Red Hat Fuse 7.9

Spring Boot へのデプロイ

スタンドアロンモードでの Spring Boot アプリケーションのビルドおよび実行

Last Updated: 2022-02-19

Red Hat Fuse 7.9 Spring Boot へのデプロイ

スタンドアロンモードでの Spring Boot アプリケーションのビルドおよび実行

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、Jar ファイルとしてパッケージ化され、JVM で直接実行 (スタンドアロンモード) される Spring Boot アプリケーションをビルドする方法について説明します。

目次

多様性を受け入れるオープンソースの強化

Red Hat では、コード、ドキュメント、Web プロパティーにおける配慮に欠ける用語の置き換えに取り組んでいます。まずは、マスター (master)、スレーブ (slave)、ブラックリスト (blacklist)、ホワイトリスト (whitelist) の 4 つの用語の置き換えから始めます。この取り組みは膨大な作業を要するため、今後の複数のリリースで段階的に用語の置き換えを実施して参ります。詳細は、CTO である Chris Wright のメッセージ をご覧ください。

第1章 スタンドアロンの SPRING BOOT の使用

1.1. SPRING BOOT スタンドアロンデプロイメントモード

スタンドアロンデプロイメントモードでは、Spring Boot アプリケーションは Jar ファイルとしてパッケージ化され、Java 仮想マシン (JVM) 内で直接実行されます。アプリケーションをパッケージ化および実行するこの方法は、サービスが最小限の要件でパッケージ化されるマイクロサービスの概念と一致しています。Spring Boot アプリケーションは、**java** コマンドに **-jar** オプションを指定すると接実行できます。以下は例になります。

```
java -jar SpringBootApplication.jar
```

Spring Boot は、実行可能な Jar のメインクラスを提供します。Fuse で Spring Boot スタンドアロンアプリケーションをビルドするには、以下が必要です。

- **Fuse BOM (Bill of Materials)** Red Hat Maven リポジトリーから、厳選された依存関係のセットを定義します。BOM は、Maven の **依存関係管理** メカニズムを利用して、適切なバージョンの Maven 依存関係を定義します。
  注記: Fuse BOM で定義された依存関係のみが Red Hat によってサポートされます。

- **Spring Boot Maven プラグイン**- Maven でスタンドアロン Spring Boot アプリケーションのビルドプロセスを実装します。このプラグインは、Spring Boot アプリケーションを実行可能な Jar ファイルとしてパッケージ化します。

1.2. SPRING BOOT 2 へのデプロイ

スタンドアロンデプロイメントモードでは、Spring Boot 2 にデプロイするオプションがあります。

注記

OpenShift のデプロイメントのモードに関する詳細は『Fuse on OpenShift ガイド』を参照してください。

1.3. SPRING BOOT 2 の新しい CAMEL コンポーネント

Spring Boot 2 は Camel バージョン **2.23** をサポートし、以下に記載されている新しい Camel コンポーネントをサポートします。

Spring Boot 2 の新しい Camel コンポーネント

- as2-component
- aws-iam-component
- fhir-component
- google-calendar-stream-component
- google-mail-stream-component
- google-sheets-component
- google-sheets-stream-component

- ipfs-component
- kubernetes-hpa-component
- kubernetes-job-component
- micrometer-component
- mybatis-bean-component
- nsq-component
- rxjava2
- service-component
- spring-cloud-consul
- spring-cloud-zookeeper
- testcontainers-spring
- testcontainers
- web3j-component

第2章 FUSE ブースターの使用

Red Hat Fuse では、Fuse アプリケーションや便利なコンポーネントを使用するために、以下のブースターが提供されます。

- 「サーキットブレーカーブースターのビルドと実行」 - ネットワーク接続の中断やバックエンドサービスの一時的な使用停止に対応するために分散アプリケーションを有効にする例。
- 「Externalized Configuration ブースターのビルドおよび実行」 - Apache Camel ルートの設定を外部化する方法の例。
- 「REST API ブースターのビルドおよび実行」 - HTTP プロトコルを使用して、Apache Camel によって公開されるリモートサービスと対話する仕組みを紹介する例。

ブースターのデモンストレーションをビルドおよび実行するための前提条件として、以下をインストールします。

- サポートされるバージョンの Java Developer Kit (JDK)。詳細は「Red Hat Fuse でサポートされる構成」を参照してください。
- Apache Maven 3.3.x 以上。Maven の Download ページを参照してください。

2.1. ブースタープロジェクトの生成

Fuse ブースタープロジェクトは、スタンドアロンアプリケーションの実行を手助けする開発者向けのプロジェクトです。ここでは、ブースタープロジェクトの 1 つである Circuit Breaker ブースターの生成手順を説明します。この演習では、Fuse on Spring Boot の便利なコンポーネントを使用します。

Netflix/Hystrix サーキットブレーカーは、ネットワーク接続の中断や、バックエンドサービスの一時的な利用停止に分散アプリケーションが対応できるようにします。サーキットブレーカーパターンの基本概念は、バックエンドサービスが一時的に利用できなくなった場合に、依存するサービスの損失が自動的に検出され、代替動作をプログラムで作成できることです。

Fuse サーキットブローカーブースターは 2 つの関連サービスで構成されます。

- 呼び名を返すバックエンドサービスである **name** サービス。
- 名前を取得するよう **name** サービスを呼び出し、文字列 **Hello, NAME** を返すフロントエンドサービスである **greetings** サービス。

このブースターデモンストレーションでは、Hystrix サーキットブレーカーは **greetings** サービスと **name** サービスとの間に挿入されます。バックエンドの **name** サービスが利用できなくなると、**name** サービスが再起動するまでの間に **greetings** サービスはブロックされず、**greetings** サービスは代替動作にフォールバックして即座にクライアントに応答します。

前提条件

- Red Hat Developer Platform にアクセスできる必要があります。
- サポートされるバージョンの Java Developer Kit (JDK) が必要です。詳細は「Red Hat Fuse でサポートされる構成」を参照してください。
- 「Setting up Maven locally」の手順に従って、Apache Maven 3.3.x 以降をインストールおよび設定している必要があります。

手順

1. https://developers.redhat.com/launch に移動します。
2. **START** をクリックします。
   ランチャーウィザードによって、Red Hat アカウントにログインするよう要求されます。
3. **Launcher** ページで **Deploy an Example Application**ボタンをクリックします。
4. **Create Example Application** ページで **Create Example Application as**フィールドに名前 **fuse-circuit-breaker** を入力します。
5. **Select an Example** をクリックします。
6. **Example** ダイアログで、**Circuit Breaker** オプションを選択します。**Select a Runtime** ドロップダウンメニューが表示されます。
   a. **Select a Runtime** ドロップダウンメニューで **Fuse** を選択します。
   b. バージョンのドロップダウンメニューで **7.9(Red Hat Fuse)** を選択します。2.21. **2(Community)** バージョンは選択しないでください。
   c. **保存** をクリックします。
7. **Create Example Application** ページで **Download** をクリックします。
8. **Your Application is Ready** ダイアログが表示されたら、**Download.zip** をクリックします。ブラウザーが生成されたブースタープロジェクト (ZIP ファイルとしてパッケージ) をダウンロードします。
9. アーカイブユーティリティーを使用して、生成されたプロジェクトをローカルファイルシステムの任意の場所に展開します。

2.2. サーキットブレーカーブースターのビルドと実行

Netflix/Hystrix サーキットブレーカーコンポーネントは、ネットワーク接続の中断や、バックエンドサービスの一時的な利用停止に分散アプリケーションが対応できるようにします。サーキットブレーカーパターンの基本概念は、バックエンドサービスが一時的に利用できなくなった場合に、依存するサービスの損失が自動的に検出され、代替動作をプログラムで作成できることです。

Fuse サーキットブローカーブースターは 2 つの関連サービスで構成されます。

- 対象の名前を返す **name サービス**。
- 名前を取得するために name サービスを呼び出し、文字列 **Hello, NAME** を返す **greetings サービス**。

このデモンストレーションでは、Hystrix サーキットブレーカーは greetings サービスと name サービスとの間に挿入されます。name サービスが利用できなくなると、greetings サービスは name サービスが再起動するまでの間にブロックまたはタイムアウトせずに、代替動作にフォールバックして即座にクライアントに応答することができます。

前提条件

- 「ブースタープロジェクトの生成」 に記載されている手順を完了している必要があります。

手順

以下の手順に従って、サーキットブレーカーブースタープロジェクトをビルドおよび実行します。

1. シェルプロンプトを開き、Maven を使用してコマンドラインからプロジェクトをビルドします。

```
cd PROJECT_DIR
mvn clean package
```

2. 新しいシェルプロンプトを開き、以下のように name サービスを起動します。

```
cd name-service
mvn spring-boot:run -DskipTests -Dserver.port=8081
```

Spring Boot が起動すると、以下のような出力が表示されます。

```
...
2017-12-08 15:44:24.223  INFO 22758 --- [           main]
o.a.camel.spring.SpringCamelContext      : Total 1 routes, of which 1 are started
2017-12-08 15:44:24.227  INFO 22758 --- [           main]
o.a.camel.spring.SpringCamelContext      : Apache Camel 2.20.0 (CamelContext: camel-1)
started in 0.776 seconds
2017-12-08 15:44:24.234  INFO 22758 --- [           main]
org.jboss.fuse.boosters.cb.Application   : Started Application in 4.137 seconds (JVM running
for 4.744)
```

