx_Controller.php
xxx_Error.php
more Applications Class…
(Etc. xxx_User.php)
bin
etc
xxx-ini.php


├ lib (File of Global Library)
├ log
├ skel (スケルトンファイル)
├ template
│ └ ja
│ ├ Index.tpl
│ └ more Templates…
└ www
├ info.php
└ index.php

Directory Structure (2)

- これらのディレクトリ構造は変更可能
  xxx_Controller.php のここを編集

```
var $directory = array(
        'action' => 'app/action',
        'etc' => 'etc',
// などなど
);
```

add Action

プロジェクトディレクトリ内で

$ ethna add-action <actionName>

- とすることで、アクションファイルが
  /app/action/<actionName>
  に生成される
- アンダーバーによるディレクトリ区切り

add View

プロジェクトディレクトリ内で

$ ethna add-view -t <viewName>

- とすることで、ビュークラスが /app/view/<viewName>.php に生成される
- -t でテンプレートが /template/ja/<viewName>.tpl に生成

add AppObject

- プロジェクトディレクトリ内で、

$ ethna add-app-object <modelName>

- とすることで、AppObjectと対応するAppManagerが/app/<project ID>_<modelName>.phpに生成される

add AppManager

- プロジェクトディレクトリ内で

$ ethna add-app-manager <managerName>

- これでマネージャが /app/<project ID>_<managerName>.php に生成される
- AppObjectが絡まないときに利用

other ethna Command

$ ethna add-action-test <actionName>

- アクション用テスト

$ ethna add-view-test <viewName>

- ビュー用テスト

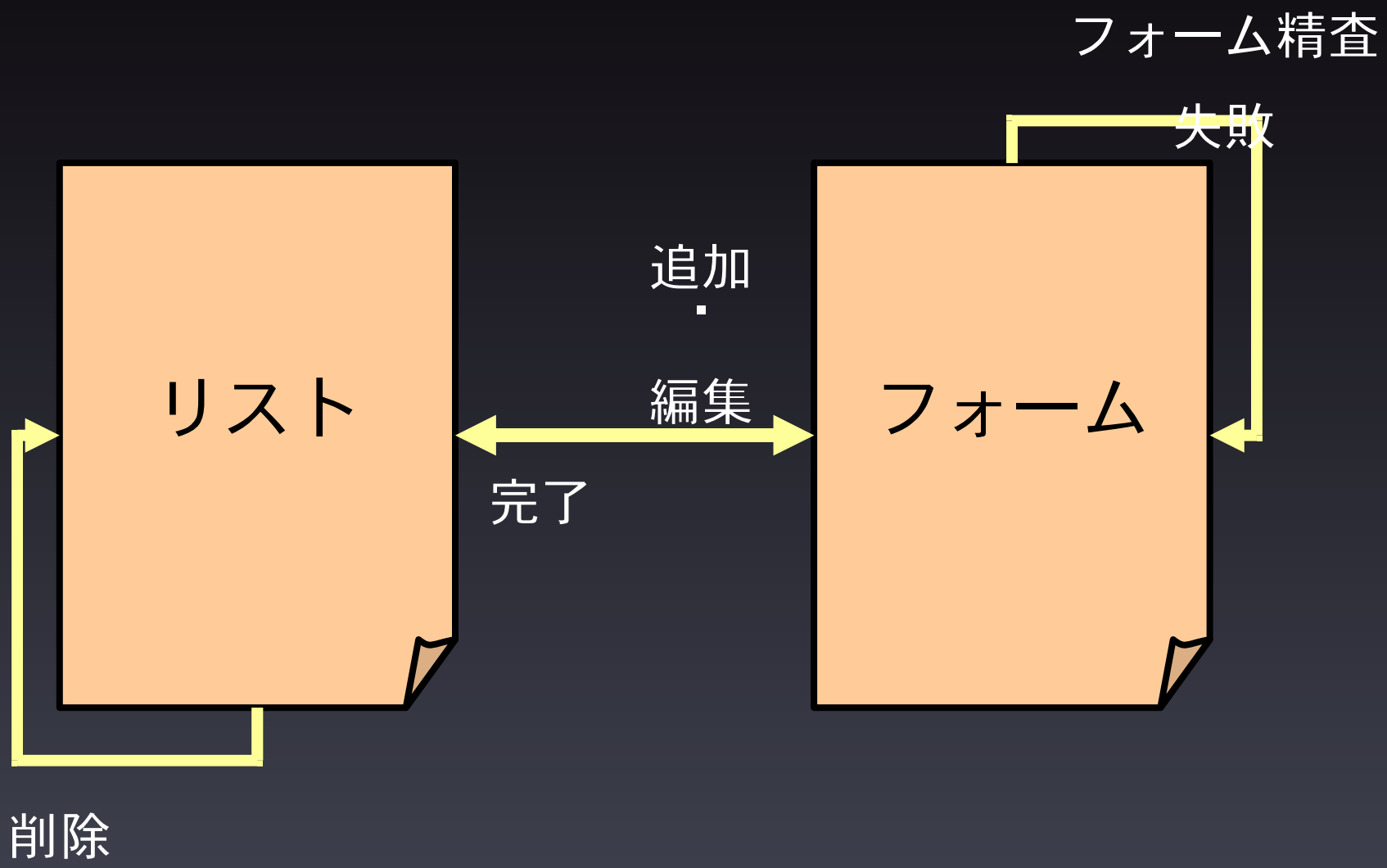
$ ethna add-action-cli <actionName>

$ ethna add-action-xmlrpc <actionName>

設計しよう

最大の難所
これはフレームワークでも
変わらない

簡易な画面遷移図



How Many “Action Classes”?

- リスト表示したい
- 書き込みフォーム表示
- 書き込まれたデータをDBに挿入する
- 編集フォーム表示
- 編集されたデータをDBで更新する
- データを削除する

Actionは6個必要

How Many “View Classes”?

- リスト画面
- 追加フォーム画面
- 編集フォーム画面

じゃあ、Viewは3枚

テーブルは？

