Red Hat support for Spring Boot 2.2

Spring Boot Runtime Guide

Spring Boot 2.2 を使用して、OpenShift およびスタンドアロン RHEL で実行するアプリケーションを開発

Last Updated: 2022-09-16

Red Hat support for Spring Boot 2.2 Spring Boot Runtime Guide

Spring Boot 2.2 を使用して、OpenShift およびスタンドアロン RHEL で実行するアプリケーションを開発

Enter your first name here. Enter your surname here.
Enter your organisation's name here. Enter your organisational division here.
Enter your email address here.

法律上の通知

概要

本ガイドでは、Spring Boot の使用方法について説明します。

目次

前書き

本ガイドでは、概念と、開発者が Spring Boot ランタイムの使用に必要な実用的な詳細情報を説明します。OpenShift 上の Linux コンテナーとしてデプロイされた Spring Boot アプリケーションの設計を管理する情報を提供します。

RED HAT ドキュメントへのフィードバック (英語のみ)

ご意見やご意見をお聞かせください。フィードバックを行うには、ドキュメント内のテキストを強調表示し、コメントを追加します。

本セクションでは、フィードバックの送信方法を説明します。

前提条件

- Red Hat カスタマーポータルにログインします。
- Red Hat カスタマーポータルで、**Multi-page HTML** 形式のドキュメントを表示します。

手順

フィードバックを提供するには、以下の手順を実施します。

1. ドキュメントの右上隅にある **Feedback** ボタンをクリックして、既存のフィードバックを表示します。

注記

フィードバック機能は、**Multi-page HTML** 形式でのみ有効です。

2. フィードバックを提供するドキュメントのセクションを強調表示します。
3. 強調表示されているテキストの近くに表示される **Add Feedback** ポップアップをクリックします。
   ページの右側のフィードバックセクションに、テキストボックスが表示されます。
4. テキストボックスにフィードバックを入力し、**Submit** をクリックします。
   ドキュメントの問題が作成されました。
5. 問題を表示するには、フィードバックビューで問題トラッカーリンクをクリックします。

第1章 SPRING BOOT でのアプリケーション開発の概要

本セクションでは、Red Hat ランタイムでのアプリケーション開発の基本概念を説明します。Spring Boot ランタイムの概要も説明します。

1.1. RED HAT RUNTIMES でのアプリケーション開発の概要

Red Hat OpenShift は、クラウドネイティブランタイムのコレクションを提供するコンテナーアプリケーションプラットフォームです。ランタイムを使用して、OpenShift で Java または JavaScript アプリケーションを開発、ビルド、およびデプロイできます。

Red Hat Runtimes for OpenShift を使用したアプリケーション開発には、以下が含まれます。

- OpenShift 上で実行されるように設計された Eclipse Vert.x、Thorntail、Spring Boot などのランタイムのコレクション。
- OpenShift でのクラウドネイティブ開発への規定的なアプローチ。

OpenShift は、アプリケーションのデプロイメントおよび監視の管理、保護、自動化に役立ちます。ビジネス上の問題を小規模なマイクロサービスに分割し、OpenShift を使用してマイクロサービスをデプロイし、監視し、維持することができます。サーキットブレーカー、ヘルスチェック、サービス検出などのパターンをアプリケーションに実装できます。

クラウドネイティブな開発は、クラウドコンピューティングを最大限に活用します。

以下でアプリケーションをビルドし、デプロイし、管理できます。

OpenShift Container Platform

Red Hat のプライベートオンプレミスクラウド。

Red Hat Container Development Kit (Minishift)

ローカルマシンにインストールおよび実行できるローカルクラウド。この機能は、Red Hat Container Development Kit (CDK) または Minishift で提供されます。

Red Hat CodeReady Studio

アプリケーションの開発、テスト、デプロイを行う統合開発環境 (IDE)。

アプリケーション開発を開始できるようにするため、サンプルアプリケーションですべてのランタイムが利用可能になります。これらのサンプルアプリケーションは Developer Launcher からアクセスできます。サンプルをテンプレートとして使用してアプリケーションを作成することができます。

本ガイドでは、Spring Boot ランタイムに関する詳細情報を提供します。その他のランタイムの詳細は、関連する ランタイムドキュメント を参照してください。

1.2. DEVELOPER LAUNCHER を使用した RED HAT OPENSHIFT でのアプリケーション開発

Developer Launcher (developers.redhat.com/launch) を使用して、OpenShift でのクラウドネイティブアプリケーションの開発を開始することができます。これは、Red Hat が提供するサービスです。

Developer Launcher はスタンドアロンのプロジェクトジェネレーターです。これを使用して、OpenShift Container Platform、Minishift、CDK などの OpenShift インスタンスでアプリケーションをビルドし、デプロイできます。

1.3. SPRING BOOT の概要

Spring Boot では、スタンドアロン Spring ベースのアプリケーションを作成できます。Spring Boot に関するドキュメントの一覧は、「関連情報」を参照してください。https://access.redhat.com/documentation/ja-jp/red_hat_support_for_spring_boot/2.2/html-single/spring_boot_runtime_guide/#additional-springboot-resources_spring-boot

OpenShift 上の Spring Boot は、以下のように、Spring Boot の合理化されたアプリケーション開発機能と OpenShift のインフラストラクチャーおよびコンテナーオーケストレーション機能を組み合わせたものです。

- ローリング更新
- サービス検出
- カナリアデプロイメント
- 一般的なマイクロサービスパターンを実装する方法: 外部化設定、ヘルスチェック、サーキットブレーカー、およびフェイルオーバー

1.3.1. Spring Boot の機能およびフレームワークの概要

本ガイドでは、Spring Boot を使用した OpenShift でのクラウドネイティブアプリケーションの開発について説明します。これ以降のセクションのアプリケーション例では、Spring Boot を他の Red Hat テクノロジーと統合する方法を説明します。これらの統合機能を使用して、クラウドネイティブの Java アプリケーションを設定する最新の設計パターンを実装することができます。

- resilient
- responsive
- scalable
- secure

通常の Web サーバースタックまたはノンブロッキングリアクティブスタックで Spring Boot アプリケーションをビルドできます。

Developer Launcher を使用して、サンプルアプリケーションを OpenShift クラスターにデプロイすることもできます。アプリケーションはパッケージ化や変更せずにデプロイできます。または、それらをカスタマイズして、追加のクラウドネイティブ機能を使用し、OpenShift に組み込まれた継続的な統合機能を使用してそれらを更新で再デプロイすることもできます。

Red Hat は、Snowdrop コミュニティープロジェクトをベースとした Spring Boot のリリースをサポートします。

サポートされるランタイムフレームワークのコンポーネントには以下が含まれます。

- Apache Tomcat (Red Hat Java Web Server 製品オファリングで提供) および JBoss Undertow (Red Hat Enterprise Application Platform で提供) をベースとしたクラウドネイティブの Java ベースのアプリケーションを開発するための Spring Boot Starter のセット。
- Spring WebFlux 非ブロッキング API、Eclipse Vert.x および Reactor Netty によって提供されるネットワークコンポーネントを使用して、リアクティブスタックでクラウドネイティブの Java ベースのアプリケーションを開発するための Spring Boot Starters のセット。
- Spring Boot と統合する OpenShift および Kubernetes のアノテーションパーサーのコレクショ

ンおよびアプリケーションテンプレートジェネレーターのコレクションである Dekorate。Dekorate を使用すると、OpenShift クラスターへのデプロイメント用にアプリケーションを設定するのに使用可能なテンプレートを自動的に作成できます。アプリケーションのビルド時、Dokorate は、アプリケーションプロジェクトの設定プロパティー (例: **application.properties**) が含まれるファイル、またはアプリケーションのソースファイルにあるアノテーションから設定パラメーターを抽出します。次に、Dekorate は抽出されたパラメーターを使用して、アプリケーションを OpenShift クラスターにデプロイするのに使用できるリソースファイルを作成し、設定します。Dekorate は、使用する言語とビルドツールとは独立して機能し、複数のクラウドネイティブなアプリケーションフレームワークと統合します。Red Hat は Dekorate を使用して、OpenShift Container Platform に Java ベースのアプリケーションをデプロイするアプリケーションテンプレートを生成するためのサポートを提供します。Red Hat は、Maven で Dekorate の使用に対するサポートを提供しています。他のビルドツールはサポートされていません。

1.3.2. Spring Boot でサポートされるアーキテクチャー

Spring Boot は以下のアーキテクチャーをサポートします。

- x86_64 (AMD64)
- OpenShift 環境の IBM Z (s390x)

アーキテクチャーによって異なるイメージがサポートされています。本ガイドのコード例は、x86_64 アーキテクチャーのコマンドを示しています。他のアーキテクチャーを使用している場合は、コマンドに該当するイメージ名を指定します。

イメージ名の詳細は、Spring Boot でサポートされる Java イメージ のセクションを参照してください。

1.3.3. サンプルアプリケーションの概要

クラウドネイティブのアプリケーションおよびサービスをビルドする方法を実証する作業アプリケーションがあります。これらは、アプリケーションの開発時に使用する必要のある規範的なアーキテクチャー、設計パターン、ツール、およびベストプラクティスを示しています。サンプルアプリケーションは、クラウドネイティブのマイクロサービスを作成するためのテンプレートとして使用できます。本ガイドで説明しているデプロイメントプロセスを使用して、例を更新および再デプロイできます。

この例では、以下のような マイクロサービスパターン を実装します。

- REST API の作成
- データベースの相互運用
- ヘルスチェックパターンの実装
- アプリケーションの設定を外部化してセキュア化し、容易なスケーリング

サンプルアプリケーションは以下のように使用できます。

- テクノロジーのデモの実行
- プロジェクトのアプリケーションの開発方法を理解するツールまたはサンドボックスの学習
- 独自のユースケースを更新または拡張するためのヒント

各サンプルアプリケーションは 1 つ以上のランタイムに実装されます。たとえば、REST API Level 0 のサンプルは以下のランタイムで利用できます。

- Node.js
- Spring Boot
- Eclipse Vert.x
- Thorntail

以降のセクションでは、Spring Boot ランタイムに実装されたサンプルアプリケーションを説明します。

すべてのサンプルアプリケーションを以下にダウンロードおよびデプロイできます。

- x86_64 アーキテクチャー - 本ガイドのアプリケーションの例では、サンプルアプリケーションを x86_64 アーキテクチャーにビルドおよびデプロイする方法を説明します。
- s390x アーキテクチャー - IBM Z インフラストラクチャーでプロビジョニングされた OpenShift 環境にサンプルアプリケーションをデプロイするには、コマンドに関連する IBM Z イメージ名を指定します。イメージ名の詳細は、Spring Boot でサポートされる Java イメージのセクションを参照してください。
  サンプルアプリケーションの一部には、ワークフローを実証するために Red Hat Data Grid などの他の製品も必要になります。この場合、これらの製品のイメージ名をサンプルアプリケーションの YAML ファイルで関連する IBM Z イメージ名に変更する必要もあります。

第2章 SPRING BOOT を使用するようにアプリケーションを設定

Red Hat ビルドの Spring Boot で提供される依存関係を使用するようにアプリケーションを設定します。BOM を使用して依存関係を管理することで、アプリケーションが Red Hat が提供するこれらの依存関係の製品バージョンを常に使用するようにしてください。アプリケーションのルートディレクトリーにある **pom.xml** ファイルの Spring Boot BOM (Bill of Materials) アーティファクトを参照します。アプリケーションプロジェクトで BOM を使用することもできます。2 種類の方法を使用できます。

- **pom.xml** の **<dependencyManagement>** セクションの 依存関係として。BOM を依存関係として使用する場合、プロジェクトは BOM の **<dependencyManagement>** セクションからすべての Spring Boot 依存関係のバージョン設定を継承します。
- **pom.xml** の **<parent>** セクションで 親 BOM として。BOM を親として使用する場合、プロジェクトの **pom.xml** は、親 BOM から以下の設定値を継承します。
  - **<dependencyManagement>** セクションのすべての Spring Boot 依存関係のバージョン
  - **<pluginManagement>** セクションのバージョンプラグイン
  - **<repositories>** セクションのリポジトリーの URL および名前
  - **<pluginRepositories>** セクションに Spring Boot プラグインが含まれるリポジトリーの URL および名前

2.1. 前提条件

- **pom.xml** ファイルを使用して設定する Maven ベースのアプリケーションプロジェクト。
- Red Hat JBoss Middleware General Availability Maven リポジトリーへのアクセス 。

2.2. SPRING BOOT BOM を使用した依存関係バージョンの管理

製品 BOM を使用して、アプリケーションプロジェクトで Spring Boot 製品依存関係のバージョンを管理します。

手順

1. **dev.snowdrop:snowdrop-dependencies** アーティファクトをプロジェクトの **pom.xml** の **<dependencyManagement>** セクションに追加し、**<type>** 属性値および **<scope>** 属性値を指定します。

```
<project>
 ...
 <dependencyManagement>
  <dependencies>
   <dependency>
    <groupId>dev.snowdrop</groupId>
    <artifactId>snowdrop-dependencies</artifactId>
    <version>2.2.11.SP1-redhat-00001</version>
    <type>pom</type>
    <scope>import</scope>
   </dependency>
  </dependencies>
```

```
  </dependencyManagement>
  ...
</project>
```

2. 以下のプロパティーを追加して、使用している Spring Boot Maven プラグインのバージョンを追跡します。

```
<project>
  ...
  <properties>
    <spring-boot-maven-plugin.version>2.2.11.RELEASE</spring-boot-maven-plugin.version>
  </properties>
  ...
</project>
```

3. BOM およびサポートされる Spring Boot Starters および Spring Boot Maven プラグインを含むリポジトリーの名前と URL を指定します。

```
  <!-- Specify the repositories containing Spring Boot artifacts. -->
  <repositories>
    <repository>
      <id>redhat-ga</id>
      <name>Red Hat GA Repository</name>
      <url>https://maven.repository.redhat.com/ga/</url>
    </repository>
  </repositories>

  <!-- Specify the repositories containing the plugins used to execute the build of your application. -->
  <pluginRepositories>
    <pluginRepository>
      <id>redhat-ga</id>
      <name>Red Hat GA Repository</name>
      <url>https://maven.repository.redhat.com/ga/</url>
    </pluginRepository>
  </pluginRepositories>
```

4. **spring-boot-maven-plugin** を Maven がアプリケーションのパッケージ化に使用するプラグインとして追加します。

```
<project>
  ...
  <build>
    ...
    <plugins>
      ...
      <plugin>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-maven-plugin</artifactId>
        <version>${spring-boot-maven-plugin.version}</version>
        <executions>
          <execution>
            <goals>
              <goal>repackage</goal>
            </goals>
```

```
          </execution>
        </executions>
          <configuration>
            <redeploy>true</redeploy>
          </configuration>
      </plugin>
      ...
    </plugins>
  ...
 </build>
 ...
</project>
```

2.3. SPRING BOOT BOM を使用したアプリケーションの親 BOM として使用

以下を自動的に管理します。

- 製品依存関係のバージョン
- Spring Boot Maven プラグインのバージョン
- 製品アーティファクトおよびプラグインが含まれる Maven リポジトリーの設定

製品の Spring Boot BOM をプロジェクトの親 BOM として含めて、アプリケーションプロジェクトで使用する内容。この方法では、BOM をアプリケーションの依存関係として使用する代替方法を提供します。

手順

1. **dev.snowdrop:snowdrop-dependencies** アーティファクトを **pom.xml** の **<parent>** セクションに追加します。

```
<project>
  ...
  <parent>
    <groupId>dev.snowdrop</groupId>
    <artifactId>snowdrop-dependencies</artifactId>
    <version>2.2.11.SP1-redhat-00001</version>
  </parent>
  ...
</project>
```

2. **spring-boot-maven-plugin** を Maven がアプリケーションを **pom.xml** の **<build>** セクションにパッケージ化するために使用するプラグインとして追加します。プラグインバージョンは、親 BOM により自動的に管理されます。

```
<project>
  ...
  <build>
    ...
    <plugins>
      ...
      <plugin>
```

```
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-maven-plugin</artifactId>
        <executions>
          <execution>
            <goals>
              <goal>repackage</goal>
            </goals>
          </execution>
        </executions>
        <configuration>
          <redeploy>true</redeploy>
        </configuration>
      </plugin>
      ...
    </plugins>
  ...
 </build>
 ...
</project>
```

2.4. 関連情報

- Spring Boot アプリケーションのパッケージ化に関する詳細は、Spring Boot Maven Plugin のドキュメントを参照してください。

第3章 DEVELOPER LAUNCHER を使用したアプリケーションのダウンロードおよびデプロイ

このセクションでは、ランタイムで提供されるサンプルアプリケーションをダウンロードおよびデプロイする方法を説明します。アプリケーションのサンプルは Developer Launcher で利用できます。

3.1. DEVELOPER LAUNCHER の使用

Developer Launcher (developers.redhat.com/launch) は OpenShift 上で実行します。アプリケーションのサンプルをデプロイする場合、Developer Launcher は以下のプロセスを説明します。

- ランタイムの選択
- アプリケーションのビルドおよび実行

選択に基づいて、Developer Launcher はカスタムプロジェクトを生成します。プロジェクトの ZIP バージョンをダウンロードするか、または OpenShift Online インスタンスでアプリケーションを直接起動できます。

Developer Launcher を使用してアプリケーションを OpenShift にデプロイする場合は、Source-to-Image (S2I) ビルドプロセスが使用されます。このビルドプロセスは、OpenShift でアプリケーションを実行するために必要なすべての設定、ビルド、およびデプロイメントのステップを処理します。

3.2. DEVELOPER LAUNCHER を使用したサンプルアプリケーションのダウンロード

Red Hat は、Spring Boot ランタイムを使い始めるのに役立つサンプルアプリケーションを提供します。これらの例は、Developer Launcher (developers.redhat.com/launch) で利用できます。

サンプルアプリケーションをダウンロードして、ビルドし、デプロイできます。本セクションでは、サンプルアプリケーションをダウンロードする方法を説明します。

サンプルアプリケーションをテンプレートとして使用し、独自のクラウドネイティブアプリケーションを作成できます。

手順

1. Developer Launcher (developers.redhat.com/launch) に移動します。
2. **Start** をクリックします。
3. **Deploy an Example Application**をクリックします。
4. **Select an Example** をクリックし、ランタイムで使用できるサンプルアプリケーションの一覧を表示します。
5. ランタイムを選択します。
6. サンプルアプリケーションを選択します。

注記

複数のランタイムで利用できるアプリケーションの例もあります。前の手順でランタイムを選択していない場合は、サンプルアプリケーションで利用できるランタイムの一覧からランタイムを選択できます。

7. ランタイムのリリースバージョンを選択します。ランタイムに一覧表示されているコミュニティーまたは製品リリースから選択できます。
8. **Save** をクリックします。
9. **Download** をクリックして、サンプルアプリケーションをダウンロードします。
ソースおよびドキュメントファイルを含む ZIP ファイルがダウンロードされます。

3.3. OPENSHIFT CONTAINER PLATFORM または CDK (MINISHIFT) へのサンプルアプリケーションのデプロイメント

サンプルアプリケーションを OpenShift Container Platform または CDK (Minishift) のいずれかにデプロイできます。アプリケーションをデプロイする場所に応じて、認証に該当する Web コンソールを使用します。

前提条件

- Developer Launcher を使用して、サンプルアプリケーションプロジェクトを作成している。
- アプリケーションを OpenShift Container Platform にデプロイする場合は、OpenShift Container Platform Web コンソールにアクセスできるようにしている。
- CDK (Minishift) にアプリケーションをデプロイする場合は、CDK (Minishift) Web コンソールにアクセスできるようにしている。
- **OC** コマンドラインクライアントがインストールされている。

手順

1. サンプルアプリケーションをダウンロードします。
2. **oc** コマンドラインクライアントを使用して、サンプルアプリケーションを OpenShift Container Platform または CDK (Minishift) にデプロイできます。
Web コンソールによって提供されるトークンを使用してクライアントを認証する必要があります。アプリケーションをデプロイする場所に応じて、OpenShift Container Platform Web コンソールまたは CDK (Minishift) Web コンソールを使用します。以下の手順を実行してクライアントの認証を取得します。
   a. Web コンソールにログインします。
   b. Web コンソールの右上隅にあるクエスチョンマークアイコンをクリックします。
   c. 一覧から **Command Line Tools** を選択します。
   d. **oc login** コマンドをコピーします。
   e. ターミナルにコマンドを貼り付け、CLI クライアント **oc** をアカウントで認証します。

```
$ oc login OPENSHIFT_URL --token=MYTOKEN
```

3. ZIP ファイルの内容を展開します。

```
$ unzip MY_APPLICATION_NAME.zip
```

4. OpenShift で新規プロジェクトを作成します。

```
$ oc new-project MY_PROJECT_NAME
```

5. **MY_APPLICATION_NAME** の root ディレクトリーに移動します。

6. Maven を使用してサンプルアプリケーションをデプロイします。

```
$ mvn clean fabric8:deploy -Popenshift
```

注記: アプリケーションの例によっては、追加の設定が必要になる場合があります。サンプルアプリケーションをビルドおよびデプロイするには、**README** ファイルに記載されている手順に従います。

7. アプリケーションのステータスを確認し、Pod が実行していることを確認します。

```
$ oc get pods -w
NAME                        READY     STATUS      RESTARTS   AGE
MY_APP_NAME-1-aaaaa         1/1       Running     0          58s
MY_APP_NAME-s2i-1-build     0/1       Completed   0          2m
```

MY_APP_NAME-1-aaaaa Pod は完全にデプロイされ起動すると **Running** ステータスになります。アプリケーションの Pod 名は異なる場合があります。Pod 名の数値は、新しいビルドごとに増加します。末尾の文字は、Pod の作成時に生成されます。

8. アプリケーションのサンプルをデプロイして起動すると、そのルートを決定します。

ルート情報の例

```
$ oc get routes
NAME            HOST/PORT                                        PATH      SERVICES
PORT      TERMINATION
MY_APP_NAME     MY_APP_NAME-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME
MY_APP_NAME     8080
```

Pod のルート情報は、アクセスできるベース URL を提供します。この例では、**http://MY_APP_NAME-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME** をベース URL として使用し、アプリケーションにアクセスできます。

第4章 SPRING BOOT ランタイムアプリケーションの開発およびデプロイ

サンプルを使用する 他に、ゼロから新しい Spring Boot アプリケーションを作成して、OpenShift にデプロイできます。

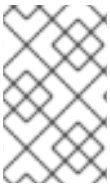
Spring Boot アプリケーションでサポート対象でテスト済みの Maven アーティファクトを指定する方法や使用の推奨される方法は、OpenShift Application Runtimes Spring Boot BOM の使用が推奨されます。

4.1. SPRING BOOT アプリケーションの開発

基本的な Spring Boot アプリケーションには、以下を作成する必要があります。

- Spring Boot メソッドが含まれる Java クラス。
- アプリケーションをビルドするために Maven が必要とする情報が含まれる **pom.xml** ファイル。

以下の手順では、応答として "{"content":"Greetings!"}" を返す単純な **Greeting** アプリケーションを作成します。



注記

アプリケーションを OpenShift にビルドおよびデプロイする場合、Spring Boot 2.2 は OpenJDK 8 および OpenJDK 11 をベースとしたビルダーイメージのみをサポートします。Oracle JDK および OpenJDK 9 のビルダーイメージはサポートされていません。

前提条件

- OpenJDK 8 または OpenJDK 11 がインストールされている。
- Maven がインストールされている。

手順

1. **myApp** ディレクトリーを新規作成し、そのディレクトリーに移動します。

```
$ mkdir myApp
$ cd myApp
```

   これは、アプリケーションのルートディレクトリーです。

2. ルートディレクトリーにディレクトリー構造 **src/main/java/com/example/** を作成し、これに移動します。

```
$ mkdir -p src/main/java/com/example/
$ cd src/main/java/com/example/
```

3. アプリケーションコードを含む Java クラスファイル **MyApp.java** を作成します。

```
package com.example;
```

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseBody;
import org.springframework.web.bind.annotation.RestController;

@SpringBootApplication
@RestController
public class MyApp {

    public static void main(String[] args) {
        SpringApplication.run(MyApp.class, args);
    }

    @RequestMapping("/")
    @ResponseBody
    public Message displayMessage() {
        return new Message();
    }

    static class Message {
        private String content = "Greetings!";

        public String getContent() {
            return content;
        }

        public void setContent(String content) {
            this.content = content;
        }
    }
}
```

4. 以下の内容を含むアプリケーションルートディレクトリー **myApp** に **pom.xml** ファイルを作成します。

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>com.example</groupId>
  <artifactId>my-app</artifactId>
  <version>1.0.0-SNAPSHOT</version>

  <name>MyApp</name>
  <description>My Application</description>

  <!-- Specify the JDK builder image used to build your application. -->
  <!-- Use OpenJDK 8 and OpenJDK 11-based images. OracleJDK-based images are not
supported. -->
  <properties>
    <fabric8.generator.from>registry.access.redhat.com/redhat-openjdk-18/openjdk18-
openshift:latest</fabric8.generator.from>
```

```
</properties>

<!-- Import dependencies from the Spring Boot BOM. -->
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>dev.snowdrop</groupId>
      <artifactId>snowdrop-dependencies</artifactId>
      <version>2.2.11.SP1-redhat-00001</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>

<dependencies>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-tomcat</artifactId>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-actuator</artifactId>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
</dependencies>

<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
      <version>2.2.11.RELEASE</version>
    </plugin>
  </plugins>
</build>

<!-- Specify the repositories containing Spring Boot artifacts -->
<repositories>
  <repository>
    <id>redhat-ga</id>
    <name>Red Hat GA Repository</name>
    <url>https://maven.repository.redhat.com/ga/</url>
  </repository>
</repositories>

<pluginRepositories>
  <pluginRepository>
    <id>redhat-ga</id>
    <name>Red Hat GA Repository</name>
    <url>https://maven.repository.redhat.com/ga/</url>
  </pluginRepository>
```

```
  </pluginRepositories>

</project>
```

5. アプリケーションのルートディレクトリーから Maven を使用してアプリケーションをビルドします。

```
$ mvn spring-boot:run
```

6. アプリケーションが実行していることを確認します。
curl またはブラウザーを使用して、アプリケーションが **http://localhost:8080** で稼働していることを確認します。

```
$ curl http://localhost:8080
{"content":"Greetings!"}
```

追加情報

- 推奨されるプラクティスとして、Liveness プローブおよび Readiness プローブを設定し、OpenShift で実行する際にアプリケーションのヘルスモニターリングを有効にできます。OpenShift でのアプリケーションのヘルスモニターリングの仕組みを確認するには、ヘルスチェックの例 を試してください。

4.2. SPRING BOOT アプリケーションの OPENSHIFT へのデプロイ

Spring Boot アプリケーションを OpenShift にデプロイするには、アプリケーションで **pom.xml** ファイルを設定し、Fabric8 Maven プラグインを使用します。**pom.xml** ファイルの **fabric8.generator.from** URL を置き換えて、Java イメージを指定できます。

イメージは Red Hat Ecosystem Catalog で利用できます。

```
<fabric8.generator.from>IMAGE_NAME</fabric8.generator.from>
```

たとえば、OpenJDK 8 を使用する RHEL 7 の Java イメージは、以下のように指定します。

```
<fabric8.generator.from>registry.access.redhat.com/redhat-openjdk-18/openjdk18-
openshift:latest</fabric8.generator.from>
```

4.2.1. Spring Boot でサポートされる Java イメージ

Spring Boot は、さまざまなオペレーティングシステムで利用可能なさまざまな Java イメージで認定およびテストされています。たとえば、Java イメージは、RHEL 7 で OpenJDK 8 または OpenJDK 11 で利用できます。

Spring Boot では、RHEL 8 上の Red Hat OpenJDK 8 および Red Hat OpenJDK 11 用の OCI 準拠の ユニバーサルベースイメージ を使用して、Spring Boot アプリケーションを OpenShift にビルドおよびデプロイできるようになりました。

同様のイメージが IBM Z で利用できます。

Red Hat Ecosystem Catalog で RHEL 8 イメージにアクセスするには、Docker または Podman 認証が必要です。

以下の表は、さまざまなアーキテクチャー向けに Spring Boot でサポートされるイメージの一覧です。また、Red Hat Ecosystem Catalog で利用可能なイメージへのリンクも提供します。イメージページには、RHEL 8 イメージへのアクセスに必要な認証手順が含まれています。

4.2.1.1. x86_64 アーキテクチャー上のイメージ

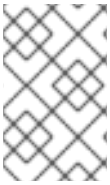
OS	Java	Red Hat Ecosystem Catalog
RHEL 7	OpenJDK 8	OpenJDK 8 を使用した RHEL 7
RHEL 7	OpenJDK 11	OpenJDK 11 を使用した RHEL 7
RHEL 8	OpenJDK 8	OpenJDK 8 での RHEL 8 Universal Base Image
RHEL 8	OpenJDK 11	OpenJDK 11 での RHEL 8 Universal Base Image

注記

RHEL 7 ホストでの RHEL 8 ベースのコンテナーの使用 (OpenShift 3 や OpenShift 4 など) は、サポートが限定されています。詳細は、Red Hat Enterprise Linux Container Compatibility Matrix を参照してください。

4.2.1.2. s390x (IBM Z) アーキテクチャー上のイメージ

OS	Java	Red Hat Ecosystem Catalog
RHEL 8	Eclipse OpenJ9 11	Eclipse OpenJ9 11 を使用した RHEL 8



注記

RHEL 7 ホストでの RHEL 8 ベースのコンテナーの使用 (OpenShift 3 や OpenShift 4 など) は、サポートが限定されています。詳細は、Red Hat Enterprise Linux Container Compatibility Matrix を参照してください。

4.2.2. OpenShift デプロイメント向け Spring Boot アプリケーションの準備

Spring Boot アプリケーションを OpenShift にデプロイするには、以下を含める必要があります。

- アプリケーションの **pom.xml** ファイルにあるランチャープロファイル情報。

以下の手順では、Fabric8 Maven プラグインを使用したプロファイルは、アプリケーションを OpenShift にビルドおよびデプロイするために使用されます。

前提条件

- Maven がインストールされている。

- Red Hat Ecosystem Catalog での Docker または Podman 認証による RHEL 8 イメージへのアクセスができる。

手順

1. 以下の内容を、アプリケーションルートディレクトリーの **pom.xml** ファイルに追加します。

```
...

<profiles>
  <profile>
    <id>openshift</id>
    <build>
      <plugins>
        <plugin>
          <groupId>io.fabric8</groupId>
          <artifactId>fabric8-maven-plugin</artifactId>
          <version>4.4.1</version>
          <executions>
            <execution>
              <goals>
                <goal>resource</goal>
                <goal>build</goal>
              </goals>
            </execution>
          </executions>
        </plugin>
      </plugins>
    </build>
  </profile>
</profiles>
```

2. **pom.xml** ファイルの **fabric8.generator.from** プロパティーを置き換えて、使用する OpenJDK イメージを指定します。

   - x86_64 アーキテクチャー
     - OpenJDK 8 を使用した RHEL 7

```
<fabric8.generator.from>registry.access.redhat.com/redhat-openjdk-18/openjdk18-openshift:latest</fabric8.generator.from>
```

     - OpenJDK 11 を使用した RHEL 7

```
<fabric8.generator.from>registry.access.redhat.com/openjdk/openjdk-11-rhel7:latest</fabric8.generator.from>
```

     - OpenJDK 8 を使用した RHEL 8

```
<fabric8.generator.from>registry.access.redhat.com/ubi8/openjdk-8:latest</fabric8.generator.from>
```

     - OpenJDK 11 を使用した RHEL 8

```
<fabric8.generator.from>registry.access.redhat.com/ubi8/openjdk-
11:latest</fabric8.generator.from>
```

- s390x (IBM Z) アーキテクチャー
  - Eclipse OpenJ9 11 を使用した RHEL 8

```
<fabric8.generator.from>registry.access.redhat.com/openj9/openj9-11-
rhel8:latest</fabric8.generator.from>
```

4.2.3. Fabric8 Maven プラグインを使用した Spring Boot アプリケーションの OpenShift へのデプロイ

Spring Boot アプリケーションを OpenShift にデプロイするには、以下を実行する必要があります。

- OpenShift インスタンスにログインします。
- OpenShift インスタンスにアプリケーションをデプロイします。

前提条件

- CLI クライアント **oc** がインストールされている。
- Maven がインストールされている。

手順

1. **oc** クライアントを使用して OpenShift インスタンスにログインします。

```
$ oc login ...
```

2. OpenShift インスタンスで新規プロジェクトを作成します。

```
$ oc new-project MY_PROJECT_NAME
```

3. アプリケーションのルートディレクトリーから Maven を使用してアプリケーションを OpenShift にデプロイします。アプリケーションのルートディレクトリーには **pom.xml** ファイルが含まれます。

```
$ mvn clean fabric8:deploy -Popenshift
```

このコマンドは Fabric8 Maven プラグインを使用して OpenShift で S2I プロセス を起動し、Pod を起動します。

4. デプロイメントを確認します。
   a. アプリケーションのステータスを確認し、Pod が実行していることを確認します。

```
$ oc get pods -w
NAME                      READY     STATUS      RESTARTS   AGE
MY_APP_NAME-1-aaaaa           1/1       Running     0          58s
MY_APP_NAME-s2i-1-build       0/1       Completed   0          2m
```

MY_APP_NAME-1-aaaaa Pod は、完全にデプロイされて起動すると、ステータスが **Running** になるはずです。

特定の Pod 名が異なります。

b. Pod のルートを確認します。

ルート情報の例

```
$ oc get routes
NAME              HOST/PORT                                   PATH      SERVICES
PORT      TERMINATION
MY_APP_NAME       MY_APP_NAME-
MY_PROJECT_NAME.OPENSHIFT_HOSTNAME      MY_APP_NAME      8080
```

Pod のルート情報には、アクセスに使用するベース URL が提供されます。

この例では、**http://MY_APP_NAME-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME** をベース URL として使用し、アプリケーションにアクセスできます。

c. アプリケーションが OpenShift で実行していることを確認します。

```
$ curl http://MY_APP_NAME-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME
{"content":"Greetings!"}
```

4.3. スタンドアロン RED HAT ENTERPRISE LINUX への SPRING BOOT アプリケーションのデプロイ

Spring Boot アプリケーションをスタンドアロンの Red Hat Enterprise Linux にデプロイするには、アプリケーションで **pom.xml** ファイルを設定し、Maven を使用してパッケージ化し、**java-jar** コマンドを使用してデプロイします。

前提条件

- RHEL 7 または RHEL 8 がインストールされている。

4.3.1. スタンドアロン Red Hat Enterprise Linux デプロイメント用の Spring Boot アプリケーションの準備

Spring Boot アプリケーションをスタンドアロンの Red Hat Enterprise Linux にデプロイするには、最初に Maven を使用してアプリケーションをパッケージ化する必要があります。

前提条件

- Maven がインストールされている。

手順

1. 以下の内容をアプリケーションの root ディレクトリーの **pom.xml** ファイルに追加します。

```
...
<!-- Specify target artifact type for the repackage goal. -->
<packaging>jar</packaging>
```

```
...
<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
      <version>${spring-boot.version}</version>
      <executions>
        <execution>
          <goals>
            <goal>repackage</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
...
```

2. Maven を使用してアプリケーションをパッケージ化します。

```
$ mvn clean package
```

作成される JAR ファイルは **target** ディレクトリーに置かれます。

4.3.2. jar を使用したスタンドアロン Red Hat Enterprise Linux への Spring Boot アプリケーションのデプロイ

Spring Boot アプリケーションをスタンドアロンの Red Hat Enterprise Linux にデプロイするには、**java -jar** コマンドを使用します。

前提条件

- RHEL 7 または RHEL 8 がインストールされている。
- OpenJDK 8 または OpenJDK 11 がインストールされている。
- アプリケーションが含まれる JAR ファイル。

手順

1. そのアプリケーションで JAR ファイルをデプロイします。

```
$ java -jar my-project-1.0.0.jar
```

2. デプロイメントを確認します。
curl またはブラウザーを使用して、アプリケーションが **http://localhost:8080** で稼働していることを確認します。

```
$ curl http://localhost:8080
```