3. 新しいシェルプロンプトを開き、以下のように greetings サービスを起動します。

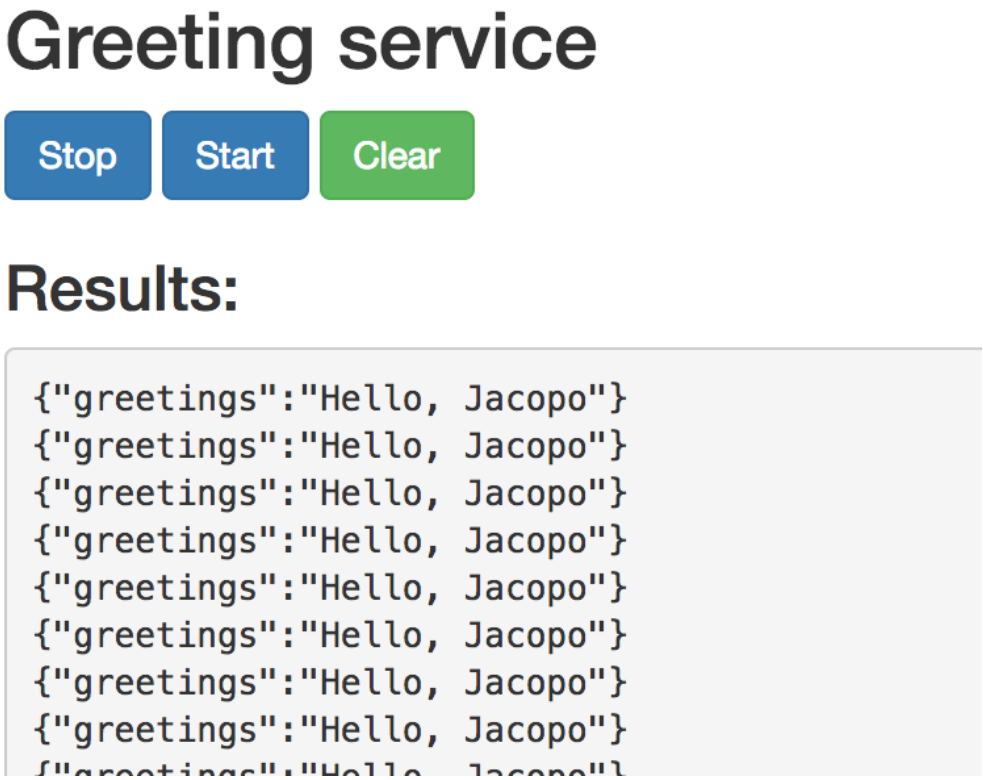
```
cd greetings-service
mvn spring-boot:run -DskipTests
```

Spring Boot が起動すると、以下のような出力が表示されます。

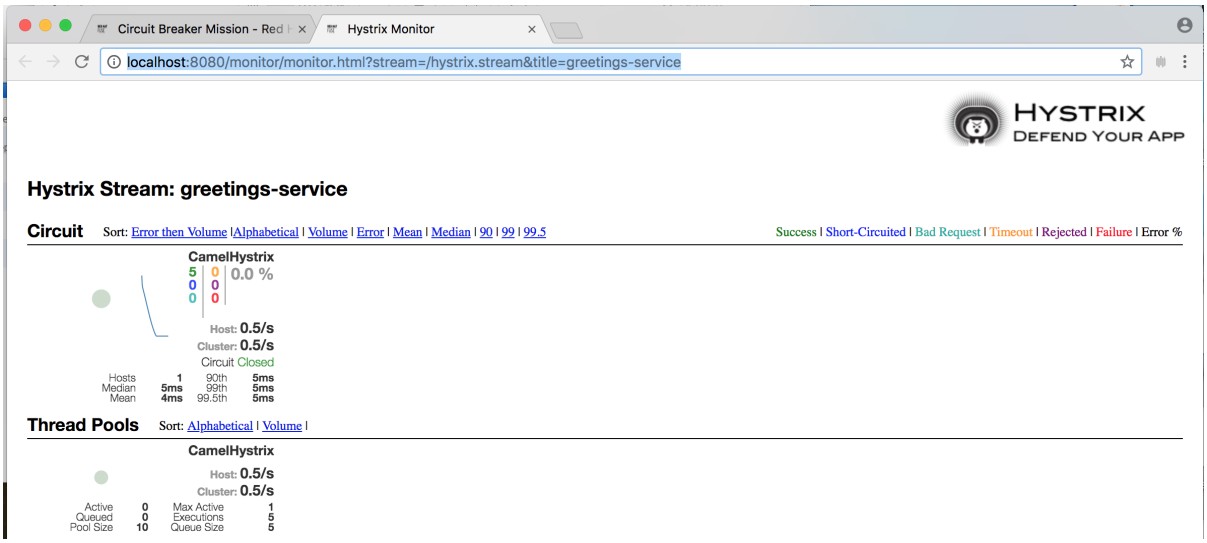
```
...
2017-12-08 15:46:58.521  INFO 22887 --- [           main] o.a.c.c.s.CamelHttpTransportServlet
: Initialized CamelHttpTransportServlet[name=CamelServlet, contextPath=]
2017-12-08 15:46:58.524  INFO 22887 --- [           main]
s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat started on port(s): 8080 (http)
2017-12-08 15:46:58.536  INFO 22887 --- [           main]
org.jboss.fuse.boosters.cb.Application   : Started Application in 6.263 seconds (JVM running
for 6.819)
```

greetings サービスは、URL **http://localhost:8080/camel/greetings** で REST エンドポイントを公開します。

4. http://localhost:8080 にアクセスします。
このページを開くと、Greeting Service が呼び出されます。

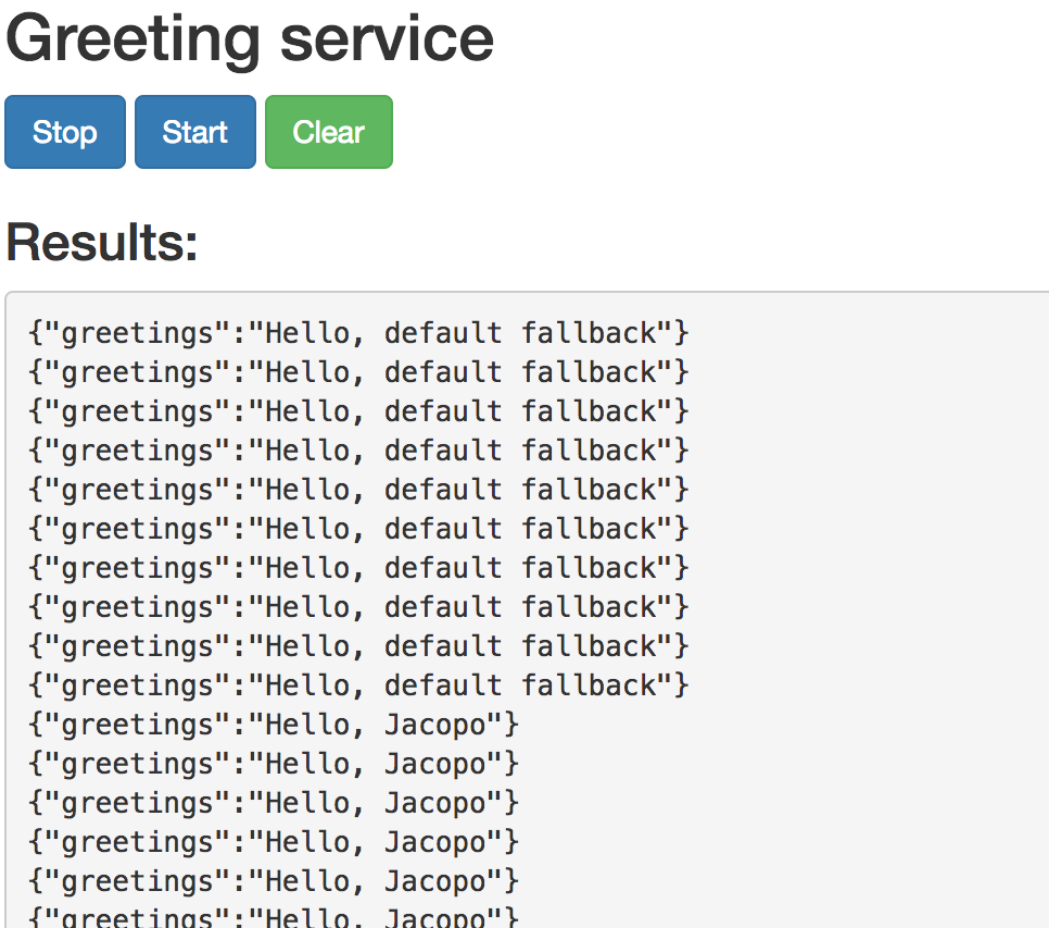


このページには、サーキットブレーカーの状態を監視する Hystrix ダッシュボードへのリンクも提供されます。



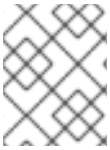
5. Camel Hystrix によって提供されるサーキットブレーカー機能を実証するには、name サービスが実行されているシェルプロンプトウインドウで **Ctrl+C** を押して、バックエンド name サービスを中止します。
これで name サービスが利用できなくなるため、呼び出されたときに greetings サービスがハングしないよう、サーキットブレーカーが作動します。