- DBのテーブルは、今回は1テーブルでいける → 「item」テーブル
  - id (INT, PK, auto_increment)
  - title (VARCHAR 255)
  - url (VARCHAR 255)
  - description (TEXT)
  - created_at (DATETIME)

AppObject, AppManager

- 今回ビジネスロジック的に管理したりしないといけないものは「item」のみなので、AppObject, AppManagerは一つでOK。
- Item Object, Item Manager.

Finaly…

- 結局、
  - Action x 6
  - View x 3
  - Template x 3
  - AppObject x 1
  - AppManager x 1
  - テーブル定義 x 1
- 他にも小さいのはあるが、これらを作る必要がある

バリバリ実装

本来は設計がまだ甘いが、、、
Actionの構成で
何をしたいかは判るはず

テーブルを作ってみる

- /schema/MySQL.sql
  というファイルを作る
  - これでバージョン管理に乗って便利
  - ローカルでやるときも
    mysql> SOURCE /schema/MySQL.sql
    これで設定される
  - 定義が変わってもすぐ対応

DBの設定をする

- データベースの設定をしてEthnaから接続できるようにする
- /etc/bookmark-ini.phpの $config 連想配列に、
  ‘dsn’ => ‘mysql://user:pass@svr/db’,
  の指定を追加

MySQL <> PHP charset

- このまま進めると、文字コードの問題で文字化けすること確率が高いので、Bookmark_DB_PEAR.php を作成
- 接続時に“SET NAMES eucjpms”を発行して、文字コード宣言をするように Connectメソッドをオーバーライド

AppObjectを生成

$ ethna add-app-object item

- これで生成できました。
- ビジネスロジック等の実装は後回し

Action, Viewを生成

$ ethna add-action index
$ ethna add-action new
$ ethna add-action create (new_do)
$ ethna add-action edit
$ ethna add-action update (edit_do)
$ ethna add-action delete
$ ethna add-view -t list
$ ethna add-view -t form_create
$ ethna add-view -t form_edit

流してみる

動くものが欲しいので
とりあえず、流してみる
顧客にこんな流れか確認するとか

とりあえず、画面フロー

- 各アクションでもっとも正規な画面に行くように設定
- テンプレートを修正して、動くような画面を作成

- 画面遷移の確認をして、システムの概観をつかむ

例・書き込みフォームを出す

/app/action/new.php new action内

```
function perform() {
    return ‘form_create’;
}
```