第5章 ECLIPSE VERT.X で SPRING BOOT を使用したリアクティブアプリケーションの開発

本セクションでは、Spring Boot および Eclipse Vert.x をベースとした Spring Boot スターターを使用して、リアクティブでアプリケーションを開発する方法を説明します。以下の例は、スターターを使用してリアクティブアプリケーションを作成する方法を表しています。

5.1. ECLIPSE VERT.X での SPRING BOOT の概要

Spring リアクティブスタックは Project Reactor 上に構築され、バックプレーシャーを実装し、Reactive Streams 仕様に準拠しています。非同期イベントストリーム処理を有効にする **Flux** および **Mono** 機能 API タイプを提供します。

Spring は、Project Reactor の上部に WebFlux (非同期イベント駆動型の Web アプリケーションフレームワーク) を提供します。WebFlux は、主に Reactor Netty と連携するように設計されていますが、Eclipse Vert.x などの他のリアクティブ HTTP サーバーでも操作できます。

Spring WebFlux および Reactor を使用すると、以下のアプリケーションを作成できます。

- ブロックなし: アプリケーションは、現在の要求の完了に必要なリモートコンポーネントまたはサービスからの応答を待機しているときに、追加の要求を引き続き処理します。
- 非同期: アプリケーションは応答イベントを生成し、それらをアプリケーション内の他のクライアントが取得できるイベントストリームに再び公開することで、イベントストリームからイベントに応答します。
- イベント駆動型: アプリケーションは、マウスクリック、HTTP 要求、ストレージに追加される新しいファイルなど、ユーザーまたは別のサービスが生成するイベントに応答します。
- スケーラブル: アプリケーションの必要なイベント処理容量のみに一致する Publisher または Subscriber の数が増えると、アプリケーションの個別クライアント間のルーティング要求の複雑性が向上します。リアクティブアプリケーションは、他のアプリケーションプログラミングモデルと比較して、コンピューティングおよびネットワークリソースを使用して多数のイベントを処理できます。
- 耐久性: アプリケーションは、サービス全体に影響を与えることなく、依存するサービス障害を処理できます。

Spring WebFlux を使用する追加の利点には以下が含まれます。

SpringMVC での類似性

SpringMVC API タイプと WebFlux API タイプは類似しており、開発者は SpringMVC の知識を WebFlux を使用したプログラミングアプリケーションに簡単に適用できます。

Red Hat による Spring Reactive 製品は、Reactor と WebFlux から OpenShift とスタンドアロンの RHEL の利点を提供し、WebFLux フレームワークの Eclipse Vert.x 拡張のセットを導入します。これにより、Spring Boot の抽象化および高速なプロトタイプ機能を保持でき、アプリケーション内のサービス間のネットワーク通信を完全にリアクティブで処理する非同期 IO API を利用できます。

アノテーション付きコントローラーのサポート

WebFlux は、SpringMVC により導入されるエンドポイントコントローラーアノテーションを保持します (SpringMVC および WebFlux はリアクティブな RxJava2 および Reactor の戻り値タイプ)。

機能的なプログラミングサポート

Reactor の Java 8 Functional API とだけでなく、**CompletablebFuture** API、および **Stream** API とも相互作用します。アノテーションベースのエンドポイントに加えて、WebFlux は機能的なエンドポイントもサポートします。

その他のリソース

Spring Reactive スタックの一部であるテクノロジーの実装に関する詳細は、以下のリソースを参照してください。

- リアクティブマニフェスト
- リアクティブストリームの仕様
- Spring Framework リファレンスドキュメント: Reactive Stack の Web アプリケーション
- Reactor Netty ドキュメント
- プロジェクト Reactor ドキュメントの **Mono** クラスの API リファレンスページ
- プロジェクト Reactor ドキュメントの **Flux** クラスの API リファレンスページ

5.2. リアクティブ SPRING WEB

spring-web モジュールは、Spring WebFlux のリアクティブ機能の基盤要素を提供します。以下が含まれます。

- **HttpHandler** API によって提供される HTTP 抽象化
- サポートされるサーバーのリアクティブストリームアダプター (Eclipse Vert.x、Undertow など)
- イベントストリームデータをエンコードおよびデコードするためのコードデックです。これには、以下が含まれます。
  - **DataBuffer**、さまざまなタイプのバイトバッファー表現 Netty **ByteBuf**、**java.nio.ByteBuffer** など) の抽象化
  - HTTP とは関係なくコンテンツをエンコードおよびデコードする低レベルの契約
  - HTTP メッセージコンテンツのエンコードおよびデコードを行う **HttpMessageReader** および **HTTPMessageWriter** コントラクト
- **WebHandler** API (非ブロッキング契約を使用する Servlet 3.1 I/O API に対応)

Web アプリケーションを設計する場合は、Spring WebFlux が以下を提供する 2 つのプログラミングモデルのいずれかを選択できます。

アノテーション付きコントローラー

Spring WebFlux のアノテーションが付けられたコントローラーは Spring MVC と一致しており、**spring-web** モジュールと同じアノテーションに基づいています。SpringMVC の **spring-web** モジュールに加えて、WebFlux の対応する **@RequestBody** 引数もサポートしています。

機能エンドポイント

Java 8 Lambda 式および機能 API の Spring WebFlux が提供する機能エンドポイント。このプログラミングモデルは、要求をルーティングおよび処理する専用のライブラリー (この場合は Reactor など) に依存します。インテントの宣言とコールバックの使用に依存してアクティビティーを完了する

アノテーションベースのエンドポイントコントローラーとは対照的に、機能エンドポイントに基づくリアクティブモデルでは、要求の処理をアプリケーションで完全に制御できます。

5.3. WEBFLUX でのリアクティブ SPRING BOOT HTTP サービスの作成

Spring Boot および WebFlux を使用して、基本的なリアクティブ Hello World HTTP Web サービスを作成します。

前提条件

- JDK 8 または JDK 11 がインストールされている。
- Maven がインストールされている。
- Spring Boot を使用するよう設定された Maven ベースのアプリケーションプロジェクト

手順

1. **vertx-spring-boot-starter-http** をプロジェクトの **pom.xml** ファイルに依存関係として追加します。

   pom.xml

```
<project>
...
  <dependencies>
  ...
    <dependency>
      <groupId>dev.snowdrop</groupId>
      <artifactId>vertx-spring-boot-starter-http</artifactId>
    </dependency>
  ...
  <dependencies>
...
</project>
```

2. アプリケーションのメインクラスを作成し、ルーターおよびハンドラーメソッドを定義します。

   HttpSampleApplication.java

```
package dev.snowdrop.vertx.sample.http;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;
import org.springframework.web.reactive.function.server.RouterFunction;
import org.springframework.web.reactive.function.server.ServerRequest;
import org.springframework.web.reactive.function.server.ServerResponse;
import reactor.core.publisher.Mono;

import static org.springframework.web.reactive.function.BodyInserters.fromObject;
import static org.springframework.web.reactive.function.server.RouterFunctions.route;
import static org.springframework.web.reactive.function.server.ServerResponse.ok;
```

```
@SpringBootApplication
public class HttpSampleApplication {

    public static void main(String[] args) {
        SpringApplication.run(HttpSampleApplication.class, args);
    }

    @Bean
    public RouterFunction<ServerResponse> helloRouter() {
        return route()
            .GET("/hello", this::helloHandler)
            .build();
    }

    private Mono<ServerResponse> helloHandler(ServerRequest request) {
        String name = request
            .queryParam("name")
            .orElse("World");
        String message = String.format("Hello, %s!", name);

        return ok()
            .body(fromObject(message));
    }
}
```

3. オプション: アプリケーションをローカルで実行し、テストします。

   a. Maven プロジェクトのルートディレクトリーへ移動します。

   ```
   $ cd myApp
   ```

   b. アプリケーションをパッケージ化します。

   ```
   $ mvn clean package
   ```

   c. コマンドラインからアプリケーションを起動します。

   ```
   $ java -jar target/vertx-spring-boot-sample-http.jar
   ```

   d. 新しいターミナルウィンドウで、**/hello** エンドポイントで HTTP 要求を発行します。

   ```
   $ curl localhost:8080/hello
   Hello, World!
   ```

   e. カスタム名と、パーソナルな応答を取得するために要求を指定します。

   ```
   $ curl http://localhost:8080/hello?name=John
   Hello, John!
   ```

その他のリソース

- Fabric8 Maven プラグインを使用して、アプリケーションを OpenShift クラスターにデプロイできます。

- また、スタンドアロンの Red Hat Enterprise Linux でのデプロイメント 用にアプリケーションを設定することもできます。
- Spring Boot でリアクティブ Web サービスの作成に関する詳細は、Spring コミュニティードキュメントの リアクティブ REST サービスの開発ガイド を参照してください。

5.4. リアクティブ SPRING BOOT WEBFLUX アプリケーションで BASIC 認証の使用

Spring Security および WebFlux スターターを使用して、リアクティブ Hello World HTTP Web サービスを作成し、基本的なフォームベースの認証を作成します。

前提条件

- JDK 8 または JDK 11 がインストールされている。
- Maven がインストールされている。
- Spring Boot を使用するよう設定されたMaven ベースのアプリケーションプロジェクト

手順

1. **vertx-spring-boot-starter-http** および **spring-boot-starter-security** をプロジェクトの **pom.xml** ファイルに依存関係として追加します。

   pom.xml

```
<project>
...
  <dependencies>
  ...
    <dependency>
      <groupId>dev.snowdrop</groupId>
      <artifactId>vertx-spring-boot-starter-http</artifactId>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-security</artifactId>
    </dependency>
  ...
  <dependencies>
...
</project>
```

2. アプリケーションのエンドポイントコントローラークラスを作成します。

   HelloController.java

```
package dev.snowdrop.vertx.sample.http.security;

import java.security.Principal;

import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;
```

```
import reactor.core.publisher.Mono;

@RestController
public class HelloController {

    @GetMapping("/")
    public Mono<String> hello(Mono<Principal> principal) {
        return principal
            .map(Principal::getName)
            .map(this::helloMessage);
    }

    private String helloMessage(String username) {
        return "Hello, " + username + "!";
    }
}
```

3. アプリケーションのメインクラスを作成します。

HttpSecuritySampleApplication.java

```
package dev.snowdrop.vertx.sample.http.security;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class HttpSecuritySampleApplication {

    public static void main(String[] args) {
        SpringApplication.run(HttpSecuritySampleApplication.class, args);
    }
}
```

4. **/hello** エンドポイントにアクセスするためのユーザー認証情報を格納する **SecurityConfiguration** クラスを作成します。

SecurityConfiguration.java

```
package dev.snowdrop.vertx.sample.http.security;

import org.springframework.context.annotation.Bean;
import org.springframework.security.config.annotation.web.reactive.EnableWebFluxSecurity;
import org.springframework.security.core.userdetails.MapReactiveUserDetailsService;
import org.springframework.security.core.userdetails.User;
import org.springframework.security.core.userdetails.UserDetails;

@EnableWebFluxSecurity
public class SecurityConfiguration {

    @Bean
    public MapReactiveUserDetailsService userDetailsService() {
        UserDetails user = User.withDefaultPasswordEncoder()
            .username("user")
            .password("user")
```

```
            .roles("USER")
            .build();

        return new MapReactiveUserDetailsService(user);
    }
}
```

5. オプション: アプリケーションをローカルで実行し、テストします。

   a. Maven プロジェクトのルートディレクトリーへ移動します。

   ```
   $ cd myApp
   ```

   b. アプリケーションをパッケージ化します。

   ```
   $ mvn clean package
   ```

   c. コマンドラインからアプリケーションを起動します。

   ```
   $ java -jar target/vertx-spring-boot-sample-http-security.jar
   ```

   d. ブラウザーを使用して **http://localhost:8080** に移動し、ログイン画面にアクセスします。

   e. 以下の認証情報を使用してログインします。

   - username: user
   - password: user

   以下のログイン時に、カスタマイズされた挨拶文を受け取ります。

   ```
   Hello, user!
   ```

   f. Web ブラウザーを使用して **http://localhost:8080/logout** に移動し、**Log out** ボタンを使用してアプリケーションからログアウトします。

   g. または、ターミナルを使用して **localhost:8080** で認証されていない HTTP 要求を実行します。アプリケーションから HTTP **401 Unauthorized** 応答を受信します。

   ```
   $ curl -I http://localhost:8080
   HTTP/1.1 401 Unauthorized
   WWW-Authenticate: Basic realm="Realm"
   Cache-Control: no-cache, no-store, max-age=0, must-revalidate
   Pragma: no-cache
   Expires: 0
   X-Content-Type-Options: nosniff
   X-Frame-Options: DENY
   X-XSS-Protection: 1 ; mode=block
   Referrer-Policy: no-referrer
   ```

   h. サンプルユーザー認証情報を使用して認証された要求を発行します。パーソナルな応答を受け取ります。

```
$ curl -u user:user http://localhost:8080
Hello, user!
```

その他のリソース

- Fabric8 Maven プラグインを使用して、アプリケーションを OpenShift クラスターにデプロイできます。
- また、スタンドアロンの Red Hat Enterprise Linux でのデプロイメント 用にアプリケーションを設定することもできます。
- Basic HTTP 認証スキームの詳細は RFC-7617 のドキュメントを参照してください。
- フォームベースの認証など、対話式クライアントの HTTP 認証拡張の完全仕様は、RFC-8053 を参照してください。

5.5. リアクティブ SPRING BOOT アプリケーションで OAUTH2 認証の使用。

リアクティブ Spring Boot アプリケーションの OAuth2 認証 を設定し、クライアント ID およびクライアントシークレットを使用して認証します。

前提条件

- JDK 8 または JDK 11 がインストールされている。
- Maven がインストールされている。
- Spring Boot を使用するよう設定されたMaven ベースのアプリケーションプロジェクト
- GitHub アカウント

手順

1. Github アカウントに 新規 OAuth 2 アプリケーションを登録 します。登録フォームに以下の値を指定するようにしてください。
   - ホームページ URL: **http://localhost:8080**
   - 承認コールバック URL: **http://localhost:8080/login/oauth2/code/github**
     登録が完了したら、クライアント ID とクライアントシークレットを保存します。
2. プロジェクトの **pom.xml** ファイルに以下の依存関係を追加します。
   - **vertx-spring-boot-starter-http**
   - **spring-boot-starter-security**
   - **spring-boot-starter-oauth2-client**
   - **reactor-netty**
     spring-boot-starter-oauth2-client が適切に動作するには、**reactor-netty** クライアントが必要です。

     pom.xml

```
<project>
...
  <dependencies>
  ...
    <dependency>
      <groupId>dev.snowdrop</groupId>
      <artifactId>vertx-spring-boot-starter-http</artifactId>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-security</artifactId>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-oauth2-client</artifactId>
    </dependency>
    <!-- Spring OAuth2 client only works with Reactor Netty client -->
    <dependency>
      <groupId>io.projectreactor.netty</groupId>
      <artifactId>reactor-netty</artifactId>
    </dependency>
  ...
  <dependencies>
...
</project>
```

3. アプリケーションのエンドポイントコントローラークラスを作成します。

HelloController.java

```
package dev.snowdrop.vertx.sample.http.oauth;

import org.springframework.security.core.annotation.AuthenticationPrincipal;
import org.springframework.security.oauth2.core.user.OAuth2User;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;
import reactor.core.publisher.Mono;

@RestController
public class HelloController {

    @GetMapping
    public Mono<String> hello(@AuthenticationPrincipal OAuth2User oauth2User) {
        return Mono.just("Hello, " + oauth2User.getAttributes().get("name") + "!");
    }
}
```

4. アプリケーションのメインクラスを作成します。

OAuthSampleApplication.java

```
package dev.snowdrop.vertx.sample.http.oauth;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
```

```
@SpringBootApplication
public class OAuthSampleApplication {

    public static void main(String[] args) {
        SpringApplication.run(OAuthSampleApplication.class, args);
    }
}
```

5. YAML 設定ファイルを作成して、アプリケーション登録時に GitHub から受け取った OAuth2 クライアント ID およびクライアントシークレットを保存します。

   src/main/resources/application.yml

```
spring:
  security:
    oauth2:
      client:
        registration:
          github:
            client-id: YOUR_GITHUB_CLIENT_ID
            client-secret: YOUR_GITHUB_CLIENT_SECRET
```

6. オプション: アプリケーションをローカルで実行し、テストします。

   a. Maven プロジェクトのルートディレクトリーへ移動します。

```
$ cd myApp
```

   b. アプリケーションをパッケージ化します。

```
$ mvn clean package
```

   c. コマンドラインからアプリケーションを起動します。

```
$ java -jar target/vertx-spring-boot-sample-http-oauth.jar
```

   d. Web ブラウザーを使用して **http://localhost:8080** に移動します。GitHub の OAuth2 アプリケーション認可画面にリダイレクトされます。プロンプトが表示されたら、GitHub アカウントの認証情報を使用してログインします。

   e. **Authorize** をクリックして確定します。パーソナライズされたグリーティングメッセージを示す画面にリダイレクトされます。

その他のリソース

- Fabric8 Maven プラグインを使用して、アプリケーションを OpenShift クラスターにデプロイできます。
- また、スタンドアロンの Red Hat Enterprise Linux でのデプロイメント 用にアプリケーションを設定することもできます。
- 詳細は、Spring コミュニティードキュメントの OAuth2 チュートリアル を参照してください。または、Spring Security の OAuth2 を使用するためのチュートリアルを参照してください。

- OAuth2 認証フレームワークの完全な仕様については、RFC-6749 を参照してください。

5.6. リアクティブ SPRING BOOT SMTP メールアプリケーションの作成

Eclipse Vert.x を使用して Spring Boot でリアクティブ SMTP メールサービスを作成します。

前提条件

- JDK 8 または JDK 11 がインストールされている。
- Maven がインストールされている。
- Spring Boot を使用するよう設定されたMaven ベースのアプリケーションプロジェクト
- マシンに設定された SMTP メールサーバー

手順

1. **vertx-spring-boot-starter-http** および **vertx-spring-boot-starter-mail** をプロジェクトの **pom.xml** ファイルに依存関係として追加します。

   pom.xml

```
<project>
...
  <dependencies>
  ...
    <dependency>
      <groupId>dev.snowdrop</groupId>
      <artifactId>vertx-spring-boot-starter-http</artifactId>
    </dependency>
    <dependency>
      <groupId>dev.snowdrop</groupId>
      <artifactId>vertx-spring-boot-starter-mail</artifactId>
    </dependency>
  ...
  <dependencies>
...
</project>
```

2. アプリケーションのメールハンドラークラスを作成します。

   MailHandler.java

```
package dev.snowdrop.vertx.sample.mail;

import dev.snowdrop.vertx.mail.MailClient;
import dev.snowdrop.vertx.mail.MailMessage;
import dev.snowdrop.vertx.mail.SimpleMailMessage;
import org.springframework.stereotype.Component;
import org.springframework.util.MultiValueMap;
import org.springframework.web.reactive.function.server.ServerRequest;
import org.springframework.web.reactive.function.server.ServerResponse;
import reactor.core.publisher.Mono;
```

```
import static org.springframework.web.reactive.function.server.ServerResponse.noContent;

@Component
public class MailHandler {

    private final MailClient mailClient;

    public MailHandler(MailClient mailClient) {
        this.mailClient = mailClient;
    }

    public Mono<ServerResponse> send(ServerRequest request) {
        return request.formData()
            .log()
            .map(this::formToMessage)
            .flatMap(mailClient::send)
            .flatMap(result -> noContent().build());
    }

    private MailMessage formToMessage(MultiValueMap<String, String> form) {
        return new SimpleMailMessage()
            .setFrom(form.getFirst("from"))
            .setTo(form.get("to"))
            .setSubject(form.getFirst("subject"))
            .setText(form.getFirst("text"));
    }

}
```

3. アプリケーションのメインクラスを作成します。

MailSampleApplication.java

```
package dev.snowdrop.vertx.sample.mail;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;
import org.springframework.core.io.ClassPathResource;
import org.springframework.web.reactive.function.server.RouterFunction;
import org.springframework.web.reactive.function.server.ServerResponse;

import static org.springframework.http.MediaType.APPLICATION_FORM_URLENCODED;
import static org.springframework.web.reactive.function.server.RequestPredicates.accept;
import static org.springframework.web.reactive.function.server.RouterFunctions.resources;
import static org.springframework.web.reactive.function.server.RouterFunctions.route;

@SpringBootApplication
public class MailSampleApplication {

    public static void main(String[] args) {
        SpringApplication.run(MailSampleApplication.class, args);
    }

    @Bean
```

```
    public RouterFunction<ServerResponse> mailRouter(MailHandler mailHandler) {
        return route()
                .POST("/mail", accept(APPLICATION_FORM_URLENCODED),
mailHandler::send)
                .build();
    }

    @Bean
    public RouterFunction<ServerResponse> staticResourceRouter() {
        return resources("/**", new ClassPathResource("static/"));
    }

}
```

4. **application.properties** ファイルを作成して、SMTP サーバー認証情報を保存します。

   application.properties

```
vertx.mail.host=YOUR_SMTP_SERVER_HOSTNAME
vertx.mail.username=YOUR_SMTP_SERVER_USERNAME
vertx.mail.password=YOUR_SMTP_SERVER_PASSWORD
```

5. アプリケーションのフロントエンドとして機能する **src/main/resources/static/index.html** ファイルを作成します。または、この手順で使用できる HTML メールフォームの例 を使用してください。

6. オプション: アプリケーションをローカルで実行し、テストします。

   a. Maven プロジェクトのルートディレクトリーへ移動します。

```
$ cd myApp
```

   b. アプリケーションをパッケージ化します。

```
$ mvn clean package
```

   c. コマンドラインからアプリケーションを起動します。

```
$ java -jar target/vertx-spring-boot-sample-mail.jar
```

   d. Web ブラウザーを使用して **http://localhost:8080/index.html** に移動し、メールフォームにアクセスします。

その他のリソース

- RHEL 7 で SMTP メールサーバーを設定する方法は、RHEL 7 ドキュメントの Mail Transport Agent Configuration セクションを参照してください。
- Fabric8 Maven プラグインを使用して、アプリケーションを OpenShift クラスターにデプロイできます。
- また、スタンドアロンの Red Hat Enterprise Linux でのデプロイメント 用にアプリケーションを設定することもできます。

5.7. サーバー向けイベント

サーバー送信イベント (SSE) は、HTTP サーバーがクライアントに単方向の更新を送信できるようにするプッシュテクノロジーです。SSE は、イベントソースとクライアント間の接続を確立することで機能します。イベントソースはこの接続を使用してイベントをクライアント側にプッシュします。サーバーがイベントをプッシュした後、接続は開いたままになり、後続のイベントをプッシュするのに使用できます。クライアントがサーバー上のリクエストを終了すると、接続は閉じられます。SSE は、クライアントが更新のためにイベントソースをポーリングするたびに新しい接続を確立する必要があるポーリングの代替手段を表します。WebSocket とは対照的に、SSE は 1 つの方向でイベントをプッシュします (つまり、ソースからクライアントへ)。イベントソースとクライアント間の双方向通信を処理しません。

SSE の指定は HTML5 に組み込まれており、レガシーバージョンを含む Web ブラウザーで広くサポートされます。SSE はコマンドラインから使用でき、他のプロトコルと比較してセットアップが比較的簡単です。

SSE は、サーバーからクライアントに頻繁に更新を必要とするユースケースに適していますが、クライアント側からサーバーへの更新の頻度は低くなることが予想されます。クライアント側からサーバーへの更新は、REST などの別のプロトコルで処理できます。このようなユースケースの例には、新規ファイルがファイルサーバーにアップロードされる時に、ソーシャルメディアフィードの更新やクライアントに送信される通知が含まれます。

5.8. リアクティブ SPRING BOOT アプリケーションでのサーバー送信イベントの使用

HTTP リクエストを受け入れ、サーバー送信イベント (SSE) のストリームを返す単純なサービスを作成します。クライアントがサーバーへの接続を確立し、ストリーミングを開始すると、接続は開いたままになります。サーバーは接続を再利用して、新しいイベントをクライアントに継続的にプッシュします。リクエストをキャンセルすると、接続が閉じられてストリームが停止し、クライアントがサーバーの更新フォームの受信を停止させます。

前提条件

- JDK 8 または JDK 11 がインストールされている。
- Maven がインストールされている。
- Spring Boot を使用するよう設定された Maven ベースのアプリケーションプロジェクト

手順

1. **vertx-spring-boot-starter-http** をプロジェクトの **pom.xml** ファイルに依存関係として追加します。

   pom.xml

```
<project>
...
  <dependencies>
  ...
    <dependency>
      <groupId>dev.snowdrop</groupId>
      <artifactId>vertx-spring-boot-starter-http</artifactId>
    </dependency>
  ...
```

```
  <dependencies>
...
</project>
```

2. アプリケーションのメインクラスを作成します。

SseExampleApplication.java

```
package dev.snowdrop.vertx.sample.sse;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class SseSampleApplication {

    public static void main(String[] args) {
        SpringApplication.run(SseSampleApplication.class, args);
    }
}
```

3. アプリケーションのサーバー送信イベントコントローラークラスを作成します。この例では、クラスはランダムな整数のストリームを生成し、それらをターミナルアプリケーションに出力します。

SseController.java

```
package dev.snowdrop.vertx.sample.sse;

import java.time.Duration;
import java.util.Random;

import org.springframework.http.MediaType;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;
import reactor.core.publisher.Flux;

@RestController
public class SseController {

    @GetMapping(produces = MediaType.TEXT_EVENT_STREAM_VALUE)
    public Flux<Integer> getRandomNumberStream() {
        Random random = new Random();

        return Flux.interval(Duration.ofSeconds(1))
            .map(i -> random.nextInt())
            .log();
    }
}
```

4. オプション: アプリケーションをローカルで実行し、テストします。

   a. Maven プロジェクトのルートディレクトリーへ移動します。

```
$ cd myApp
```

b. アプリケーションをパッケージ化します。

```
$ mvn clean package
```

c. コマンドラインからアプリケーションを起動します。

```
$ java -jar target/vertx-spring-boot-sample-sse.jar
```

d. 新しいターミナルウィンドウで、HTTP 要求を **localhost** に発行します。サーバーから送信されたイベントコントローラーからランダムな整数の連続ストリームの受信を開始します。

```
$ curl localhost:8080
data:-2126721954

data:-573499422

data:1404187823

data:1338766210

data:-666543077
...
```

Ctrl+C を押して HTTP 要求をキャンセルし、応答のストリームを終了します。

その他のリソース

- Fabric8 Maven プラグインを使用して、アプリケーションを OpenShift クラスターにデプロイできます。
- また、スタンドアロンの Red Hat Enterprise Linux でのデプロイメント 用にアプリケーションを設定することもできます。

5.9. WEBSOCKET プロトコル

WebSocket プロトコルは標準の HTTP 接続をアップグレードして、これを永続化し、その接続を使用してアプリケーションのクライアントとサーバーの間に特別にフォーマットされたメッセージを渡します。プロトコルは、TCP を介してクライアントとサーバー間で最初の接続を確立するためにハンドシェイクなどの HTTP に依存しますが、クライアントとサーバー間の通信に特別なメッセージ形式が使用されます。

標準の HTTP 接続、WebSocket 接続とは異なり、以下を実行します。

- 両方の方向でメッセージを送信するために使用できます。
- 最初の要求が完了した後も開いたままになります。
- メッセージで特別なフレーミングヘッダーを使用します。これにより、HTTP 要求内で HTTP 形式ではないメッセージペイロード (制御データなど) を送信できます。

その結果、WebSockets プロトコルは、必要なネットワークリソースを減らし、ネットワークタイムアウトが原因でサービスが失敗するリスクを減らしながら、標準の HTTP 接続の可能性を拡張します (HTTP Long Polling などのリアルタイムメッセージング機能を提供する代替方法と比較して)。

Websocket 接続は、デフォルトで、さまざまなオペレーティングシステムやハードウェアアーキテクチャー全体で現在利用可能なほとんどの Web ブラウザーでサポートされます。これにより、WebSocket は Web ブラウザーのみを使用するために接続できるクロスプラットフォームの Web ベースのアプリケーションを作成するのに適しています。

5.10. WEBFLUX をベースとしたリアクティブアプリケーションでの WEBSOCKET の使用

以下の例は、Web ブラウザーを使用して接続できるバックエンドサービスを提供するアプリケーションで WebSocket プロトコルを使用する方法を示しています。Web ブラウザーを使用してアプリケーションの Web フロントエンド URL にアクセスすると、フロントエンドはバックエンドサービスへの WebSocket 接続を開始します。Web サイトで利用可能な Web フォームを使用して、WebSocket 接続を使用して、テキスト文字列としてフォーマットされた値をバックエンドサービスに送信できます。アプリケーションは、すべての文字を大文字に変換して受信した値を処理し、同じ WebSocket 接続を使用して結果をフロントエンドに送信します。

以下で設定される Reactive Stack で Spring を使用してアプリケーションを作成します。

- WebSocket ハンドラーを使用したバックエンド Java ベースのサービス
- HTML および JavaScript に基づく Web フロントエンド

前提条件

- Spring Boot を使用する Maven ベースの Java アプリケーションプロジェクト
- JDK 8 または JDK 11 がインストールされている。
- Maven がインストールされている。

手順：

1. **vertx-spring-boot-starter-http** をアプリケーションプロジェクトの **pom.xml** ファイルに依存関係として追加します。

   pom.xml

   ```
   ...
   <dependencies>
     ...
     <dependency>
       <groupId>dev.snowdrop</groupId>
       <artifactId>vertx-spring-boot-starter-http</artifactId>
     </dependency>
     ...
   </dependencies>
   ...
   ```

2. バックエンドアプリケーションコードが含まれるクラスファイルを作成します。

   /src/main/java/webSocketSampleApplication.java

   ```
   package dev.snowdrop.WebSocketSampleApplication;
   ```

```
import java.util.Collections;
import java.util.Map;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;
import org.springframework.web.reactive.HandlerMapping;
import org.springframework.web.reactive.handler.SimpleUrlHandlerMapping;
import org.springframework.web.reactive.socket.WebSocketHandler;
import org.springframework.web.reactive.socket.WebSocketMessage;
import org.springframework.web.reactive.socket.WebSocketSession;

import reactor.core.publisher.Flux;
import reactor.core.publisher.Mono;

@SpringBootApplication
public class WebSocketSampleApplication {

    public static void main(String[] args) {
        SpringApplication.run(WebSocketSampleApplication.class, args);
    }

    @Bean
    public HandlerMapping handlerMapping() {
        // Define URL mapping for the socket handlers
        Map<String, WebSocketHandler> handlers = Collections.singletonMap("/echo-upper",
this::toUppercaseHandler);

        SimpleUrlHandlerMapping handlerMapping = new SimpleUrlHandlerMapping();
        handlerMapping.setUrlMap(handlers);
        // Set a higher precedence than annotated controllers (smaller value means higher
precedence)
        handlerMapping.setOrder(-1);

        return handlerMapping;
    }

    private Mono<Void> toUppercaseHandler(WebSocketSession session) {
        Flux<WebSocketMessage> messages = session.receive() // Get incoming messages
stream
            .filter(message -> message.getType() == WebSocketMessage.Type.TEXT) // Filter
out non-text messages
            .map(message -> message.getPayloadAsText().toUpperCase()) // Execute service
logic
            .map(session::textMessage); // Create a response message

        return session.send(messages); // Send response messages
    }
}
```

3. アプリケーションのフロントエンドとして機能する HTML ドキュメントを作成します。以下の例の **<script>** 要素には、アプリケーションのバックエンドとの通信を処理する JavaScript コードが含まれます。

/src/main/resources/static/index.html

```
<!doctype html>
<html>
<head>
  <meta charset="utf-8"/>
  <title>WebSocket Example</title>
  <script>
  const socket = new WebSocket("ws://localhost:8080/echo-upper");

  socket.onmessage = function(e) {
    console.log("Received a value: " + e.data);
    const messages = document.getElementById("messages");
    const message = document.createElement("li");
    message.innerHTML = e.data;
    messages.append(message);
  }

  window.onbeforeunload = function(e) {
    console.log("Closing socket");
    socket.close();
  }

  function send(event) {
    event.preventDefault();

    const value = document.getElementById("value-to-send").value.trim();
    if (value.length > 0) {
      console.log("Sending value to socket: " + value);
      socket.send(value);
    }
  }
  </script>
</head>
<body>
<div>
  <h1>Vert.x Spring Boot WebSocket example</h1>
  <p>
    Enter a value to the form below and click submit. The value will be sent via socket to a
backend service.
    The service will then uppercase the value and send it back via the same socket.
  </p>
</div>
<div>
  <form onsubmit="send(event)">
    <input type="text" id="value-to-send" placeholder="A value to be sent"/>
    <input type="submit"/>
  </form>
</div>
<div>
  <ol id="messages"></ol>
</div>
</body>
</html>
```

4. オプション: アプリケーションをローカルで実行し、テストします。

    a. Maven プロジェクトのルートディレクトリーへ移動します。

```
$ cd myApp
```

b. アプリケーションをパッケージ化します。

```
$ mvn clean package
```

c. コマンドラインからアプリケーションを起動します。

```
$ java -jar target/vertx-spring-boot-sample-websocket.jar
```

d. Web ブラウザーを使用して **http://localhost:8080/index.html** に移動します。Web サイトには、次を含む Web インターフェイスが表示されます。

- 入力テキストボックス
- 処理された結果のリスト
- **Submit** ボタン。

5. テキストボックスに文字列値を入力して **Submit** を選択します。
6. 入力テキストボックスの配下にあるリストで大文字でレンダリングされた値を確認します。

その他のリソース

- Fabric8 Maven プラグインを使用して、アプリケーションを OpenShift クラスターにデプロイできます。
- また、スタンドアロンの Red Hat Enterprise Linux でのデプロイメント 用にアプリケーションを設定することもできます。

5.11. ADVANCED MESSAGE QUEUING PROTOCOL

Advanced Message Queuing Protocol (AMQP) は、ブロック以外の方法でアプリケーション間でメッセージを移動するために設計された通信プロトコルです。このプロトコルは AMQP 1.0 として標準化され、異なるネットワークトポロジーと環境における新しいアプリケーションとレガシーアプリケーションとの間の相互運用性とメッセージング統合を提供します。AMQP は複数のブローカーアーキテクチャーと連携し、さまざまな方法でメッセージの配信、受信、キューイング、およびルーティングを提供します。また、AMQP はブローカーを使用しない場合にピアツーピアも機能します。ハイブリッドクラウド環境では、AMQP を使用してさまざまな異なるメッセージ形式を処理することなく、サービスをレガシーアプリケーションに統合できます。AMQP はリアルタイムの非同期メッセージ処理機能をサポートするため、リアクティブアプリケーションでの使用に適しています。

5.12. AMQP リアクティブサンプルの仕組み

この例の機能があるメッセージング統合パターンは、2 つのキューとブローカーを持つ Publisher-Subscriber パターンです。

- Request キューは、テキスト文字列プロセッサーによって処理される Web インターフェイスを使用して入力する文字列を含む HTTP リクエストを保存します。
- Result キューは、Uppercase に変換され、表示される準備が整う文字列が含まれる応答を保存します。

アプリケーションで設定されるコンポーネントは、以下のとおりです。

- テキスト文字列をアプリケーションに送信するのに使用できるフロントエンドサービス。
- 文字列を大文字に変換するバックエンドサービス。
- Spring Boot HTTP Starter によって設定され、提供される HTTP コントローラー。
- 2 つのメッセージングキュー間でメッセージをルーティングする組み込み Artemis AMQP Broker インスタンス。