6. Hystrix Monitor ダッシュボードおよび Greeting Service の出力で変更を確認します。



2.3. EXTERNALIZED CONFIGURATION ブースターのビルドおよび実行

Externalized Configuration (外部化設定) ブースターは、Apache Camel ルートの設定を外部化する方法の例を提供します。Spring Boot スタンドアロンデプロイメントでは、設定データは **application.properties** ファイルに保存されます。



注記

Fuse on OpenShift デプロイメントでは、設定データは ConfigMap オブジェクトに保存されます。

前提条件

- 「ブースタープロジェクトの生成」 に記載されている手順を完了している必要があります。

手順

Externalized Configuration ミッションの 「ブースタープロジェクトの生成」 の手順に従った後、以下のステップに従って Externalized Configuration ブースターをローカルマシンのスタンドアロンプロジェクトとしてビルドおよび実行します。

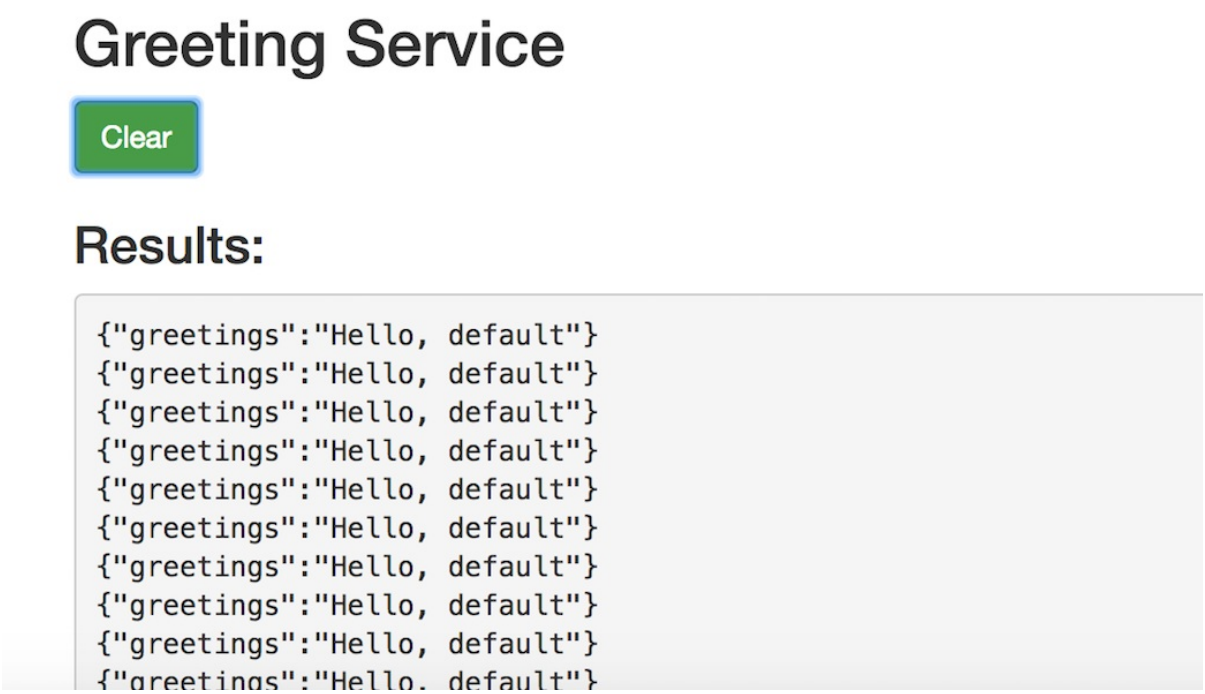
1. プロジェクトをダウンロードし、ローカルファイルシステムでアーカイブを展開します。
2. プロジェクトをビルドします。

```
cd PROJECT_DIR
mvn clean package
```

3. サービスを実行します。

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

4. Web ブラウザーで http://localhost:8080 を開きます。このページは、5 秒ごとに Greeting Service を呼び出します。Greetings サービスは、**target/classes/application.properties** ファイルから **booster.nameToGreetvalue** を使用して応答します。
Greeting Service は「default」へのグリーティングを 5 秒ごとに出力します。



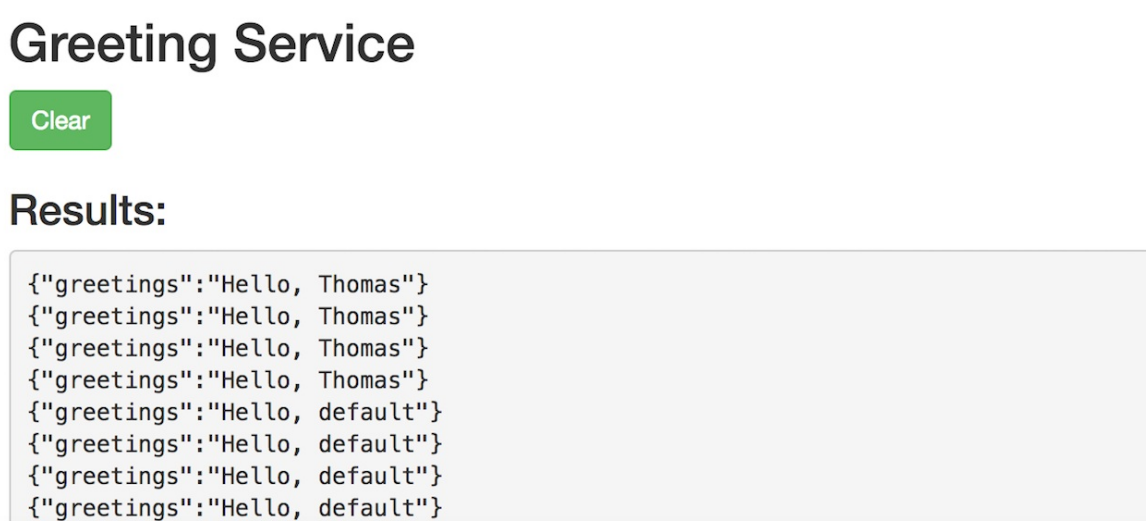
5. **booster.nameToGreet** の値を変更します。
   a. エディターで **target/classes/application.properties** ファイルを開きます。
   b. **booster.nameToGreetvalue** の値を、**default** から他の値 (例: **Thomas**) に変更します。

```
booster.nameToGreetvalue=Thomas
```

6. ターミナルウィンドウで **CTRL+C** を押してサービスを停止します。
7. サービスを再実行します。

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

8. Web ブラウザーで http://localhost:8080 ページに戻り、Greeting Service の Results ウィンドウで変更された値を確認します。



2.4. REST API ブースターのビルドおよび実行

REST API Level 0 のミッションでは、REST フレームワークを使用して、HTTP 経由でリモートプロシージャー呼び出しエンドポイントにビジネスオペレーションをマッピングする方法を示します。このミッションは、Richardson Maturity Model の Level 0 に該当します。

REST API ブースターは、HTTP プロトコルを使用して、Apache Camel によって公開されるリモートサービスと対話する仕組みを紹介します。この Fuse ブースターを使用すると、迅速に REST API のプロトタイプを作成し、柔軟に REST API を設定することができます。

このブースターを使用して、以下を行います。

- **camel/greetings/{name}** エンドポイントで HTTP GET 要求を実行します。このリクエストは、ペイロード **Hello, $name!** を使用して JSON 形式の応答を生成します (**$name** は HTTP GET リクエストからの URL パラメーターの値に置き換えられます)。
- URL **{name}** パラメーターの値を変更すると、変更後の値が応答に反映されます。
- REST API の Swagger ページを表示します。

前提条件

- 「ブースタープロジェクトの生成」 に記載されている手順を完了している必要があります。

手順

以下の手順に従って、REST API ブースターをローカルマシンのスタンドアロンプロジェクトとしてビルドおよび実行します。

1. プロジェクトをダウンロードし、ローカルファイルシステムでアーカイブを展開します。
2. プロジェクトをビルドします。

```
cd PROJECT_DIR
mvn clean package
```

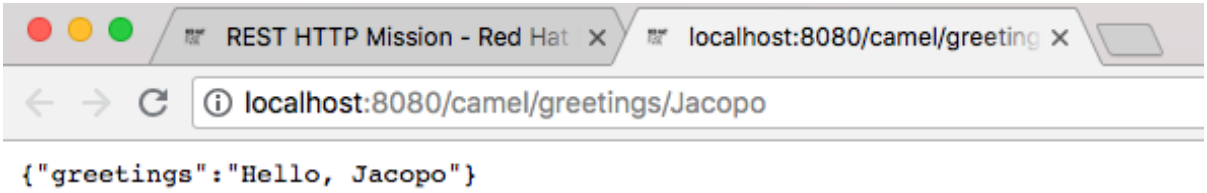
3. サービスを実行します。

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

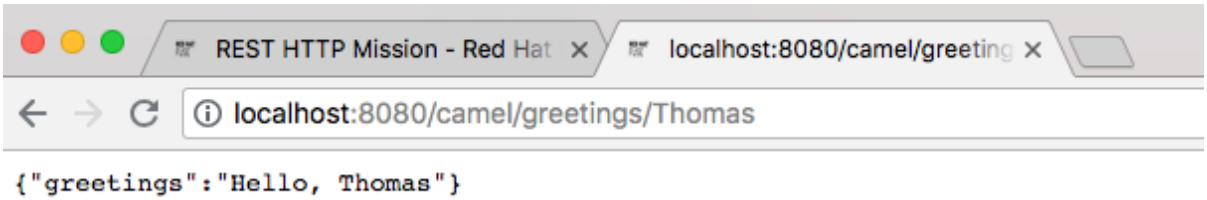
4. Web ブラウザーで http://localhost:8080 を開きます。