- actionのメソッドの返り値が文字列ならそれは実行される**View名**になる

フォームから飛ぶには？

- フォームから飛ばすには、
  - hiddenやsubmitのnameを“action_<アクション名>” とかにしておく
  - その画面から飛ばすアクションが一つの場合はhiddenでもOK
  - その画面から飛ばすアクションが複数ある場合はsubmitにあてて押されるボタンで判定

GET、POSTに注意

- アクション名送信についてはGETで送っているか、POSTで送っているかに注意
  - 参考: BoBlog http://blog.as-roma.com/BoBlog/?itemid=1037

Smarty. {include}

- フォーム部分は共通なので、一つに
- form/_form.tpl.html として保存
  - 拡張子を変えておくことでViewとして扱われない
- 呼び出しアクションのhiddenのnameは変数化 → {$action}
- {include file="form/_form.tpl.html" action="new_do"} として呼び出す

初めは追加！

データがないことには
どうしようもないでしょ

何が必要？

- そこで
  - new, new_do Action
  - form_create View
  - form_create Template
  - 書き込みに必要なビジネスロジック

  を書く必要がある

フォームテンプレートを修正

- form/_form.tpl.html を
  修正してフォームを作成
- このとき、フォーム各エレメントの
  name属性をちゃんと決める
  - title
  - url
  - description

  こうすることにした

フォームの受け手

- 先のフォームのデータを受けるのは **new_do Action**
- フォームの処理はActionFormを用いる
- フォーム値の精査もActionFormで行う

- 精査や処理をするには、フォームの定義が必要！

フォームを定義する

- 受けるアクションクラスファイルにあるActionFormClassの $formメンバを修正して定義を行う
- 定義内容の詳細は公式サイト参照

- Ethna ActionForm Builder
  http://as-roma.com/ethna/formBuilder/

フォームの精査

- ***ActionForm::validate();***
  フォーム定義に従ってフォーム値の精査を行い問題があるフォームエレメント数をintで返す

精査する場所

- 一般に、Action::prepare(); 内に書く
  - http://ethna.jp/ethna-document-tutorial-prac

  これでエラーは登録され、returnされたView+Templateにいく
- ここでは、formCreateViewに戻るようにする

エラーを表示する

- エラーは {$errors} に入っている
  - http://ethna.jp/ethna-document-tutorial-practice3

  このように
  　リスト形式, 個別
  に表示することができる
- そのようなテンプレートを作成して
  {include file="parts/errorMessages.tpl.html"}

特殊なエラー

- ActionFormの定義だけで行えないエラーチェックは自前で精査して、ActionErrorに投げる (ことが多い)

```
ActionError::add(
    $elementName,
    $message, $errorCode
);
```

書き直しとかいやじゃね?

- 追加画面をエラーが出るように実行してみてください
  - 入力した値、表示されていましたか?
    - すごく感じ悪いですね
- inputのvalue属性がないので当たり前

- では、どういう値を設定すれば？？

Template - {$form.*} 変数

- {$form.*(name属性)} に、そのとき入力された値が入っています
- value属性に使ってみると・・・

```
<input name="a" value="{$form.a}" />
```

受け手は何をする？

- 定義をコピペで完成させる
- 受け手がすること･･･
  データを**DB**に登録する
  です
- そのことをビジネスロジックとしてコーディングしていく

Ethna_DBを使う

- Ethna_DBを用いてクエリを投げる形式

```
$db =& $this->backend->getDB();
$r =& $db->query(“SELECT * FROM item”);
```

- 通常だと、SQLinjection対策必要
- PEAR_DB::prepare(), execute() も検討

Ethna_DB_PEAR Prepared-Query

```
$db =& $this->backend->getDB();
$insQid = $db->db->prepare("INSERT
   INTO item VALUES (NULL, ?, ?, ?, ?)");
$r =& $db->db->execute($insQid, array(
   $title, $url, $description, $created_at));
```

- これで安全

追加コードを記述

```
$title = $this->af->get(‘title’);
$url = $this->af->get(‘url’);
・・・
$created_at = date(MYSQL_DATE);
```