要求キューは、テキスト文字列を含むメッセージをフロントエンドからテキスト文字列プロセッサーサービスに渡します。処理用の文字列を送信する場合は、以下を行います。

1. フロントエンドサービスは、リクエストのペイロードとして文字列を含む HTTP **POST** リクエストを HTTP コントローラーに送信します。
2. リクエストは、AMQP ブローカーへメッセージをルーティングするメッセージングマネージャーによって選択されます。
3. ブローカーはメッセージをテキスト文字列プロセッサーサービスにルーティングします。テキストプロセッサーサービスがリクエストを取得できない場合には、このようなインスタンスが利用可能な場合、ブローカーはメッセージを次の利用可能なプロセッサーインスタンスにルーティングします。または、ブローカーは、再度利用可能になったときに、同じインスタンスにリクエストを再送信するまで待機します。
4. テキスト文字列プロセッサーサービスはメッセージを取得して、文字列の文字を大文字に変換します。プロセッサーサービスは処理済みのリクエストを大文字で AMQP ブローカーに送信します。
5. AMQP ブローカーは、処理された結果とともにリクエストをメッセージングマネージャーにルーティングします。
6. メッセージングマネージャーは、処理済みのリクエストを、フロントエンドサービスがアクセスできる送信キューに格納します。

応答キューは、文字列プロセッサーサービスによって処理される結果が含まれる HTTP 応答を保存します。フロントエンドアプリケーションは、このキューを一定間隔でポーリングして結果を取得します。処理された結果が表示される準備が整ったら、以下を行います。

1. フロントエンドサービスは、HTTP **GET** リクエストを Spring Boot HTTP Starter によって提供される HTTP コントローラーに送信します。
2. HTTP コントローラーはリクエストをメッセージングマネージャーにルーティングします。
3. フロントエンドによって以前に送信されたリクエストが処理の準備ができ、送信キューで利用可能な場合、メッセージングマネージャーは結果を HTTP **GET** リクエストとして HTTP コントローラーに送信します。
4. HTTP コントローラーは、応答をフロントエンドサービスにルーティングし、結果を表示します。

5.13. リアクティブアプリケーションでの AMQP の使用

Spring Boot HTTP コントローラーで AMQP Client Starter を使用して、簡易メッセージングリアクティブアプリケーションを開発します。このサンプルアプリケーションは、2 つのメッセージングキューとブローカーを使用する Publisher-Subscriber メッセージング統合パターンで 2 つのサービスを統合しま

す。

この例は、AMQP メッセージングを使用して統合された 2 つのサービスで設定される Reactor Netty 上の Spring Boot および Eclipse Vert.x で基本的なアプリケーションを作成する方法を示しています。アプリケーションは以下のコンポーネントで設定されます。

- テキスト文字列をアプリケーションに送信するために使用できるフロントエンドサービス
- 文字列を大文字に変換するバックエンドサービス
- サービス間でマッサージをルーティングし、要求キューと応答キューを管理する Artemis AMQP ブローカー。
- Spring Boot HTTP Starter によって提供される HTTP コントローラー

前提条件

- Spring Boot を使用するよう設定された Maven ベースの Java アプリケーションプロジェクト
- JDK 8 または JDK 11 がインストールされている。
- Maven がインストールされている。

手順

1. 以下の依存関係をアプリケーションプロジェクトの **pom.xml** ファイルに追加します。

   pom.xml

```
...
<dependencies>
  ...
  <dependency>
    <groupId>dev.snowdrop</groupId>
    <artifactId>vertx-spring-boot-starter-http</artifactId>
  </dependency>
  <dependency>
    <groupId>dev.snowdrop</groupId>
    <artifactId>vertx-spring-boot-starter-amqp</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-artemis</artifactId>
  </dependency>
  <dependency>
    <groupId>org.apache.activemq</groupId>
    <artifactId>artemis-jms-server</artifactId>
  </dependency>
  <dependency>
    <groupId>org.apache.activemq</groupId>
    <artifactId>artemis-amqp-protocol</artifactId>
   <exclusions>
    <exclusion>
      <groupId>org.apache.qpid</groupId>
      <artifactId>proton-j</artifactId>
    </exclusion>
```

```
    </exclusions>
  </dependency>
  ...
</dependencies>
...
```

2. サンプルアプリケーションのメインクラスファイルを作成します。このクラスには、要求と結果に対応する処理キューを定義するメソッドが含まれています。

/src/main/java/AmqpExampleApplication.java

```
package dev.snowdrop.AmqpExampleApplication.java;

import java.util.HashMap;
import java.util.Map;

import dev.snowdrop.vertx.amqp.AmqpProperties;
import org.apache.activemq.artemis.api.core.TransportConfiguration;
import org.apache.activemq.artemis.core.remoting.impl.netty.NettyAcceptorFactory;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.boot.autoconfigure.jms.artemis.ArtemisConfigurationCustomizer;
import org.springframework.context.annotation.Bean;

@SpringBootApplication
public class AmqpExampleApplication {

    final static String PROCESSING_REQUESTS_QUEUE = "processing-requests";

    final static String PROCESSING_RESULTS_QUEUE = "processing-results";

    public static void main(String[] args) {
        SpringApplication.run(AmqpExampleApplication.class, args);
    }

    /**
     * Add Netty acceptor to the embedded Artemis server.
     */
    @Bean
    public ArtemisConfigurationCustomizer artemisConfigurationCustomizer(AmqpProperties
properties) {
        Map<String, Object> params = new HashMap<>();
        params.put("host", properties.getHost());
        params.put("port", properties.getPort());

        return configuration -> configuration
            .addAcceptorConfiguration(new
TransportConfiguration(NettyAcceptorFactory.class.getName(), params));
    }
}
```

3. GET および POST リクエストを処理する REST エンドポイントを公開して、要求キューと応答キューを管理する HTTP REST コントローラーのコードを含むクラスファイルを作成します。

/src/main/java/Controller.java

```
package dev.snowdrop.vertx.sample.amqp;

import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RestController;
import reactor.core.publisher.Flux;
import reactor.core.publisher.Mono;

import static org.springframework.http.MediaType.TEXT_EVENT_STREAM_VALUE;

/**
 * Rest controller exposing GET and POST resources to receive processed messages and
submit messages for processing.
 */
@RestController
public class Controller {

    private final MessagesManager messagesManager;

    public Controller(MessagesManager messagesManager) {
        this.messagesManager = messagesManager;
    }

    /**
     * Get a flux of messages processed up to this point.
     */
    @GetMapping(produces = TEXT_EVENT_STREAM_VALUE)
    public Flux<String> getProcessedMessages() {
        return Flux.fromIterable(messagesManager.getProcessedMessages());
    }

    /**
     * Submit a message for processing by publishing it to a processing requests queue.
     */
    @PostMapping
    public Mono<Void> submitMessageForProcessing(@RequestBody String body) {
        return messagesManager.processMessage(body.trim());
    }
}
```

4. メッセージングマネージャーを含むクラスファイルを作成します。Manager は、アプリケーションコンポーネントがリクエストキューにリクエストを公開する方法を制御し、その後に応答キューにサブスクライブして処理された結果を取得します。

/src/main/java/MessagesManager.java:

```
package dev.snowdrop.vertx.sample.amqp;

import java.util.List;
import java.util.concurrent.CopyOnWriteArrayList;

import dev.snowdrop.vertx.amqp.AmqpClient;
import dev.snowdrop.vertx.amqp.AmqpMessage;
import dev.snowdrop.vertx.amqp.AmqpSender;
```

```
import org.apache.activemq.artemis.core.server.embedded.EmbeddedActiveMQ;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.DisposableBean;
import org.springframework.beans.factory.InitializingBean;
import org.springframework.stereotype.Component;
import reactor.core.Disposable;
import reactor.core.publisher.Mono;

import static
dev.snowdrop.vertx.sample.amqp.AmqpSampleApplication.PROCESSING_REQUESTS_QUE
UE;
import static
dev.snowdrop.vertx.sample.amqp.AmqpSampleApplication.PROCESSING_RESULTS_QUEU
E;

/**
 * Processor client submits messages to the requests queue and subscribes to the results
queue for processed messages.
 */
@Component
public class MessagesManager implements InitializingBean, DisposableBean {

    private final Logger logger = LoggerFactory.getLogger(MessagesManager.class);

    private final List<String> processedMessages = new CopyOnWriteArrayList<>();

    private final AmqpClient client;

    private Disposable receiverDisposer;

    // Injecting EmbeddedActiveMQ to make sure it has started before creating this
component.
    public MessagesManager(AmqpClient client, EmbeddedActiveMQ server) {
        this.client = client;
    }

    /**
     * Create a processed messages receiver and subscribe to its messages publisher.
     */
    @Override
    public void afterPropertiesSet() {
        receiverDisposer = client.createReceiver(PROCESSING_RESULTS_QUEUE)
            .flatMapMany(receiver -> receiver.flux()
                .doOnCancel(() -> receiver.close().block())) // Close the receiver once subscription
is disposed
            .subscribe(this::handleMessage);
    }

    /**
     * Cancel processed messages publisher subscription.
     */
    @Override
    public void destroy() {
        if (receiverDisposer != null) {
            receiverDisposer.dispose();
```

```
        }
    }

    /**
     * Get messages which were processed up to this moment.
     *
     * @return List of processed messages.
     */
    public List<String> getProcessedMessages() {
        return processedMessages;
    }

    /**
     * Submit a message for processing by publishing it to a processing requests queue.
     *
     * @param body Message body to be processed.
     * @return Mono which is completed once the message is sent.
     */
    public Mono<Void> processMessage(String body) {
        logger.info("Sending message '{}' for processing", body);

        AmqpMessage message = AmqpMessage.create()
            .withBody(body)
            .build();

        return client.createSender(PROCESSING_REQUESTS_QUEUE)
            .map(sender -> sender.send(message))
            .flatMap(AmqpSender::close);
    }

    private void handleMessage(AmqpMessage message) {
        String body = message.bodyAsString();

        logger.info("Received processed message '{}'", body);
        processedMessages.add(body);
    }
}
```

5. 要求キューからテキスト文字列を受け取る大文字プロセッサーを含むクラスファイルを作成し、それを大文字に変換します。その後、プロセッサーは結果を応答キューに公開します。

/src/main/java/UppercaseProcessor.java

```
package dev.snowdrop.vertx.sample.amqp;

import dev.snowdrop.vertx.amqp.AmqpClient;
import dev.snowdrop.vertx.amqp.AmqpMessage;
import dev.snowdrop.vertx.amqp.AmqpSender;
import org.apache.activemq.artemis.core.server.embedded.EmbeddedActiveMQ;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.DisposableBean;
import org.springframework.beans.factory.InitializingBean;
import org.springframework.stereotype.Component;
import reactor.core.Disposable;
import reactor.core.publisher.Mono;
```

```
import static
dev.snowdrop.vertx.sample.amqp.AmqpSampleApplication.PROCESSING_REQUESTS_QUE
UE;
import static
dev.snowdrop.vertx.sample.amqp.AmqpSampleApplication.PROCESSING_RESULTS_QUEU
E;

/**
 * Uppercase processor subscribes to the requests queue, converts each received message
to uppercase and send it to the
 * results queue.
 */
@Component
public class UppercaseProcessor implements InitializingBean, DisposableBean {

    private final Logger logger = LoggerFactory.getLogger(UppercaseProcessor.class);

    private final AmqpClient client;

    private Disposable receiverDisposer;

    // Injecting EmbeddedActiveMQ to make sure it has started before creating this
component.
    public UppercaseProcessor(AmqpClient client, EmbeddedActiveMQ server) {
        this.client = client;
    }

    /**
     * Create a processing requests receiver and subscribe to its messages publisher.
     */
    @Override
    public void afterPropertiesSet() {
        receiverDisposer = client.createReceiver(PROCESSING_REQUESTS_QUEUE)
            .flatMapMany(receiver -> receiver.flux()
                .doOnCancel(() -> receiver.close().block())) // Close the receiver once subscription
is disposed
            .flatMap(this::handleMessage)
            .subscribe();
    }

    /**
     * Cancel processing requests publisher subscription.
     */
    @Override
    public void destroy() {
        if (receiverDisposer != null) {
            receiverDisposer.dispose();
        }
    }

    /**
     * Convert the message body to uppercase and send it to the results queue.
     */
    private Mono<Void> handleMessage(AmqpMessage originalMessage) {
        logger.info("Processing '{}'", originalMessage.bodyAsString());
```

```
        AmqpMessage processedMessage = AmqpMessage.create()
            .withBody(originalMessage.bodyAsString().toUpperCase())
            .build();

        return client.createSender(PROCESSING_RESULTS_QUEUE)
            .map(sender -> sender.send(processedMessage))
            .flatMap(AmqpSender::close);
    }
}
```

6. オプション: アプリケーションをローカルで実行し、テストします。

   a. Maven プロジェクトのルートディレクトリーへ移動します。

   ```
   $ cd myApp
   ```

   b. アプリケーションをパッケージ化します。

   ```
   $ mvn clean package
   ```

   c. コマンドラインからアプリケーションを起動します。

   ```
   $ java -jar target/vertx-spring-boot-sample-amqp.jar
   ```

   d. 新しいターミナルウィンドウで、**localhost** へ処理されるテキスト文字列が含まれる HTTP **POST** リクエストを多数送信します。

   ```
   $ curl -H "Content-Type: text/plain" -d 'Hello, World' -X POST http://localhost:8080
   $ curl -H "Content-Type: text/plain" -d 'Hello again' -X POST http://localhost:8080
   ```

   e. HTTP **GET** 要求を **localhost** に送信します。文字列を大文字で HTTP 応答を受け取ります。

   ```
   $ curl http://localhost:8080
   HTTP/1.1 200 OK
   Content-Type: text/event-stream;charset=UTF-8
   transfer-encoding: chunked

   data:HELLO, WORLD

   data:HELLO AGAIN
   ```

その他のリソース

- Fabric8 Maven プラグインを使用して、アプリケーションを OpenShift クラスターにデプロイできます。
- また、スタンドアロンの Red Hat Enterprise Linux でのデプロイメント 用にアプリケーションを設定することもできます。

5.14. APACHE KAFKA

Apache Kafka は、プロセス、アプリケーション、およびサービスの間でメッセージを交換するために設計されたスケーラブルなメッセージング統合システムです。Kafka は、トピックのセットを維持する 1 つ以上のブローカーを持つクラスターを基にします。基本的に、トピックは、トピック ID を使用してすべてのクラスターに定義できるカテゴリーです。各トピックには、アプリケーション内で行うイベントに関する情報が含まれるレコードと呼ばれるデータの一部が含まれています。システムに接続されているアプリケーションは、これらのトピックにレコードを追加したり、先に追加したメッセージを処理したり、再処理したりできます。

ブローカーは、クライアントアプリケーションとの通信を処理し、トピックでレコードを管理します。レコードが失われないように、ブローカーはコミットログのすべてのレコードを追跡し、各アプリケーションのオフセット値を追跡します。オフセットは、最後に追加したレコードを示すポインターと似ています。

アプリケーションは、トピックから最新のレコードをプルするか、オフセットを変更して以前のメッセージが追加されたレコードを読み取ることができます。この機能は、クライアントアプリケーションがリアルタイムで処理できない場合に、着信要求に圧倒されるのを防ぎます。これが発生した場合、Kafka は、リアルタイムで処理できないレコードをコミットログに保存することにより、データの損失を防ぎます。クライアントアプリケーションが着信要求に追いつくことができると、リアルタイムでレコードの処理を再開します

ブローカーは、トピックパーティションに分類することで、複数のトピックのレコードを管理できます。Apache Kafka はこれらのパーティションを複製して、単一のトピックからのレコードを複数のブローカーで並行して処理できるようにし、アプリケーションがトピック内のレコードを処理する速度をスケーリングできるようにします。レプリケートされたトピックパーティション (別名フォロワー) は、レコード処理の冗長性を回避するために、元のトピックパーティション (別名 Leader) と同期します。新しいレコードが Leader パーティションにコミットされ、フォロワーはリーダーに加えられた変更のみを複製します。

5.15. APACHE KAFKA REACTIVE サンプルの仕組み

このサンプルアプリケーションは、Apache Kafka を使用して実装された Publisher-Subscriber メッセージストリーミングパターンに基づいています。アプリケーションで設定されるコンポーネントは、以下のとおりです。

- ログメッセージプロデューサーおよびコンシューマーをインスタンス化する **KafkaExampleApplication** クラス
- Spring Boot HTTP Starter によって設定され、提供される WebFlux HTTP コントローラー。コントローラーは、メッセージの公開および読み取りに使用される残りのリソースを提供します。
- プロデューサーが Kafka の **log** トピックにメッセージを公開する方法を定義する **KafkaLogger** クラス。
- Kafka の **log** トピックからサンプルアプリケーションが受け取るメッセージを表示する **KafkaLog** クラス。

メッセージを公開します。

1. ログメッセージをペイロードとして使用して、サンプルアプリケーションに対して HTTP POST 要求を実行します。
2. HTTP コントローラーはメッセージを公開するのに使用される REST エンドポイントにルーティングし、そのメッセージをロガーインスタンスに渡します。
3. HTTP コントローラーは、受け取ったメッセージを Kafka の **log** トピックに公開します。

4. KafkaLog インスタンスは、Kafka トピックからログメッセージを受け取ります。

メッセージの読み取り:

1. HTTP **GET** リクエストをサンプルアプリケーション URL に送信します。
2. コントローラーは **KafkaLog** インスタンスからメッセージを取得し、HTTP 応答のボディーとして返します。

5.16. リアクティブアプリケーションでの KAFKA の使用

この例は、Reactor Netty 上の Spring Boot および Eclipse Vert.x で Apache Kafka を使用するメッセージングアプリケーションのサンプルを作成する方法を示しています。アプリケーションはメッセージを Kafka トピックに公開してから、リクエストの送信時にメッセージを取得して表示します。

Kafka クラスターによって使用されるメッセージトピック、URL、およびメタデータの Kafka 設定プロパティーは **src/main/resources/application.yml** に保存されます。

前提条件

- Spring Boot を使用するよう設定された Maven ベースの Java アプリケーションプロジェクト
- JDK 8 または JDK 11 がインストールされている。
- Maven がインストールされている。

手順

1. WebFlux HTTP Starter および Apache Kafka Starter をアプリケーションプロジェクトの **pom.xml** ファイルに依存関係として追加します。

   pom.xml

```
...
<dependencies>
  ...
  <!-- Vert.x WebFlux starter used to handle HTTP requests -->
  <dependency>
    <groupId>dev.snowdrop</groupId>
    <artifactId>vertx-spring-boot-starter-http</artifactId>
  </dependency>
  <!-- Vert.x Kafka starter used to send and receive messages to/from Kafka cluster -->
  <dependency>
    <groupId>dev.snowdrop</groupId>
    <artifactId>vertx-spring-boot-starter-kafka</artifactId>
  </dependency>
  ...
</dependencies>
...
```

1. **KafkaLogger** クラスを作成します。このクラスはプロデューサーと sendas メッセージを機能させます。**KafkaLogger** クラスは、Producer がメッセージ (別名レコード) をトピックに公開する方法を定義します。

   /src/main/java/KafkaLogger.java

```
...
final class KafkaLogger {

    private final KafkaProducer<String, String> producer;

    KafkaLogger(KafkaProducer<String, String> producer) {
        this.producer = producer;
    }

    public Mono<Void> logMessage(String body) {
        // Generic key and value types can be inferred if both key and value are used to create a
builder
        ProducerRecord<String, String> record = ProducerRecord.<String,
String>builder(LOG_TOPIC, body).build();

        return producer.send(record)
            .log("Kafka logger producer")
            .then();
    }
}
...
```

2. **KafkaLog** クラスを作成します。このクラスは、kafka メッセージのコンシューマーとして機能します。**KafkaLog** は、トピックからメッセージを取得してターミナルに表示されるメッセージを取得します。

/src/main/java/KafkaLog.java

```
...
final class KafkaLog implements InitializingBean, DisposableBean {

    private final List<String> messages = new CopyOnWriteArrayList<>();

    private final KafkaConsumer<String, String> consumer;

    private Disposable consumerDisposer;

    KafkaLog(KafkaConsumer<String, String> consumer) {
        this.consumer = consumer;
    }

    @Override
    public void afterPropertiesSet() {
        consumerDisposer = consumer.subscribe(LOG_TOPIC)
            .thenMany(consumer.flux())
            .log("Kafka log consumer")
            .map(ConsumerRecord::value)
            .subscribe(messages::add);
    }

    @Override
    public void destroy() {
        if (consumerDisposer != null) {
            consumerDisposer.dispose();
        }
```

```
        consumer.unsubscribe()
            .block(Duration.ofSeconds(2));
    }

    public List<String> getMessages() {
        return messages;
    }
}
...
```

3. HTTP REST コントローラーが含まれるクラスファイルを作成します。アプリケーションがメッセージのロギングや読み取りを処理するために使用する REST リソースを公開するコントローラー。

/src/main/java/Controller.java

```
package dev.snowdrop.vertx.sample.kafka;

import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RestController;
import reactor.core.publisher.Flux;
import reactor.core.publisher.Mono;

import static org.springframework.http.MediaType.TEXT_EVENT_STREAM_VALUE;

/**
 * HTTP controller exposes GET and POST resources to log messages and to receive the
previously logged ones.
 */
@RestController
public class Controller {

    private final KafkaLogger logger;

    private final KafkaLog log;

    public Controller(KafkaLogger logger, KafkaLog log) {
        this.logger = logger;
        this.log = log;
    }

    /**
     * Get a Flux of previously logged messages.
     */
    @GetMapping(produces = TEXT_EVENT_STREAM_VALUE)
    public Flux<String> getMessages() {
        return Flux.fromIterable(log.getMessages());
    }

    /**
     * Log a message.
     */
    @PostMapping
    public Mono<Void> logMessage(@RequestBody String body) {
```

```
        return logger.logMessage(body.trim());
    }
}
```

4. Apache Kafka Cluster のプロデューサーおよびコンシューマーが使用する URL が含まれる YAML テンプレートを作成して、メッセージのログおよび読み取りを行います。この例では、Apache Kafka Cluster のコンシューマーおよびプロデューサーは、デフォルトで **localhost** でポート **9092** を使用して通信します。以下の例に示すように、プロデューサーとコンシューマーを個別に設定する必要があります。

/src/main/resources/application.yml

```
vertx:
  kafka:
    producer:
      bootstrap:
        # The producer in your cluster uses this URL to publish messages to the log.
        servers: localhost:9092
      key:
        # This class assigns the mandatory key attribute that is assigned to each message.
        serializer: org.apache.kafka.common.serialization.StringSerializer
      value:
        # This class assigns the mandatory value attribute that is assigned to each message.
        serializer: org.apache.kafka.common.serialization.StringSerializer
    consumer:
      bootstrap:
        servers: localhost:9092 # The consumer in your cluster uses this URL to read messages from the log.
      group:
        id: log # The consumer group IDs used to define a group of consumers that subscribe to the same topic. In this example, all consumers belong in the same consumer group.
      key:
        deserializer: org.apache.kafka.common.serialization.StringDeserializer # This class generates the mandatory key attribute that is assigned to each message.
      value:
        deserializer: org.apache.kafka.common.serialization.StringDeserializer # This class generates the mandatory value attribute that is assigned to each message.
```

5. オプション: アプリケーションをローカルで実行し、テストします。

   a. Maven プロジェクトのルートディレクトリーへ移動します。

   ```
   $ cd vertx-spring-boot-sample-kafka
   ```

   b. アプリケーションをパッケージ化します。

   ```
   $ mvn clean package
   ```

   c. コマンドラインからアプリケーションを起動します。

   ```
   $ java -jar target/vertx-spring-boot-sample-kafka.jar
   ```

d. 新しいターミナルウィンドウで、テキスト文字列としてフォーマットされたメッセージが含まれる HTTP **POST** リクエストを **localhost** に送信します。メッセージはすべて **log** トピックに公開されます。

```
$ curl -H "Content-Type: text/plain" -d 'Hello, World' -X POST http://localhost:8080
$ curl -H "Content-Type: text/plain" -d 'Hello again' -X POST http://localhost:8080
...
```

e. HTTP **GET** 要求を **localhost** に送信します。コンシューマーがサブスクライブするトピック内の全メッセージが含まれる HTTP 応答を受け取ります。

```
$ curl http://localhost:8080
HTTP/1.1 200 OK
Content-Type: text/event-stream;charset=UTF-8
transfer-encoding: chunked

data:Hello, World

data:Hello, again
...
```

その他のリソース

- Fabric8 Maven プラグインを使用して、アプリケーションを OpenShift クラスターにデプロイできます。
- また、スタンドアロンの Red Hat Enterprise Linux でのデプロイメント 用にアプリケーションを設定することもできます。

サンプルを使用 する他に、Eclipse Vert.x と Spring Boot を使用して、ゼロから新しい Spring Boot アプリケーションを作成し、それらを OpenShift にデプロイすることもできます。

第6章 SPRING BOOT アプリケーションでの DEKORATE の使用

6.1. DEKORATE の概要

Dekorate は、Red Hat ビルドの Spring Boot で提供される compile-time アノテーションパーサーおよびアプリケーションリソースジェネレーターのコレクションです。これは、アプリケーションをビルドして設定プロパティーを抽出する際に、コードでアノテーションを解析することで機能します。次に、Dekorate は、展開したプロパティーの値を使用して、アプリケーションを Kubernetes または OpenShift クラスターにデプロイするために使用できるアプリケーション設定リソースを生成します。

開発者は、コードにアノテーションを付けた後に Dekorate を使用してアプリケーションのビルド時にアプリケーションマニフェストを自動的に生成できます。これにより、アプリケーションをデプロイするためのリソースファイルを手動で作成する必要がなくなります。アプリケーションが、Spring Boot などのリッチアプリケーションのランタイムフレームワークに基づいていると、Spring Boot、Dokorate はフレームワークと直接統合でき、フレームワークによって提供される API から設定パラメーターを抽出できるため、コードにアノテーションを付ける必要がなくなります。Dekorate は、以下を使用してアプリケーションを自動的に設定できます。

- マニフェストファイルの設定に使用される値およびメタデータを取得するためにアプリケーションコードで Dekorate 固有のアノテーションを解析します。
- **application.properties**、**application.yaml** などの設定リソースから情報の抽出
- リッチアプリケーションフレームワークから必要なメタデータを取得し、**application.properties** ファイルまたは **application.yml** ファイルから設定値を展開します。

アプリケーションのリソース定義の生成に加え、Dekorate はビルドフックを生成することもできます。これにより、OpenShift クラスターの Dekorate でアプリケーションをビルドおよびデプロイできます。これは、アプリケーションを作成する言語とは独立して機能し、幅広いビルドシステムで使用できます。Dekorate は、Maven BOM として配布されるライブラリーのセットで設定されます。ライブラリーをアプリケーションプロジェクトの依存関係として追加し、アプリケーションと Dekorate を使用することができます。

Red Hat は、Dekorate を使用したリソースファイルの生成およびビルドフックのサポートを提供します。このフックは、Spring Boot をベースとした Java アプリケーションを Apache Maven で OpenShift Container Platform にデプロイするために使用できます。

6.2. DEKORATE を使用するようにアプリケーションプロジェクトを設定します。

Dekorate BOM および OpenShift Annotations Starter をアプリケーションプロジェクトの **pom.xml** ファイルに追加します。基本的なアノテーションをソースファイルに追加し、Maven でアプリケーションをパッケージ化してアプリケーションマニフェストを生成します。

前提条件

- Spring Bootを使用するよう設定された Maven ベースのアプリケーションプロジェクト
- Spring Boot を使用する Java ベースのアプリケーション
- Java JDK 8 または JDK 11 がインストールされている。
- Maven がインストールされている。

手順

1. OpenShift アノテーション Starter をアプリケーションの **pom.xml** ファイルに追加します。

```
<project>
 ...
 <dependencies>
   ...
  <dependency>
   <groupId>io.dekorate</groupId>
   <artifactId>openshift-annotations</artifactId>
  </dependency>
  <dependency>
   <groupId>io.dekorate</groupId>
   <artifactId>openshift-spring-starter</artifactId>
  </dependency>
  ...
 </dependencies>
...
<project>
```

2. **@Dekorate** アノテーションをアプリケーションプロジェクトのソースファイルに追加します。

```
package org.acme;

import io.dekorate.annotation.Dekorate;

@Dekorate
public class Application {
}
```

3. アプリケーションをパッケージ化します。

```
mvn clean package
```

4. 生成されたアプリケーションテンプレートファイルを含む **target/classes/META-INF/dekorate** ディレクトリーに移動します。

6.3. DEKORATE を使用したアプリケーション設定のカスタマイズ

Dekorate を使用して、OpenShift でのデプロイメント用にアプリケーションの設定をカスタマイズします。

- アプリケーションのソースのアノテーションで設定パラメーターを指定
- **application.properties** ファイルでのプロパティーの設定

以下の例は、OpenShift へのデプロイ時に 2 つのレプリカで始まるようにアプリケーションを設定する方法を示しています。

前提条件

- Spring Boot および Dekorate を使用するよう設定された Maven ベースのアプリケーションプロジェクト

- Spring Boot を使用する Java ベースのアプリケーション
- Java JDK 8 または JDK 11 がインストールされている。
- Maven がインストールされている。

手順

1. Dekorate OpenShift Annotations モジュールを、アプリケーションの **pom.xml** ファイルの依存関係として追加します。

```
<project>
 ...
 <dependencies>
   ...
  <dependency>
   <groupId>io.dekorate</groupId>
   <artifactId>openshift-annotations</artifactId>
  </dependency>
  ...
 </dependencies>
...
<project>
```

2. OpenShift へのデプロイ時に、アプリケーションが開始するデフォルトのレプリカ数を設定します。

   a. **@OpenshiftApplication** アノテーションをアプリケーションのメインソースファイルに追加し、レプリカの数を **2** に指定します。

```
package org.acme;

import io.dekorate.openshift.annotation.OpenshiftApplication;

// include the parameter for the number of replicas to
@OpenshiftApplication(replicas=2)
public class Application {
}
```

   b. または、アプリケーションの **application.properties** ファイルに **dekorate.openshift.replicas=2** プロパティーを設定します。

   /src/main/resources/application.properties

```
dekorate.openshift.replicas=2
```

3. アプリケーションをパッケージ化します。

```
mvn clean package
```

4. **target/classes/META-INF/dekorate** に移動して、Dekorate によって生成されたテンプレートを表示します。デプロイメント設定 YAML テンプレートのレプリカ数は 2 に設定されます。

```
...
```

```
spec:
  replicas: 2
  selector:
    matchLabels:
      app: acme
...
```

6.4. SPRING BOOT アプリケーションでのアノテーションレス設定の使用

Dekorate を使用して、**application.properties** ファイルおよび **application.yml** ファイルから dekorate 設定プロパティーを抽出して、Spring Boot アプリケーションプロジェクトの OpenShift リソース設定ファイルを生成します。Dekorate は、Spring Boot から必要なメタデータとプロパティーファイルから設定パラメーターを取得できるため、このメソッドにはアプリケーションソースにアノテーションを付ける必要はありません。アノテーションのない設定は、Spring Boot と Dekorate の間のリッチフレームワーク統合機能です。

前提条件

- Spring Boot および Dekorate を使用するよう設定された Maven ベースのアプリケーションプロジェクト
- **@SpringBootApplication** アノテーションが付けられたアプリケーションプロジェクトの 1 つ以上のクラス。
- Java JDK 8 または JDK 11 がインストールされている。
- Maven がインストールされている。

手順

1. アプリケーションの **pom.xml** ファイルに以下の依存関係を追加します。

```
<project>
  ...
  <dependencies>
    ...
    <!-- The OpenShift Spring Starter automatically adds "io.dekorate:openshift-annotations"
as a transitive dependency -->
    <dependency>
      <groupId>io.dekorate</groupId>
      <artifactId>openshfit-spring-starter</artifactId>
    </dependency>
    ...
  </dependencies>
...
<project>
```

2. Dekorate 設定プロパティーをプロジェクトの **application.properties** ファイルまたは **application.yml** ファイルに追加します。Dekorate プロパティーアノテーションをソースファイルに追加する必要はありません。ソースファイルのアノテーションを使用できますが、これを行う場合は、アノテーションで提供されるパラメーターを **application.properties** ファイルまたは **application.yml** ファイルにあるパラメーターで上書きします。

3. アプリケーションをパッケージ化します。

```
mvn clean package
```

アプリケーションの Dekorate をビルドすると、アプリケーションプロジェクト内の以下のリソースの設定が解析されます。設定リソースは優先順に解析されます。つまり、異なるタイプの 2 つの異なるリソースが同じ設定パラメーターに異なる値を持っている場合、Dekorate は優先度のリストにあるリソースから取得した値を使用します。たとえば、ソースのアノテーションでパラメーター値が指定されていても、**application.yml** の同じパラメーターに別の値が指定されている場合、Dekorate は **application.yml** の取得形式を使用します。Dekorate は、以下の優先順位でプロジェクトリソースを解析します。

1. アノテーション
2. **application.properties**
3. **application.yaml**
4. **application.yml**

4. 生成された **application.json** または **application.yml** マニフェストファイルを含む **target/classes/META-INF/dekorate** ディレクトリーに移動します。

6.5. DEKORATE を使用した OPENSHIFT SOURCE-TO-IMAGE ビルドの自動実行

Maven でアプリケーションをコンパイルした後に、Dekorate を使用して OpenShift コンテナーイメージビルドを自動的に実行できます。

Dekorate を使用して Source-to-Image ビルドを自動的にトリガーする機能は テクノロジープレビュー として利用できます。Red Hat は、実稼働環境でのこの機能の使用に対するサポートを提供していません。

前提条件

- Spring Boot および Dekorate を使用するよう設定された Maven ベースのアプリケーションプロジェクト
- プロジェクトのソースファイルに追加された @OpenShiftApplication アノテーション
- Java JDK 8 または JDK 11 がインストールされている。
- Maven がインストールされている。
- **oc** コマンドラインツールがインストールされている
- **oc** コマンドラインツールを使用して OpenShift クラスターにログインしている。

手順

1. Dekorate OpenShift Annotations モジュールを依存関係としてアプリケーションの **pom.xml** ファイルに追加します。このモジュールは、すべての Dekorate Starters の推移的な依存関係として含まれています。

```
<project>
  ...
  <dependencies>
```

```
    ...
    <dependency>
      <groupId>io.dekorate</groupId>
      <artifactId>openshift-annotations</artifactId>
    </dependency>
    ...
  </dependencies>
...
<project>
```

2. アプリケーションをビルドしてデプロイします。Maven がアプリケーションをコンパイルした後にコンテナーイメージビルドを実行するための **-Ddekorate.build=true** プロパティーを含めます。Source-to-Image ビルドを自動的に実行する機能は テクノロジープレビュー として提供されることに注意してください。

```
$ mvn clean install -Ddekorate.build=true
```

Maven でアプリケーションをコンパイルした後に、コマンドラインから Source-to-image ビルドを手動で実行することもできます。

```
# Process your application YAML template that is generated by Dekorate:
$ oc apply -f target/classes/META-INF/dekorate/openshift.yml
# Execute the Source-to-image build and deploy your application to the OpenShift cluster:
$ oc start-build example --from-dir=./target --follow
```

6.6. OPENSHIFT での SPRING BOOT での DEKORATE の使用

以下の例は、方法を示しています。

1. アプリケーションで **openshift-spring-stater** を使用できます。
2. Dekorate は、アプリケーションのタイプを自動的に識別し、OpenShift サービスルートおよびプローブを随時設定できます。
3. Maven がアプリケーションをコンパイルした後に、source-to-image ビルドをトリガーするようにアプリケーションを設定できます。
4. 前提条件
   - Spring Boot および Dekorate を使用するよう設定された Maven ベースのアプリケーションプロジェクト
   - **@SpringBootApplication** アノテーションがプロジェクトのソースファイルに追加されました。
   - Java JDK 8 または JDK 11 がインストールされている。
   - Maven がインストールされている。
   - **oc** コマンドラインツールがインストールされている
   - **oc** コマンドラインツールを使用して OpenShift クラスターにログインしている。

手順

1. Dekorate Spring Starter は、アプリケーションプロジェクトの **pom.xml** ファイルの依存関係として追加します。

pom.xml

```
<project>
  ...
  <dependencies>
    ...
    <dependency>
      <groupId>io.dekorate</groupId>
      <artifactId>openshift-spring-starter</artifactId>
    </dependency>
    ...
  </dependencies>
...
<project>
```