5. HTTP GET リクエストの例を実行するには、**camel/greetings/{name}** ボタンをクリックします。
localhost:8080/camel/greetings/Jacopo URL で新しい Web ブラウザーウインドウが開きます。URL **{name}** パラメーターのデフォルト値は **Jacopo** です。

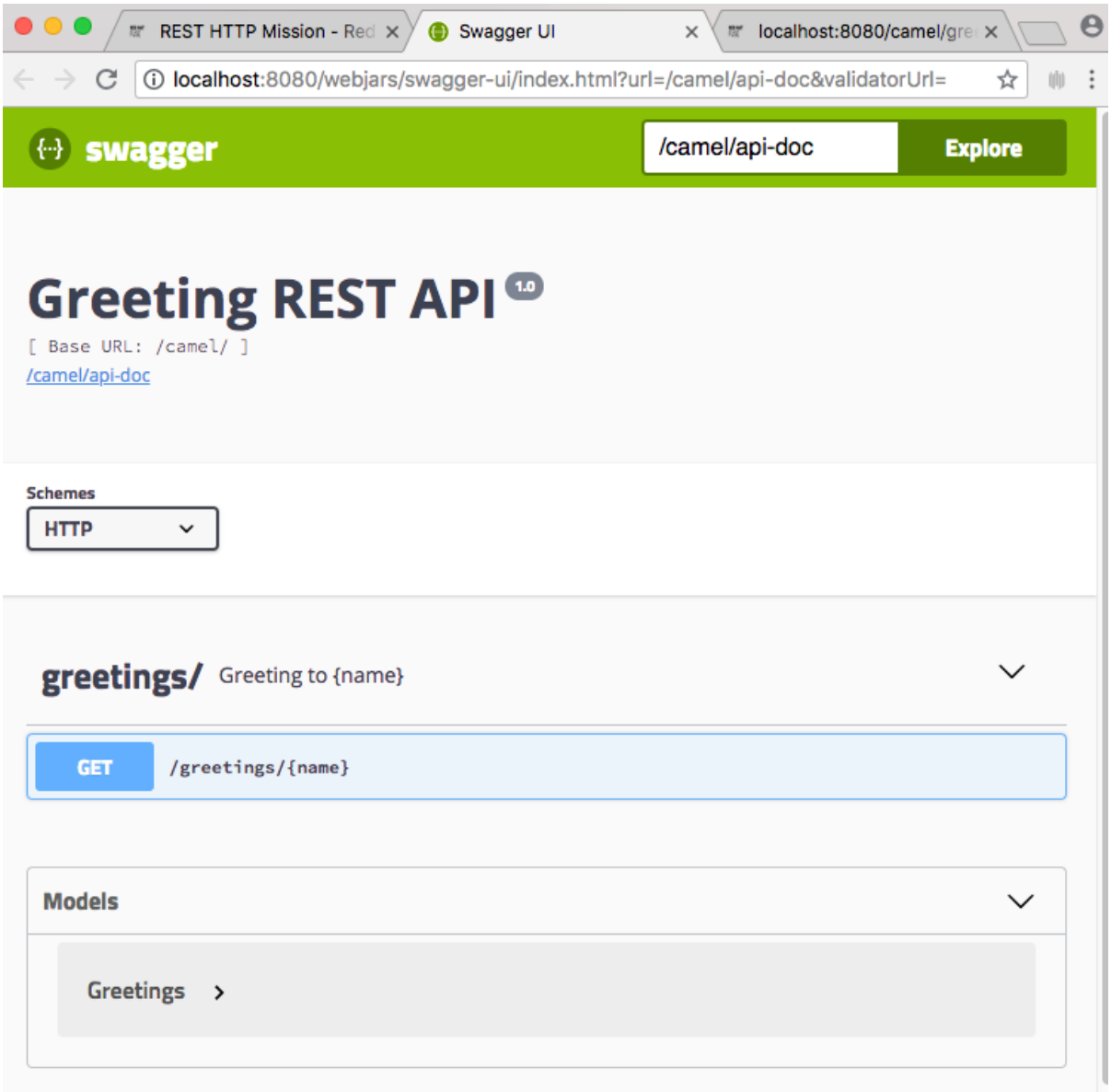
   ブラウザーウインドウに JSON 応答が表示されます。



6. **{name}** パラメーターの値を変更するには、URL を変更します。たとえば、名前を **Thomas** に変更するには、URL **localhost:8080/camel/greetings/Thomas** を使用します。
ブラウザーウインドウに更新された JSON 応答が表示されます。



7. REST API の Swagger ページを表示するには、API Swagger ページボタンをクリックします。
ブラウザーウィンドウに API swagger ページが表示されます。


REST HTTP Mission - Red
Swagger UI
localhost:8080/camel/gre
localhost:8080/webjars/swagger-ui/index.html?url=/camel/api-doc&validatorUrl=
swagger
/camel/api-doc
Explore
Greeting REST API
1.0
[Base URL: /camel/]
/camel/api-doc
Schemes
HTTP
greetings/ Greeting to {name}
GET
/greetings/{name}
Models
Greetings

第3章 SPRING BOOT での RED HAT SINGLE SIGN-ON の使用

Red Hat Single Sign-On クライアントアダプターは、Red Hat Single Sign-On でアプリケーションとサービスのセキュリティーを簡単に保護するためのライブラリーです。Keycloak Spring Boot アダプターを使用して、Spring Boot プロジェクトをセキュアにすることができます。

3.1. SPRING BOOT コンテナーでの RED HAT SINGLE SIGN-ON の使用

Spring Boot アプリケーションをセキュアにするには、Keycloak Spring Boot アダプター JAR をプロジェクトに追加します。Keycloak Spring Boot アダプターは Spring Boot の自動設定機能を活用するため、Keycloak Spring Boot スターターをプロジェクトに追加することのみが必要になります。

手順

1. Keycloak Spring Boot スターターを手動で追加するには、以下をプロジェクトの **pom.xml** に追加します。

```
<dependency>
      <groupId>org.keycloak</groupId>
      <artifactId>keycloak-spring-boot-starter</artifactId>
</dependency>
```

2. アダプター BOM 依存関係を追加します。

```
<dependencyManagement>
 <dependencies>
      <dependency>
        <groupId>org.keycloak.bom</groupId>
        <artifactId>keycloak-adapter-bom</artifactId>
        <version>3.4.17.Final-redhat-00001</version>
        <type>pom</type>
        <scope>import</scope>
      </dependency>
 </dependencies>
</dependencyManagement>
```

3. Keycloak を使用するように Spring Boot プロジェクトを設定します。**keycloak.json** ファイルの代わりに、通常の Spring Boot 設定を使用して Spring Boot アダプターのレルムを設定できます。たとえば、以下の設定を **src/main/resources/application.properties** ファイルに追加します。

```
keycloak.realm = demorealm
keycloak.auth-server-url = http://127.0.0.1:8080/auth
keycloak.ssl-required = external
keycloak.resource = demoapp
keycloak.credentials.secret = 11111111-1111-1111-1111-111111111111
keycloak.use-resource-role-mappings = true
```

keycloak.enabled = false を設定して、Keycloak Spring Boot Adapter (テストなど) を無効できます。Policy Enforcer を設定するには、**keycloak.json** とは異なり、**policy-enforcer** ではなく **policy-enforcer-config** を使用する必要があります。

4. **web.xml** に Java EE セキュリティー設定を指定します。Spring Boot Adapter は **login-method** を KEYCLOAK に設定し、起動時に **security-constraints** を設定します。設定例を以下に示します。

```
keycloak.securityConstraints[0].authRoles[0] = admin
keycloak.securityConstraints[0].authRoles[1] = user
keycloak.securityConstraints[0].securityCollections[0].name = insecure stuff
keycloak.securityConstraints[0].securityCollections[0].patterns[0] = /insecure


keycloak.securityConstraints[1].authRoles[0] = admin
keycloak.securityConstraints[1].securityCollections[0].name = admin stuff
keycloak.securityConstraints[1].securityCollections[0].patterns[0] = /admin
```

注記: Spring アプリケーションを WAR としてデプロイする場合は、Spring Boot アダプターを使用しないでください。使用しているアプリケーションサーバーまたはサーブレットコンテナーの専用アダプターを使用します。Spring Boot には **web.xml** ファイルも含まれている必要があります。

3.2. SPRING BOOT CXF JAXRS KEYCLOAK クイックスタートのビルドおよびデプロイ

この例は、Keycloak でセキュアにされた Apache CXF JAXRS を Spring Boot と使用する方法を示しています。クイックスタートは Spring Boot を使用して、Keycloak によってセキュアにされた、Swagger が有効な CXF JAXRS エンドポイントが含まれるアプリケーションを設定します。このクイックスタートはスタンドアロンモードで実行できます。