- そして先のprePared-Query
- 定数MYSQL_DATEは日付フォーマット

ErrorObject Check

- Ethna::isError($r) === true
  $r がErrorオブジェクト
- DB等でなにかしたあとは必ずチェックする必要がある
- 適切なエラーハンドリングのためにチェックが必要

DAO化

- これをDAO(Data Access Object)化
- なんかDB部が分離できていいらしい
- AppManagerで実装 or 個別にClass
  - 重いけど、AppManagerを推奨
  - インスタンス生成時に
    $this->db 等にDBインスタンスが参照！

追加したのかわかりません

- データとして見えないというより
  完了しているのかわからない
- 完了画面は面倒くさい

- 例えば、リストの上部に
  「完了しました」って出せばいいよね

Actionで画面に文字列

- 画面を構築するために必要な動的要素はViewで処理しようと伝えてきました
- でも、例外もあると伝えました

- これが例外です
  アクションの結果などそのアクションのみが出す出力をそれ以外の状態でも利用するビューに表示するとき

アプリケーション値をSmartyへ

- アプリケーションの処理結果などをビューというか(Smarty)に渡す際
  Smarty::assign(*key*, *value*);

- Code: $this->af->setApp(*key*, *value*);
  Tpl: {$app.*key*}
  - *key* string テンプレート側のキー
  - *value* mixed 値

_noticeMessages.tpl.html

- 例えば {$app.notices} は予約語として
- そこにアクションからのメッセージをリスト表示するとか

- それもテンプレート切り出せそう
  - _noticeMessages.tpl.html
  - {foreach from=$app.notices item=notice}

AppObjectとか使ってみる

今日の本題
すごく便利だよ

DAO化しても…

- itemDAO::add($title, $url, $description);

- スキーマが変われば引数の形も変わる
- なんか梱包するオブジェクトとかあればいいよね → DTO

- DTO + O/R Mapping = AppObject !

Ethna_AppObject

- Ethnaでは、先ほどのような要件を満たすために、Ethna_AppObject というオブジェクトを用意しています

- コンストラクタと
  set(), get(), add(), update(), remove();
  をつかみましょう

AppObject - Table Define

- テーブルとの連携をするので、連携の設定が必要です
- $table_def, $prop_def メンバを修正

- 今回は自動設定に任せます
  - なお、設定されたデータは /tmp/cache/default/cache_ethna_app_object

AppObject - ClassFile Include

- AppObjectもただのクラスなので、includeしておかないと使えません

- 今回は、itemObjectなので、
  - /app/Bookmark_Contoller.phpに include_once (“Bookmark_Item.php”);

AppObject - Constructor

- new AppObject(*backend*);
  - 新規に生成する
- new AppObject(*backend*, *key*, *key-value*);
  - 既存のレコードを取得する SELECTする

  - *backend* Object Ethna_Backend Object
  - *key* mixed 主キー 複合主キーなら配列
  - *key-value* mixed 検索値 複合なら配列

AppObject - set(), get()

- $appObject->get(*key*);
  - 値が返る
- $appObject->set(*key*, *value*);
  - 値を設定する

  - *key* string プロパティ名
  - *value* mixed 値

AppObject
add(), update(), remove()

- $appObject->add();
  - このレコードをINSERTする
- $appObject->update();
  - このレコードをUPDATEする
- $appObject->remove();
  - このレコードをDELETEする

set して add

- 各プロパティの値をsetしていって、addすればレコードが追加されそうだ
- とりあえず、Actionに書いてみる

- しかし、、、日付までActionに書く必要があるのか？
  - フォーム値の設定はわかるけど

AppManagerで登録

- item追加はManagerを通じて行う
  - しかし、追加の処理を行うのは AppObject
  - AppManagerでAppObjectを叩く形
- なぜActionに書かないか?
  - データ追加する場所が一箇所ならそれでもいいけど・・・
  - どうせ増えるし
  - そこで変更あったらややこしい

マネージャに必要な裏処理

```
function add (&$item) {
    $item->set('created_at',
                        date(MYSQL_DATETIME));
    $r =& $item->add();

    return $item->isValid() && !Ethna::isError($r);
}
```