2. **Main.java** クラスファイルに **@OpenShiftApplication** アノテーションを付けます。これにより、アプリケーションをコンパイルする際に Source-to-Image ビルドを起動できます。

/src/main/java/io/dekorate/example/sbonopenshift/Main.java

```
package io.dekorate.example.sbonopenshift;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@OpenShiftApplication
@SpringBootApplication
public class Main {

  public static void main(String[] args) {
    SpringApplication.run(Main.class, args);
  }

}
```

3. Rest コントローラーをアプリケーションに追加します。

/src/main/java/io/dekorate/example/sbonopenshift/Controller.java

```
package io.dekorate.example.sbonopenshift;

import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class Controller {

  @RequestMapping("/")
  public String hello() {
    return "Hello world";
  }
}
```

Dekorate Spring スターターによって提供される Spring アプリケーションプロセッサーは Rest コントローラーを自動的に検出し、アプリケーションタイプを Web アプリケーションとして識別します。Web アプリケーションの場合、Dekorate は OpenShift アプリケーションテンプレートを自動的に生成し、以下を設定します。

- アプリケーションの OpenShift Service ルート
- アプリケーションのルートでサービスを公開します。
- Liveness および Readiness プローブの設定

4. アプリケーションをビルドしてデプロイします。Maven がアプリケーションをコンパイルした後に source-to-image ビルドを自動的に実行するには、**-Ddekorate.deploy=true** プロパティーを含めます。

```
mvn clean install -Ddekorate.deploy=true
```

第7章 SPRING BOOT ベースのアプリケーションのデバッグ

本セクションでは、ローカルデプロイメントとリモートデプロイメントの両方で Spring Boot ベースのアプリケーションのデバッグについて説明します。

7.1. リモートのデバッグ

アプリケーションをリモートでデバッグするには、まずデバッグモードで開始するように設定してから、デバッガーを割り当てる必要があります。

7.1.1. デバッグモードでの Spring Boot アプリケーションのローカルでの開始

Maven ベースのプロジェクトのデバッグ方法の 1 つは、デバッグポートを指定している間にアプリケーションを手動で起動し、その後にリモートデバッガーをそのポートに接続することです。この方法は、**mvn spring-boot:run** ゴールを使用してアプリケーションを手動で起動する際に適用できます。

前提条件

- Maven ベースのアプリケーション

手順

1. コンソールで、アプリケーションでディレクトリーに移動します。
2. アプリケーションを起動し、以下の構文を使用して必要な JVM 引数とデバッグポートを指定します。

```
$ mvn spring-boot:run -Drun.jvmArguments="-Xdebug -
Xrunjdwp:transport=dt_socket,server=y,suspend=n,address=$PORT_NUMBER"
```

$PORT_NUMBER は未使用のポート番号です。リモートデバッガー設定のこの番号を覚えておいてください。

JVM を一時停止してリモートデバッガー接続を待つ場合は、アプリケーションを起動する前に **suspend** を **y** に変更します。

7.1.2. デバッグモードでの uberjar の起動

アプリケーションを Spring Boot uberjar としてパッケージ化する場合は、以下のパラメーターで実行してデバッグします。

前提条件

- アプリケーションによる uberjar

手順

1. コンソールで、uberjar のディレクトリーに移動します。
2. 以下のパラメーターで uberjar を実行します。行上の uberjar の名前の前に、すべてのパラメーターが指定されていることを確認してください。

```
$ java -agentlib:jdwp=transport=dt_socket,server=y,suspend=n,address=$PORT_NUMBER -jar $UBERJAR_FILENAME
```

$PORT_NUMBER は未使用のポート番号です。リモートデバッガー設定のこの番号を覚えておいてください。

JVM を一時停止してリモートデバッガー接続を待つ場合は、アプリケーションを起動する前に **suspend** を **y** に変更します。

7.1.3. デバッグモードでの OpenShift でのアプリケーションの起動

OpenShift で Spring Boot ベースのアプリケーションをリモートでデバッグするには、コンテナー内で **JAVA_DEBUG** 環境変数を **true** に設定し、リモートデバッガーからアプリケーションに接続できるようにポート転送を設定する必要があります。

前提条件

- アプリケーションが OpenShift で実行している。
- **oc** バイナリーがインストールされている。
- ターゲット OpenShift 環境で **oc port-forward** コマンドを実行する機能。

手順

1. **oc** コマンドを使用して、利用可能なデプロイメント設定を一覧表示します。

```
$ oc get dc
```

2. アプリケーションのデプロイメント設定で **JAVA_DEBUG** 環境変数を **true** に設定します。これにより、JVM がデバッグ用にポート番号 **5005** を開くように設定されます。以下は例になります。

```
$ oc set env dc/MY_APP_NAME JAVA_DEBUG=true
```

3. 設定変更時に自動的に再デプロイするように設定されていない場合は、アプリケーションを再デプロイします。以下は例になります。

```
$ oc rollout latest dc/MY_APP_NAME
```

4. ローカルマシンからアプリケーション Pod へのポート転送を設定します。
   a. 現在実行中の Pod を一覧表示し、アプリケーションが含まれる Pod を検索します。

```
$ oc get pod
NAME                        READY     STATUS      RESTARTS   AGE
MY_APP_NAME-3-1xrsp         0/1       Running     0          6s
...
```

   b. ポート転送を設定します。

```
$ oc port-forward MY_APP_NAME-3-1xrsp $LOCAL_PORT_NUMBER:5005
```

ここでは、**$LOCAL_PORT_NUMBER** はローカルマシンで選択した未使用のポート番号になります。リモートデバッガー設定のこの番号をメモします。

5. デバッグが完了したら、アプリケーション Pod の **JAVA_DEBUG** 環境変数の設定を解除します。以下は例になります。

```
$ oc set env dc/MY_APP_NAME JAVA_DEBUG-
```

追加リソース

デバッグポートをデフォルト (**5005**) から変更する場合は、**JAVA_DEBUG_PORT** 環境変数を設定することもできます。

7.1.4. アプリケーションへのリモートデバッガーの割り当て

デバッグ用にアプリケーションが設定されている場合は、選択したリモートデバッガーを割り当てます。本ガイドでは、Red Hat CodeReady Studio について説明していますが、他のプログラムを使用する場合も手順は同じようになります。

前提条件

- ローカルまたは OpenShift 上で実行し、デバッグ用に設定されたアプリケーション。
- アプリケーションがデバッグをリッスンしているポート番号。
- Red Hat CodeReady Studio がマシンにインストールされている。Red Hat CodeReady Studio ダウンロードページ からダウンロードできます。

手順

1. Red Hat CodeReady Studio を開始します。
2. アプリケーションの新規デバッグ設定を作成します。
   a. **Run→Debug Configurations** をクリックします。
   b. 設定のリストで、**Remote Java アプリケーション** をダブルクリックします。これにより、新しいリモートデバッグ設定が作成されます。
   c. **Name** フィールドに設定に適した名前を入力します。
   d. アプリケーションが含まれるディレクトリーへのパスを **Project** フィールドに入力します。便宜上、**Browse...** ボタンを使用できます。
   e. **Connection Type** フィールドを **Standard (Socket Attach)** に設定していない場合は、これを設定します。
   f. **Port** フィールドを、アプリケーションがデバッグをリッスンしているポート番号に設定します。
   g. **Apply** をクリックします。
3. Debug Configurations ウィンドウの **Debug** ボタンをクリックして、デバッグを開始します。初めてデバッグ設定を迅速に起動するには **Run→Debug History** をクリックし、一覧から設定を選択します。

追加リソース

- Red Hat ナレッジベースの Debug an OpenShift Java Application with JBoss Developer Studio
  Red Hat CodeReady Studio はこれまで JBoss Developer Studio と呼ばれていました。
- OpenShift ブログの記事 Debugging Java Applications On OpenShift and Kubernetes

7.2. デバッグロギング

7.2.1. Spring Boot デバッグロギングの追加

デバッグロギングをアプリケーションに追加します。

前提条件

- デバッグするアプリケーション。たとえば、REST API Level 0 のサンプル です。

手順

1. ロギングを追加するクラスの **org.apache.commons.logging.LogFactory** を使用して **org.apache.commons.logging.Log** オブジェクトを宣言します。

```
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
...
private static Log logger = LogFactory.getLog(TheClass.class);
```

   たとえば、REST API Level 0 サンプル の **GreetingEndpoint** クラスにロギングを追加する場合は、**GreetingEndpoint.class** を使用します。

2. **logger.debug("my logging message")** を使用してデバッグステートメントを追加します。

   ロギングステートメントの例

```
@GET
@Path("/greeting")
@Produces("application/json")
public Greeting greeting(@QueryParam("name") @DefaultValue("World") String name) {
    String message = String.format(properties.getMessage(), name);

    logger.debug("Message: " + message);

    return new Greeting(message);
}
```

3. **src/main/resources/application.properties** に **logging.level.fully.qualified.name.of.TheClass=DEBUG** を追加します。
   たとえば、ロギングステートメントを **io.openshift.booster.service.GreetingEndpoint** に追加した場合は、以下を使用します。

```
logging.level.io.openshift.booster.service.GreetingEndpoint=DEBUG
```

これにより、**DEBUG** レベル以上のログメッセージをクラスのログに表示できるようになります。

7.2.2. localhost での Spring Boot デバッグログへのアクセス

アプリケーションを起動し、これと対話してデバッグステートメントを確認します。

前提条件

- デバッグロギングが有効になっているアプリケーション。

手順

1. アプリケーションを起動します。

```
$ mvn spring-boot:run
```

2. アプリケーションをテストしてデバッグロギングを呼び出します。
たとえば、REST API Level 0 サンプル をテストするには、**/api/greeting** メソッドを呼び出すことができます。

```
$ curl http://localhost:8080/api/greeting?name=Sarah
```

3. アプリケーションログを表示して、デバッグメッセージを表示します。

```
i.o.booster.service.GreetingEndpoint     : Message: Hello, Sarah!
```

デバッグロギングを無効にするには、**src/main/resources/application.properties** から **logging.level.fully.qualified.name.of.TheClass=DEBUG** を削除し、アプリケーションを再起動します。

7.2.3. OpenShift でのデバッグログへのアクセス

アプリケーションを起動し、これと対話して、OpenShift のデバッグステートメントを確認します。

前提条件

- CLI クライアント **oc** がインストールされ、認証されている。
- デバッグロギングが有効になっている Maven ベースのアプリケーション。

手順

1. アプリケーションを OpenShift にデプロイします。

```
$ mvn clean fabric8:deploy -Popenshift
```

2. ログを表示します。
    1. アプリケーションと共に Pod の名前を取得します。

```
$ oc get pods
```

2. ログ出力の監視を開始します。

```
$ oc logs -f pod/MY_APP_NAME-2-aaaaa
```

ログ出力を確認できるように、端末ウィンドウにログ出力が表示されます。

3. アプリケーションと対話します。
たとえば、REST API Level 0 の例 にデバッグロギングがあり、**/api/greeting** メソッドで **message** 変数をログに記録します。

1. アプリケーションのルートを取得します。

```
$ oc get routes
```

2. アプリケーションの **/api/greeting** エンドポイントで HTTP 要求を作成します。

```
$ curl $APPLICATION_ROUTE/api/greeting?name=Sarah
```

4. Pod ログのあるウィンドウに戻り、ログでデバッグロギングメッセージを検査します。

```
i.o.booster.service.GreetingEndpoint     : Message: Hello, Sarah!
```

5. デバッグロギングを無効にするには、**src/main/resources/application.properties** から **logging.level.fully.qualified.name.of.TheClass=DEBUG** を削除し、アプリケーションを再デプロイします。

第8章 アプリケーションのモニターリング

このセクションでは、OpenShift で実行する Spring Boot ベースのアプリケーションのモニターリングについて説明します。

8.1. OPENSHIFT でのアプリケーションの JVM メトリクスへのアクセス

8.1.1. OpenShift で Jolokia を使用した JVM メトリクスへのアクセス

Jolokia は、OpenShift 上の HTTP (Java Management Extension) メトリクスにアクセスするための組み込みの軽量ソリューションです。Jolokia を使用すると、HTTP ブリッジ上で JMX によって収集される CPU、ストレージ、およびメモリー使用状況データにアクセスできます。Jolokia は REST インターフェイスおよび JSON 形式のメッセージペイロードを使用します。これは、非常に高速で、リソース要件が低いため、クラウドアプリケーションのモニターリングに適しています。

Java ベースのアプリケーションの場合、OpenShift Web コンソールは、アプリケーションを実行している JVM によって関連するすべてのメトリクス出力を収集し、表示する統合 **hawt.io コンソール** を提供します。

前提条件

- **oc** クライアントが認証されている。
- OpenShift のプロジェクトで実行している Java ベースのアプリケーションコンテナー
- 最新の JDK 1.8.0 イメージ

手順

1. プロジェクト内の Pod のデプロイメント設定をリストし、アプリケーションに対応するものを選択します。

```
oc get dc
```

```
NAME        REVISION   DESIRED   CURRENT   TRIGGERED BY
MY_APP_NAME   2          1         1         config,image(my-app:6)
...
```

2. アプリケーションを実行している Pod の YAML デプロイメントテンプレートを開いて編集します。

```
oc edit dc/MY_APP_NAME
```

3. 以下のエントリーをテンプレートの **ports** セクションに追加し、変更を保存します。

```
...
spec:
  ...
  ports:
  - containerPort: 8778
    name: jolokia
```

```
      protocol: TCP
    ...
...
```

4. アプリケーションを実行する Pod を再デプロイします。

```
oc rollout latest dc/MY_APP_NAME
```

Pod は更新されたデプロイメント設定で再デプロイされ、ポート **8778** を公開します。

5. OpenShift Web コンソールにログインします。
6. サイドバーで、**Applications** > **Pods** に移動し、アプリケーションを実行する Pod の名前をクリックします。
7. Pod の詳細画面で **Open Java Console** をクリックし、hawt.io コンソールにアクセスします。

追加リソース

- hawt.io ドキュメント

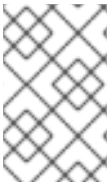
第9章 利用可能なサンプル SPRING BOOT

Spring Boot ランタイムは、サンプルアプリケーションを提供します。OpenShift でアプリケーションの開発を開始すると、サンプルアプリケーションをテンプレートとして使用できます。

これらのサンプルアプリケーションは Developer Launcher でアクセスできます。

すべてのサンプルアプリケーションを以下にダウンロードおよびデプロイできます。

- x86_64 アーキテクチャー - 本ガイドのアプリケーションの例では、サンプルアプリケーションを x86_64 アーキテクチャーにビルドおよびデプロイする方法を説明します。
- s390x アーキテクチャー - IBM Z インフラストラクチャーでプロビジョニングされた OpenShift 環境にサンプルアプリケーションをデプロイするには、コマンドに関連する IBM Z イメージ名を指定します。イメージ名の詳細は、Spring Boot でサポートされる Java イメージのセクションを参照してください。
  サンプルアプリケーションの一部には、ワークフローを実証するために Red Hat Data Grid などの他の製品も必要になります。この場合、これらの製品のイメージ名をサンプルアプリケーションの YAML ファイルで関連する IBM Z イメージ名に変更する必要もあります。



注記

Spring Boot の Secured サンプルアプリケーションでは、Red Hat SSO 7.3 が必要です。Red Hat SSO 7.3 は IBM Z でサポートされていないため、IBM Z では Secured の例は利用できません。

9.1. SPRING BOOT の REST API LEVEL 0 サンプル

重要

以下の例は、実稼働環境での実行を目的としていません。

上達度レベルの例: **Foundational**

REST API Level 0 サンプルでできること

REST API Level 0 サンプルでは、REST フレームワークを使用して、HTTP 経由でビジネスオペレーションをリモートプロシージャコールエンドポイントにマッピングする方法が示されています。これは、Richardson Maturity Model の Level 0 に対応します。REST およびその基本的な原則を使用して HTTP エンドポイントを作成すると、API を柔軟にプロトタイプおよび設計することができます。

この例では、HTTP プロトコルを使用してリモートサービスと対話するためのメカニズムが導入されました。これにより、以下が可能になります。

- **api/greeting** エンドポイントで HTTP **GET** 要求を実行します。
- **Hello, World!** で設定されるペイロードを使用して JSON 形式でレスポンスを受け取ります。文字列。
- String 引数を渡し、**api/greeting** エンドポイントで HTTP **GET** 要求を実行します。これにより、クエリー文字列に **name** 要求パラメーターが使用されます。
- **Hello, $name!** のペイロードを含む JSON 形式の応答を受信します。**$name** は、要求に渡された **name** パラメーターの値に置き換えられます。

9.1.1. REST API Level 0 設計トレードオフ

表9.1 設計トレードオフ

利点	悪い点

• アプリケーション例では、高速なプロトタイプを有効にします。
• API Design は柔軟性があります。
• HTTP エンドポイントにより、クライアントは言語に依存しません。
	• アプリケーションまたはサービスが成熟するにつれて、REST API Level 0 のアプローチは適切にスケーリングされない可能性があります。クリーンな API 設計や、データベースの対話に関するユースケースをサポートしない場合があります。
	  ◦ 共有された変更可能な状態を含むすべての操作は、適切なバッキングデータストアと統合する必要があります。
	  ◦ この API 設計で処理されるすべての要求は、要求に対応するコンテナーにのみスコープ指定されます。これ以降の要求は、同じコンテナーで処理されない可能性があります。

9.1.2. REST API Level 0 サンプルアプリケーションの OpenShift Online へのデプロイメント

以下のオプションのいずれかを使用して、OpenShift Online で REST API Level 0 サンプルアプリケーションを実行します。

- developers.redhat.com/launch の使用
- CLI クライアント **oc** の使用

各メソッドは、同じ **oc** コマンドを使用してアプリケーションをデプロイしますが、developers.redhat.com/launch を使用すると、**oc** コマンドを実行する自動デプロイメントワークフローが提供されます。

9.1.2.1. developers.redhat.com/launch を使用したサンプルアプリケーションのデプロイメント

このセクションでは、REST API Level 0 のサンプルアプリケーションをビルドし、Red Hat Developer Launcher Web インターフェイスから OpenShift にデプロイする方法を説明します。

前提条件

- OpenShift Online のアカウント。

手順

1. ブラウザーで developers.redhat.com/launch URL に移動します。
2. 画面の指示に従って、Spring Boot でサンプルアプリケーションを作成して起動します。

9.1.2.2. CLI クライアント oc の認証

oc コマンドラインクライアントを使用して OpenShift Online でアプリケーションのサンプルを使用するには、OpenShift Online の Web インターフェイスによって提供されるトークンを使用してクライアントを認証する必要があります。

前提条件

- OpenShift Online のアカウント。

手順

1. ブラウザーで OpenShift Online URL に移動します。
2. ユーザー名の横にある Web コンソールの右上にあるクエスチョンマークアイコンをクリックします。
3. ドロップダウンメニューで **Command Line Tools** を選択します。
4. **oc login** コマンドをコピーします。
5. 端末にコマンドを貼り付けます。このコマンドは、認証トークンを使用して CLI クライアント **oc** を OpenShift Online アカウントで認証します。

```
$ oc login OPENSHIFT_URL --token=MYTOKEN
```

9.1.2.3. CLI クライアント oc を使用した REST API Level 0 サンプルアプリケーションのデプロイメント

このセクションでは、REST API Level 0 サンプルアプリケーションをビルドし、コマンドラインから OpenShift にデプロイする方法を説明します。

前提条件

- developers.redhat.com/launch を使用して作成されたサンプルアプリケーション。詳細は「developers.redhat.com/launch を使用したサンプルアプリケーションのデプロイメント」を参照してください。
- 認証された **oc** クライアント。詳細は「CLI クライアント **oc** の認証」を参照してください。

手順

1. GitHub からプロジェクトのクローンを作成します。

```
$ git clone git@github.com:USERNAME/MY_PROJECT_NAME.git
```

または、プロジェクトの ZIP ファイルをダウンロードして、展開します。

```
$ unzip MY_PROJECT_NAME.zip
```

2. OpenShift で新規プロジェクトを作成します。

```
$ oc new-project MY_PROJECT_NAME
```

3. アプリケーションの root ディレクトリーに移動します。

4. Maven を使用して OpenShift へのデプロイメントを開始します。

```
$ mvn clean fabric8:deploy -Popenshift
```

このコマンドは、Fabric8 Maven プラグインを使用して OpenShift で S2I プロセス を起動し、Pod を起動します。

5. アプリケーションのステータスを確認し、Pod が実行していることを確認します。

```
$ oc get pods -w
NAME                      READY     STATUS      RESTARTS   AGE
MY_APP_NAME-1-aaaaa            1/1       Running     0          58s
MY_APP_NAME-s2i-1-build        0/1       Completed   0          2m
```

MY_APP_NAME-1-aaaaa Pod は、完全にデプロイされて起動すると、ステータスが **Running** になるはずです。特定の Pod 名が異なります。中間の数字は新規ビルドごとに増えます。末尾の文字は、Pod の作成時に生成されます。

6. アプリケーションのサンプルをデプロイして起動すると、そのルートを決定します。

ルート情報の例

```
$ oc get routes
NAME              HOST/PORT                                   PATH      SERVICES
PORT      TERMINATION
MY_APP_NAME       MY_APP_NAME-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME
MY_APP_NAME      8080
```

Pod のルート情報には、アクセスに使用するベース URL が提供されます。上記の例では、**http://MY_APP_NAME-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME** をベース URL として使用し、アプリケーションにアクセスします。

9.1.3. REST API Level 0 サンプルアプリケーションの Minishift または CDK へのデプロイメント

以下のオプションのいずれかを使用して、REST API Level 0 サンプルアプリケーションを Minishift または CDK でローカルに実行します。

- Fabric8 Launcher の使用
- CLI クライアント **oc** の使用

各メソッドは、同じ **oc** コマンドを使用してアプリケーションをデプロイしますが、Fabric8 Launcher を使用すると、**oc** コマンドを実行する自動デプロイメントワークフローが提供されます。

9.1.3.1. Fabric8 Launcher ツールの URL および認証情報の取得

Minishift または CDK にサンプルアプリケーションを作成してデプロイするには、Fabric8 Launcher ツール URL とユーザー認証情報が必要です。この情報は、Minishift または CDK の起動時に提供されます。

前提条件

- Fabric8 Launcher ツールがインストールされ、設定され、実行している。

手順

1. Minishift または CDK を起動したコンソールに移動します。
2. 実行中の Fabric8 Launcher にアクセスするのに使用できる URL およびユーザー認証情報のコンソール出力を確認します。

Minishift または CDK 起動時のコンソール出力の例

```
...
-- Removing temporary directory ... OK
-- Server Information ...
   OpenShift server started.
   The server is accessible via web console at:
      https://192.168.42.152:8443

   You are logged in as:
      User:     developer
      Password: developer

   To login as administrator:
      oc login -u system:admin
```

9.1.3.2. Fabric8 Launcher ツールを使用したサンプルアプリケーションのデプロイメント

このセクションでは、REST API Level 0 のサンプルアプリケーションをビルドし、Fabric8 Launcher Web インターフェイスから OpenShift にデプロイする方法を説明します。

前提条件

- 実行中の Fabric8 Launcher インスタンスの URL と、Minishift または CDK のユーザー認証情報。詳細は「Fabric8 Launcher ツールの URL および認証情報の取得」 を参照してください。

手順

1. ブラウザーで Fabric8 Launcher URL に移動します。
2. 画面の指示に従って、Spring Boot でサンプルアプリケーションを作成して起動します。

9.1.3.3. CLI クライアント oc の認証

oc コマンドラインクライアントを使用して Minishift または CDK でサンプルアプリケーションを使用するには、Minishift または CDK Web インターフェイスが提供するトークンを使用してクライアントを認証する必要があります。

前提条件

- 実行中の Fabric8 Launcher インスタンスの URL と、Minishift または CDK のユーザー認証情報。詳細は「Fabric8 Launcher ツールの URL および認証情報の取得」 を参照してください。

手順

1. ブラウザーで Minishift または CDK URL に移動します。

2. ユーザー名の横にある Web コンソールの右上にあるクエスチョンマークアイコンをクリックします。
3. ドロップダウンメニューで **Command Line Tools** を選択します。
4. **oc login** コマンドをコピーします。
5. 端末にコマンドを貼り付けます。このコマンドは、認証トークンを使用して、Minishift または CDK アカウントで CLI クライアント **oc** を認証します。

```
$ oc login OPENSHIFT_URL --token=MYTOKEN
```

9.1.3.4. CLI クライアント oc を使用した REST API Level 0 サンプルアプリケーションのデプロイメント

このセクションでは、REST API Level 0 サンプルアプリケーションをビルドし、コマンドラインから OpenShift にデプロイする方法を説明します。

前提条件

- Minishift または CDK の Fabric8 Launcher ツールを使用して作成されたサンプルアプリケーション。詳細は「Fabric8 Launcher ツールを使用したサンプルアプリケーションのデプロイメント」を参照してください。
- Fabric8 Launcher ツールの URL。
- 認証された **oc** クライアント。詳細は「CLI クライアント **oc** の認証」を参照してください。

手順

1. GitHub からプロジェクトのクローンを作成します。

```
$ git clone git@github.com:USERNAME/MY_PROJECT_NAME.git
```

   または、プロジェクトの ZIP ファイルをダウンロードして、展開します。

```
$ unzip MY_PROJECT_NAME.zip
```

2. OpenShift で新規プロジェクトを作成します。

```
$ oc new-project MY_PROJECT_NAME
```

3. アプリケーションの root ディレクトリーに移動します。
4. Maven を使用して OpenShift へのデプロイメントを開始します。

```
$ mvn clean fabric8:deploy -Popenshift
```

   このコマンドは、Fabric8 Maven プラグインを使用して OpenShift で S2I プロセス を起動し、Pod を起動します。

5. アプリケーションのステータスを確認し、Pod が実行していることを確認します。

```
$ oc get pods -w
```

```
NAME                        READY     STATUS      RESTARTS   AGE
MY_APP_NAME-1-aaaaa         1/1       Running     0          58s
MY_APP_NAME-s2i-1-build     0/1       Completed   0          2m
```

MY_APP_NAME-1-aaaaa Pod は、完全にデプロイされて起動すると、ステータスが **Running** になるはずです。特定の Pod 名が異なります。中間の数字は新規ビルドごとに増えます。末尾の文字は、Pod の作成時に生成されます。

6. アプリケーションのサンプルをデプロイして起動すると、そのルートを決定します。

ルート情報の例

```
$ oc get routes
NAME              HOST/PORT                                        PATH      SERVICES
PORT      TERMINATION
MY_APP_NAME       MY_APP_NAME-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME
MY_APP_NAME      8080
```

Pod のルート情報には、アクセスに使用するベース URL が提供されます。上記の例では、**http://MY_APP_NAME-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME** をベース URL として使用し、アプリケーションにアクセスします。

9.1.4. REST API Level 0 サンプルアプリケーションの OpenShift Container Platform へのデプロイメント

サンプルアプリケーションを OpenShift Container Platform に作成し、デプロイするプロセスは OpenShift Online に似ています。

前提条件

- developers.redhat.com/launch を使用して作成されたサンプルアプリケーション。

手順

- 「REST API Level 0 サンプルアプリケーションの OpenShift Online へのデプロイメント」 の説明に従い、OpenShift Container Platform Web コンソールの URL およびユーザー認証情報のみを使用します。

9.1.5. Spring Boot の未変更の REST API Level 0 サンプルアプリケーションとの対話

この例では、GET 要求を受け入れるデフォルトの HTTP エンドポイントを提供します。

前提条件

- アプリケーションの実行
- **curl** バイナリーまたは Web ブラウザー

手順

1. **curl** を使用して、サンプルに **GET** 要求を実行します。これを行うには、ブラウザーを使用することもできます。

```
$ curl http://MY_APP_NAME-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME/api/greeting
{"content":"Hello, World!"}
```

2. **curl** を使用して、例に対して URL パラメーター **name** を付けて **GET** 要求を実行します。これを行うには、ブラウザーを使用することもできます。

```
$ curl http://MY_APP_NAME-
MY_PROJECT_NAME.OPENSHIFT_HOSTNAME/api/greeting?name=Sarah
{"content":"Hello, Sarah!"}
```

注記

ブラウザーから、例で提供されているフォームを使用して、これらの同じ対話を実行することもできます。フォームは、プロジェクト **http://MY_APP_NAME-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME** の root にあります。

9.1.6. REST API Level 0 のサンプルアプリケーション統合テストの実行

このサンプルアプリケーションには、自己完結型の統合テストセットが含まれます。OpenShift プロジェクト内で実行する場合、テストは以下を行います。

- アプリケーションのテストインスタンスをプロジェクトにデプロイします。
- そのインスタンスで個別のテストを実行します。
- テストが完了したら、プロジェクトからアプリケーションのすべてのインスタンスを削除します。

警告

統合テストを実行すると、サンプルアプリケーションの既存インスタンスがすべて、ターゲット OpenShift プロジェクトから削除されます。サンプルアプリケーションが正しく削除されないようにするには、テストを実行するために別の OpenShift プロジェクトを作成して選択してください。

前提条件

- 認証された **oc** クライアント。
- 空の OpenShift プロジェクト。

手順

次のコマンドを実行して統合テストを実行します。

```
$ mvn clean verify -Popenshift,openshift-it
```

9.1.7. REST リソース

REST の背景および関連情報は、以下を参照してください。

- Architectural Styles and the Design of Network-based Software Architectures - Representational State Transfer (REST)
- Richardson Maturity Model
- JSR 311: JAX-RS: The JavaTM API for RESTful Web Services
- Spring での RESTful サービスの構築
- Eclipse Vert.x の REST API Level 0
- Thorntail の REST API Level 0
- Node.js の REST API Level 0

9.2. SPRING BOOT の外部化設定の例

重要

以下の例は、実稼働環境での実行を目的としていません。

上達度レベルの例: **Foundational**

外部化された設定は、ConfigMap を使用して設定を外部化する基本的な例を提供します。**ConfigMap** は、コンテナーを OpenShift に依存しないようにする一方で、単純なキーと値のペアとして設定データを 1 つ以上の Linux コンテナーに挿入するために OpenShift で使用されるオブジェクトです。

この例では、以下の方法を示しています。

- **ConfigMap** をセットアップし、設定します。
- アプリケーション内で **ConfigMap** によって提供される設定を使用します。
- 実行中のアプリケーションの **ConfigMap** 設定に変更をデプロイします。

9.2.1. 外部化された設定の設計パターン

可能な限り、アプリケーション設定を外部化し、アプリケーションコードから分離させます。これにより、異なる環境を通過する際にアプリケーション設定を変更できますが、コードは変更されません。設定を外部化すると、機密情報や内部情報がコードベースやバージョン管理から除外されます。多くの言語およびアプリケーションサーバーは、アプリケーション設定の外部化をサポートする環境変数を提供します。

マイクロサービスアーキテクチャーおよび多言語 (polyglot) 環境は、アプリケーションの設定を管理する複雑な層を追加します。アプリケーションは独立した分散サービスで設定され、それぞれ独自の設定を持つことができます。すべての設定データを同期し、アクセス可能な状態に維持すると、メンテナーンスの課題が発生します。

ConfigMap により、アプリケーション設定を外部化でき、OpenShift 上の個々の Linux コンテナーおよび Pod で使用できます。YAML ファイルの使用を含むさまざまな方法で ConfigMap オブジェクトを作成し、これを Linux コンテナーに挿入できます。ConfigMap を使用すると、設定データのグループ化およびスケーリングが可能です。これにより、基本的な **開発**、**ステージ**、および **実稼働** 以外の多くの環境を設定できます。ConfigMap の詳細は、 OpenShift ドキュメント を参照してください。

9.2.2. 外部化設定設計のトレードオフ

表9.2 設計のトレードオフ

利点	悪い点
• 設定がデプロイメントとは分離される • 個別に更新可能 • サービス間で共有できる	• 環境への設定の追加には追加のステップが必要 • 別々に維持する必要がある • サービスのスコープを超える調整が必要

9.2.3. 外部化設定のサンプルアプリケーションの OpenShift Online へのデプロイメント

以下のいずれかのオプションを使用して、OpenShift Online で外部化設定アプリケーションを実行します。

- developers.redhat.com/launch の使用
- CLI クライアント **oc** の使用

各メソッドは、同じ **oc** コマンドを使用してアプリケーションをデプロイしますが、developers.redhat.com/launch を使用すると、**oc** コマンドを実行する自動デプロイメントワークフローが提供されます。

9.2.3.1. developers.redhat.com/launch を使用したサンプルアプリケーションのデプロイメント

このセクションでは、REST API Level 0 のサンプルアプリケーションをビルドし、Red Hat Developer Launcher Web インターフェイスから OpenShift にデプロイする方法を説明します。

前提条件

- OpenShift Online のアカウント。

手順

1. ブラウザーで developers.redhat.com/launch URL に移動します。
2. 画面の指示に従って、Spring Boot でサンプルアプリケーションを作成して起動します。

9.2.3.2. CLI クライアント oc の認証

oc コマンドラインクライアントを使用して OpenShift Online でアプリケーションのサンプルを使用するには、OpenShift Online の Web インターフェイスによって提供されるトークンを使用してクライアントを認証する必要があります。

前提条件

- OpenShift Online のアカウント。

手順

1. ブラウザーで OpenShift Online URL に移動します。
2. ユーザー名の横にある Web コンソールの右上にあるクエスチョンマークアイコンをクリックします。
3. ドロップダウンメニューで **Command Line Tools** を選択します。
4. **oc login** コマンドをコピーします。
5. 端末にコマンドを貼り付けます。このコマンドは、認証トークンを使用して CLI クライアント **oc** を OpenShift Online アカウントで認証します。

```
$ oc login OPENSHIFT_URL --token=MYTOKEN
```

9.2.3.3. CLI クライアント oc を使用した Externalized Configuration アプリケーションのデプロイメント

このセクションでは、外部化設定のサンプルアプリケーションをビルドし、コマンドラインから OpenShift にデプロイする方法を説明します。

前提条件

- developers.redhat.com/launch を使用して作成されたサンプルアプリケーション。詳細は「developers.redhat.com/launch を使用したサンプルアプリケーションのデプロイメント」を参照してください。
- 認証された **oc** クライアント。詳細は 「CLI クライアント **oc** の認証」 を参照してください。

手順

1. GitHub からプロジェクトのクローンを作成します。

```
$ git clone git@github.com:USERNAME/MY_PROJECT_NAME.git
```

   または、プロジェクトの ZIP ファイルをダウンロードして、展開します。

```
$ unzip MY_PROJECT_NAME.zip
```

2. 新しい OpenShift プロジェクトを作成します。

```
$ oc new-project MY_PROJECT_NAME
```

3. サンプルアプリケーションをデプロイする前に、サービスアカウントへの view アクセス権を割り当て、ConfigMap の内容を読み取るためにアプリケーションが OpenShift API にアクセスできるようにします。

```
$ oc policy add-role-to-user view -n $(oc project -q) -z default
```

4. アプリケーションの root ディレクトリーに移動します。
5. **application.yml** を使用して ConfigMap 設定を OpenShift にデプロイします。

```
$ oc create configmap app-config --from-file=application.yml
```

6. ConfigMap 設定がデプロイされていることを確認します。

```
$ oc get configmap app-config -o yaml

apiVersion: template.openshift.io/v1
data:
  application.yml: |
    # This properties file should be used to initialise a ConfigMap
    greeting:
      message: "Hello %s from a ConfigMap!"
...
```

7. Maven を使用して OpenShift へのデプロイメントを開始します。

```
$ mvn clean fabric8:deploy -Popenshift
```

このコマンドは、Fabric8 Maven プラグインを使用して OpenShift で S2I プロセス を起動し、Pod を起動します。

8. アプリケーションのステータスを確認し、Pod が実行していることを確認します。

```
$ oc get pods -w
NAME                              READY     STATUS      RESTARTS   AGE
MY_APP_NAME-1-aaaaa               1/1       Running     0          58s
MY_APP_NAME-s2i-1-build           0/1       Completed   0          2m
```