注記

これはアップストリームのデモであり、Red Hat からのサポートはありません。
『Deploying into Spring Boot』の「Using Spring Boot BOM」セクションを参照してください。

手順

このクイックスタートをローカルマシンでスタンドアロンプロジェクトとして実行するには、以下を行います。

1. Spring Boot CXF JAXRS Keycloak クイックスタートを https://github.com/ffang/spring-boot-cxf-keycloak からダウンロードし、ローカルファイルシステムでアーカイブを展開します。
2. クイックスタートディレクトリーに移動し、プロジェクトをビルドします。

```
cd PROJECT_DIR
mvn clean package
```

3. 以下のコマンドを実行して Spring Boot CXF JAXRS Keycloak クイックスタートをビルドし、デプロイします。

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

これにより、CXF JAXRS SB2 エンドポイントとともに事前定義された設定 (./src/main/resources/keycloak-config/realm-export-new.json) で Keycloak 認証サーバーが起動されます。

4. その後、Web ブラウザーから CXF JAXRS エンドポイントに直接アクセスできます。たとえば、**http://localhost:8080/services/helloservice/sayHello/FIS** を開いてエンドポイントにアクセスします。CXF JAXRS エンドポイントは Keycloak によってセキュア化されるため、リクエストが Keycloak 認証サーバーにリダイレクトされます。

5. ユーザー名には **admin**、パスワードには **passw0rd** を入力します。これにより OAuth2 JWT トークンが取得され、CXF JAXRS エンドポイントにリダイレクトされます。ブラウザーに **Hello FIS, Welcome to CXF RS Spring Boot World!!!** メッセージが表示されます。

第4章 SPRING BOOT で暗号化プロパティープレースホルダーを使用する方法

コンテナーのセキュリティーを保護する場合、設定ファイルでプレーンテキストのパスワードを使用することは推奨されません。プレーンテキストのパスワードを使用しないようにする方法の 1 つは、可能な限り暗号化プロパティープレースホルダーを使用することです。

4.1. 値を暗号化するマスターパスワードについて

Jasypt を使用して値を暗号化するには、マスターパスワードが必要です。マスターパスワードを選択するのは、ユーザーまたは管理者です。Jasypt は、マスターパスワードを設定する複数の方法を提供します。Jasypt を Spring 設定フレームワークに統合することにより、設定ファイルが読み込まれる際にプロパティーの値が復号化されます。1 つの方法として、Spring Boot 設定でマスターパスワードをプレーンテキストで指定できます。

Spring は **PropertyPlaceholder** フレームワークを使用して、トークンをプロパティーファイルからの値に置き換え、Jasypt のアプローチは **PropertyPlaceholderConfigurer** クラスを、暗号化された文字列を認識し、それを復号化するものに置き換えます。

例

```
<bean id="propertyPlaceholderConfigurer"
    class="org.jasypt.spring.properties.EncryptablePropertyPlaceholderConfigurer">
  <constructor-arg ref="configurationEncryptor" />
  <property name="location" value="/WEB-INF/application.properties" />
</bean>

<bean id="configurationEncryptor" class="org.jasypt.encryption.pbe.StandardPBEStringEncryptor">
  <property name="config" ref="environmentVariablesConfiguration" />
</bean>

<bean id="environmentVariablesConfiguration"
    class="org.jasypt.encryption.pbe.config.EnvironmentStringPBEConfig">
  <property name="algorithm" value="PBEWithMD5AndDES" />
  <property name="password" value="myPassword" />
</bean>
```

プレーンテキストでマスターパスワードを指定する代わりに、環境変数を使用してマスターパスワードを設定できます。Spring Boot 設定ファイルで、この環境変数を **passwordEnvName** プロパティーの値として指定します。たとえば、**MASTER_PW** 環境変数をマスターパスワードに設定すると、Spring Boot 設定ファイルにこのエントリーが作成されます。

```
<property name="passwordEnvName" value="MASTER_PW">
```

4.2. SPRING BOOT での暗号化プロパティープレースホルダーの使用

Jasypt を使用すると、プロパティーソースを暗号化し、アプリケーションは暗号化されたプロパティーを復号化し、元の値を取得できます。以下の手順では、Spring Boot でプロパティーソースを暗号化および復号化する方法を説明します。

手順

1. **jasypt** 依存関係をプロジェクトの **pom.xml** ファイルに追加します。

```
<dependency>
   <groupId>com.github.ulisesbocchio</groupId>
   <artifactId>jasypt-spring-boot-starter</artifactId>
   <version>3.0.3</version>
</dependency>
```

2. Maven リポジトリーをプロジェクトの pom.xml に追加します。

```
<repository>
<id>jasypt-basic</id>
<name>Jasypt Repository</name>
<url>https://repo1.maven.org/maven2/</url>
</repository>
```

3. Jasypt Maven プラグインをプロジェクトに追加すると、暗号化と復号化に Maven コマンドを使用できるようになります。

```
<plugin>
  <groupId>com.github.ulisesbocchio</groupId>
  <artifactId>jasypt-maven-plugin</artifactId>
  <version>3.0.3</version>
 </plugin>
```

4. プラグインリポジトリーを **pom.xml** に追加します。

```
<pluginRepository>
  <id>jasypt-basic</id>
  <name>Jasypt Repository</name>
  <url>https://repo1.maven.org/maven2/</url>
</pluginRepository>
```

5. **application.properties** ファイルに一覧表示されているユーザー名とパスワードを暗号化するには、以下のように **DEC()** 内でこれらの値をラップします。

```
spring.datasource.username=DEC(root)
spring.datasource.password=DEC(Password@1)
```

6. 以下のコマンドを実行して、ユーザー名とパスワードを暗号化します。

```
mvn jasypt:encrypt -Djasypt.encryptor.password=mypassword
```

これは、**application.properties** ファイルの DEC() プレースホルダーを、暗号化された値に置き換えます。以下に例を示します。

```
spring.datasource.username=ENC(3UtB1NhSZdVXN9xQBwkT0Gn+UxR832XP+tOOfFTlNL5
7FiMM7BWPRTeychVtLLhB)
spring.datasource.password=ENC(4ErqElyCHjjFnqPOCZNAaTdRC7u7yJSy16UsHtVkwPIr+3z
LyabNmQwwpFo7F7LU)
```

7. Spring アプリケーションの設定ファイルの認証情報を復号化するには、以下のコマンドを実行します。

```
mvn jasypt:decrypt -Djasypt.encryptor.password=mypassword
```

これにより、暗号化前の **application.properties** ファイルの内容が出力されます。ただし、これにより設定ファイルは更新されません。

第5章 MAVEN でのビルド

Fuse で Spring Boot のアプリケーションを開発する場合、Apache Maven ビルドツールを使用して、ソースコードを Maven プロジェクトとして構築することが標準的な方法です。すぐに開発できるようにするため、Fuse には Maven クイックスタートが提供されています。また、多くの Fuse ビルドツールは Maven プラグインとして提供されています。このため、Fuse の Spring Boot プロジェクトのビルドツールとして Maven を採用することが強く推奨されます。

5.1. MAVEN プロジェクトの生成

Fuse には、Maven アーキタイプを基にした複数の Spring Boot アプリケーションが提供され、これらを使用して Spring Boot アプリケーションの最初の Maven プロジェクトを生成できます。さまざまな Maven アーキタイプの場所情報とバージョンを把握する必要をなくすため、Fuse はスタンドアロン Spring Boot プロジェクトの Maven プロジェクトを生成するためのツールを提供します。

5.1.1. developers.redhat.com/launch のプロジェクトジェネレーター

Fuse で Spring Boot スタンドアロンを使い始める最も簡単な方法は、developers.redhat.com/launch にアクセスし、Spring Boot スタンドアロンランタイムの手順に従って、新しい Maven プロジェクトを生成することです。画面の指示に従うと、ローカルにビルドおよび実行できる完全な Maven プロジェクトが含まれるアーカイブファイルをダウンロードするように指示されます。

5.1.2. Developer Studio の Fuse ツールウィザード

また、Fuse Tooling が含まれる Red Hat JBoss Developer Studio をダウンロードおよびインストールすることもできます。**Fuse New Integration Project** ウィザードを使用すると、新しい Spring Boot スタンドアロンプロジェクトを生成し、Eclipse ベースの IDE 内で開発を継続できます。

5.2. SPRING BOOT BOM の使用

最初の Spring Boot プロジェクトを作成およびビルドした後、コンポーネントをすぐ追加したくなるでしょう。しかし、プロジェクトに追加する Maven 依存関係のバージョンはどのように判断したらよいのでしょうか。最も簡単な方法は、すべてのバージョンの依存関係を自動的に定義する、BOM (Bill of Materials) ファイルを使用することです。これは推奨される方法でもあります。

5.2.1. Spring Boot の BOM ファイル

Maven BOM (Bill of Materials) ファイルの目的は、正常に動作する Maven 依存関係バージョンのセットを提供し、各 Maven アーティファクトに対して個別にバージョンを定義する必要をなくすことです。

重要

使用している Spring Boot のバージョンに適した Fuse BOM を使用するようにしてください。

Spring Boot の Fuse BOM には以下の利点があります。

- Maven 依存関係のバージョンを定義するため、依存関係を POM に追加するときにバージョンを指定する必要がありません。
- 特定バージョンの Fuse に対して完全にテストされ、完全にサポートする依存関係のセットを定義します。