AppManager on Action

- /app/Bookmark_Controller.php
  var $manager = array(
  ‘item’ => “Item”,
  );
  「呼び出しキー => マネージャ名」
- Action, View, Manager, AppObjにて、

  $this->item->add();

Actionに呼び出しを記述

```
$item =& new
        Bookmark_Item($this->backend);
$item->set('title', $this->af->get('title'));
$item->set('url', $this->af->get('url'));
$item->set('description',
                    $this->af->get('description'));
$ret = $this->item->add($item);
```

set, get… **orz**

Ethna_AppObject::importForm();

- これは、ActionFormとAppObjectの連結
  ActionFormにはいったフォーム値を
  AppObjectに格納するメソッド
- AppObjectのプロパティ設定の
  ‘form_name’ の値のフォーム値が
  AppObjectにsetされる

import があれば

Ethna_AppObject::exportForm();

- importForm()とは逆に、フォームにAppObjectの値をセットする
- 編集とかに使える

そこで・・・

```
$item =& new
        Bookmark_Item($this->backend);
$item->importForm();
$ret = $this->item->add($item);
```

これでOK

Form Renderer Helper. {form_input}, {form_name}

- フォームの名称を
  {form_name name="afにつけた名前"}
- フォームのエレメントを
  {form_input name="afにつけた名前"}

- 詳細は,
  http://blog.as-roma.com/BoBlog/?itemid=1028
  をみてください

Action と ActionForm

- ActionとActionFormはくっついているので、現在は new_do Actionにしか今回用いるActionFormはない
- 例えばnew Actionでも、{form_input}を使ってフォームは出したいけど、定義がないので利用できない
  - 実際に /templates/ja/form/_form.tpl.html を編集して動作させてみるとわかる

Helper ActionForm

- 表示するためのフォーム定義を追加する方法として form_create ViewClassで

  $action = ‘new_do’;
  $this->addActionFormHelper($action);

  - http://blog.as-roma.com/BoBlog/index.php?itemid=1019

- 表記用のActionFormが設定される

違いは問題領域

- フォームの値を処理するという概念は
  Action
- フォームの表示するという概念は
  View

リストを出したい

とりあえずデータがあれば
みたいよね

リスト表示したい

- 追加できれば表示したいですね
- そこで、
  - index Action
  - list View
  - list Template
  - 必要なデータを返すビジネスロジック

  を書きましょう

Actionは簡単 でも・・・

- ただ、list Viewを呼び出すだけ
  - return ‘list’;
- Viewが面倒
  - データを取得して > DB叩く, AppManager
  - テンプレートに当てはめる > AF::setApp

まずは、SQLで

- 今までと同じように、DBを直接Viewに

```
$db =& $this->backend->getDB();
$r =& $db->query(“SELECT * FROM item”);
while ($temp = $r->fetchRow()) {
  $datas[] = $temp;
}
```

Template: foreach等で回して・・・

- リスト表示できますね
  -
    {foreach from=$app.items item=item}

    {$item.title}
…
    {/foreach}

形式はこだわらずに なんでもOKです。

AppManagerにまとめたい

```
function getAll () {
   // さっき書いたコード
   // AppManagerなので getDB不要
   return $arr;
}
```

$this->item->getAll() とかで呼べる！

AppObjectは？

- 追加でAppObject使った ここでは？
- 当然使いますよ

- AppObjectの取得にAppManager！
  - AppObject::searchId(); でもいいけど
  - AppManager::getObjectList();
  - AppManager::getObjectPropList();

AppManager::getObjectList()

AppManager::getObjectList (
name, *filter*, *order*, *offset*, *count*
);

- *name* string オブジェクト名
- *filter* array 検索条件
- *order* array 並び替え条件
- *offset* int 出力オフセット
- *count* int 出力個数

- 返り値注意 array
  - [0] int 取得個数
  - [1] array Objects

or Ethna_Error

AppManager::getObjectPropList()

AppManager::getObjectPropList (
name, *keys*, *filter*, *order*, *offset*, *count*
);

- *name* string オブジェクト名
- *keys* array プロパティ名
- *filter* array 検索条件
- *order* array 並び替え条件
- *offset* int 出力オフセット
- *count* int 出力個数