MY_APP_NAME-1-aaaaa Pod は、完全にデプロイされて起動すると、ステータスが **Running** である必要があります。特定の Pod 名が異なります。中間の数字は新規ビルドごとに増えます。末尾の文字は、Pod の作成時に生成されます。

9. アプリケーションのサンプルをデプロイして起動すると、そのルートを決定します。

ルート情報の例

```
$ oc get routes
NAME              HOST/PORT                                        PATH      SERVICES
PORT      TERMINATION
MY_APP_NAME       MY_APP_NAME-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME
MY_APP_NAME      8080
```

Pod のルート情報には、アクセスに使用するベース URL が提供されます。上記の例では、**http://MY_APP_NAME-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME** をベース URL として使用し、アプリケーションにアクセスします。

9.2.4. 外部化設定アプリケーションの Minishift または CDK へのデプロイメント

以下のオプションのいずれかを使用して、Minishift または CDK で外部設定サンプルアプリケーションをローカルで実行します。

- Fabric8 Launcher の使用
- CLI クライアント **oc** の使用

各メソッドは、同じ **oc** コマンドを使用してアプリケーションをデプロイしますが、Fabric8 Launcher を使用すると、**oc** コマンドを実行する自動デプロイメントワークフローが提供されます。

9.2.4.1. Fabric8 Launcher ツールの URL および認証情報の取得

Minishift または CDK にサンプルアプリケーションを作成してデプロイするには、Fabric8 Launcher ツール URL とユーザー認証情報が必要です。この情報は、Minishift または CDK の起動時に提供されます。

前提条件

- Fabric8 Launcher ツールがインストールされ、設定され、実行している。

手順

1. Minishift または CDK を起動したコンソールに移動します。
2. 実行中の Fabric8 Launcher にアクセスするのに使用できる URL およびユーザー認証情報のコンソール出力を確認します。

Minishift または CDK 起動時のコンソール出力の例

```
...
-- Removing temporary directory ... OK
-- Server Information ...
   OpenShift server started.
   The server is accessible via web console at:
      https://192.168.42.152:8443

   You are logged in as:
      User:     developer
      Password: developer

   To login as administrator:
      oc login -u system:admin
```

9.2.4.2. Fabric8 Launcher ツールを使用したサンプルアプリケーションのデプロイメント

このセクションでは、REST API Level 0 のサンプルアプリケーションをビルドし、Fabric8 Launcher Web インターフェイスから OpenShift にデプロイする方法を説明します。

前提条件

- 実行中の Fabric8 Launcher インスタンスの URL と、Minishift または CDK のユーザー認証情報。詳細は「Fabric8 Launcher ツールの URL および認証情報の取得」 を参照してください。

手順

1. ブラウザーで Fabric8 Launcher URL に移動します。
2. 画面の指示に従って、Spring Boot でサンプルアプリケーションを作成して起動します。

9.2.4.3. CLI クライアント oc の認証

oc コマンドラインクライアントを使用して Minishift または CDK でサンプルアプリケーションを使用するには、Minishift または CDK Web インターフェイスが提供するトークンを使用してクライアントを認証する必要があります。

前提条件

- 実行中の Fabric8 Launcher インスタンスの URL と、Minishift または CDK のユーザー認証情報。詳細は「Fabric8 Launcher ツールの URL および認証情報の取得」 を参照してください。

手順

1. ブラウザーで Minishift または CDK URL に移動します。
2. ユーザー名の横にある Web コンソールの右上にあるクエスチョンマークアイコンをクリックします。
3. ドロップダウンメニューで **Command Line Tools** を選択します。
4. **oc login** コマンドをコピーします。
5. 端末にコマンドを貼り付けます。このコマンドは、認証トークンを使用して、Minishift または CDK アカウントで CLI クライアント **oc** を認証します。

```
$ oc login OPENSHIFT_URL --token=MYTOKEN
```

9.2.4.4. CLI クライアント oc を使用した Externalized Configuration アプリケーションのデプロイメント

このセクションでは、外部化設定のサンプルアプリケーションをビルドし、コマンドラインから OpenShift にデプロイする方法を説明します。

前提条件

- Minishift または CDK の Fabric8 Launcher ツールを使用して作成されたサンプルアプリケーション。詳細は「Fabric8 Launcher ツールを使用したサンプルアプリケーションのデプロイメント」を参照してください。
- Fabric8 Launcher ツールの URL。
- 認証された **oc** クライアント。詳細は「CLI クライアント **oc** の認証」を参照してください。

手順

1. GitHub からプロジェクトのクローンを作成します。

```
$ git clone git@github.com:USERNAME/MY_PROJECT_NAME.git
```

   または、プロジェクトの ZIP ファイルをダウンロードして、展開します。

```
$ unzip MY_PROJECT_NAME.zip
```

2. 新しい OpenShift プロジェクトを作成します。

```
$ oc new-project MY_PROJECT_NAME
```

3. サンプルアプリケーションをデプロイする前に、サービスアカウントへの view アクセス権を割り当て、ConfigMap の内容を読み取るためにアプリケーションが OpenShift API にアクセスできるようにします。

```
$ oc policy add-role-to-user view -n $(oc project -q) -z default
```

4. アプリケーションの root ディレクトリーに移動します。

5. **application.yml** を使用して ConfigMap 設定を OpenShift にデプロイします。

```
$ oc create configmap app-config --from-file=application.yml
```

6. ConfigMap 設定がデプロイされていることを確認します。

```
$ oc get configmap app-config -o yaml

apiVersion: template.openshift.io/v1
data:
  application.yml: |
    # This properties file should be used to initialise a ConfigMap
    greeting:
      message: "Hello %s from a ConfigMap!"
...
```

7. Maven を使用して OpenShift へのデプロイメントを開始します。

```
$ mvn clean fabric8:deploy -Popenshift
```

このコマンドは、Fabric8 Maven プラグインを使用して OpenShift で S2I プロセス を起動し、Pod を起動します。

8. アプリケーションのステータスを確認し、Pod が実行していることを確認します。

```
$ oc get pods -w
NAME                          READY     STATUS      RESTARTS   AGE
MY_APP_NAME-1-aaaaa           1/1       Running     0          58s
MY_APP_NAME-s2i-1-build       0/1       Completed   0          2m
```

MY_APP_NAME-1-aaaaa Pod は、完全にデプロイされて起動すると、ステータスが **Running** である必要があります。特定の Pod 名が異なります。中間の数字は新規ビルドごとに増えます。末尾の文字は、Pod の作成時に生成されます。

9. アプリケーションのサンプルをデプロイして起動すると、そのルートを決定します。

ルート情報の例

```
$ oc get routes
NAME              HOST/PORT                                        PATH      SERVICES
PORT      TERMINATION
MY_APP_NAME       MY_APP_NAME-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME
MY_APP_NAME      8080
```

Pod のルート情報には、アクセスに使用するベース URL が提供されます。上記の例で

Pod のルート情報には、アクセスに使用するベース URL が提供されます。上記の例では、**http://MY_APP_NAME-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME** をベース URL として使用し、アプリケーションにアクセスします。

9.2.5. 外部設定サンプルアプリケーションの OpenShift Container Platform へのデプロイメント

サンプルアプリケーションを OpenShift Container Platform に作成し、デプロイするプロセスは OpenShift Online に似ています。

前提条件

- developers.redhat.com/launch を使用して作成されたサンプルアプリケーション。

手順

- 「外部化設定のサンプルアプリケーションの OpenShift Online へのデプロイメント」 の説明に従い、OpenShift Container Platform Web コンソールの URL およびユーザー認証情報のみを使用します。

9.2.6. Spring Boot の未変更の外部化設定サンプルアプリケーションとの対話

この例では、GET 要求を受け入れるデフォルトの HTTP エンドポイントを提供します。

前提条件

- アプリケーションの実行
- **curl** バイナリーまたは Web ブラウザー

手順

1. **curl** を使用して、サンプルに **GET** 要求を実行します。これを行うには、ブラウザーを使用することもできます。

```
$ curl http://MY_APP_NAME-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME/api/greeting
{"content":"Hello World from a ConfigMap!"}
```

2. デプロイされた ConfigMap 設定を更新します。

```
$ oc edit configmap app-config
```

greeting.message キーの値を **Bonjour!** に変更し、ファイルを保存します。これを保存すると、変更は OpenShift インスタンスに伝播されます。

3. アプリケーションの新規バージョンをデプロイし、ConfigMap 設定の変更が反映されます。

```
$ oc rollout latest dc/MY_APP_NAME
```

4. 例のステータスを確認し、新規 Pod が実行されていることを確認します。

```
$ oc get pods -w
NAME                      READY     STATUS      RESTARTS   AGE
MY_APP_NAME-1-aaaaa       1/1       Running     0          58s
```

```
MY_APP_NAME-s2i-1-build   0/1       Completed   0         2m
```

MY_APP_NAME-1-aaaaa Pod は、完全にデプロイされて起動すると、ステータスが **Running** になるはずです。特定の Pod 名が異なります。中間の数字は新規ビルドごとに増えます。末尾の文字は、Pod の作成時に生成されます。

5. 更新された ConfigMap 設定の例に対して **curl** を使用して **GET** リクエストを実行し、更新されたグリーティングを確認します。また、アプリケーションが提供する Web フォームを使用して、ブラウザーから実行することもできます。

```
$ curl http://MY_APP_NAME-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME/api/greeting
{"content":"Bonjour!"}
```

9.2.7. 外部化設定のサンプルアプリケーションの統合テストの実行

このサンプルアプリケーションには、自己完結型の統合テストセットが含まれます。OpenShift プロジェクト内で実行する場合、テストは以下を行います。

- アプリケーションのテストインスタンスをプロジェクトにデプロイします。
- そのインスタンスで個別のテストを実行します。
- テストが完了したら、プロジェクトからアプリケーションのすべてのインスタンスを削除します。

警告

統合テストを実行すると、サンプルアプリケーションの既存インスタンスがすべて、ターゲット OpenShift プロジェクトから削除されます。サンプルアプリケーションが正しく削除されないようにするには、テストを実行するために別の OpenShift プロジェクトを作成して選択してください。

前提条件

- 認証された **oc** クライアント。
- 空の OpenShift プロジェクト。
- サンプルアプリケーションのサービスアカウントに割り当てられたアクセスパーミッションを表示します。これにより、アプリケーションは ConfigMap から設定を読み取ることができます。

```
$ oc policy add-role-to-user view -n $(oc project -q) -z default
```

手順

次のコマンドを実行して統合テストを実行します。

```
$ mvn clean verify -Popenshift,openshift-it
```

9.2.8. 外部化設定リソース

外部化設定および ConfigMap の背景および関連情報は、以下を参照してください。

- OpenShift ConfigMap ドキュメント
- OpenShift での ConfigMap に関するブログ投稿
- Spring Boot での外部化設定
- Eclipse Vert.x の外部化設定
- Thorntail の Externalized Configuration
- Node.js の Externalized Configuration

9.3. SPRING BOOT のリレーショナルデータベースバックエンドのサンプル

重要

以下の例は、実稼働環境での実行を目的としていません。

制限: このサンプルアプリケーションは、Minishift または CDK で実行してください。また、手動のワークフローを使用して、この例を OpenShift Online Pro および OpenShift Container Platform にデプロイすることもできます。この例は、現在 OpenShift Online Starter では使用できません。

上達度レベルの例: **Foundational**

リレーショナルデータベースバックエンドのサンプル

リレーショナルデータベースバックエンドのサンプルは、REST API Level 0 アプリケーションを拡張して、単純な HTTP API を使用して PostgreSQL データベースで **作成**、**読み取り**、**更新**、**削除** (**CRUD**) 操作を実行する基本的な例を提供します。**CRUD** 操作は永続ストレージの 4 つの基本的な機能であり、データベースを処理する HTTP API の開発時に広く使用されます。

また、この例では、HTTP アプリケーションが OpenShift のデータベースを特定し、接続する機能も示しています。各ランタイムは、指定のケースで最も適した接続ソリューションを実装する方法を示しています。ランタイムは、API の **JDBC**、**JPA** などを使用、または **ORM** に直接アクセスするなどのオプションを選択できます。

アプリケーションのサンプルは HTTP API を公開し、HTTP で **CRUD** 操作を実行してデータを操作できるようにするエンドポイントを提供します。**CRUD** 操作は HTTP **Verbs** にマップされます。API は JSON フォーマットを使用して要求を受け取りし、ユーザーに応答を返します。また、ユーザーは、サンプルが提供するユーザーインターフェイスを使用して、アプリケーションを使用することもできます。具体的には、この例では以下を可能にするアプリケーションを提供します。

- ブラウザーでアプリケーション Web インターフェイスに移動します。これにより、**my_data** データベースのデータで **CRUD** 操作を実行する簡単な Web サイトが公開されます。
- **api/fruits** エンドポイントで HTTP **GET** 要求を実行します。
- データベース内のすべての fruits のリストが含まれる JSON 配列としてフォーマットされたレスポンスを受け取ります。

- 有効なアイテム ID を引数として渡しながら、**api/fruits/*** エンドポイントで HTTP **GET** 要求を実行します。
- 指定した ID を持つ fruit の名前が含まれる JSON 形式で応答を受け取ります。指定された ID に項目がない場合は、呼び出しにより HTTP エラー 404 が発生します。
- **api/fruits** エンドポイントで HTTP **POST** 要求を実行し、有効な **name** 値を渡してデータベースの新規エントリーを作成します。
- 有効な ID と名前を引数として渡して、**api/fruits/*** エンドポイントで HTTP **PUT** 要求を実行します。これにより、要求に指定された名前に一致するように、指定の ID を持つ項目の名前が更新されます。
- **api/fruits/*** エンドポイントで HTTP **DELETE** 要求を実行し、有効な ID を引数として渡します。これにより、指定された ID の項目がデータベースから削除され、応答として HTTP コード **204** (No Content) を返します。無効な ID を渡すと、呼び出しにより HTTP エラー **404** が発生します。

この例には、アプリケーションがデータベースと完全に統合されていることを検証するために使用できる自動化された 統合テスト のセットも含まれています。

この例では、完全に成熟した RESTful モデル (レベル 3) を紹介していませんが、推奨される HTTP API プラクティスに従って、互換性のある HTTP 動詞とステータスを使用しています。

9.3.1. リレーショナルデータベースバックエンドの設計トレードオフ

表9.3 設計のトレードオフ

利点	悪い点
• 各ランタイムは、データベースの対話の実装方法を決定します。1つは JDBC などの低レベルの接続 API を使用し、他の接続では JPA を使用できますが、別の接続は ORM API に直接アクセスできます。各ランタイムは、最適な方法を決定します。 • 各ランタイムはスキーマの作成方法を決定します。	• このサンプルアプリケーションで提供される PostgreSQL データベースは永続ストレージではバックアップされていません。データベース Pod を停止または再デプロイすると、データベースへの変更は失われます。変更を保持するために、サンプルアプリケーションの Pod で外部データベースを使用するには、OpenShift ドキュメントの Creating an application with a database を参照してください。また、OpenShift 上のデータベースコンテナーで永続ストレージを設定することもできます。OpenShift およびコンテナーで永続ストレージを使用する方法は、OpenShift ドキュメントの Persistent Storage、Managing Volumes および Persistent Volumes の章を参照してください。

9.3.2. リレーショナルデータベースバックエンドのサンプルアプリケーションの OpenShift Online へのデプロイメント

以下のオプションのいずれかを使用して、OpenShift Online でリレーショナルデータベースバックエンドアプリケーションのサンプルアプリケーションを実行します。

- developers.redhat.com/launch の使用

- CLI クライアント **oc** の使用

各メソッドは、同じ **oc** コマンドを使用してアプリケーションをデプロイしますが、developers.redhat.com/launch を使用すると、**oc** コマンドを実行する自動デプロイメントワークフローが提供されます。

9.3.2.1. developers.redhat.com/launch を使用したサンプルアプリケーションのデプロイメント

このセクションでは、REST API Level 0 のサンプルアプリケーションをビルドし、Red Hat Developer Launcher Web インターフェイスから OpenShift にデプロイする方法を説明します。

前提条件

- OpenShift Online のアカウント。

手順

1. ブラウザーで developers.redhat.com/launch URL に移動します。
2. 画面の指示に従って、Spring Boot でサンプルアプリケーションを作成して起動します。

9.3.2.2. CLI クライアント oc の認証

oc コマンドラインクライアントを使用して OpenShift Online でアプリケーションのサンプルを使用するには、OpenShift Online の Web インターフェイスによって提供されるトークンを使用してクライアントを認証する必要があります。

前提条件

- OpenShift Online のアカウント。

手順

1. ブラウザーで OpenShift Online URL に移動します。
2. ユーザー名の横にある Web コンソールの右上にあるクエスチョンマークアイコンをクリックします。
3. ドロップダウンメニューで **Command Line Tools** を選択します。
4. **oc login** コマンドをコピーします。
5. 端末にコマンドを貼り付けます。このコマンドは、認証トークンを使用して CLI クライアント **oc** を OpenShift Online アカウントで認証します。

```
$ oc login OPENSHIFT_URL --token=MYTOKEN
```

9.3.2.3. CLI クライアント oc を使用したリレーショナルデータベースバックエンドのサンプルアプリケーションのデプロイメント

このセクションでは、リレーショナルデータベースバックエンドのサンプルアプリケーションを構築し、コマンドラインから OpenShift にデプロイする方法を説明します。

前提条件

- developers.redhat.com/launch を使用して作成されたサンプルアプリケーション。詳細は「developers.redhat.com/launch を使用したサンプルアプリケーションのデプロイメント」を参照してください。
- 認証された **oc** クライアント。詳細は「CLI クライアント **oc** の認証」を参照してください。

手順

1. GitHub からプロジェクトのクローンを作成します。

```
$ git clone git@github.com:USERNAME/MY_PROJECT_NAME.git
```

   または、プロジェクトの ZIP ファイルをダウンロードして、展開します。

```
$ unzip MY_PROJECT_NAME.zip
```

2. 新しい OpenShift プロジェクトを作成します。

```
$ oc new-project MY_PROJECT_NAME
```

3. アプリケーションの root ディレクトリーに移動します。

4. PostgreSQL データベースを OpenShift にデプロイします。データベースアプリケーションの作成時に、ユーザー名、パスワード、およびデータベース名に以下の値を使用するようにしてください。これらの値を使用するサンプルアプリケーションが事前設定されています。異なる値を使用すると、アプリケーションがデータベースと統合できなくなります。

```
$ oc new-app -e POSTGRESQL_USER=luke -ePOSTGRESQL_PASSWORD=secret -
ePOSTGRESQL_DATABASE=my_data registry.access.redhat.com/rhscl/postgresql-10-rhel7
--name=my-database
```

5. データベースのステータスを確認し、Pod が実行中であることを確認します。

```
$ oc get pods -w
my-database-1-aaaaa   1/1       Running   0         45s
my-database-1-deploy   0/1       Completed   0         53s
```

   my-database-1-aaaaa Pod のステータスは **Running** で、完全にデプロイされて起動すると ready と示される必要があります。特定の Pod 名が異なります。中間の数字は新規ビルドごとに増えます。末尾の文字は、Pod の作成時に生成されます。

6. maven を使用して OpenShift へのデプロイメントを開始します。

```
$ mvn clean fabric8:deploy -Popenshift
```

   このコマンドは、Fabric8 Maven プラグインを使用して OpenShift で S2I プロセス を起動し、Pod を起動します。

7. アプリケーションのステータスを確認し、Pod が実行していることを確認します。

```
$ oc get pods -w
NAME                     READY     STATUS      RESTARTS   AGE
```

```
MY_APP_NAME-1-aaaaa       1/1       Running     0          58s
MY_APP_NAME-s2i-1-build   0/1       Completed   0          2m
```

MY_APP_NAME-1-aaaaa Pod のステータスは **Running** で、完全にデプロイされて起動すると ready と示される必要があります。

8. アプリケーションのサンプルをデプロイして起動すると、そのルートを決定します。

ルート情報の例

```
$ oc get routes
NAME              HOST/PORT                                   PATH      SERVICES          PORT
TERMINATION
MY_APP_NAME   MY_APP_NAME-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME
MY_APP_NAME   8080
```

Pod のルート情報には、アクセスに使用するベース URL が提供されます。上記の例では、**http://MY_APP_NAME-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME** をベース URL として使用し、アプリケーションにアクセスします。

9.3.3. リレーショナルデータベースバックエンドのサンプルアプリケーションの Minishift または CDK へのデプロイメント

以下のオプションのいずれかを使用して、Minishift または CDK でローカルで Relational Database Backend サンプルアプリケーションを実行します。

- Fabric8 Launcher の使用
- CLI クライアント **oc** の使用

各メソッドは、同じ **oc** コマンドを使用してアプリケーションをデプロイしますが、Fabric8 Launcher を使用すると、**oc** コマンドを実行する自動デプロイメントワークフローが提供されます。

9.3.3.1. Fabric8 Launcher ツールの URL および認証情報の取得

Minishift または CDK にサンプルアプリケーションを作成してデプロイするには、Fabric8 Launcher ツール URL とユーザー認証情報が必要です。この情報は、Minishift または CDK の起動時に提供されます。

前提条件

- Fabric8 Launcher ツールがインストールされ、設定され、実行している。

手順

1. Minishift または CDK を起動したコンソールに移動します。
2. 実行中の Fabric8 Launcher にアクセスするのに使用できる URL およびユーザー認証情報のコンソール出力を確認します。

Minishift または CDK 起動時のコンソール出力の例

```
...
-- Removing temporary directory ... OK
```

```
-- Server Information ...
   OpenShift server started.
   The server is accessible via web console at:
      https://192.168.42.152:8443

   You are logged in as:
      User:     developer
      Password: developer

   To login as administrator:
      oc login -u system:admin
```

9.3.3.2. Fabric8 Launcher ツールを使用したサンプルアプリケーションのデプロイメント

このセクションでは、REST API Level 0 のサンプルアプリケーションをビルドし、Fabric8 Launcher Web インターフェイスから OpenShift にデプロイする方法を説明します。

前提条件

- 実行中の Fabric8 Launcher インスタンスの URL と、Minishift または CDK のユーザー認証情報。詳細は「Fabric8 Launcher ツールの URL および認証情報の取得」 を参照してください。

手順

1. ブラウザーで Fabric8 Launcher URL に移動します。
2. 画面の指示に従って、Spring Boot でサンプルアプリケーションを作成して起動します。

9.3.3.3. CLI クライアント oc の認証

oc コマンドラインクライアントを使用して Minishift または CDK でサンプルアプリケーションを使用するには、Minishift または CDK Web インターフェイスが提供するトークンを使用してクライアントを認証する必要があります。

前提条件

- 実行中の Fabric8 Launcher インスタンスの URL と、Minishift または CDK のユーザー認証情報。詳細は「Fabric8 Launcher ツールの URL および認証情報の取得」 を参照してください。

手順

1. ブラウザーで Minishift または CDK URL に移動します。
2. ユーザー名の横にある Web コンソールの右上にあるクエスチョンマークアイコンをクリックします。
3. ドロップダウンメニューで **Command Line Tools** を選択します。
4. **oc login** コマンドをコピーします。
5. 端末にコマンドを貼り付けます。このコマンドは、認証トークンを使用して、Minishift または CDK アカウントで CLI クライアント **oc** を認証します。

```
$ oc login OPENSHIFT_URL --token=MYTOKEN
```

9.3.3.4. CLI クライアント oc を使用したリレーショナルデータベースバックエンドのサンプルアプリケーションのデプロイメント

このセクションでは、リレーショナルデータベースバックエンドのサンプルアプリケーションを構築し、コマンドラインから OpenShift にデプロイする方法を説明します。

前提条件

- Minishift または CDK の Fabric8 Launcher ツールを使用して作成されたサンプルアプリケーション。詳細は「Fabric8 Launcher ツールを使用したサンプルアプリケーションのデプロイメント」を参照してください。
- Fabric8 Launcher ツールの URL。
- 認証された **oc** クライアント。詳細は「CLI クライアント **oc** の認証」を参照してください。

手順

1. GitHub からプロジェクトのクローンを作成します。

```
$ git clone git@github.com:USERNAME/MY_PROJECT_NAME.git
```

   または、プロジェクトの ZIP ファイルをダウンロードして、展開します。

```
$ unzip MY_PROJECT_NAME.zip
```

2. 新しい OpenShift プロジェクトを作成します。

```
$ oc new-project MY_PROJECT_NAME
```

3. アプリケーションの root ディレクトリーに移動します。

4. PostgreSQL データベースを OpenShift にデプロイします。データベースアプリケーションの作成時に、ユーザー名、パスワード、およびデータベース名に以下の値を使用するようにしてください。これらの値を使用するサンプルアプリケーションが事前設定されています。異なる値を使用すると、アプリケーションがデータベースと統合できなくなります。

```
$ oc new-app -e POSTGRESQL_USER=luke -ePOSTGRESQL_PASSWORD=secret -
ePOSTGRESQL_DATABASE=my_data registry.access.redhat.com/rhscl/postgresql-10-rhel7
--name=my-database
```

5. データベースのステータスを確認し、Pod が実行中であることを確認します。

```
$ oc get pods -w
my-database-1-aaaaa   1/1       Running   0         45s
my-database-1-deploy   0/1       Completed   0         53s
```

   my-database-1-aaaaa Pod のステータスは **Running** で、完全にデプロイされて起動すると ready と示される必要があります。特定の Pod 名が異なります。中間の数字は新規ビルドごとに増えます。末尾の文字は、Pod の作成時に生成されます。

6. maven を使用して OpenShift へのデプロイメントを開始します。

```
$ mvn clean fabric8:deploy -Popenshift
```

このコマンドは、Fabric8 Maven プラグインを使用して OpenShift で S2I プロセス を起動し、Pod を起動します。

7. アプリケーションのステータスを確認し、Pod が実行していることを確認します。

```
$ oc get pods -w
NAME                      READY     STATUS      RESTARTS   AGE
MY_APP_NAME-1-aaaaa       1/1       Running     0          58s
MY_APP_NAME-s2i-1-build   0/1       Completed   0          2m
```

MY_APP_NAME-1-aaaaa Pod のステータスは **Running** で、完全にデプロイされて起動すると ready と示される必要があります。

8. アプリケーションのサンプルをデプロイして起動すると、そのルートを決定します。

ルート情報の例

```
$ oc get routes
NAME            HOST/PORT                                   PATH      SERVICES          PORT
TERMINATION
MY_APP_NAME   MY_APP_NAME-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME
MY_APP_NAME   8080
```

Pod のルート情報には、アクセスに使用するベース URL が提供されます。上記の例では、**http://MY_APP_NAME-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME** をベース URL として使用し、アプリケーションにアクセスします。

9.3.4. リレーショナルデータベースバックエンドのサンプルアプリケーションの OpenShift Container Platform へのデプロイメント

サンプルアプリケーションを OpenShift Container Platform に作成し、デプロイするプロセスは OpenShift Online に似ています。

前提条件

- developers.redhat.com/launch を使用して作成されたサンプルアプリケーション。

手順

- 「リレーショナルデータベースバックエンドのサンプルアプリケーションの OpenShift Online へのデプロイメント」 の説明に従い、OpenShift Container Platform Web コンソールの URL およびユーザー認証情報のみを使用します。

9.3.5. Relational Database Backend API との対話

サンプルアプリケーションの作成が完了したら、以下のように対話できます。

前提条件

- アプリケーションの実行
- **curl** バイナリーまたは Web ブラウザー

手順

1. 以下のコマンドを実行して、アプリケーションの URL を取得します。

```
$ oc get route MY_APP_NAME
```

```
NAME            HOST/PORT                                   PATH    SERVICES        PORT
TERMINATION
MY_APP_NAME     MY_APP_NAME-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME
MY_APP_NAME     8080
```

2. データベースアプリケーションの Web インターフェイスにアクセスするには、ブラウザーで **アプリケーション URL** に移動します。

```
http://MY_APP_NAME-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME
```

また、**curl** を使用して **api/fruits/*** エンドポイントで要求を直接作成できます。

データベースのエントリーの一覧表示

```
$ curl http://MY_APP_NAME-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME/api/fruits
```

```
[ {
  "id" : 1,
  "name" : "Apple",
  "stock" : 10
}, {
  "id" : 2,
  "name" : "Orange",
  "stock" : 10
}, {
  "id" : 3,
  "name" : "Pear",
  "stock" : 10
} ]
```

特定の ID のあるエントリーの取得

```
$ curl http://MY_APP_NAME-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME/api/fruits/3
```

```
{
  "id" : 3,
  "name" : "Pear",
  "stock" : 10
}
```

エントリーの新規作成

```
$ curl -H "Content-Type: application/json" -X POST -d '{"name":"Peach","stock":1}'
http://MY_APP_NAME-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME/api/fruits
```

```
{
  "id" : 4,
```

```
  "name" : "Peach",
  "stock" : 1
}
```

エントリーの更新

```
$ curl -H "Content-Type: application/json" -X PUT -d '{"name":"Apple","stock":100}'
http://MY_APP_NAME-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME/api/fruits/1
```

```
{
  "id" : 1,
  "name" : "Apple",
  "stock" : 100
}
```

エントリーの削除

```
$ curl -X DELETE http://MY_APP_NAME-
MY_PROJECT_NAME.OPENSHIFT_HOSTNAME/api/fruits/1
```

トラブルシューティング

- これらのコマンドを実行後に HTTP エラーコード **503** を応答として受け取った場合は、アプリケーションが準備状態にないことを意味します。

9.3.6. リレーショナルデータベースバックエンドのサンプルアプリケーション統合テストの実行

このサンプルアプリケーションには、自己完結型の統合テストセットが含まれます。OpenShift プロジェクト内で実行する場合、テストは以下を行います。

- アプリケーションのテストインスタンスをプロジェクトにデプロイします。
- そのインスタンスで個別のテストを実行します。
- テストが完了したら、プロジェクトからアプリケーションのすべてのインスタンスを削除します。

警告

統合テストを実行すると、サンプルアプリケーションの既存インスタンスがすべて、ターゲット OpenShift プロジェクトから削除されます。サンプルアプリケーションが正しく削除されないようにするには、テストを実行するために別の OpenShift プロジェクトを作成して選択してください。

前提条件

- 認証された **oc** クライアント。

- 空の OpenShift プロジェクト。

手順

次のコマンドを実行して統合テストを実行します。

```
$ mvn clean verify -Popenshift,openshift-it
```

9.3.7. リレーショナルデータベースリソース

OpenShift、CRUD、HTTP API、および REST でのリレーショナルデータベースの実行に関する背景および関連情報は、以下を参照してください。

- HTTP Verbs
- Architectural Styles and the Design of Network-based Software Architectures - Representational State Transfer (REST)
- REST API 設計のディベースが終了しない
- REST APIs must be Hypertext driven
- Richardson Maturity Model
- JSR 311: JAX-RS: The JavaTM API for RESTful Web Services
- Spring での RESTful サービスの構築
- Eclipse Vert.x の Relational Database Backend
- Thorntail の Relational Database Backend
- Node.js の Relational Database Backend

9.4. SPRING BOOT のヘルスチェックの例

重要

以下の例は、実稼働環境での実行を目的としていません。

上達度レベルの例: **Foundational**

アプリケーションをデプロイする場合は、アプリケーションが利用可能かどうかを確認し、受信した要求の処理を開始することが重要です。**ヘルスチェック** パターンを実装すると、アプリケーションが利用できるかどうかや要求に対応できるかどうかなど、アプリケーションの健全性を監視できます。

注記

ヘルスチェックの用語に慣れていない場合は、最初に 「ヘルスチェックの概念」 セクションを参照してください。

このユースケースの目的は、プローブを使用してヘルスチェックパターンを実証することです。プロービングは、アプリケーションの Liveness および Readiness を報告するために使用されます。このユースケースでは、HTTP **health** エンドポイントを公開して HTTP 要求を発行するアプリケーションを設定

します。コンテナーが動作している場合は、HTTP エンドポイント **health** の Liveness プローブによると、管理プラットフォームは **200** を戻りコードとして受け取り、それ以上のアクションは必要ありません。HTTP エンドポイント **health** が応答を返さない場合、たとえばスレッドがブロックされている場合は、Liveness プローブにより、アプリケーションが稼働しているとは見なされません。この場合、プラットフォームはそのアプリケーションに対応する Pod を強制終了し、アプリケーションを再起動するために新規 Pod を再作成します。

このユースケースでは、Readiness プローブを実証し、使用することもできます。アプリケーションが実行していても要求を処理できない場合 (再起動中にアプリケーションが HTTP 応答コード **503** を返す場合など)、このアプリケーションは Readiness プローブにより準備ができていないと見なされます。Readiness プローブによってアプリケーションが準備状態にあるとみなされないと、要求は Readiness プローブにより準備が整っているとみなされるまで、要求はそのアプリケーションにルーティングされません。

9.4.1. ヘルスチェックの概念

ヘルスチェックパターンを理解するには、まず以下の概念を理解する必要があります。

Liveness

Liveness は、アプリケーションが実行しているかどうかを定義します。実行中のアプリケーションが応答しない状態または停止状態に移行する可能性があるため、再起動する必要があります。Liveness の確認は、アプリケーションを再起動する必要があるかどうかを判断するのに役立ちます。

Readines

Readiness は、実行中のアプリケーションが要求を処理できるかどうかを定義します。実行中のアプリケーションがエラーまたは破損状態に切り替わり、要求にサービスを提供することができません。Readiness を確認すると、要求が引き続きそのアプリケーションにルーティングされるべきかどうかが判断されます。

フェイルオーバー

フェイルオーバーにより、サービス要求の失敗が適切に処理されるようになります。アプリケーションが要求のサービスに失敗した場合、その要求と今後の要求は **フェイルオーバー** または他のアプリケーションにルーティングできます (通常、同じアプリケーションの冗長コピーです)。

耐障害性および安定性

耐障害性および安定性により、要求処理の失敗を適切に処理できます。接続の損失によりアプリケーションが要求を処理できない場合、耐久性のあるシステムでは、接続が再確立された後にその要求を再試行できます。

プローブ

プローブは実行中のコンテナーで定期的に実行する Kubernetes の動作です。

9.4.2. Health Check サンプルアプリケーションの OpenShift Online へのデプロイメント

以下のオプションのいずれかを使用して、OpenShift Online で Health Check のサンプルアプリケーションを実行します。

- developers.redhat.com/launch の使用
- CLI クライアント **oc** の使用

各メソッドは、同じ **oc** コマンドを使用してアプリケーションをデプロイしますが、developers.redhat.com/launch を使用すると、**oc** コマンドを実行する自動デプロイメントワークフローが提供されます。

9.4.2.1. developers.redhat.com/launch を使用したサンプルアプリケーションのデプロイメント

このセクションでは、REST API Level 0 のサンプルアプリケーションをビルドし、Red Hat Developer Launcher Web インターフェイスから OpenShift にデプロイする方法を説明します。

前提条件

- OpenShift Online のアカウント。

手順

1. ブラウザーで developers.redhat.com/launch URL に移動します。
2. 画面の指示に従って、Spring Boot でサンプルアプリケーションを作成して起動します。

9.4.2.2. CLI クライアント oc の認証

oc コマンドラインクライアントを使用して OpenShift Online でアプリケーションのサンプルを使用するには、OpenShift Online の Web インターフェイスによって提供されるトークンを使用してクライアントを認証する必要があります。

前提条件

- OpenShift Online のアカウント。

手順

1. ブラウザーで OpenShift Online URL に移動します。
2. ユーザー名の横にある Web コンソールの右上にあるクエスチョンマークアイコンをクリックします。
3. ドロップダウンメニューで **Command Line Tools** を選択します。
4. **oc login** コマンドをコピーします。
5. 端末にコマンドを貼り付けます。このコマンドは、認証トークンを使用して CLI クライアント **oc** を OpenShift Online アカウントで認証します。

```
$ oc login OPENSHIFT_URL --token=MYTOKEN
```