- Fuse のアップグレードを簡素化します。

重要

Fuse BOM によって定義される依存関係のセットのみが Red Hat によってサポートされます。

5.2.2. BOM ファイルの組み込み

Maven プロジェクトに BOM ファイルを組み込むには、以下の Spring Boot 2 の例のように、プロジェクトの **pom.xml** ファイル (または親 POM ファイル内の) **dependencyManagement** 要素を指定します。

- Spring Boot 2 の BOM

Spring Boot 2 の BOM

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<project ...>
  ...
  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

    <!-- configure the versions you want to use here -->
    <fuse.version>7.9.0.fuse-sb2-790065-redhat-00001</fuse.version>
  </properties>

  <dependencyManagement>
    <dependencies>
      <dependency>
        <groupId>org.jboss.redhat-fuse</groupId>
        <artifactId>fuse-springboot-bom</artifactId>
        <version>${fuse.version}</version>
        <type>pom</type>
        <scope>import</scope>
      </dependency>
    </dependencies>
  </dependencyManagement>
  ...
</project>
```

依存関係管理のメカニズムを使用して BOM を指定した後、アーティファクトのバージョンを指定**しなくても**、Maven 依存関係を POM に追加できるようになります。たとえば、 **camel-hystrix** コンポーネントの依存関係を追加するには、以下の XML フラグメントを POM の **dependencies** 要素に追加します。

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-hystrix-starter</artifactId>
</dependency>
```

Camel アーティファクト ID が **-starter** 接尾辞とともに追加されていることに注意してください。つまり、Camel Hystrix コンポーネントを **camel-hystrix** ではなく **camel-hystrix-starter** として指定します。Camel スターターコンポーネントは、Spring Boot 環境に対して最適化されるようにパッケージ化

されています。

5.2.3. Spring Boot Maven プラグイン

Spring Boot Maven プラグインは Spring Boot によって提供されます。これは、Spring Boot プロジェクトをビルドおよび実行するための開発者ユーティリティーです。

- **ビルド**: プロジェクトディレクトリーでコマンド **mvn package** を入力し、Spring Boot アプリケーションの実行可能な Jar パッケージを作成します。ビルドの出力は、Maven プロジェクトの **target/** サブディレクトリーに格納されます。

- **実行**: 新規ビルドされたアプリケーションは **mvn spring-boot:start** コマンドで実行することができます。

Spring Boot Maven プラグインをプロジェクトの POM ファイルに組み込むには、以下の例のように、プラグイン設定を **pom.xml** ファイルの **project/build/plugins** セクションに追加します。

例

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<project ...>
  ...
  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

    <!-- configure the versions you want to use here -->
    <fuse.version>7.9.0.fuse-sb2-790065-redhat-00001</fuse.version>

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

第6章 SPRING BOOT での APACHE CAMEL アプリケーションの実行

Apache Camel Spring Boot コンポーネントは、自動的に Camel コンテキストを Spring Boot に設定します。Camel コンテキストの自動設定によって、Spring コンテキストで使用できる Camel ルートが自動検出され、プロデューサーテンプレート、コンシューマーテンプレート、タイプコンバーターなどの主な Camel ユーティリティーが Bean として登録されます。Apache Camel コンポーネントには、スターターを使用して Spring Boot アプリケーションを開発できるようにする Spring Boot スターターモジュールが含まれます。

6.1. CAMEL SPRING BOOT コンポーネント

Camel Spring Boot アプリケーションはすべてプロジェクトの **pom.xml** にある **dependencyManagement** を使用して、依存関係の製品化バージョンを指定する必要があります。これらの依存関係は Red Hat Fuse BOM で定義され、特定バージョンの Red Hat Fuse でサポートされます。BOM からのバージョンをオーバーライドしないようにするため、追加のスターターのバージョン番号を省略することができます。詳細は、「quickstart pom 」を参照してください。

例

```
<dependencyManagement>
 <dependencies>
  <dependency>
   <groupId>org.jboss.redhat-fuse</groupId>
   <artifactId>fuse-springboot-bom</artifactId>
   <version>${fuse.version}</version>
   <type>pom</type>
   <scope>import</scope>
  </dependency>
 </dependencies>
</dependencyManagement>
```

注記

camel-spring-boot jar には **spring.factories** ファイルが含まれています。このファイルにより、依存関係をクラスパスに追加できるため、Spring Boot によって Camel コンテキストが自動的に設定されます。

6.2. CAMEL SPRING BOOT スターターモジュール

スターターは、Spring Boot アプリケーションでの使用を目的とする Apache Camel モジュールです。「スターターモジュールのない Camel コンポーネント一覧」 に記載されている一部の例外を除き、**camel-xxx-starter** モジュールは各 Camel コンポーネントにあります。

スターターは以下の要件を満たしています。

- IDE ツールと互換性のあるネイティブ Spring Boot 設定システムを使用して、コンポーネントの自動設定を可能にします。
- データ形式および言語の自動設定を可能にします。
- 推移的なログの依存関係を管理し、Spring Boot ロギングシステムと統合します。

- 追加の依存関係を含め、推移的な依存関係を合わせることで Spring Boot アプリケーションを作成するための労力を最小限にします。

各スターターの **tests/camel-itest-spring-boot** に独自の統合テストがあり、現在のリリースの Spring Boot との互換性を検証します。

6.3. スターターモジュールのない CAMEL コンポーネント一覧

互換性の問題があるため、以下のコンポーネントにはスターターモジュールがありません。

- **camel-blueprint** (OSGi のみを対象)
- **camel-cdi** (CDI のみを対象)
- **camel-core-osgi** (OSGi のみを対象)
- **camel-ejb** (JEE のみを対象)
- **camel-eventadmin** (OSGi のみを対象)
- **camel-ibatis** (**camel-mybatis-starter** が含まれます)
- **camel-jclouds**
- **camel-mina** (**camel-mina2-starter** が含まれます)
- **camel-paxlogging** (OSGi のみを対象)
- **camel-quartz** (**camel-quartz2-starter** が含まれます)
- **camel-spark-rest**
- **camel-openapi-java** (**camel-openapi-java-starter** が含まれます)

6.4. CAMEL SPRING BOOT スターターの使用

Apache Camel では、Spring Boot アプリケーションをすぐに開発できるようにするスターターモジュールが提供されます。

手順

1. 以下の依存関係を Spring Boot の pom.xml に追加します。

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-spring-boot-starter</artifactId>
</dependency>
```

2. 以下のスニペットのように、Camel ルートでクラスを追加します。これらのルートがクラスパスに追加されると、ルートは自動的に開始されます。

```
package com.example;

import org.apache.camel.builder.RouteBuilder;
import org.springframework.stereotype.Component;
```

```
@Component
public class MyRoute extends RouteBuilder {

    @Override
    public void configure() throws Exception {
        from("timer:foo")
          .to("log:bar");
    }
}
```

3. 任意手順:Camel の稼働を維持するためにメインスレッドがブロックされた状態を維持するには、以下の 1 つを行います。

    a. **spring-boot-starter-web** 関係が含まれるようにします。

    b. または、**camel.springboot.main-run-controller=true** を **application.properties** または **application.yml** ファイルに追加します。
    application.properties または **application.yml** ファイルで **camel.springboot.* properties** を使用すると Camel アプリケーションをカスタマイズできます。

4. 任意手順: Bean の ID 名を使用してカスタム Bean を参照するには、**src/main/resources/application.properties** または **application.yml** ファイルのオプションを設定します。以下の例は、Bean ID を使用して xslt コンポーネントがカスタム Bean を参照する方法を示しています。

    a. ID **myExtensionFactory** でカスタム Bean を参照します。

    ```
    camel.component.xslt.saxon-extension-functions=myExtensionFactory
    ```

    b. 次に、Spring Boot の @Bean アノテーションを使用してカスタム Bean を作成します。

    ```
    @Bean(name = "myExtensionFactory")
    public ExtensionFunctionDefinition myExtensionFactory() {
        }
    ```

    または、Jackson ObjectMapper の場合は、**camel-jackson** データ形式を以下のようにします。

    ```
    camel.dataformat.json-jackson.object-mapper=myJacksonMapper
    ```

6.5. SPRING BOOT の CAMEL コンテキストの自動設定

Camel Spring Boot auto-configuration は、**CamelContext** インスタンスを提供し、**SpringCamelContext** を作成します。また、コンテキストの初期化およびシャットダウンを実行します。この Camel コンテキストは、**camelContext** Bean 名で Spring アプリケーションコンテキストに登録され、他の Spring Bean と同様にアクセスできます。**camelContext** には次のようにアクセスできます。

例

```
@Configuration
public class MyAppConfig {
```

```
  @Autowired
  CamelContext camelContext;

  @Bean
  MyService myService() {
    return new DefaultMyService(camelContext);
  }

}
```

6.6. SPRING BOOT アプリケーションでの CAMEL ルートの自動検出

Camel auto-configuration は、Spring コンテキストからすべての **RouteBuilder** インスタンスを収集し、自動的に **CamelContext** にインジェクトします。これにより、Spring Boot スターターで新しい Camel ルートを作成する処理が簡単になります。以下のようにルートを作成できます。

例

@Component アノテーションが付けられたクラスをクラスパスに追加します。

```
@Component
public class MyRouter extends RouteBuilder {

  @Override
  public void configure() throws Exception {
    from("jms:invoices").to("file:/invoices");
  }