- 返り値注意 array
  - [0] int 取得個数
  - [1] array Props

or Ethna_Error

AppManager::getObjectProp

AppManager::getObjectProp (
name, *keys*, *filter*
);

- *name* string オブジェクト名
- keys array プロパティ名
- *filter* array 検索条件

- 返り値
  - array Props
  - null noneEntry
  - Ethna_Error

徹底比較！ Props or Objects

- Propsのほうが軽い！
  - これはかなり大きなメリット
  - Propsなら一発で setApp出来そう
  - 通常時はPropsでよくね？
- Objectsは getName($prop) 使える！
  - 例えばカテゴリ名と登録件数を表示とか
  - JOINしないでこういうのやるとき
  - getNameObject(); で一括 便利

比較続き: setAppleには?

- Props・・・
  - $propsData = getObjectPropList(・・・);
    $this->af->setApp(“items”, $propsData[1]);

- Objects・・・
  - $objsData = getObjectList(・・・);
    foreach($objsData[1] as $obj)
    $temp[] = $obj->getNameObject();
    $this->af->setApp(“items”, $temp);

比較: 結論

- 整形しないでOKと判っているなら
  断然 ***getObjectPropList!!***
- この場面のみでないよくある表示用の
  変換処理があってそれを今回も使う
  となれば ***getObjectList!!***

- 今回は getObjectPropList();

ORDER BY created_at DESC

- order で行う 形式として
  array(
  　　“created_at” => OBJECT_SORT_DESC,
)

- 検索のfilterも同じような形式
  array(
  　　“id” => new Ethna_AppSearchObject(10, OBJECT_CONDITION_EQ),
)

- 定数は出来るだけ覚えてください

編集できればいいな

そろそろ編集もありでないと

編集もしたい･･･

- 編集するのに必要なもの･･･
  - Actions
    - edit
    - edit_do
  - View
    - form_edit
  - Business Logic
    - Data Update

編集って事は元の値は？

- もちろん入っている必要があります

- edit Actionで・・・
- AppObject::exportForm();
  - これを使ってItemObjectからフォーム値へ
  - set, getしなくて楽

edit_do Action

- FormをValidateして
- DBのUPDATE文とかもいいけど
- idをもとに Item Objectつくって
  - $item->importForm();
    $item->update();

    で終わり！

Noticeしようね

- 以前使った「Notice Message Pattern」をここでも使いましょう
- 「編集が完了しました」とか出ればいいんじゃないでしょうか

Sessionで編集IDを管理

- Sessionに現在編集中のIDを入れてチェックさせる
- 簡単なCSRFとかの対策
- OneTimeTicket系のPOST自体を特定できる機能があったほうがもっとセキュアですが
  - 今回は省略します

Session Class

- start(), destroy()
  - 開始, 終了
- set(*key*, *value*), get(*key*), remove(*key*)
  - 設定, 取得, 削除
- isStart()
  - 開始チェック

配列でいれたほうが

- 利便性下げるのは・・・
  例えば、
  $_SESSION[‘edit’] = 5;
  とかすると、同時に複数の編集Windowを立ち上げることが出来ない
- 配列で入れて、
  in_array(S::get(‘edits’), AF::get(‘item_id’))

どこで設定する?

- どこで設定して、どこでチェックすればよいでしょう

- 設定場所
  - edit
- チェック
  - edit_do

削除で完成！

作って消せないってどうよ

delete Action

- FormをValidateして
- idを元に Item Object作って
  - $item->remove();

    で終わり！

Noticeしようね

- ここでも、「Notice Message Pattern」を使いましょう
- 「削除が完了しました」とか表示すればいいんじゃないでしょうか

JavaScriptで確認

- イキナリ消えるという挙動はちょっとユーザ的にもよくなさそう
- JavaScriptのconfirm(“削除してOK?”);とかを用いる

そんなわけで、開発手順

重要ポイントを抑える

例えば

- 画面がきてそれにあったシステム
  - 画面の遷移から必要なActionとViewを抜き出す
  - それで画面遷移出来るようにActionとViewをガンガン作る
  - その間にビジネスロジック組み立てる人とか居たらハッピー
  - くっつけて完成とかとか

End.

ご清聴ありがとうございました。
おつかれさまでした。