9.4.2.3. CLI クライアント oc を使用した Health Check サンプルアプリケーションのデプロイメント

このセクションでは、Health Check のサンプルアプリケーションをビルドし、コマンドラインから OpenShift にデプロイする方法を説明します。

前提条件

- developers.redhat.com/launch を使用して作成されたサンプルアプリケーション。詳細は「developers.redhat.com/launch を使用したサンプルアプリケーションのデプロイメント」を参照してください。

- 認証された **oc** クライアント。詳細は「CLI クライアント **oc** の認証」を参照してください。

手順

1. GitHub からプロジェクトのクローンを作成します。

```
$ git clone git@github.com:USERNAME/MY_PROJECT_NAME.git
```

または、プロジェクトの ZIP ファイルをダウンロードして、展開します。

```
$ unzip MY_PROJECT_NAME.zip
```

2. 新しい OpenShift プロジェクトを作成します。

```
$ oc new-project MY_PROJECT_NAME
```

3. アプリケーションの root ディレクトリーに移動します。

4. Maven を使用して OpenShift へのデプロイメントを開始します。

```
$ mvn clean fabric8:deploy -Popenshift
```

このコマンドは、Fabric8 Maven プラグインを使用して OpenShift で S2I プロセス を起動し、Pod を起動します。

5. アプリケーションのステータスを確認し、Pod が実行していることを確認します。

```
$ oc get pods -w
NAME                      READY     STATUS      RESTARTS   AGE
MY_APP_NAME-1-aaaaa             1/1       Running     0          58s
MY_APP_NAME-s2i-1-build         0/1       Completed   0          2m
```

MY_APP_NAME-1-aaaaa Pod は、完全にデプロイされて起動すると、ステータスが **Running** になるはずです。また、続行する前に Pod が準備状態になるまで待機する必要があります。これは **READY** 列に表示されます。たとえば、**MY_APP_NAME-1-aaaaa** は、**READY** 列が **1/1** の場合に準備状態になります。特定の Pod 名が異なります。中間の数字は新規ビルドごとに増えます。末尾の文字は、Pod の作成時に生成されます。

6. アプリケーションのサンプルをデプロイして起動すると、そのルートを決定します。

ルート情報の例

```
$ oc get routes
NAME              HOST/PORT                                        PATH      SERVICES
PORT      TERMINATION
MY_APP_NAME       MY_APP_NAME-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME
MY_APP_NAME     8080
```

Pod のルート情報には、アクセスに使用するベース URL が提供されます。上記の例では、**http://MY_APP_NAME-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME** をベース URL として使用し、アプリケーションにアクセスします。

9.4.3. Health Check のサンプルアプリケーションの Minishift または CDK へのデプロイメント

以下のオプションのいずれかを使用して、Minishift または CDK で Health Check サンプルアプリケーションをローカルで実行します。

- Fabric8 Launcher の使用
- CLI クライアント **oc** の使用

各メソッドは、同じ **oc** コマンドを使用してアプリケーションをデプロイしますが、Fabric8 Launcher を使用すると、**oc** コマンドを実行する自動デプロイメントワークフローが提供されます。

9.4.3.1. Fabric8 Launcher ツールの URL および認証情報の取得

Minishift または CDK にサンプルアプリケーションを作成してデプロイするには、Fabric8 Launcher ツール URL とユーザー認証情報が必要です。この情報は、Minishift または CDK の起動時に提供されます。

前提条件

- Fabric8 Launcher ツールがインストールされ、設定され、実行している。

手順

1. Minishift または CDK を起動したコンソールに移動します。
2. 実行中の Fabric8 Launcher にアクセスするのに使用できる URL およびユーザー認証情報のコンソール出力を確認します。

Minishift または CDK 起動時のコンソール出力の例

```
...
-- Removing temporary directory ... OK
-- Server Information ...
   OpenShift server started.
   The server is accessible via web console at:
      https://192.168.42.152:8443

   You are logged in as:
      User:     developer
      Password: developer

   To login as administrator:
      oc login -u system:admin
```

9.4.3.2. Fabric8 Launcher ツールを使用したサンプルアプリケーションのデプロイメント

このセクションでは、REST API Level 0 のサンプルアプリケーションをビルドし、Fabric8 Launcher Web インターフェイスから OpenShift にデプロイする方法を説明します。

前提条件

- 実行中の Fabric8 Launcher インスタンスの URL と、Minishift または CDK のユーザー認証情報。詳細は「Fabric8 Launcher ツールの URL および認証情報の取得」 を参照してください。

手順

1. ブラウザーで Fabric8 Launcher URL に移動します。
2. 画面の指示に従って、Spring Boot でサンプルアプリケーションを作成して起動します。

9.4.3.3. CLI クライアント oc の認証

oc コマンドラインクライアントを使用して Minishift または CDK でサンプルアプリケーションを使用するには、Minishift または CDK Web インターフェイスが提供するトークンを使用してクライアントを認証する必要があります。

前提条件

- 実行中の Fabric8 Launcher インスタンスの URL と、Minishift または CDK のユーザー認証情報。詳細は「Fabric8 Launcher ツールの URL および認証情報の取得」 を参照してください。

手順

1. ブラウザーで Minishift または CDK URL に移動します。
2. ユーザー名の横にある Web コンソールの右上にあるクエスチョンマークアイコンをクリックします。
3. ドロップダウンメニューで **Command Line Tools** を選択します。
4. **oc login** コマンドをコピーします。
5. 端末にコマンドを貼り付けます。このコマンドは、認証トークンを使用して、Minishift または CDK アカウントで CLI クライアント **oc** を認証します。

```
$ oc login OPENSHIFT_URL --token=MYTOKEN
```

9.4.3.4. CLI クライアント oc を使用した Health Check サンプルアプリケーションのデプロイメント

このセクションでは、Health Check のサンプルアプリケーションをビルドし、コマンドラインから OpenShift にデプロイする方法を説明します。

前提条件

- Minishift または CDK の Fabric8 Launcher ツールを使用して作成されたサンプルアプリケーション。詳細は「Fabric8 Launcher ツールを使用したサンプルアプリケーションのデプロイメント」を参照してください。
- Fabric8 Launcher ツールの URL。
- 認証された **oc** クライアント。詳細は「CLI クライアント **oc** の認証」を参照してください。

手順

1. GitHub からプロジェクトのクローンを作成します。

```
$ git clone git@github.com:USERNAME/MY_PROJECT_NAME.git
```

または、プロジェクトの ZIP ファイルをダウンロードして、展開します。

```
$ unzip MY_PROJECT_NAME.zip
```

2. 新しい OpenShift プロジェクトを作成します。

```
$ oc new-project MY_PROJECT_NAME
```

3. アプリケーションの root ディレクトリーに移動します。

4. Maven を使用して OpenShift へのデプロイメントを開始します。

```
$ mvn clean fabric8:deploy -Popenshift
```

このコマンドは、Fabric8 Maven プラグインを使用して OpenShift で S2I プロセス を起動し、Pod を起動します。

5. アプリケーションのステータスを確認し、Pod が実行していることを確認します。

```
$ oc get pods -w
NAME                      READY     STATUS      RESTARTS   AGE
MY_APP_NAME-1-aaaaa       1/1       Running     0          58s
MY_APP_NAME-s2i-1-build   0/1       Completed   0          2m
```

MY_APP_NAME-1-aaaaa Pod は、完全にデプロイされて起動すると、ステータスが **Running** になるはずです。また、続行する前に Pod が準備状態になるまで待機する必要があります。これは **READY** 列に表示されます。たとえば、**MY_APP_NAME-1-aaaaa** は、**READY** 列が **1/1** の場合に準備状態になります。特定の Pod 名が異なります。中間の数字は新規ビルドごとに増えます。末尾の文字は、Pod の作成時に生成されます。

6. アプリケーションのサンプルをデプロイして起動すると、そのルートを決定します。

ルート情報の例

```
$ oc get routes
NAME              HOST/PORT                                        PATH      SERVICES
PORT      TERMINATION
MY_APP_NAME       MY_APP_NAME-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME
MY_APP_NAME      8080
```

Pod のルート情報には、アクセスに使用するベース URL が提供されます。上記の例では、**http://MY_APP_NAME-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME** をベース URL として使用し、アプリケーションにアクセスします。

9.4.4. Health Check サンプルアプリケーションの OpenShift Container Platform へのデプロイメント

サンプルアプリケーションを OpenShift Container Platform に作成し、デプロイするプロセスは OpenShift Online に似ています。

前提条件

- developers.redhat.com/launch を使用して作成されたサンプルアプリケーション。

手順

- 「Health Check サンプルアプリケーションの OpenShift Online へのデプロイメント」 の説明に従い、OpenShift Container Platform Web コンソールの URL およびユーザー認証情報のみを使用します。

9.4.5. 未変更の Health Check サンプルアプリケーションとの対話

サンプルアプリケーションをデプロイすると、**MY_APP_NAME** サービスが実行します。**MY_APP_NAME** サービスは、以下の REST エンドポイントを公開します。

/api/greeting
名前を String として返します。

/api/stop
障害をシミュレーションする手段として、サービスが強制的に応答しなくなります。

以下の手順は、サービスの可用性を確認し、障害をシミュレートする方法を示しています。この利用可能なサービスの障害により、そのサービスで OpenShift の自己修復機能が実行します。

Web インターフェイスを使用して、これらの手順を実施することができます。

1. **curl** を使用して、**MY_APP_NAME** サービスに対して **GET** 要求を実行します。これを行うには、ブラウザーを使用することもできます。

```
$ curl http://MY_APP_NAME-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME/api/greeting
```

```
{"content":"Hello, World!"}
```

2. **/api/stop** エンドポイントを呼び出して、その直後に **/api/greeting** エンドポイントの可用性を確認します。
/api/stop エンドポイントを呼び出すと、内部サービスの障害をシミュレートし、OpenShift の自己修復機能をトリガーします。障害のシミュレーション後に **/api/greeting** を呼び出すと、サービスは **Application is not available** ページに戻るはずです。

```
$ curl http://MY_APP_NAME-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME/api/stop
```

(続く)

```
$ curl http://MY_APP_NAME-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME/api/greeting
```

```
<html>
  <head>
  ...
  </head>
  <body>
    <div>
      <h1>Application is not available</h1>
      ...
    </div>
  </body>
</html>
```

注記

/api/stop エンドポイントを呼び出すと、OpenShift が Pod を削除するタイミングによっては、最初に 404 エラーコードが表示される可能性があります。引き続き **/api/greeting** エンドポイントを呼び出すと、OpenShift が Pod を削除した後に **Application is not available** ページが表示されます。

3. **oc get pods -w** を使用して、動作中の自己修復機能を継続的に監視します。
サービス障害の呼び出し中に、OpenShift コンソールまたは **oc** クライアントツールで機能する自己修復機能を確認できます。**READY** 状態の Pod 数がゼロ (**0/1**) になり、その後短時間 (1 分未満) で 1 (**1/1**) に戻ることが確認できるはずです。さらに、サービスの失敗を呼び出すたびに **RESTARTS** カウントが増加します。

```
$ oc get pods -w
NAME                    READY    STATUS    RESTARTS   AGE
MY_APP_NAME-1-26iy7   0/1      Running   5          18m
MY_APP_NAME-1-26iy7   1/1      Running   5         19m
```

4. 必要に応じて、Web インターフェイスを使用してサービスを呼び出します。
端末ウィンドウを使用した対話では、サービスが提供する Web インターフェイスを使用して、異なるメソッドを起動し、サービスがライフサイクルフェーズを進めるのを監視できます。

```
http://MY_APP_NAME-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME
```

5. 必要に応じて、Web コンソールを使用して、自己修復プロセスの各段階でアプリケーションによって生成されたログ出力を表示します。
    1. プロジェクトに移動します。
    2. サイドバーで **Monitoring** をクリックします。
    3. 画面右上にある **Events** をクリックし、ログメッセージを表示します。
    4. 必要に応じて、**View Details** をクリックし、Event ログの詳細なビューを表示します。

    ヘルスチェックアプリケーションは以下のメッセージを生成します。

メッセージ	Status
Unhealthy	readiness プローブが失敗しました。このメッセージは予想され、**/api/greeting** エンドポイントのシミュレートされた失敗が検出され、自己修復プロセスが開始することを示します。
Killing	サービスを実行している利用不可の Docker コンテナーは、再作成前に強制終了されています。
Pulling	最新バージョンの Docker イメージをダウンロードして、コンテナーを再作成します。
Pulled	Docker イメージが正常にダウンロードされました。

メッセージ	Status
Created	Docker コンテナーが正常に作成されました。
Started	Docker コンテナーが要求を処理する準備が整っている。

9.4.6. Health Check のサンプルアプリケーション統合テストの実行

このサンプルアプリケーションには、自己完結型の統合テストセットが含まれます。OpenShift プロジェクト内で実行する場合、テストは以下を行います。

- アプリケーションのテストインスタンスをプロジェクトにデプロイします。
- そのインスタンスで個別のテストを実行します。
- テストが完了したら、プロジェクトからアプリケーションのすべてのインスタンスを削除します。

警告

統合テストを実行すると、サンプルアプリケーションの既存インスタンスがすべて、ターゲット OpenShift プロジェクトから削除されます。サンプルアプリケーションが正しく削除されないようにするには、テストを実行するために別の OpenShift プロジェクトを作成して選択してください。

前提条件

- 認証された **oc** クライアント。
- 空の OpenShift プロジェクト。

手順

次のコマンドを実行して統合テストを実行します。

```
$ mvn clean verify -Popenshift,openshift-it
```

9.4.7. ヘルスチェックリソース

ヘルスチェックの背景および関連情報は、以下を参照してください。

- OpenShift のアプリケーションの正常性
- Kubernetes Liveness and Readiness Probes
- Eclipse Vert.x のヘルスチェック

- Thorntail のヘルスチェック
- Node.js のヘルスチェック

9.5. SPRING BOOT の CIRCUIT BREAKER の例

重要

以下の例は、実稼働環境での実行を目的としていません。

制限: このサンプルアプリケーションは、Minishift または CDK で実行してください。また、手動のワークフローを使用して、この例を OpenShift Online Pro および OpenShift Container Platform にデプロイすることもできます。この例は、現在 OpenShift Online Starter では使用できません。

上達度レベルの例: **Foundational**

Circuit Breaker 例は、サービスの障害を報告し、要求の処理に使用できるまで、失敗したサービスへのアクセスを制限する一般的なパターンを示しています。これにより、障害が発生したサービスの機能に依存する他のサービスでのカスケード障害を防ぐことができます。

この例では、サービスに Circuit Breaker および Fallback パターンを実装する方法を表しています。

9.5.1. サーキットブレーカー設計パターン

Circuit Breaker は、次のことも目的としたパターンです。

- ネットワーク障害の影響を低減し、サービスが他のサービスを同期的に呼び出すサービスアーキテクチャーにおける高いレイテンシーを削減します。
  いずれかのサービスが以下の状態では:
  - ネットワーク障害により利用できない
  - 圧倒的なトラフィックが原因で、レイテンシーが異常に高くなる

  エンドポイントを呼び出そうとする他のサービスは、エンドポイントに到達しようとして重要なリソースを使い果たし、使用できなくなる可能性があります。
- マイクロサービスアーキテクチャー全体が使用できなくなる可能性があるカスケード障害とも呼ばれる状態を防止します。
- 障害を監視する保護機能とリモート機能の間のプロキシーとして機能します。
- 障害が特定のしきい値に達すると作動し、サーキットブレーカーへのそれ以降のすべての呼び出しは、保護された呼び出しがまったく行われずに、エラーまたは事前定義されたフォールバック応答を返します。

Circuit Breaker には通常、Circuit Breaker が作動した時に通知するエラー報告メカニズムも含まれます。

Circuit Breaker の実装

- Circuit Breaker パターンが実装されると、サービスクライアントは一定間隔でプロキシー経由でリモートサービスエンドポイントを呼び出します。

- リモートサービスエンドポイントへの呼び出しが繰り返し一貫して失敗すると、Circuit Breaker が作動し、そのサービスへのすべての呼び出しが、設定されたタイムアウト期間中に即座に失敗し、事前定義されたフォールバック応答を返します。
- タイムアウト期間が終了すると、限られた数のテストコールがリモートサービスを通過し、リモートサービスが回復したか、使用できないままであるかを判断できます。
  - テストの呼び出しに失敗した場合、Circuit Breaker はサービスを利用できない状態を保ち、受信呼び出しへのフォールバック応答を返し続けます。
  - テストに成功すると、Circuit Breaker が閉じ、トラフィックが再度リモートサービスに到達できるようにします。

9.5.2. Circuit Breaker 設計のトレードオフ

表9.4 設計のトレードオフ

利点	悪い点

• サービスが、自身が呼び出す他のサービスの障害を処理できるようになる。	• タイムアウト値の最適化が困難な場合がある。
	  ◦ タイムアウト値を必要以上に大きくすると、レイテンシーが過度に生成される可能性があります。
	  ◦ タイムアウトを必要な値より小さくすると、誤検知が発生する可能性があります。

9.5.3. Circuit Breaker サンプルアプリケーションの OpenShift Online へのデプロイメント

以下のオプションのいずれかを使用して、OpenShift Online で Circuit Breaker サンプルアプリケーションを実行します。

- developers.redhat.com/launch の使用
- CLI クライアント **oc** の使用

各メソッドは、同じ **oc** コマンドを使用してアプリケーションをデプロイしますが、developers.redhat.com/launch を使用すると、**oc** コマンドを実行する自動デプロイメントワークフローが提供されます。

9.5.3.1. developers.redhat.com/launch を使用したサンプルアプリケーションのデプロイメント

このセクションでは、REST API Level 0 のサンプルアプリケーションをビルドし、Red Hat Developer Launcher Web インターフェイスから OpenShift にデプロイする方法を説明します。

前提条件

- OpenShift Online のアカウント。

手順

手順

1. ブラウザーで developers.redhat.com/launch URL に移動します。
2. 画面の指示に従って、Spring Boot でサンプルアプリケーションを作成して起動します。

9.5.3.2. CLI クライアント oc の認証

oc コマンドラインクライアントを使用して OpenShift Online でアプリケーションのサンプルを使用するには、OpenShift Online の Web インターフェイスによって提供されるトークンを使用してクライアントを認証する必要があります。

前提条件

- OpenShift Online のアカウント。

手順

1. ブラウザーで OpenShift Online URL に移動します。
2. ユーザー名の横にある Web コンソールの右上にあるクエスチョンマークアイコンをクリックします。
3. ドロップダウンメニューで **Command Line Tools** を選択します。
4. **oc login** コマンドをコピーします。
5. 端末にコマンドを貼り付けます。このコマンドは、認証トークンを使用して CLI クライアント **oc** を OpenShift Online アカウントで認証します。

```
$ oc login OPENSHIFT_URL --token=MYTOKEN
```

9.5.3.3. CLI クライアント oc を使用した Circuit Breaker サンプルアプリケーションのデプロイメント

このセクションでは、Circuit Breaker サンプルアプリケーションをビルドし、コマンドラインから OpenShift にデプロイする方法を説明します。

前提条件

- developers.redhat.com/launch を使用して作成されたサンプルアプリケーション。詳細は「developers.redhat.com/launch を使用したサンプルアプリケーションのデプロイメント」を参照してください。
- 認証された **oc** クライアント。詳細は「CLI クライアント **oc** の認証」を参照してください。

手順

1. GitHub からプロジェクトのクローンを作成します。

```
$ git clone git@github.com:USERNAME/MY_PROJECT_NAME.git
```

   または、プロジェクトの ZIP ファイルをダウンロードして、展開します。

```
$ unzip MY_PROJECT_NAME.zip
```

2. 新しい OpenShift プロジェクトを作成します。

```
$ oc new-project MY_PROJECT_NAME
```

3. アプリケーションの root ディレクトリーに移動します。

4. Maven を使用して OpenShift へのデプロイメントを開始します。

```
$ mvn clean fabric8:deploy -Popenshift
```

このコマンドは、Fabric8 Maven プラグインを使用して OpenShift で S2I プロセス を起動し、Pod を起動します。

5. アプリケーションのステータスを確認し、Pod が実行していることを確認します。

```
$ oc get pods -w
NAME                          READY     STATUS      RESTARTS   AGE
MY_APP_NAME-greeting-1-aaaaa     1/1       Running   0          17s
MY_APP_NAME-greeting-1-deploy    0/1       Completed 0          22s
MY_APP_NAME-name-1-aaaaa         1/1       Running   0          14s
MY_APP_NAME-name-1-deploy        0/1       Completed 0          28s
```

MY_APP_NAME-greeting-1-aaaaa Pod および **MY_APP_NAME-name-1-aaaaa** Pod の両方には、完全にデプロイおよび起動すると **Running** のステータスが必要になります。また、続行する前に Pod が準備状態になるまで待機する必要があります。これは **READY** 列に表示されます。たとえば、**MY_APP_NAME-greeting-1-aaaaa** は、**READY** 列が **1/1** の場合に準備状態になります。特定の Pod 名が異なります。中間の数字は新規ビルドごとに増えます。末尾の文字は、Pod の作成時に生成されます。

6. アプリケーションのサンプルをデプロイして起動すると、そのルートを決定します。

ルート情報の例

```
$ oc get routes
NAME                HOST/PORT                                        PATH      SERVICES
PORT      TERMINATION
MY_APP_NAME-greeting   MY_APP_NAME-greeting-
MY_PROJECT_NAME.OPENSHIFT_HOSTNAME          MY_APP_NAME-greeting   8080
None
MY_APP_NAME-name       MY_APP_NAME-name-
MY_PROJECT_NAME.OPENSHIFT_HOSTNAME          MY_APP_NAME-name       8080
None
```

Pod のルート情報には、アクセスに使用するベース URL が提供されます。上記の例では、**http://MY_APP_NAME-greeting-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME** をベース URL として使用し、アプリケーションにアクセスします。

9.5.4. Circuit Breaker サンプルアプリケーションの Minishift または CDK へのデプロイメント

以下のいずれかのオプションを使用して、Minishift または CDK で Circuit Breaker サンプルアプリケーションをローカルで実行します。

- Fabric8 Launcher の使用
- CLI クライアント **oc** の使用

各メソッドは、同じ **oc** コマンドを使用してアプリケーションをデプロイしますが、Fabric8 Launcher を使用すると、**oc** コマンドを実行する自動デプロイメントワークフローが提供されます。

9.5.4.1. Fabric8 Launcher ツールの URL および認証情報の取得

Minishift または CDK にサンプルアプリケーションを作成してデプロイするには、Fabric8 Launcher ツール URL とユーザー認証情報が必要です。この情報は、Minishift または CDK の起動時に提供されます。

前提条件

- Fabric8 Launcher ツールがインストールされ、設定され、実行している。

手順

1. Minishift または CDK を起動したコンソールに移動します。
2. 実行中の Fabric8 Launcher にアクセスするのに使用できる URL およびユーザー認証情報のコンソール出力を確認します。

Minishift または CDK 起動時のコンソール出力の例

```
...
-- Removing temporary directory ... OK
-- Server Information ...
   OpenShift server started.
   The server is accessible via web console at:
      https://192.168.42.152:8443

   You are logged in as:
      User:     developer
      Password: developer

   To login as administrator:
      oc login -u system:admin
```

9.5.4.2. Fabric8 Launcher ツールを使用したサンプルアプリケーションのデプロイメント

このセクションでは、REST API Level 0 のサンプルアプリケーションをビルドし、Fabric8 Launcher Web インターフェイスから OpenShift にデプロイする方法を説明します。

前提条件

- 実行中の Fabric8 Launcher インスタンスの URL と、Minishift または CDK のユーザー認証情報。詳細は「Fabric8 Launcher ツールの URL および認証情報の取得」 を参照してください。

手順

1. ブラウザーで Fabric8 Launcher URL に移動します。

2. 画面の指示に従って、Spring Boot でサンプルアプリケーションを作成して起動します。

9.5.4.3. CLI クライアント oc の認証

oc コマンドラインクライアントを使用して Minishift または CDK でサンプルアプリケーションを使用するには、Minishift または CDK Web インターフェイスが提供するトークンを使用してクライアントを認証する必要があります。

前提条件

- 実行中の Fabric8 Launcher インスタンスの URL と、Minishift または CDK のユーザー認証情報。詳細は「Fabric8 Launcher ツールの URL および認証情報の取得」 を参照してください。

手順

1. ブラウザーで Minishift または CDK URL に移動します。
2. ユーザー名の横にある Web コンソールの右上にあるクエスチョンマークアイコンをクリックします。
3. ドロップダウンメニューで **Command Line Tools** を選択します。
4. **oc login** コマンドをコピーします。
5. 端末にコマンドを貼り付けます。このコマンドは、認証トークンを使用して、Minishift または CDK アカウントで CLI クライアント **oc** を認証します。

```
$ oc login OPENSHIFT_URL --token=MYTOKEN
```

9.5.4.4. CLI クライアント oc を使用した Circuit Breaker サンプルアプリケーションのデプロイメント

このセクションでは、Circuit Breaker サンプルアプリケーションをビルドし、コマンドラインから OpenShift にデプロイする方法を説明します。

前提条件

- Minishift または CDK の Fabric8 Launcher ツールを使用して作成されたサンプルアプリケーション。詳細は「Fabric8 Launcher ツールを使用したサンプルアプリケーションのデプロイメント」を参照してください。
- Fabric8 Launcher ツールの URL。
- 認証された **oc** クライアント。詳細は「CLI クライアント **oc** の認証」を参照してください。

手順

1. GitHub からプロジェクトのクローンを作成します。

```
$ git clone git@github.com:USERNAME/MY_PROJECT_NAME.git
```

または、プロジェクトの ZIP ファイルをダウンロードして、展開します。

```
$ unzip MY_PROJECT_NAME.zip
```

2. 新しい OpenShift プロジェクトを作成します。

```
$ oc new-project MY_PROJECT_NAME
```

3. アプリケーションの root ディレクトリーに移動します。

4. Maven を使用して OpenShift へのデプロイメントを開始します。

```
$ mvn clean fabric8:deploy -Popenshift
```

このコマンドは、Fabric8 Maven プラグインを使用して OpenShift で S2I プロセス を起動し、Pod を起動します。

5. アプリケーションのステータスを確認し、Pod が実行していることを確認します。

```
$ oc get pods -w
NAME                          READY     STATUS      RESTARTS   AGE
MY_APP_NAME-greeting-1-aaaaa     1/1       Running   0          17s
MY_APP_NAME-greeting-1-deploy    0/1       Completed 0          22s
MY_APP_NAME-name-1-aaaaa         1/1       Running   0          14s
MY_APP_NAME-name-1-deploy        0/1       Completed 0          28s
```

MY_APP_NAME-greeting-1-aaaaa Pod および **MY_APP_NAME-name-1-aaaaa** Pod の両方には、完全にデプロイおよび起動すると **Running** のステータスが必要になります。また、続行する前に Pod が準備状態になるまで待機する必要があります。これは **READY** 列に表示されます。たとえば、**MY_APP_NAME-greeting-1-aaaaa** は、**READY** 列が **1/1** の場合に準備状態になります。特定の Pod 名が異なります。中間の数字は新規ビルドごとに増えます。末尾の文字は、Pod の作成時に生成されます。

6. アプリケーションのサンプルをデプロイして起動すると、そのルートを決定します。

ルート情報の例

```
$ oc get routes
NAME                HOST/PORT                                        PATH      SERVICES
PORT      TERMINATION
MY_APP_NAME-greeting   MY_APP_NAME-greeting-
MY_PROJECT_NAME.OPENSHIFT_HOSTNAME          MY_APP_NAME-greeting   8080
None
MY_APP_NAME-name       MY_APP_NAME-name-
MY_PROJECT_NAME.OPENSHIFT_HOSTNAME          MY_APP_NAME-name       8080
None
```

Pod のルート情報には、アクセスに使用するベース URL が提供されます。上記の例では、**http://MY_APP_NAME-greeting-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME** をベース URL として使用し、アプリケーションにアクセスします。

9.5.5. Circuit Breaker サンプルアプリケーションの OpenShift Container Platform へのデプロイメント

サンプルアプリケーションを OpenShift Container Platform に作成し、デプロイするプロセスは OpenShift Online に似ています。

前提条件

- developers.redhat.com/launch を使用して作成されたサンプルアプリケーション。

手順

- 「Circuit Breaker サンプルアプリケーションの OpenShift Online へのデプロイメント」 の説明に従い、OpenShift Container Platform Web コンソールの URL およびユーザー認証情報のみを使用します。

9.5.6. 未変更の Spring Boot Circuit Breaker サンプルアプリケーションとの対話

Spring Boot のサンプルアプリケーションをデプロイした後に、以下のサービスが実行されます。

MY_APP_NAME-name

以下のエンドポイントを公開します。

- このサービスの稼働時に名前を返す **/api/name** エンドポイントと、このサービスが失敗を実証するように設定されたときにエラーが返されます。
- **/api/state** エンドポイント: **/api/name** エンドポイントの動作を制御し、サービスが正しく機能するか、または失敗を実証するかどうかを判断します。

MY_APP_NAME-greeting

以下のエンドポイントを公開します。

- パーソナライズされたグリーティング応答を取得するために呼び出す **/api/greeting** エンドポイント。
  /api/greeting エンドポイントを呼び出すと、要求の処理の一環として、**MY_APP_NAME-name** サービスの **/api/name** エンドポイントに対して呼び出しを実行します。**/api/name** エンドポイントに対する呼び出しは、Circuit Breaker によって保護されます。

  リモートエンドポイントが利用可能な場合、**name** サービスは HTTP コード **200** (**OK**) で応答し、**/api/greeting** エンドポイントから以下のグリーティングを受け取ります。

  ```
  {"content":"Hello, World!"}
  ```

  リモートエンドポイントが利用できない場合、**name** サービスは HTTP コード **500** (**Internal server error**) で応答し、**/api/greeting** エンドポイントから事前定義されたフォールバック応答を受け取ります。

  ```
  {"content":"Hello, Fallback!"}
  ```

- Circuit Breaker の状態を返す **/api/cb-state** エンドポイント。状態は次のいずれかです。
  - **open**: サーキットブレーカーにより、失敗したサービスに要求が到達できないようになります。
  - **closed**: サーキットブレーカーにより、要求がサービスに到達できるようになります。

以下の手順は、サービスの可用性を確認し、障害をシミュレートしてフォールバック応答を受け取る方法を示しています。

1. **curl** を使用して、**MY_APP_NAME-greeting** サービスに対して **GET** 要求を実行します。これを行うには、Web インターフェイスの **Invoke** ボタンを使用することもできます。

```
$ curl http://MY_APP_NAME-greeting-
MY_PROJECT_NAME.LOCAL_OPENSHIFT_HOSTNAME/api/greeting
{"content":"Hello, World!"}
```

2. **MY_APP_NAME-name** サービスの失敗をシミュレートするには、以下を行います。
   - Web インターフェイスで **トグル** ボタンを使用します。
   - **MY_APP_NAME-name** サービスを実行している Pod のレプリカ数を 0 にスケーリングします。
   - **MY_APP_NAME-name** サービスの **/api/state** エンドポイントに対して HTTP **PUT** 要求を実行し、その状態を **fail** に設定します。

     ```
     $ curl -X PUT -H "Content-Type: application/json" -d '{"state": "fail"}'
     http://MY_APP_NAME-name-
     MY_PROJECT_NAME.LOCAL_OPENSHIFT_HOSTNAME/api/state
     ```

3. **/api/greeting** エンドポイントを呼び出します。**/api/name** エンドポイントで複数の要求が失敗する場合:
   a. Circuit Breaker が開きます。
   b. Web インターフェイスの状態インジケーターが **CLOSED** から **OPEN** に変わります。
   c. **/api/greeting** エンドポイントを呼び出すと、Circuit Breaker はフォールバック応答を実行します。

      ```
      $ curl http://MY_APP_NAME-greeting-
      MY_PROJECT_NAME.LOCAL_OPENSHIFT_HOSTNAME/api/greeting
      {"content":"Hello, Fallback!"}
      ```

4. 名前 **MY_APP_NAME-name** サービスを、可用性に戻します。これを行うには、以下を行います。
   - Web インターフェイスで **トグル** ボタンを使用します。
   - **MY_APP_NAME-name** サービスを実行する Pod のレプリカ数を 1 にスケーリングします。
   - **MY_APP_NAME-name** サービスの **/api/state** エンドポイントに対して HTTP **PUT** 要求を実行し、その状態を **OK** に戻します。

     ```
     $ curl -X PUT -H "Content-Type: application/json" -d '{"state": "ok"}'
     http://MY_APP_NAME-name-
     MY_PROJECT_NAME.LOCAL_OPENSHIFT_HOSTNAME/api/state
     ```

5. **/api/greeting** エンドポイントを再度呼び出します。**/api/name** エンドポイントでの複数の要求が正常に実行される場合:
   a. Circuit Breaker が閉じます。
   b. Web インターフェイスの状態インジケーターが **OPEN** から **CLOSED** に変わります。
   c. Circuit Breaker は、**/api/greeting** エンドポイントを呼び出す際に **Hello World!** グリーティングを返します。

```
$ curl http://MY_APP_NAME-greeting-
MY_PROJECT_NAME.LOCAL_OPENSHIFT_HOSTNAME/api/greeting
{"content":"Hello, World!"}
```

9.5.7. Circuit Breaker サンプルアプリケーション統合テストの実行

このサンプルアプリケーションには、自己完結型の統合テストセットが含まれます。OpenShift プロジェクト内で実行する場合、テストは以下を行います。

- アプリケーションのテストインスタンスをプロジェクトにデプロイします。
- そのインスタンスで個別のテストを実行します。
- テストが完了したら、プロジェクトからアプリケーションのすべてのインスタンスを削除します。

警告

統合テストを実行すると、サンプルアプリケーションの既存インスタンスがすべて、ターゲット OpenShift プロジェクトから削除されます。サンプルアプリケーションが正しく削除されないようにするには、テストを実行するために別の OpenShift プロジェクトを作成して選択してください。

前提条件

- 認証された **oc** クライアント。
- 空の OpenShift プロジェクト。

手順

次のコマンドを実行して統合テストを実行します。

```
$ mvn clean verify -Popenshift,openshift-it
```

9.5.8. Hystrix Dashboard を使用したサーキットブレーカーの監視

Histrix Dashboard を使用すると、イベントストリームから Histrix メトリクスデータを集計し、それを 1 つの画面で表示することで、サービスの正常性を簡単にリアルタイムで監視できます。

前提条件

- アプリケーションがデプロイされている。

手順

1. Minishift または CDK クラスターにログインします。

```
$ oc login OPENSHIFT_URL --token=MYTOKEN
```

2. Web コンソールにアクセスするには、ブラウザーを使用して Minishift または CDK URL に移動します。

3. Circuit Breaker アプリケーションが含まれるプロジェクトに移動します。

```
$ oc project MY_PROJECT_NAME
```

4. Hystrix Dashboard アプリケーションの YAML テンプレート をインポートします。そのためには、**Add to Project** をクリックしてから **Import YAML / JSON** タブを選択し、YAML ファイルの内容をテキストボックスにコピーします。または、次のコマンドを実行できます。

```
$ oc create -f https://raw.githubusercontent.com/snowdrop/openshift-
templates/master/hystrix-dashboard/hystrix-dashboard.yml
```

5. **Create** ボタンをクリックして、テンプレートに基づいて Histrix Dashboard アプリケーションを作成します。または、次のコマンドを実行できます。

```
$ oc new-app --template=hystrix-dashboard
```

6. Hystrix Dashboard が含まれる Pod がデプロイされるまで待機します。

7. Hystrix Dashboard アプリケーションのルートを取得します。

```
$ oc get route hystrix-dashboard
NAME                HOST/PORT                                                  PATH      SERVICES
PORT      TERMINATION   WILDCARD
hystrix-dashboard   hystrix-dashboard-
MY_PROJECT_NAME.LOCAL_OPENSHIFT_HOSTNAME                  hystrix-dashboard
<all>                 None
```

8. Dashboard にアクセスするには、ブラウザーで Dashboard アプリケーションルート URL を開きます。Web コンソールの **Overview** 画面に移動し、Hystrix Dashboard アプリケーションが含まれる Pod の上にあるヘッダーのルート URL をクリックします。