}
```

または、新しいルート **RouteBuilder** Bean を **@Configuration** クラスで作成します。

```
@Configuration
public class MyRouterConfiguration {

  @Bean
  RoutesBuilder myRouter() {
    return new RouteBuilder() {

      @Override
      public void configure() throws Exception {
        from("jms:invoices").to("file:/invoices");
      }

    };
  }

}
```

6.7. CAMEL SPRING BOOT AUTO CONFIGURATION の CAMEL プロパティーの設定

Spring Boot auto-configuration は、プロパティーのプレースホルダー、OS 環境変数、Camel プロパティーがサポートされるシステムプロパティーなどの Spring Boot 外部設定に接続します。

手順

1. **application.properties** ファイルにプロパティーを定義します。

```
route.from = jms:invoices
```

または、以下の例のように Camel プロパティーをシステムプロパティーとして設定します。

```
java -Droute.to=jms:processed.invoices -jar mySpringApp.jar
```

2. 次のように、設定されたプロパティーを Camel ルートのプレースホルダーとして使用します。

```
@Component
public class MyRouter extends RouteBuilder {

  @Override
  public void configure() throws Exception {
    from("{{route.from}}").to("{{route.to}}");
  }

}
```

6.8. カスタム CAMEL コンテキストの設定

Camel Spring Boot auto-configuration によって作成された **CamelContext** Bean でオペレーションを実行するには、Spring コンテキストで **CamelContextConfiguration** インスタンスを登録します。

手順

- 以下のように、Spring コンテキストで **CamelContextConfiguration** のインスタンスを登録します。

```
@Configuration
public class MyAppConfig {

  ...

  @Bean
  CamelContextConfiguration contextConfiguration() {
    return new CamelContextConfiguration() {
      @Override
      void beforeApplicationStart(CamelContext context) {
        // your custom configuration goes here
      }
    };
  }

}
```

Spring コンテキストの開始前に **CamelContextConfiguration** および **beforeApplicationStart(CamelContext)** メソッドが呼び出され、このコールバックに渡された **CamelContext** インスタンスは完全に自動設定されます。複数のインスタンスの **CamelContextConfiguration** を Spring コンテキストに追加でき、すべてが実行されます。

6.9. 自動設定された CAMELCONTEXT での JMX の無効化

自動設定された **CamelContext** で JMX を無効にするには、**camel.springboot.jmxEnabled** プロパティーを使用できます。JMX はデフォルトで有効になっています。

手順

- 以下のプロパティーを **application.properties** ファイルに追加し、**false** に設定します。

```
camel.springboot.jmxEnabled = false
```

6.10. 自動設定されたコンシューマーおよびプロデューサーテンプレートの SPRING 管理 BEAN へのインジェクト

Camel auto configuration によって、事前設定された **ConsumerTemplate** および **ProducerTemplate** インスタンスが提供されます。これらを Spring 管理の Bean にインジェクトすることができます。

例

```
@Component
public class InvoiceProcessor {

  @Autowired
  private ProducerTemplate producerTemplate;

  @Autowired
  private ConsumerTemplate consumerTemplate;
  public void processNextInvoice() {
    Invoice invoice = consumerTemplate.receiveBody("jms:invoices", Invoice.class);
    ...
    producerTemplate.sendBody("netty-http:http://invoicing.com/received/" + invoice.id());
  }

}
```

デフォルトでは、コンシューマーテンプレートとプロデューサーテンプレートのエンドポイントキャッシュサイズは 1000 に設定されています。これらの値を変更するには、以下の Spring プロパティーを希望するキャッシュサイズに設定します。例を以下に示します。

```
camel.springboot.consumerTemplateCacheSize = 100
camel.springboot.producerTemplateCacheSize = 200
```

6.11. SPRING コンテキストの自動設定された TYPECONVERTER

Camel auto configuration では、Spring コンテキストの **typeConverter** という名前の **TypeConverter** インスタンスが登録されます。

例

```
@Component
public class InvoiceProcessor {
```

```
  @Autowired
  private TypeConverter typeConverter;

  public long parseInvoiceValue(Invoice invoice) {
    String invoiceValue = invoice.grossValue();
    return typeConverter.convertTo(Long.class, invoiceValue);
  }

}
```

6.12. SPRING タイプコンバージョン API ブリッジ

Spring は、強力な タイプコンバージョン API で構成されます。Spring API は Camel の タイプコンバーター API と似ています。これらの API は似ているため、Camel Spring Boot は Spring コンバージョン API に委譲するブリッジコンバーター (**SpringTypeConverter**) を自動的に登録します。つまり、追加設定のない Camel は Spring コンバーターを Camel と同様に扱います。

これにより、以下のように Camel **TypeConverter** API を使用して、Camel および Spring コンバーターの両方にアクセスできます。

例

```
@Component
public class InvoiceProcessor {

  @Autowired
  private TypeConverter typeConverter;

  public UUID parseInvoiceId(Invoice invoice) {
    // Using Spring's StringToUUIDConverter
    UUID id = invoice.typeConverter.convertTo(UUID.class, invoice.getId());
  }

}
```

ここでは、Spring Boot はアプリケーションコンテキストで使用できる Spring の **ConversionService** インスタンスに変換を委譲します。**ConversionService** インスタンスがない場合は、Camel Spring Boot の自動設定が **ConversionService** のインスタンスを作成します。

6.13. タイプ変換機能の無効化

Camel Spring Boot のタイプ変換機能を無効にするには、**camel.springboot.typeConversion** プロパティーを **false** に設定します。このプロパティーが **false** に設定されると、auto-configuration によってタイプコンバーターインスタンスが登録されず、Spring Boot タイプコンバージョン API へのタイプ変換の委譲が有効になりません。

手順

- Camel Spring Boot コンポーネントのタイプ変換機能を無効にするには、以下のように **camel.springboot.typeConversion** プロパティーを **false** に設定します。

```
camel.springboot.typeConversion = false
```

6.14. 自動設定の XML ルートのクラスパスへの追加

デフォルトでは、**camel** ディレクトリーのクラスパスにある Camel XML ルートは Camel Spring Boot コンポーネントによって自動検出され、含まれます。設定オプションを使用すると、ディレクトリー名を設定でき、設定オプションを使用してこの機能を無効化できます。

手順

- 以下のようにクラスパスの Camel Spring Boot XML ルートを設定します。

```
// turn off
camel.springboot.xmlRoutes = false
// scan in the com/foo/routes classpath
camel.springboot.xmlRoutes = classpath:com/foo/routes/*.xml
```

注記

XML ファイルによって Camel XML ルート要素が定義され、**CamelContext** 要素は定義されないはずです。例を以下に示します。

```
<routes xmlns="http://camel.apache.org/schema/spring">
    <route id="test">
        <from uri="timer://trigger"/>
        <transform>
            <simple>ref:myBean</simple>
        </transform>
        <to uri="log:out"/>
    </route>
</routes>
```

Spring XML ファイルの使用

<camelContext> で Spring XML ファイルを使用するには、Spring XML ファイルまたは **application.properties** ファイルの Camel コンテキストを設定します。Camel コンテキストの名前を設定し、ストリームキャッシングを有効にするには、以下を **application.properties** ファイルに追加します。

```
camel.springboot.name = MyCamel
camel.springboot.stream-caching-enabled=true
```

6.15. 自動設定の XML REXT-DSL ルートの追加

Camel Spring Boot コンポーネントによって、**camel-rest** ディレクトリー以下のクラスパスに追加される Camel Rest-DSL XML ルートが自動検出され、組み込まれます。設定オプションを使用すると、ディレクトリー名を設定でき、設定オプションを使用してこの機能を無効化できます。

手順

- 以下のように、クラスパスの Camel Spring Boot Rest-DSL XML ルートを設定します。

```
// turn off
camel.springboot.xmlRests = false
// scan in the com/foo/routes classpath
```

```
camel.springboot.xmlRests = classpath:com/foo/rests/*.xml
```

注記

Rest-DSL ファイルによって、Camel XML REST 要素が定義され、**CamelContext** 要素は定義されないはずです。例を以下に示します。

```
<rests xmlns="http://camel.apache.org/schema/spring">
  <rest>
    <post uri="/persons">
      <to uri="direct:postPersons"/>
    </post>
    <get uri="/persons">
      <to uri="direct:getPersons"/>
    </get>
    <get uri="/persons/{personId}">
       <to uri="direct:getPersionId"/>
    </get>
    <put uri="/persons/{personId}">
       <to uri="direct:putPersionId"/>
    </put>
    <delete uri="/persons/{personId}">
       <to uri="direct:deletePersionId"/>
    </delete>
  </rest>
</rests>
```

6.16. CAMEL SPRING BOOT でのテスト

Camel を Spring Boot で実行すると、Spring Boot は自動的に Camel と **@Component** アノテーションが付けられたそのルートを組み込みします。Spring Boot でテストする場合、**@ContextConfiguration** ではなく **@SpringBootTest** を使用して、使用する設定クラスを指定します。

異なる RouteBuilder クラスに複数の Camel ルートがある場合、アプリケーションの実行時に Camel Spring Boot コンポーネントによってこれらのルートがすべて自動的に組み込まれます。1 つの RouteBuilder クラスのみからルートをテストする場合は、以下のパターンを使用して、有効にする RouteBuilder を include (含める) または exclude (除外) することができます。

- java-routes-include-pattern: パターンに一致する RouteBuilder クラスを include (含める)ために使用されます。
- java-routes-exclude-pattern: パターンに一致する RouteBuilder クラスを exclude (除外) するために使用されます。exclude は include よりも優先されます。

手順

1. 以下のように、ユニットテストクラスの **include** または **exclude** パターンを **@SpringBootTest** アノテーションへのプロパティーとして指定します。