9. Dashboard を使用して **MY_APP_NAME-greeting** サービスを監視するには、デフォルトのイベントストリームアドレスを以下のアドレスに置き換え、**Monitor Stream** ボタンをクリックします。

```
http://MY_APP_NAME-greeting-
MY_PROJECT_NAME.LOCAL_OPENSHIFT_HOSTNAME/hystrix.stream
```

関連情報

- Histrix Dashboard wiki ページ

9.5.9. サーキットブレーカーリソース

Circuit Breaker パターンの背後にある設計原則に関する背景情報については、以下のリンクを参照してください。

- microservices.io: Microservice Patterns: Circuit Breaker
- Martin Fowler: CircuitBreaker

- Eclipse Vert.x の Circuit Breaker
- Node.js の Circuit Breaker
- Thorntail の Circuit Breaker

9.6. SPRING BOOT のセキュアなサンプルアプリケーション

重要

以下の例は、実稼働環境での実行を目的としていません。

制限: このサンプルアプリケーションは、Minishift または CDK で実行してください。また、手動のワークフローを使用して、この例を OpenShift Online Pro および OpenShift Container Platform にデプロイすることもできます。この例は、現在 OpenShift Online Starter では使用できません。

注記

Spring Boot の Secured サンプルアプリケーションでは、Red Hat SSO 7.3 が必要です。Red Hat SSO 7.3 は IBM Z でサポートされていないため、IBM Z では Secured の例は利用できません。

上達度レベルの例: **Advanced**

Secured サンプルアプリケーションは、Red Hat SSO を使用して REST エンドポイントを保護します。この例では、REST API Level 0 の例に展開されます。

Red Hat SSO:

- OAuth 2.0 仕様のエクステンションである Open ID Connect プロトコルを実装します。
- アクセストークンを発行し、セキュアなリソースに対するさまざまなアクセス権限をクライアントに提供します。

SSO でアプリケーションをセキュアにすると、セキュリティー設定を一元化しつつ、アプリケーションにセキュリティーを追加することができます。

重要

この例では、デモ目的で Red Hat SSO が事前設定されたので、その原則、使用方法、または設定を説明しません。この例を使用する前に、Red Hat SSO に関する基本的な概念を理解していることを確認してください。

9.6.1. Secured プロジェクト構造

SSO のサンプルには以下が含まれます。

- 保護するつもりである Greeting サービスのソース
- SSO サーバーをデプロイするテンプレートファイル (**service.sso.yaml**)
- サービスのセキュリティーを保護するように Keycloak アダプターの設定

9.6.2. Red Hat SSO デプロイメントの設定

この例の **service.sso.yaml** ファイルには、事前設定された Red Hat SSO サーバーをデプロイするすべての OpenShift 設定項目が含まれます。SSO サーバーの設定は、このサンプルのために簡略化されており、ユーザーとセキュリティー設定が事前に設定された、すぐに使用できる設定を提供します。**service.sso.yaml** ファイルには非常に長い行が含まれ、gedit などの一部のテキストエディターでは、このファイルの読み取りに問題がある場合があります。

警告

実稼働環境では、この SSO 設定を使用することは推奨されません。具体的には、セキュリティー設定の例に追加された単純化は、実稼働環境での使用に影響を及ぼします。

表9.5 SSO サンプルの簡素化

変更点	理由	推奨事項
デフォルト設定には、**yaml 設定ファイルに公開鍵と秘密鍵** の両方が含まれます。	これを行ったのは、エンドユーザーが Red Hat SSO モジュールをデプロイして、内部や Red Hat SSO の設定方法を知らなくても使用可能な状態にすることができるためです。	実稼働環境では、秘密鍵をソース制御に保存しないでください。サーバー管理者が追加する必要があります。
設定済みの **クライアントがコールバック URL を受け入れます。**	各ランタイムにカスタム設定を使用しないように、OAuth2 仕様で必要なコールバックの検証を回避します。	アプリケーション固有のコールバック URL には、有効なドメイン名を指定する必要があります。
クライアントには SSL/TLS が必要ありません。また、セキュアなアプリケーションは HTTPS 上で公開されません。	この例は、ランタイムごとに証明書を生成する必要がないことで単純化されます。	実稼働環境では、セキュアなアプリケーションは単純な HTTP ではなく HTTPS を使用する必要があります。
トークンのタイムアウトがデフォルトの 1 分から 10 分に増えました。	コマンドラインの例を使用した場合のユーザーエクスペリエンスを向上します。	セキュリティーの観点から、攻撃者はアクセストークンが拡張されていると推測する必要があるウィンドウ。潜在的な攻撃者が現在のトークンを推測するのがより困難になるため、このウィンドウを短くすることが推奨されます。

9.6.3. Red Hat SSO レルムモデル

master レルムは、この例のセキュリティーを保護するために使用されます。コマンドラインクライアントとセキュアな REST エンドポイントのモデルを提供する事前設定されたアプリケーションクライアント定義は 2 つあります。

また、Red Hat SSO の **master** レルムには、**admin** および **alice** のさまざまな認証および認可の結果の検証に使用できる 2 つの事前設定されたユーザーも存在します。

9.6.3.1. Red Hat SSO ユーザー

セキュリティーが保護された例のレルムモデルには、ユーザーが 2 つ含まれます。

admin

admin ユーザーのパスワードは **admin** で、レルム管理者です。このユーザーは Red Hat SSO 管理コンソールに完全アクセスできますが、セキュアなエンドポイントへのアクセスに必要なロールマッピングはありません。このユーザーを使用して、認証されていて認可されていないユーザーの動作を説明できます。

alice

alice ユーザーには **password** のパスワードがあり、正規のアプリケーションユーザーです。このユーザーは、セキュアなエンドポイントへの認証および認可が成功したアクセスを実証します。ロールマッピングの例は、デコードされた JWT ベアラートークンにあります。

```
{
  "jti": "0073cfaa-7ed6-4326-ac07-c108d34b4f82",
  "exp": 1510162193,
  "nbf": 0,
  "iat": 1510161593,
  "iss": "https://secure-sso-sso.LOCAL_OPENSHIFT_HOSTNAME/auth/realms/master", (1)
  "aud": "demoapp",
  "sub": "c0175ccb-0892-4b31-829f-dda873815fe8",
  "typ": "Bearer",
  "azp": "demoapp",
  "nonce": "90ff5d1a-ba44-45ae-a413-50b08bf4a242",
  "auth_time": 1510161591,
  "session_state": "98efb95a-b355-43d1-996b-0abcb1304352",
  "acr": "1",
  "client_session": "5962112c-2b19-461e-8aac-84ab512d2a01",
  "allowed-origins": [
    "*"
  ],
  "realm_access": {
    "roles": [ (2)
      "example-admin"
    ]
  },
  "resource_access": { (3)
    "secured-example-endpoint": {
      "roles": [
        "example-admin" (4)
      ]
    },
    "account": {
      "roles": [
        "manage-account",
        "view-profile"
      ]
    }
  },
  "name": "Alice InChains",
```

```
  "preferred_username": "alice", 5
  "given_name": "Alice",
  "family_name": "InChains",
  "email": "alice@keycloak.org"
}
```

1. **iss** フィールドは、トークンを発行する Red Hat SSO レルムインスタンス URL に対応します。トークンを検証するには、セキュアなエンドポイントデプロイメントで設定する必要があります。

2. **roles** オブジェクトは、グローバルレルムレベルでユーザーに付与されたロールを提供します。この例では、**alice** には **example-admin** ロールが付与されています。セキュリティーが保護されたエンドポイントが認可されたロールのレルムレベルを検索していることを確認できます。

3. **resource_access** オブジェクトには、リソースの固有のロール付与が含まれます。このオブジェクトの下には、セキュアな各エンドポイントのオブジェクトを見つけます。

4. **resource_access.secured-example-endpoint.roles** オブジェクトには、**secured-example-endpoint** リソースの **alice** に付与されるロールが含まれます。

5. **preferred_username** フィールドは、アクセストークンの生成に使用したユーザー名を提供します。

9.6.3.2. アプリケーションクライアント

OAuth 2.0 仕様では、リソースの所有者の代わりにセキュアなリソースにアクセスするアプリケーションクライアントのロールを定義することができます。**master** レルムには、以下のアプリケーションクライアントが定義されています。

demoapp

これは、アクセストークンの取得に使用されるクライアントシークレットを持つ **confidential** タイプのクライアントです。トークンには **alice** ユーザーの権限が含まれ、**alice** が Thorntail、Eclipse Vert.x、Node.js、および Spring Boot ベースの REST サンプルアプリケーションデプロイメントにアクセスできるようにします。

secured-example-endpoint

secured-example-endpoint は、関連するリソース (特に Greeting サービス) にアクセスするために **example-admin** ロールを必要とするベアラーのみのクライアントタイプです。

9.6.4. Spring Boot SSO アダプターの設定

SSO アダプターは **クライアント側**、または SSO サーバーのクライアントで、Web リソースのセキュリティーを強制するコンポーネントです。ここでは、これは Greeting サービスです。

SSO アダプターとエンドポイントセキュリティーの両方が **src/main/resources/application.properties** で設定されます。

application.properties ファイルの例

```
$ # Adapter configuration
keycloak.realm=${realm:master} 1
keycloak.realm-key=...
```

```
keycloak.auth-server-url=${sso.auth.server.url} 2
keycloak.resource=${client.id:secured-example-endpoint} 3
keycloak.credentials.secret=${secret:1daa57a2-b60e-468b-a3ac-25bd2dc2eadc} 4
keycloak.use-resource-role-mappings=true 5
keycloak.bearer-only=true 6
# Endpoint security configuration
keycloak.securityConstraints[0].securityCollections[0].name=admin stuff 7
keycloak.securityConstraints[0].securityCollections[0].authRoles[0]=example-admin 8
keycloak.securityConstraints[0].securityCollections[0].patterns[0]=/api/greeting 9
```

1. 使用するセキュリティーレルム。
2. Red Hat SSO サーバーのアドレス (ビルド時の補間)。
3. 実際の keycloak **クライアント** の設定。
4. 認証サーバーにアクセスするためのシークレット
5. ユーザーのアプリケーションレベルのロールマッピングのトークンを確認します。
6. 有効な場合、アダプターはユーザーの認証を試行しませんが、ベアラートークンのみを検証します。
7. セキュリティー制約の単純名。
8. セキュアなエンドポイントへのアクセスに必要なロール。
9. セキュアなエンドポイントパスパターン。

9.6.5. Secured サンプルアプリケーションの Minishift または CDK へのデプロイメント

9.6.5.1. Fabric8 Launcher ツールの URL および認証情報の取得

Minishift または CDK にサンプルアプリケーションを作成してデプロイするには、Fabric8 Launcher ツール URL とユーザー認証情報が必要です。この情報は、Minishift または CDK の起動時に提供されます。

前提条件

- Fabric8 Launcher ツールがインストールされ、設定され、実行している。

手順

1. Minishift または CDK を起動したコンソールに移動します。
2. 実行中の Fabric8 Launcher にアクセスするのに使用できる URL およびユーザー認証情報のコンソール出力を確認します。

   Minishift または CDK 起動時のコンソール出力の例

   ```
   ...
   -- Removing temporary directory ... OK
   -- Server Information ...
   ```

```
OpenShift server started.
The server is accessible via web console at:
    https://192.168.42.152:8443

You are logged in as:
    User:     developer
    Password: developer

To login as administrator:
    oc login -u system:admin
```

9.6.5.2. Fabric8 Launcher を使用した Secured サンプルアプリケーションの作成

前提条件

- 実行中の Fabric8 Launcher インスタンスの URL およびユーザー認証情報。詳細は「Fabric8 Launcher ツールの URL および認証情報の取得」を参照してください。

手順

- ブラウザーで Fabric8 Launcher URL に移動します。
- 画面の指示に従って、Spring Boot にサンプルを作成します。デプロイメントタイプについて尋ねられたら、**ローカルで構築して実行します**。
- 画面の指示に従ってください。
  完了したら、**Download as ZIP file**ボタンをクリックし、ファイルをハードドライブに保存します。

9.6.5.3. CLI クライアント oc の認証

oc コマンドラインクライアントを使用して Minishift または CDK でサンプルアプリケーションを使用するには、Minishift または CDK Web インターフェイスが提供するトークンを使用してクライアントを認証する必要があります。

前提条件

- 実行中の Fabric8 Launcher インスタンスの URL と、Minishift または CDK のユーザー認証情報。詳細は「Fabric8 Launcher ツールの URL および認証情報の取得」 を参照してください。

手順

1. ブラウザーで Minishift または CDK URL に移動します。
2. ユーザー名の横にある Web コンソールの右上にあるクエスチョンマークアイコンをクリックします。
3. ドロップダウンメニューで **Command Line Tools** を選択します。
4. **oc login** コマンドをコピーします。
5. 端末にコマンドを貼り付けます。このコマンドは、認証トークンを使用して、Minishift または CDK アカウントで CLI クライアント **oc** を認証します。

```
$ oc login OPENSHIFT_URL --token=MYTOKEN
```

9.6.5.4. CLI クライアント oc を使用した Secured サンプルアプリケーションのデプロイメント

このセクションでは、Secured サンプルアプリケーションをビルドし、コマンドラインから OpenShift にデプロイする方法を説明します。

前提条件

- Minishift または CDK の Fabric8 Launcher ツールを使用して作成されたサンプルアプリケーション。詳細は「Fabric8 Launcher を使用した Secured サンプルアプリケーションの作成」 を参照してください。
- Fabric8 Launcher URL。
- 認証された **oc** クライアント。詳細は「CLI クライアント **oc** の認証」を参照してください。

手順

1. GitHub からプロジェクトのクローンを作成します。

   ```
   $ git clone git@github.com:USERNAME/MY_PROJECT_NAME.git
   ```

   または、プロジェクトの ZIP ファイルをダウンロードして、展開します。

   ```
   $ unzip MY_PROJECT_NAME.zip
   ```

2. 新しい OpenShift プロジェクトを作成します。

   ```
   $ oc new-project MY_PROJECT_NAME
   ```

3. アプリケーションの root ディレクトリーに移動します。

4. サンプルの ZIP ファイルから **service.sso.yaml** ファイルを使用して Red Hat SSO サーバーをデプロイします。

   ```
   $ oc create -f service.sso.yaml
   ```

5. Maven を使用して Minishift または CDK へのデプロイメントを開始します。

   ```
   $ mvn clean fabric8:deploy -Popenshift -DskipTests \
       -DSSO_AUTH_SERVER_URL=$(oc get route secure-sso -o jsonpath='{"https://"}
   {.spec.host}{"/auth\n"}')
   ```

   このコマンドは、Fabric8 Maven Plugin を使用して Minishift または CDK で S2I プロセス を起動し、Pod を起動します。

このプロセスは、uberjar ファイルと OpenShift リソースを生成し、それらを Minishift または CDK サーバーの現在のプロジェクトにデプロイします。

9.6.6. Secured サンプルアプリケーションの OpenShift Container Platform へのデプロイメント

Minishift または CDK の他に、マイナーな相違点のみが OpenShift Container Platform にサンプルを作成し、デプロイできます。最も重要な相違点は、OpenShift Container Platform でデプロイする前に、Minishift または CDK にサンプルアプリケーションを作成する必要があります。

前提条件

- Minishift または CDK を使用して作成された例

9.6.6.1. CLI クライアント oc の認証

コマンドラインクライアント **oc** を使用して OpenShift Container Platform のサンプルアプリケーションを使用するには、OpenShift Container Platform Web インターフェイスによって提供されるトークンを使用してクライアントを認証する必要があります。

前提条件

- Red Hat OpenShift Container Platform のアカウント

手順

1. ブラウザーで OpenShift Container Platform URL に移動します。
2. ユーザー名の横にある Web コンソールの右上にあるクエスチョンマークアイコンをクリックします。
3. ドロップダウンメニューで **Command Line Tools** を選択します。
4. **oc login** コマンドをコピーします。
5. 端末にコマンドを貼り付けます。このコマンドは、認証トークンを使用して CLI クライアント **oc** を OpenShift Container Platform アカウントで認証します。

```
$ oc login OPENSHIFT_URL --token=MYTOKEN
```

9.6.6.2. CLI クライアント oc を使用した Secured サンプルアプリケーションのデプロイメント

このセクションでは、Secured サンプルアプリケーションをビルドし、コマンドラインから OpenShift にデプロイする方法を説明します。

前提条件

- Minishift または CDK の Fabric8 Launcher ツールを使用して作成されたサンプルアプリケーション。
- 認証された **oc** クライアント。詳細は「CLI クライアント **oc** の認証」を参照してください。

手順

1. GitHub からプロジェクトのクローンを作成します。

```
$ git clone git@github.com:USERNAME/MY_PROJECT_NAME.git
```

または、プロジェクトの ZIP ファイルをダウンロードして、展開します。

```
$ unzip MY_PROJECT_NAME.zip
```

2. 新しい OpenShift プロジェクトを作成します。

```
$ oc new-project MY_PROJECT_NAME
```

3. アプリケーションの root ディレクトリーに移動します。

4. サンプルの ZIP ファイルから **service.sso.yaml** ファイルを使用して Red Hat SSO サーバーをデプロイします。

```
$ oc create -f service.sso.yaml
```

5. Maven を使用して OpenShift Container Platform へのデプロイメントを開始します。

```
$ mvn clean fabric8:deploy -Popenshift -DskipTests \
    -DSSO_AUTH_SERVER_URL=$(oc get route secure-sso -o jsonpath='{"https://"}
{.spec.host}{"/auth\n"}')
```

このコマンドは、Fabric8 Maven プラグインを使用して OpenShift Container Platform で S2I プロセス を起動し、Pod を起動します。

このプロセスは、uberjar ファイルと OpenShift リソースを生成し、それらを OpenShift Container Platform サーバーの現在のプロジェクトにデプロイします。

9.6.7. Secured サンプルアプリケーション API エンドポイントへの認証

Secured サンプルアプリケーションは、呼び出し元が認証および認可されている場合に **GET** 要求を受け入れるデフォルトの HTTP エンドポイントを提供します。クライアントは最初に Red Hat SSO サーバーに対して認証を行い、認証手順によって返されるアクセストークンを使用して Secured サンプルアプリケーションに対して **GET** 要求を実行します。

9.6.7.1. Secured サンプルアプリケーション API エンドポイントの取得

クライアントを使用して例を操作する場合は、**PROJECT_ID** サービスである Secured サンプルアプリケーションのエンドポイントを指定する必要があります。

前提条件

- Secured サンプルアプリケーションがデプロイされ、実行します。
- 認証された **oc** クライアント。

手順

1. 端末アプリケーションで、**oc get routes** コマンドを実行します。
以下の表の出力例を以下に示します。

例9.1 Secured エンドポイントの一覧

名前	ホスト/ポート	パス	サービス	ポート	終了
secure-sso	secure-sso-myproject.LOCAL_OPENSHIFT_HOSTNAME		secure-sso	(すべて)	passthrough
PROJECT_ID	PROJECT_ID-myproject.LOCAL_OPENSHIFT_HOSTNAME		PROJECT_ID	(すべて)	
sso	sso-myproject.LOCAL_OPENSHIFT_HOSTNAME		sso	(すべて)	

上記の例では、サンプルエンドポイントは **http://PROJECT_ID-myproject.LOCAL_OPENSHIFT_HOSTNAME** になります。**PROJECT_ID** は、developers.redhat.com/launch または Fabric8 Launcher ツールを使用してサンプルを生成する際に入力した名前に基づいています。

9.6.7.2. コマンドラインを使用した HTTP 要求の認証

HTTP POST 要求を Red Hat SSO サーバーに送信してトークンを要求します。以下の例では、CLI ツール jq を使用して JSON 応答からトークン値を抽出します。

前提条件

- セキュアなサンプルエンドポイント URL。詳細は「Secured サンプルアプリケーション API エンドポイントの取得」を参照してください。
- **jq** コマンドラインツール (任意)。ツールのダウンロードと、詳細な情報は、https://stedolan.github.io/jq/ を参照してください。

手順

1. **curl**、認証情報、**<SSO_AUTH_SERVER_URL>** でアクセストークンを要求し、**jq** コマンドを使用して応答からトークンを展開します。

```
curl -sk -X POST https://<SSO_AUTH_SERVER_URL>/auth/realms/master/protocol/openid-connect/token \
  -d grant_type=password \
  -d username=alice\
  -d password=password \
  -d client_id=demoapp \
```

```
  -d client_secret=1daa57a2-b60e-468b-a3ac-25bd2dc2eadc \
  | jq -r '.access_token'

eyJhbGciOiJSUzI1NiIsInR5cCIgOiAiSldUIiwia2lkIiA6ICJRek1nbXhZMUhrQnpxTnR0SnkwMm5j
NTNtMGNiWDQxV1hNSTU1MFo4MGVBIn0.eyJqdGkiOiI0NDA3YTliNC04YWRhLTRlMTctOD
Q2ZS03YjI5MjMyN2RmYTIiLCJleHAiOjE1MDc3OTM3ODcsIm5iZiI6MCwiaWF0IjoxNTA3Nzkz
NzI3LCJpc3MiOiJodHRwczovL3NlY3VyZS1zc28tc3NvLWRlbW8uYXBwcy5jYWZlLWJhYmUub
3JnL2F1dGgvcmVhbG1zL21hc3RlciIsImF1ZCI6ImRlbW9hcHAiLCJzdWIiOiJjMDE3NWNjYi0w
ODkyLTRiMzEtODI5Zi1kZGE4NzM4MTVmZTgiLCJ0eXAiOiJCZWFyZXIiLCJhenAiOiJkZW1vY
XBwIiwiYXV0aF90aW1lIjowLCJzZXNzaW9uX3N0YXRlIjoiMDFjOTkzNGQtNmZmOS00NWYzL
WJkNWUtMTU4NDI5ZDZjNDczIiwiYWNyIjoiMSIsImNsaWVudF9zZXNzaW9uIjoiMzM3Yzk0MT
YtYTdlZS00ZWUzLThjZWQtODhlODI0MGJjNTAyIiwiYWxsb3dlZC1vcmlnaW5zIjpbIioiXSwicmV
hbG1fYWNjZXNzIjp7InJvbGVzIjpbImJvb3N0ZXItYWRtaW4iXX0sInJlc291cmNlX2FjY2VzcyI6ey
JzZWN1cmVkLWJvb3N0ZXItZW5kcG9pbnQiOnsicm9sZXMiOlsiYm9vc3Rlci1hZG1pbiJdfSwiY
WNjb3VudCI6eyJyb2xlcyI6WyJtYW5hZ2UtYWNjb3VudCIsInZpZXctcHJvZmlsZSJdfX0sIm5hbW
UiOiJBbGljZSBJbkNoYWlucyIsInByZWZlcnJlZF91c2VybmFtZSI6ImFsaWNlIiwiZ2l2ZW5fbmFtZ
SI6IkFsaWNlIiwiZmFtaWx5X25hbWUiOiJJbkNoYWlucyIsImVtYWlsIjoiYWxpY2VAa2V5Y2xvYW
sub3JnIn0.mjmZe37enHpigJv0BGuIitOj-
kfMLPNwYzNd3n0Ax4Nga7KpnfytGyuPSvR4KAG8rzkfBNN9klPYdy7pJEeYlfmnFUkM4EDrZY
gn4qZAznP1Wzy1RfVRdUFi0-
GqFTMPb37o5HRldZZ09QljX_j3GHnoMGXRtYW9RZN4eKkYkcz9hRwgfJoTy2CuwFqeJwZY
UyXifrfA-JoTr0UmSUed-0NMksGrtJjjPggUGS-
qOn6OgKcmN2vaVAQlxW32y53JqUXctfLQ6DhJzIMYTmOfIIPy0sgG1mG7sovQhw1xTg0vTjdx
8zQ-EJcexkj7IivRevRZsslKgqRFWs67jQAFQA
```

<SSO_AUTH_SERVER_URL> は **secure-sso** サービスの URL です。

通常、**username**、**password**、**client_secret** などの属性は、通常、秘密になりますが、上記のコマンドはこの例で提供されているデフォルトの認証情報をデモの目的で使用します。

jq を使用してトークンを抽出しない場合は、**curl** コマンドのみを実行し、アクセストークンを手動で抽出できます。

注記

-sk オプションは curl に対し、自己署名証明書の失敗を無視します。実稼働環境ではこのオプションを使用しないでください。macOS では、**curl** バージョン **7.56.1** 以降がインストールされている必要があります。また、OpenSSL でビルドする必要があります。

1. Secured サービスを起動します。アクセス (ベアラー) トークンを HTTP ヘッダーに割り当てます。

```
$ curl -v -H "Authorization: Bearer <TOKEN>" http://<SERVICE_HOST>/api/greeting

{
   "content": "Hello, World!",
   "id": 2
}
```

例9.2 Access (Bearer) トークンが含まれる GET 要求ヘッダーのサンプル

```
> GET /api/greeting HTTP/1.1
> Host: <SERVICE_HOST>
> User-Agent: curl/7.51.0
```

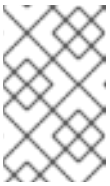
```
> Accept: */*
> Authorization: Bearer <TOKEN>
```

<SERVICE_HOST> は、セキュアなサンプルエンドポイントの URL です。詳細は「Secured サンプルアプリケーション API エンドポイントの取得」を参照してください。

2. アクセストークンの署名を確認します。
アクセストークンは JSON Web トークン であるため、JWT デバッガー を使用してデコードできます。

   a. Web ブラウザーで、JWT デバッガー の Web サイトに移動します。

   b. **Algorithm** ドロップダウンメニューから **RS256** を選択します。

   注記

   選択後に Web フォームが更新され、正しい RSASHA256(...) 情報が Signature セクションに表示されることを確認します。そうでない場合は、HS256 に切り替えてから RS256 に戻ります。

   c. 上部のテキストボックスの **VERIFY SIGNATURE** セクションに以下のコンテンツを貼り付けます。

   ```
   -----BEGIN PUBLIC KEY-----
   MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAoETnPmN55xBJjRzN/cs30OzJ
   9olkteLVNRjzdTxFOyRtS2ovDfzdhhO9XzUcTMbIsCOAZtSt8K+6yvBXypOSYvI75EUdypm
   kcK1KoptqY5KEBQ1KwhWuP7IWQ0fshUwD6jI1QWDfGxfM/h34FvEn/0tJ71xN2P8TI2Yan
   wuDZgosdobx/PAvlGREBGuk4BgmexTOkAdnFxIUQcCkiEZ2C41uCrxiS4CEe5OX91aK9
   HKZV4ZJX6vnqMHmdDnsMdO+UFtxOBYZio+a1jP4W3d7J5fGeiOaXjQCOpivKnP2yU2D
   PdWmDMyVb67l8DRA+jh0OJFKZ5H2fNgE3II59vdsRwIDAQAB
   -----END PUBLIC KEY-----
   ```

   注記

   これは、Secured サンプルアプリケーションの Red Hat SSO サーバーデプロイメントからのマスターレルム公開鍵です。

   d. クライアント出力から **Encoded** されたボックスに **token** 出力を貼り付けます。
   Signature Verified 記号がデバッガーページに表示されます。

9.6.7.3. Web インターフェイスを使用した HTTP 要求の認証

セキュアなエンドポイントには、HTTP API の他に、対話する Web インターフェイスも含まれます。

以下の手順は、セキュリティーの実施方法、認証方法、および認証トークンの使用方法を確認するための演習です。

前提条件

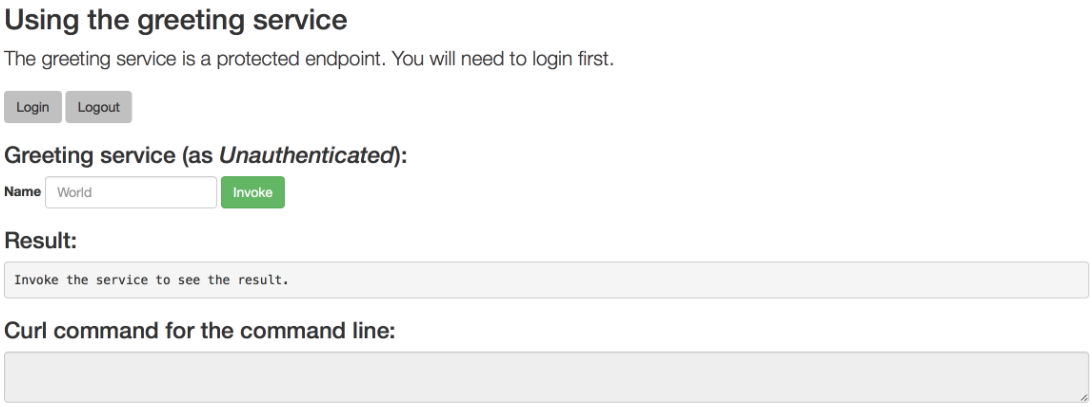
- セキュアなエンドポイント URL。詳細は「Secured サンプルアプリケーション API エンドポイントの取得」を参照してください。

手順

手順

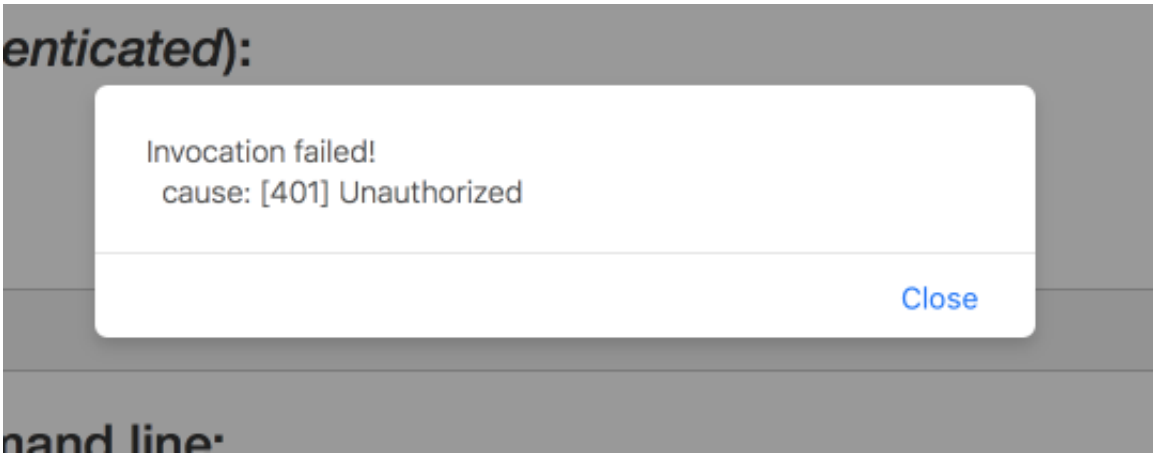
1. Web ブラウザーで、エンドポイント URL に移動します。
2. 認証されていない要求を実行します。
   a. **Invoke** ボタンをクリックします。

図9.1 認証されていないセキュアなサンプル Web インターフェイス

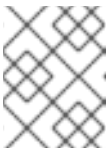


サービスは、**HTTP 401 Unauthorized** ステータスコードで応答します。

図9.2 未認証のエラーメッセージ



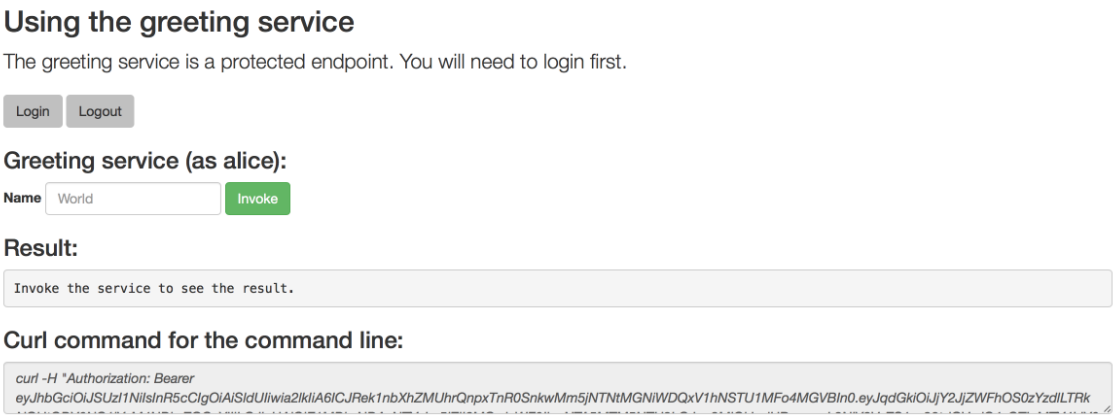
3. 認証された要求をユーザーとして実行します。
   a. **Login** ボタンをクリックして Red Hat SSO に対して認証を行います。SSO サーバーにリダイレクトされます。
   b. Alice ユーザー としてログインします。Web インターフェイスにリダイレクトされます。



注記

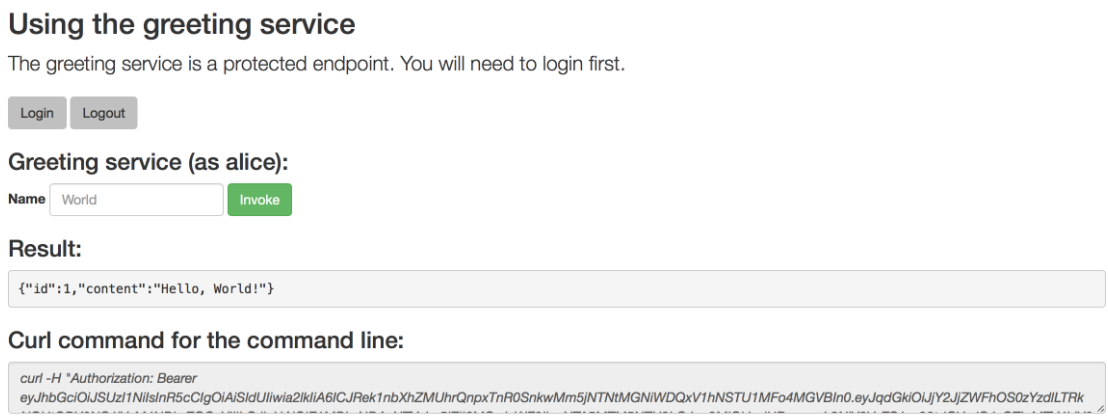
ページ下部のコマンドライン出力に、アクセス (ベアラー) トークンが表示されます。

図9.3 (Alice として) 認証されたセキュアなサンプル Web インターフェイス



c. 再度 **Invoke** をクリックして Greeting サービスにアクセスします。
例外がなく、JSON 応答ペイロードが表示されていることを確認します。つまり、サービスはアクセス (ベアラー) トークンを許可し、Greeting サービスへのアクセスが認可されていることを意味します。

図9.4 (Alice として) 認証されたグリーティング要求の結果



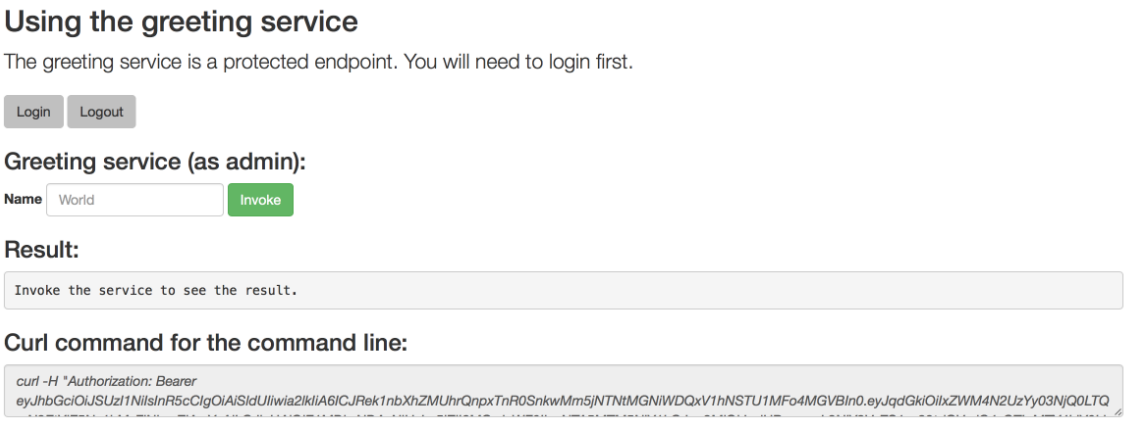
d. ログアウトします。

4. 認証された要求を管理者として実行します。

a. **Invoke** ボタンをクリックします。
これにより、認証されていない要求が Greeting サービスに送信されていることを確認します。

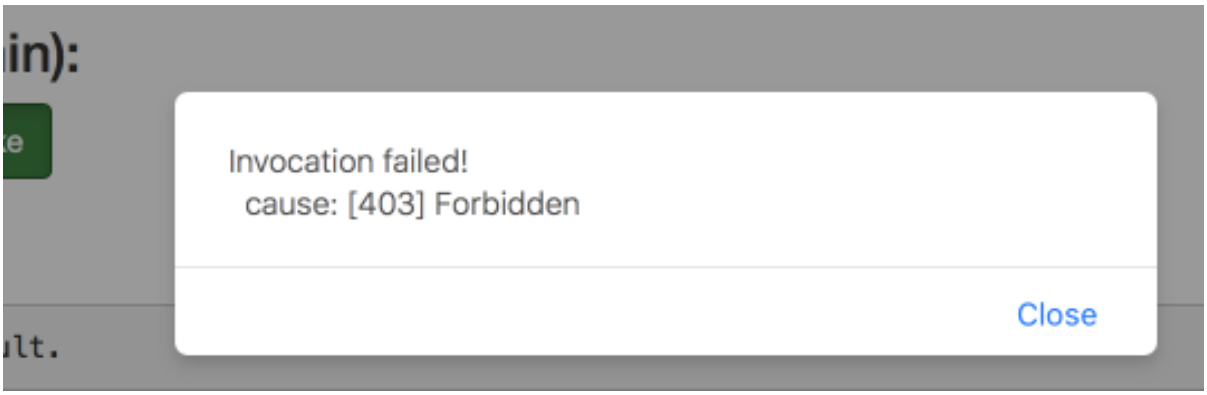
b. **Login** ボタンをクリックして admin ユーザー としてログインします。

図9.5 (admin として) 認証されたセキュアな Web インターフェイスの例



5. **Invoke** ボタンをクリックします。
   このサービスは、**admin** ユーザーが Greeting サービスへのアクセスが許可されていないため、**HTTP 403 Forbidden** ステータスコードで応答します。

   図9.6 承認されていないエラーメッセージ



9.6.8. Spring Boot Secured サンプルアプリケーション統合テストの実行

ここでは、認証された oc クライアントによって提供される Red Hat SSO サーバーを使用して統合テストを実行する方法を説明します。

前提条件

- 認証された **oc** クライアント。

手順

警告

統合テストを実行すると、サンプルアプリケーションの既存インスタンスがすべて、ターゲット OpenShift プロジェクトから削除されます。サンプルアプリケーションを誤って削除しないようにするには、テストを実行するために別の OpenShift プロジェクトを作成して選択してください。

1. 端末アプリケーションで、プロジェクトのディレクトリーに移動します。
2. Red Hat SSO サーバーアプリケーションを作成します。

```
oc create -f service.sso.yaml
```

3. Red Hat SSO サーバーの準備ができるまで待ちます。Web コンソールに移動するか、または **oc get pods** の出力を表示して、Red Hat SSO サーバーを実行する Pod が準備状態にあるかどうかを確認します。
4. 統合テストを実行します。

```
mvn clean verify -Popenshift,openshift-it -DSSO_AUTH_SERVER_URL=$(oc get route
secure-sso -o jsonpath='{"https://"}{.spec.host}{"/auth\n"}')
```

9.6.9. セキュアな SSO リソース

OAuth2 仕様の背後にある原則に関する追加情報や、Red Hat SSO および Keycloak を使用してアプリケーションのセキュリティーを保護する方法については、以下のリンクを参照してください。

- Aaron Parecki: OAuth2 Simplified
- Red Hat SSO 7.1 ドキュメント
- Keycloak 3.2 のドキュメント
- Eclipse Vert.x におけるセキュリティー保護
- Thorntail におけるセキュリティー保護
- Node.js におけるセキュリティー保護

9.7. SPRING BOOT のキャッシュの例

重要

以下の例は、実稼働環境での実行を目的としていません。

制限: このサンプルアプリケーションは、Minishift または CDK で実行してください。また、手動のワークフローを使用して、この例を OpenShift Online Pro および OpenShift Container Platform にデプロイすることもできます。この例は、現在 OpenShift Online Starter では使用できません。

上達度レベルの例: **Advanced**

キャッシュサンプルは、キャッシュを使用してアプリケーションの応答時間を長くする方法を示しています。

この例では、以下の方法を示しています。

- キャッシュを OpenShift にデプロイします。
- アプリケーション内でキャッシュを使用します。

9.7.1. キャッシュの仕組みおよび必要なタイミング

キャッシュを使用すると、特定の期間情報を保存し、アクセスすることができます。元のサービスを繰り返し呼び出すよりも、キャッシュの情報により迅速またはより確実にアクセスすることができます。キャッシュを使用する欠点は、キャッシュされた情報が最新ではないことです。ただし、キャッシュに保存されている各値に **有効期限** または TTL (time to live) に設定すると、この問題を軽減できます。

> **例9.3 キャッシュの例**
>
> service1 および service2 の 2 つのアプリケーションがあるとします。
>
> - Service1 は service2 からの値によって異なります。
>   - service2 からの値が頻繁に変更されると、service1 は一定期間 service2 からの値をキャッシュする可能性があります。
>   - キャッシュされた値を使用すると、service2 が呼び出される回数を減らすことができます。
> - service1 が service2 から直接値を取得するのに 500 ミリ秒かかり、キャッシュされた値を取得するのに 100 ミリ秒かかる場合、service1 はキャッシュされた各呼び出しに対してキャッシュされた値を使用することで 400 ミリ秒短くすることができます。
> - service1 が service2 にキャッシュされていない呼び出しを 1 秒あたり 5 回、10 秒以上行うとすると、呼び出しは 50 になります。
> - 代わりに service1 が 1 秒の TTL でキャッシュされた値の使用を開始した場合は、10 秒で 10 コールに削減されます。

キャッシュサンプルの仕組み

1. cache サービス、cute name サービス、および greeting サービスがデプロイされ、公開されます。
2. ユーザーは greeting サービスの Web フロントエンドにアクセスします。
3. ユーザーは、Web フロントエンドのボタンを使用して greeting HTTP API を呼び出します。
4. greeting サービスは、cute name サービスの値によって異なります。
   - greeting サービスは、最初にその値が cache サービスに保存されているかどうかを確認します。これが存在する場合は、キャッシュされた値が返されます。
   - 値がキャッシュされていない場合、greeting サービスは cute name サービスを呼び出し、値を返し、その値を 5 秒の TTL で cache サービスに保存します。
5. Web フロントエンドは、greeting サービスからの応答と、操作の合計時間を表示します。
6. ユーザーはサービスを複数回呼び出し、キャッシュされた操作とキャッシュされていない操作の違いを確認します。
   - キャッシュされた操作は、キャッシュされていない操作よりもはるかに高速です。
   - ユーザーは、TTL の有効期限が切れる前にキャッシュを強制的に消去することができます。

9.7.2. キャッシュサンプルアプリケーションの OpenShift Online へのデプロイ

以下のオプションのいずれかを使用して、OpenShift Online で Cache サンプルアプリケーションを実行します。

- developers.redhat.com/launch の使用
- CLI クライアント **oc** の使用

各メソッドは、同じ **oc** コマンドを使用してアプリケーションをデプロイしますが、developers.redhat.com/launch を使用すると、**oc** コマンドを実行する自動デプロイメントワークフローが提供されます。

9.7.2.1. developers.redhat.com/launch を使用したサンプルアプリケーションのデプロイメント

このセクションでは、REST API Level 0 のサンプルアプリケーションをビルドし、Red Hat Developer Launcher Web インターフェイスから OpenShift にデプロイする方法を説明します。

前提条件

- OpenShift Online のアカウント。

手順

1. ブラウザーで developers.redhat.com/launch URL に移動します。
2. 画面の指示に従って、Spring Boot でサンプルアプリケーションを作成して起動します。

9.7.2.2. CLI クライアント oc の認証

oc コマンドラインクライアントを使用して OpenShift Online でアプリケーションのサンプルを使用するには、OpenShift Online の Web インターフェイスによって提供されるトークンを使用してクライアントを認証する必要があります。

前提条件

- OpenShift Online のアカウント。

手順

1. ブラウザーで OpenShift Online URL に移動します。
2. ユーザー名の横にある Web コンソールの右上にあるクエスチョンマークアイコンをクリックします。
3. ドロップダウンメニューで **Command Line Tools** を選択します。
4. **oc login** コマンドをコピーします。
5. 端末にコマンドを貼り付けます。このコマンドは、認証トークンを使用して CLI クライアント **oc** を OpenShift Online アカウントで認証します。

```
$ oc login OPENSHIFT_URL --token=MYTOKEN
```

9.7.2.3. CLI クライアント oc を使用したキャッシュサンプルアプリケーションのデプロイメント

このセクションでは、キャッシュサンプルアプリケーションをビルドし、コマンドラインから OpenShift にデプロイする方法を説明します。

前提条件

- developers.redhat.com/launch を使用して作成されたサンプルアプリケーション。詳細は「developers.redhat.com/launch を使用したサンプルアプリケーションのデプロイメント」を参照してください。
- 認証された **oc** クライアント。詳細は「CLI クライアント **oc** の認証」を参照してください。

手順

1. GitHub からプロジェクトのクローンを作成します。

```
$ git clone git@github.com:USERNAME/MY_PROJECT_NAME.git
```

   または、プロジェクトの ZIP ファイルをダウンロードして、展開します。

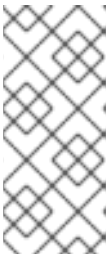
```
$ unzip MY_PROJECT_NAME.zip
```

2. 新規プロジェクトを作成します。

```
$ oc new-project MY_PROJECT_NAME
```

3. アプリケーションの root ディレクトリーに移動します。
4. キャッシュサービスをデプロイします。

```
$ oc apply -f service.cache.yml
```



注記

x86_64 以外のアーキテクチャーを使用している場合は、YAML ファイルで Red Hat Data Grid のイメージ名を、そのアーキテクチャー内の関連するイメージ名に更新します。たとえば、s390x アーキテクチャーの場合は、イメージ名を IBM Z イメージ名 **registry.access.redhat.com/jboss-datagrid-7/datagrid73-openj9-11-openshift-rhel8** に更新します。

5. Maven を使用して OpenShift へのデプロイメントを開始します。

```
$ mvn clean fabric8:deploy -Popenshift
```

6. アプリケーションのステータスを確認し、Pod が実行していることを確認します。

```
$ oc get pods -w
NAME                          READY     STATUS      RESTARTS   AGE
cache-server-123456789-aaaaa          1/1       Running     0          8m
MY_APP_NAME-cutename-1-bbbbb      1/1       Running     0          4m
```

```
MY_APP_NAME-cutename-s2i-1-build   0/1       Completed   0          7m
MY_APP_NAME-greeting-1-ccccc       1/1       Running     0          3m
MY_APP_NAME-greeting-s2i-1-build   0/1       Completed   0          3m
```

3 つの Pod が完全にデプロイされて起動すると、ステータスは **Running** でなければなりません。

7. アプリケーションのサンプルをデプロイして起動すると、そのルートを決定します。

ルート情報の例

```
$ oc get routes
NAME                HOST/PORT                                        PATH      SERVICES
PORT      TERMINATION
MY_APP_NAME-cutename   MY_APP_NAME-cutename-
MY_PROJECT_NAME.OPENSHIFT_HOSTNAME            MY_APP_NAME-cutename   8080
None
MY_APP_NAME-greeting   MY_APP_NAME-greeting-
MY_PROJECT_NAME.OPENSHIFT_HOSTNAME            MY_APP_NAME-greeting   8080
None
```

Pod のルート情報には、アクセスに使用するベース URL が提供されます。上記の例では、**http://MY_APP_NAME-greeting-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME** をベース URL として使用して greeting サービスにアクセスします。

9.7.3. Cache サンプルアプリケーションの Minishift または CDK へのデプロイ

以下のオプションのいずれかを使用して、Minishift または CDK でキャッシュサンプルアプリケーションをローカルで実行します。

- Fabric8 Launcher の使用
- CLI クライアント **oc** の使用

各メソッドは、同じ **oc** コマンドを使用してアプリケーションをデプロイしますが、Fabric8 Launcher を使用すると、**oc** コマンドを実行する自動デプロイメントワークフローが提供されます。

9.7.3.1. Fabric8 Launcher ツールの URL および認証情報の取得

Minishift または CDK にサンプルアプリケーションを作成してデプロイするには、Fabric8 Launcher ツール URL とユーザー認証情報が必要です。この情報は、Minishift または CDK の起動時に提供されます。

前提条件

- Fabric8 Launcher ツールがインストールされ、設定され、実行している。

手順

1. Minishift または CDK を起動したコンソールに移動します。
2. 実行中の Fabric8 Launcher にアクセスするのに使用できる URL およびユーザー認証情報のコンソール出力を確認します。

Minishift または CDK 起動時のコンソール出力の例

```
...
-- Removing temporary directory ... OK
-- Server Information ...
   OpenShift server started.
   The server is accessible via web console at:
      https://192.168.42.152:8443

   You are logged in as:
      User:     developer
      Password: developer

   To login as administrator:
      oc login -u system:admin
```

9.7.3.2. Fabric8 Launcher ツールを使用したサンプルアプリケーションのデプロイメント

このセクションでは、REST API Level 0 のサンプルアプリケーションをビルドし、Fabric8 Launcher Web インターフェイスから OpenShift にデプロイする方法を説明します。

前提条件

- 実行中の Fabric8 Launcher インスタンスの URL と、Minishift または CDK のユーザー認証情報。詳細は「Fabric8 Launcher ツールの URL および認証情報の取得」 を参照してください。

手順

1. ブラウザーで Fabric8 Launcher URL に移動します。
2. 画面の指示に従って、Spring Boot でサンプルアプリケーションを作成して起動します。

9.7.3.3. CLI クライアント oc の認証

oc コマンドラインクライアントを使用して Minishift または CDK でサンプルアプリケーションを使用するには、Minishift または CDK Web インターフェイスが提供するトークンを使用してクライアントを認証する必要があります。

前提条件

- 実行中の Fabric8 Launcher インスタンスの URL と、Minishift または CDK のユーザー認証情報。詳細は「Fabric8 Launcher ツールの URL および認証情報の取得」 を参照してください。

手順

1. ブラウザーで Minishift または CDK URL に移動します。
2. ユーザー名の横にある Web コンソールの右上にあるクエスチョンマークアイコンをクリックします。
3. ドロップダウンメニューで **Command Line Tools** を選択します。
4. **oc login** コマンドをコピーします。
5. 端末にコマンドを貼り付けます。このコマンドは、認証トークンを使用して、Minishift または CDK アカウントで CLI クライアント **oc** を認証します。

```
$ oc login OPENSHIFT_URL --token=MYTOKEN
```

9.7.3.4. CLI クライアント oc を使用したキャッシュサンプルアプリケーションのデプロイメント

このセクションでは、キャッシュサンプルアプリケーションをビルドし、コマンドラインから OpenShift にデプロイする方法を説明します。

前提条件

- Minishift または CDK の Fabric8 Launcher ツールを使用して作成されたサンプルアプリケーション。詳細は「Fabric8 Launcher ツールを使用したサンプルアプリケーションのデプロイメント」を参照してください。
- Fabric8 Launcher ツールの URL。
- 認証された **oc** クライアント。詳細は「CLI クライアント **oc** の認証」を参照してください。

手順

1. GitHub からプロジェクトのクローンを作成します。

```
$ git clone git@github.com:USERNAME/MY_PROJECT_NAME.git
```

   または、プロジェクトの ZIP ファイルをダウンロードして、展開します。

```
$ unzip MY_PROJECT_NAME.zip
```

2. 新規プロジェクトを作成します。

```
$ oc new-project MY_PROJECT_NAME
```

3. アプリケーションの root ディレクトリーに移動します。
4. キャッシュサービスをデプロイします。

```
$ oc apply -f service.cache.yml
```

注記

x86_64 以外のアーキテクチャーを使用している場合は、YAML ファイルで Red Hat Data Grid のイメージ名を、そのアーキテクチャー内の関連するイメージ名に更新します。たとえば、s390x アーキテクチャーの場合は、イメージ名を IBM Z イメージ名 **registry.access.redhat.com/jboss-datagrid-7/datagrid73-openj9-11-openshift-rhel8** に更新します。

5. Maven を使用して OpenShift へのデプロイメントを開始します。

```
$ mvn clean fabric8:deploy -Popenshift
```

6. アプリケーションのステータスを確認し、Pod が実行していることを確認します。

```
$ oc get pods -w
NAME                          READY     STATUS      RESTARTS   AGE
cache-server-123456789-aaaaa            1/1       Running     0          8m
MY_APP_NAME-cutename-1-bbbbb       1/1       Running     0          4m
MY_APP_NAME-cutename-s2i-1-build   0/1       Completed   0          7m
MY_APP_NAME-greeting-1-ccccc       1/1       Running     0          3m
MY_APP_NAME-greeting-s2i-1-build   0/1       Completed   0          3m
```

3 つの Pod が完全にデプロイされて起動すると、ステータスは **Running** でなければなりません。

7. アプリケーションのサンプルをデプロイして起動すると、そのルートを決定します。

ルート情報の例

```
$ oc get routes
NAME                HOST/PORT                                        PATH      SERVICES
PORT      TERMINATION
MY_APP_NAME-cutename    MY_APP_NAME-cutename-
MY_PROJECT_NAME.OPENSHIFT_HOSTNAME            MY_APP_NAME-cutename    8080
None
MY_APP_NAME-greeting    MY_APP_NAME-greeting-
MY_PROJECT_NAME.OPENSHIFT_HOSTNAME            MY_APP_NAME-greeting    8080
None
```

Pod のルート情報には、アクセスに使用するベース URL が提供されます。上記の例では、**http://MY_APP_NAME-greeting-MY_PROJECT_NAME.OPENSHIFT_HOSTNAME** をベース URL として使用して greeting サービスにアクセスします。

9.7.4. キャッシュサンプルアプリケーションの OpenShift Container Platform へのデプロイメント

サンプルアプリケーションを OpenShift Container Platform に作成し、デプロイするプロセスは OpenShift Online に似ています。

前提条件

- developers.redhat.com/launch を使用して作成されたサンプルアプリケーション。

手順

- 「キャッシュサンプルアプリケーションの OpenShift Online へのデプロイ」 の説明に従い、OpenShift Container Platform Web コンソールの URL およびユーザー認証情報のみを使用します。

9.7.5. 未変更の Cache サンプルアプリケーションとの対話

デフォルトの Web インターフェイスを使用して、未変更の Cache サンプルアプリケーションを操作し、頻繁にアクセスされるデータの保存方法により、サービスへのアクセスに必要な時間を短縮できます。

前提条件

- アプリケーションがデプロイされている。

手順

1. ブラウザーを使用して **greeting** サービスに移動します。
2. **サービスの起動** を一度クリックします。
   duration の値が **2000** を超えることに注意してください。また、キャッシュ状態が **No cached value** から **A value is cached** になっていることに注意してください。
3. 5 秒間待機して、キャッシュ状態が **No cached value** に戻されるのを確認します。
   キャッシュされた値の TTL は 5 秒に設定されています。TTL の期限が切れると、値はキャッシュされなくなります。
4. **Invoke the service** を一度クリックして、値をキャッシュします。
5. キャッシュ状態が **A value is cached** になっている数秒の間に **Invoke the service** をさらに数回クリックします。
   キャッシュされた値を使用しているため、**duration** の値が大幅に低いことに注意してください。**Clear the cache** をクリックすると、キャッシュは空になります。

9.7.6. キャッシュサンプルアプリケーション統合テストの実行

このサンプルアプリケーションには、自己完結型の統合テストセットが含まれます。OpenShift プロジェクト内で実行する場合、テストは以下を行います。

- アプリケーションのテストインスタンスをプロジェクトにデプロイします。
- そのインスタンスで個別のテストを実行します。
- テストが完了したら、プロジェクトからアプリケーションのすべてのインスタンスを削除します。

警告

統合テストを実行すると、サンプルアプリケーションの既存インスタンスがすべて、ターゲット OpenShift プロジェクトから削除されます。サンプルアプリケーションが正しく削除されないようにするには、テストを実行するために別の OpenShift プロジェクトを作成して選択してください。

前提条件

- 認証された **oc** クライアント。
- 空の OpenShift プロジェクト。

手順

次のコマンドを実行して統合テストを実行します。

```
$ mvn clean verify -Popenshift,openshift-it
```

9.7.7. リソースのキャッシュ

キャッシュに関する背景および関連情報は、以下を参照してください。

- Eclipse Vert.x のキャッシュ
- Thorntail のキャッシュ
- Node.js のキャッシュ

付録A SOURCE-TO-IMAGE (S2I) ビルドプロセス

Source-to-Image (S2I) は、アプリケーションソースのあるオンライン SCM リポジトリーから再現可能な Docker 形式のコンテナーイメージを生成するビルドツールです。S2I ビルドを使用すると、ビルド時間を短縮し、リソースとネットワークの使用量を減らし、セキュリティーを改善し、その他の多くの利点を活用して、アプリケーションの最新バージョンを本番環境に簡単に配信できます。OpenShift は、複数の ビルドストラテジーおよび入力ソース をサポートします。

詳細は、OpenShift Container Platform ドキュメントの Source-to-Image (S2I) ビルド の章を参照してください。

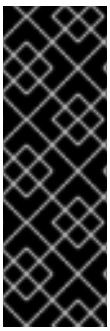
最終的なコンテナーイメージをアセンブルするには、S2I プロセスに 3 つの要素を提供する必要があります。

- GitHub などのオンライン SCM リポジトリーでホストされるアプリケーションソース。
- S2I Builder イメージ。アセンブルされたイメージの基盤となり、アプリケーションが実行しているエコシステムを提供します。
- 必要に応じて、S2I スクリプト によって使用される環境変数およびパラメーターを指定することもできます。

このプロセスは、S2I スクリプトで指定された指示に従ってアプリケーションソースと依存関係を Builder イメージに挿入し、アセンブルされたアプリケーションを実行する Docker 形式のコンテナーイメージを生成します。詳細は、OpenShift Container Platform ドキュメントの S2I build requirements、build options、および how builds work を参照してください。

付録B サンプルアプリケーションのデプロイメント設定の更新

サンプルアプリケーションのデプロイメント設定には、ルート情報や Readiness プローブの場所など、OpenShift でのアプリケーションのデプロイおよび実行に関連する情報が含まれます。サンプルアプリケーションのデプロイメント設定は YAML ファイルのセットに保存されます。Fabric8 Maven プラグインを使用する例では、YAML ファイルは **src/main/fabric8/** ディレクトリーにあります。Nodeshift を使用する例として、YAML ファイルは **.nodeshift** ディレクトリーにあります。



重要

Fabric8 Maven Plugin および Nodeshift が使用するデプロイメント設定ファイルは完全な OpenShift リソース定義である必要はありません。Fabric8 Maven Plugin と Nodeshift の両方がデプロイメント設定ファイルを取り、不足している情報を追加して完全な OpenShift リソース定義を作成できます。Fabric8 Maven Plugin によって生成されるリソース定義は **target/classes/META-INF/fabric8/** ディレクトリーにあります。Nodeshift によって生成されるリソース定義は **tmp/nodeshift/resource/** ディレクトリーにあります。

前提条件

- 既存のサンプルプロジェクト。
- CLI クライアント **oc** がインストールされている。

手順

1. 既存の YAML ファイルを編集したり、設定を更新して追加の YAML ファイルを作成します。
   - たとえば、サンプルに **readinessProbe** が設定された YAML ファイルがすでにある場合は、**path** の値を別の利用可能なパスに変更し、Readiness の有無を確認することができます。

```
spec:
  template:
    spec:
      containers:
        readinessProbe:
          httpGet:
            path: /path/to/probe
            port: 8080
            scheme: HTTP
...
```

   - **readinessProbe** が既存の YAML ファイルで設定されていない場合は、**readinessProbe** 設定を使用して新規 YAML ファイルを同じディレクトリーに作成することもできます。
2. Maven または npm を使用して、サンプルの更新バージョンをデプロイします。
3. 設定更新が、デプロイ済みのサンプルに表示されることを確認します。

```
$ oc export all --as-template='my-template'

apiVersion: template.openshift.io/v1
kind: Template
metadata:
```

```
  creationTimestamp: null
  name: my-template
objects:
- apiVersion: template.openshift.io/v1
  kind: DeploymentConfig
  ...
  spec:
    ...
    template:
      ...
      spec:
        containers:
          ...
          livenessProbe:
            failureThreshold: 3
            httpGet:
              path: /path/to/different/probe
              port: 8080
              scheme: HTTP
            initialDelaySeconds: 60
            periodSeconds: 30
            successThreshold: 1
            timeoutSeconds: 1
          ...
```

追加リソース

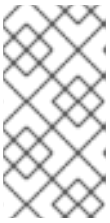
Web ベースのコンソール、または CLI クライアント **oc** を使用してアプリケーションの設定を直接更新した場合は、これらの変更を YAML ファイルへエクスポートして追加します。**oc export all** コマンドを使用して、デプロイされたアプリケーションの設定を表示します。

付録C FABRIC8 MAVEN PLUGIN でアプリケーションをデプロイする JENKINS フリースタイルプロジェクトの設定

ローカルホストの Maven および Fabric8 Maven Plugin を使用してアプリケーションをデプロイするのと同様に、Jenkins を設定して Maven および Fabric8 Maven プラグインを使用してアプリケーションをデプロイすることができます。

前提条件

- OpenShift クラスターへのアクセス。
- 同じ OpenShift クラスターで実行している Jenkins コンテナーイメージ。
- Jenkins サーバーに JDK および Maven がインストールされ、設定されている。
- Maven (**pom.xml** の Fabric8 Maven Plugin) を使用し、RHEL ベースイメージを使用してビルドするように設定されたアプリケーション。



注記

アプリケーションを OpenShift にビルドおよびデプロイする場合、Spring Boot 2.2 は OpenJDK 8 および OpenJDK 11 をベースとしたビルダーイメージのみをサポートします。Oracle JDK および OpenJDK 9 のビルダーイメージはサポートされていません。

pom.xml の例

```
<properties>
  ...
  <fabric8.generator.from>registry.access.redhat.com/redhat-openjdk-18/openjdk18-
openshift:latest</fabric8.generator.from>
</properties>
```

- GitHub で利用可能なアプリケーションのソース。

手順

1. アプリケーションの新規 OpenShift プロジェクトを作成します。
   a. OpenShift Web コンソールを開き、ログインします。
   b. **Create Project** をクリックし、新規 OpenShift プロジェクトを作成します。
   c. プロジェクト情報を入力し、**Create** をクリックします。
2. Jenkins がそのプロジェクトにアクセスできるようにします。
   たとえば、Jenkins のサービスアカウントを設定した場合は、アカウントに、アプリケーションのプロジェクトへの **edit** アクセスがあることを確認してください。
3. Jenkins サーバーで新しい フリースタイルの Jenkins プロジェクト を作成します。
   a. **New Item** をクリックします。
   b. 名前を入力し、**Freestyle プロジェクト** を選択して **OK** をクリックします。

c. **Source Code Management** で **Git** を選択し、アプリケーションの GitHub URL を追加します。

d. **Build** で **Add build step** を選択し、**Invoke top-level Maven target** を選択します。

e. 以下を **Goals** に追加します。

```
clean fabric8:deploy -Popenshift -Dfabric8.namespace=MY_PROJECT
```

MY_PROJECT をアプリケーションの OpenShift プロジェクトの名前に置き換えます。

f. **Save** をクリックします。

4. Jenkins プロジェクトのメインページから **Build Now** をクリックし、アプリケーションの OpenShift プロジェクトへのアプリケーションのビルドおよびデプロイを確認します。アプリケーションの OpenShift プロジェクトでルートを開いて、アプリケーションがデプロイされていることを確認することもできます。

次のステップ

- GITSCM ポーリング を追加すること、または the **Poll SCM** ビルドトリガー を使用することを検討してください。これらのオプションにより、新規コミットが GitHub リポジトリーにプッシュされるたびにビルドを実行できます。
- デプロイ前にテストを実行するビルドステップを追加することを検討してください。

付録D WAR ファイルを使用した SPRING BOOT アプリケーションのデプロイ

ファット JAR ファイルを使用してサポートされているアプリケーションのパッケージ化とデプロイのワークフローの代わりに、Spring Boot アプリケーションを WAR (Web アプリケーションアーカイブ) ファイルとしてパッケージ化してデプロイできます。アプリケーションが OpenShift で正しくビルドし、デプロイするように、ビルドおよびデプロイメントの設定を行う必要があります。

前提条件

- サンプル などの Spring Boot アプリケーション。
- アプリケーションを OpenShift にデプロイするのに使用される Fabric8 Maven Plugin。
- アプリケーションのパッケージ化に使用される Spring Boot Maven プラグイン。

手順

1. **war** パッケージをプロジェクトの **pom.xml** ファイルに追加します。

   pom.xml の例

   ```
   <project ...>
     ...
     <packaging>war</packaging>
     ...
   </project>
   ```

2. **spring-boot-starter-tomcat** をアプリケーションの依存関係として指定します。

   pom.xml の例

   ```
   <project ...>
     ...
     <dependencies>
       ...
       <dependency>
         <groupId>org.springframework.boot</groupId>
         <artifactId>spring-boot-starter-tomcat</artifactId>
       </dependency>
       ...
     </dependencies>
     ...
   </project>
   ```

3. Spring Boot Maven プラグインの Maven ゴール **repackage** が **pom.xml** ファイルで定義されているようにしてください。

   pom.xml の例

   ```
   <project ...>
   ...
     <build>
   ```

```
    ...
    <plugins>
      ...
      <plugin>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-maven-plugin</artifactId>
        <executions>
          <execution>
            <goals>
              <goal>repackage</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
...
</project>
```

これにより、アプリケーションの起動に使用される Spring Boot クラスが WAR ファイルに含まれ、これらのクラスの対応するプロパティーが WAR ファイルの **MANIFEST.mf** ファイルで定義されているようになります。

- **Main-Class: org.springframework.boot.loader.WarLauncher**
- **Spring-Boot-Classes: WEB-INF/classes/**
- **Spring-Boot-Lib: WEB-INF/lib/**
- **Spring-Boot-Version: 2.2.11**

4. **ARTIFACT_COPY_ARGS** 環境変数を **pom.xml** ファイルに追加します。
Fabric8 Maven プラグインは、ビルドプロセス中にこの変数を使用し、**Build および Deploy** ツールは WAR ファイル (デフォルトのファット JAR ファイルよりも) を使用してアプリケーションコンテナーイメージを作成します。

pom.xml の例

```
    ...
    <profile>
      <id>openshift</id>
      <build>
        <plugins>
          <plugin>
            <groupId>io.fabric8</groupId>
            <artifactId>fabric8-maven-plugin</artifactId>
            <executions>
              ...
            </executions>
            <configuration>
              <images>
                <image>
                  <name>${project.artifactId}:%t</name>
                  <alias>${project.artifactId}</alias>
                  <build>
                    <from>registry.access.redhat.com/redhat-openjdk-18/openjdk18-
```

```
openshift:${openjdk18-openshift.version}</from>
                    <assembly>
                        <basedir>/deployments</basedir>
                        <descriptorRef>artifact</descriptorRef>
                    </assembly>
                    <env>
                        <ARTIFACT_COPY_ARGS>*.war</ARTIFACT_COPY_ARGS>
                        <JAVA_APP_DIR>/deployments</JAVA_APP_DIR>
                    </env>
                    <ports>
                        <port>8080</port>
                    </ports>
                </build>
            </image>
        </images>
    </configuration>
  </plugin>
 </plugins>
</build>
</profile>
...
```

5. **JAVA_APP_JAR** 環境変数を **src/main/fabric8/deployment.yml** ファイルに追加します。この変数は Fabric8 Maven Plugin に対して、コンテナーに含まれる WAR ファイルを使用してアプリケーションを起動します。**src/main/fabric8/deployment.yml** が存在しない場合は、これを作成できます。

   deployment.yml の例

```
spec:
 template:
  spec:
   containers:
    ...
     env:
     - name: JAVA_APP_JAR
       value: ${project.artifactId}-${project.version}.war
```

6. アプリケーションをビルドしてデプロイします。

```
mvn clean fabric8:deploy -Popenshift
```

付録E 追加の SPRING BOOT リソース

- OpenShift Architecture Overview
- Spring Boot Microservices On Red Hat OpenShift Container Platform 3
- Spring Cloud Kubernetes
- Spring Boot プロジェクト
- Spring Framework プロジェクト
- OpenShift Spring Boot Lab Microservices
- Fabric8 を使用した Spring Boot アプリケーションの作成
- Fabric8 Maven Plugin

付録F アプリケーション開発リソース

OpenShift でのアプリケーション開発に関する詳細は、以下を参照してください。

- OpenShift インタラクティブラーニングポータル

ネットワークの負荷を削減し、アプリケーションのビルド時間を短縮するには、Minishift または CDK に Maven の Nexus ミラーを設定します。

- Maven 用の Nexus ミラーリングの設定

付録G 上達度レベル

使用できる各例は、特定の最小知識を必要とする概念について言及しています。この要件は例によって異なります。最小要件と概念は、いくつかの上達度レベルで設定されています。ここで説明するレベルの他に、各例に固有の追加情報が必要になる場合があります。

Foundational

Foundational とされている例は、通常、主題に関する事前の知識を必要としません。この例では、主要な要素、概念、および用語の一般的な認識とデモンストレーションを提供します。この例の説明で直接言及されているものを除き、特別な要件はありません。

Advanced

Advanced サンプルを使用する場合は、Kubernetes および OpenShift の他に、例の主題の領域の一般的な概念および用語について理解していることを前提としています。サービスやアプリケーションの設定、ネットワークの管理など、独自の基本的なタスクを実行できるようにする必要があります。この例ではサービスが必要で、設定が例の範囲に含まれていない場合には、適切に設定する知識があり、サービスの結果として生じる状態のみがドキュメントに記載されていることを前提とします。

Expert

Expert サンプルでは、主題について最も高いレベルの知識が必要です。機能ベースのドキュメントとマニュアルに基づいて多くのタスクを実行することが期待されており、ドキュメントは最も複雑なシナリオを対象としています。

付録H 用語

H.1. 製品およびプロジェクト名

Developer Launcher (developers.redhat.com/launch)

Developer Launcher (developers.redhat.com/launch) は、Red Hat が提供するスタンドアロンの入門エクスペリエンスです。これは、OpenShift でのクラウドネイティブ開発の開始に役立ちます。これには、OpenShift にダウンロード、ビルド、およびデプロイできる機能のサンプルアプリケーションが含まれます。

Minishift または CDK

Minishift を使用してマシンで実行している OpenShift クラスター。

H.2. DEVELOPER LAUNCHER に固有の用語

例

アプリケーション仕様 (例: **REST API を持つ Web サービス**)。
この例では、通常、実行する言語やプラットフォームを指定していません。説明には意図した機能のみが含まれています。

サンプルアプリケーション

特定の ランタイム における特定の サンプル の言語固有の実装。サンプルアプリケーションは、サンプルカタログ に記載されています。
たとえば、サンプルアプリケーションは Thorntail ランタイムを使用して実装される REST API を持つ Web サービスです。

サンプルカタログ

サンプルアプリケーションに関する情報が含まれる Git リポジトリー。

ランタイム

アプリケーションの例 を実行するプラットフォーム。たとえば、Thorntail または Eclipse Vert.x などです。