```
@RunWith(CamelSpringBootRunner.class)
@SpringBootTest(classes = {MyApplication.class);
  properties = {"camel.springboot.java-routes-include-pattern=**/Foo*"})
public class FooTest {
```

FooTest クラスの include パターンは Ant スタイルパターンを表す **/**Foo*** です。このパターンは、すべてのパッケージ名と一致する 2 つのアスタリスクで始まります。/**Foo*** は、FooRoute のようにクラス名が Foo で始まる必要があることを意味します。

2. 以下の maven コマンドを使用してテストを実行します。

```
mvn test -Dtest=FooTest
```

その他のリソース

- Configuring Camel (Camel の設定)
- Component (コンポーネント)
- Endpoint (エンドポイント)
- Getting Started (スタートガイド)

6.17. SPRING BOOT、APACHE CAMEL、および外部メッセージングブローカーの使用

Fuse は外部メッセージングブローカーを使用します。サポートされるブローカー、クライアント、および Camel コンポーネントの組み合わせに関する詳細は「Red Hat Fuse でサポートされる構成」を参照してください。

Camel コンポーネントは JMS 接続ファクトリーに接続されている必要があります。以下の例は、**camel-amqp** コンポーネントを JMS 接続ファクトリーに接続する方法を示しています。

```
import org.apache.activemq.jms.pool.PooledConnectionFactory;
import org.apache.camel.component.amqp.AMQPComponent;
import org.apache.qpid.jms.JmsConnectionFactory;
...

AMQPComponent amqpComponent(AMQPConfiguration config) {
    JmsConnectionFactory qpid = new JmsConnectionFactory(config.getUsername(),
config.getPassword(), "amqp://"+ config.getHost() + ":" + config.getPort());
    qpid.setTopicPrefix("topic://");

    PooledConnectionFactory factory = new PooledConnectionFactory();
    factory.setConnectionFactory(qpid);

AMQPComponent amqpcomp = new AMQPComponent(factory);
```

第7章 RED HAT FUSE アプリケーションへのパッチ適用

新しい **patch-maven-plugin** メカニズムを使用すると、パッチを Red Hat Fuse アプリケーションに適用できます。このメカニズムにより、異なる Red Hat Fuse の BOM によって提供される個々のバージョンを変更できます (たとえば、**fuse-springboot-bom** と **fuse-karaf-bom** など)。

7.1. PATCH-MAVEN-PLUGIN

patch-maven-plugin は以下の操作を実行します。

- 現在の Red Hat Fuse BOM に関連するパッチメタデータを取得します。
- BOM からインポートされた **<dependencyManagement>** に、バージョンの変更を適用します。

patch-maven-plugin がメタデータを取得したら、プラグインが宣言されたプロジェクトの管理された依存関係および直接の依存関係すべてに対して繰り返し処理を行い、CVE/patch メタデータを使用して、一致する依存関係バージョンを置き換えます。バージョンが置き換えられたら、Maven ビルドが続行され、標準の Maven プロジェクトのステージに進みます。

7.2. RED HAT FUSE アプリケーションへのパッチ適用

patch-maven-plugin の目的は、Red Hat Fuse BOM にある依存関係のバージョンを、アプリケーションに適用するパッチのパッチメタデータに指定されたバージョンに更新することです。

手順

以下の手順では、アプリケーションにパッチを適用する方法を説明します。

1. **patch-maven-plugin** をプロジェクトの **pom.xml** ファイルに追加します。**patch-maven-plugin** のバージョンは、Fuse BOM のバージョンと同じである必要があります。

```
<build>
  <plugins>
    <plugin>
      <groupId>org.jboss.redhat-fuse</groupId>
      <artifactId>patch-maven-plugin</artifactId>
      <version>${version.org.jboss-redhat-fuse}</version>
      <extensions>true</extensions>
    </plugin>
  </plugins>
</build>
```

2. **mvn clean deploy**、**mvn validate**、または **mvn dependency:tree** コマンドの 1 つを実行すると、プラグインはプロジェクトモジュールを検索して、Red Hat Fuse BOM のいずれかが使用されているかどうかを確認します。以下の 2 つのみがサポートされる BOM とみなされます。
   - **org.jboss.redhat-fuse:fuse-karaf-bom**: Fuse Karaf BOM の場合
   - **org.jboss.redhat-fuse:fuse-springboot-bom**: Fuse Spring Boot BOM の場合
3. 上記の BOM がいずれも見つからない場合は、プラグインによって以下のメッセージが表示されます。

```
$ mvn clean install
```

```
[INFO] Scanning for projects...
[INFO]

========== Red Hat Fuse Maven patching ==========

[INFO] [PATCH] No project in the reactor uses Fuse Karaf or Fuse Spring Boot BOM.
Skipping patch processing.
[INFO] [PATCH] Done in 3ms
```

4. Fuse BOM が両方が見つかった場合は、**patch-maven-plugin** は以下の警告により停止します。

```
$ mvn clean install
[INFO] Scanning for projects...
[INFO]

========== Red Hat Fuse Maven patching ==========

[WARNING] [PATCH] Reactor uses both Fuse Karaf and Fuse Spring Boot BOMs. Please
use only one. Skipping patch processing.
[INFO] [PATCH] Done in 3ms
```

5. **patch-maven-plugin** は以下の Maven メタデータ値のいずれかを取得しようと試みます。

   - Fuse Karaf BOM を使用するプロジェクトの場合、**org.jboss.redhat-fuse/fuse-karaf-patch-metadata/maven-metadata.xml** が解決されています。これは、**org.jboss.redhat-fuse:fuse-karaf-patch-metadata:RELEASE** のアーティファクトのメタデータです。

   - Fuse Spring Boot BOM プロジェクトを使用するプロジェクトでは、**org.jboss.redhat-fuse/fuse-springboot-patch-metadata/maven-metadata.xml** が解決されています。これは、**org.jboss.redhat-fuse:fuse-springboot-patch-metadata:RELEASE** のアーティファクトのメタデータです。

     Maven によって生成されたメタデータの例

```
<?xml version="1.0" encoding="UTF-8"?>
<metadata>
  <groupId>org.jboss.redhat-fuse</groupId>
  <artifactId>fuse-springboot-patch-metadata</artifactId>
  <versioning>
    <release>7.8.1.fuse-sb2-781025</release>
    <versions>
      <version>7.8.0.fuse-sb2-780025</version>
      <version>7.7.0.fuse-sb2-770010</version>
      <version>7.7.0.fuse-770010</version>
      <version>7.8.1.fuse-sb2-781025</version>
    </versions>
    <lastUpdated>20201023131724</lastUpdated>
  </versioning>
</metadata>
```

6. **patch-maven-plugin** はメタデータを解析し、現在のプロジェクトに適用可能なバージョンを選択します。これは、バージョン **7.8.xxx** の Fuse BOM を使用する Maven プロジェクトでのみ可能です。バージョン範囲 7.8 および 7.9 以降と一致するメタデータのみが適用可能で、メタデータの最新バージョンのみが取得されます。

7. **patch-maven-plugin** は、前の手順で見つかった **groupId**、**artifactId**、および **version** によって特定されたパッチメタデータをダウンロードする際に使用されるリモート Maven リポジトリーのリストを収集します。これらの Maven リポジトリーは、アクティブなプロファイルのプロジェクトの **<repositories>** 要素にリストされているもので、**settings.xml** ファイルからのリポジトリーもリストされています。

```
$ mvn clean install
[INFO] Scanning for projects...
[INFO]

========== Red Hat Fuse Maven patching ==========

[INFO] [PATCH] Reading patch metadata and artifacts from 2 project repositories
[INFO] [PATCH]  - local-nexus: http://everfree.forest:8081/repository/maven-releases/
[INFO] [PATCH]  - central: https://repo.maven.apache.org/maven2
Downloading from local-nexus: http://everfree.forest:8081/repository/maven-
releases/org/jboss/redhat-fuse/fuse-springboot-patch-metadata/maven-metadata.xml
...
```

8. 任意で、オフラインリポジトリーを使用する場合は、**-Dpatch** オプションを使用して、**jboss-fuse/redhat-fuse** プロジェクトの **fuse-karaf/fuse-karaf-patch-repository** モジュールまたは **fuse-springboot/fuse-springboot-patch-repository** モジュールによって生成される ZIP ファイルを指定できます。これらの ZIP ファイルの内部構造は、Maven リポジトリーの構造と同じです。以下に例を示します。

```
$ mvn clean install -Dpatch=../../../test/resources/patch-3.zip
[INFO] Scanning for projects...
[INFO]

========== Red Hat Fuse Maven patching ==========

[INFO] [PATCH] Reading metadata and artifacts from /data/sources/github.com/jboss-
fuse/redhat-fuse/fuse-tools/patch-maven-plugin/src/test/resources/patch-3.zip
Downloading from fuse-patch: zip:file:/tmp/patch-3.zip-
1742974214598205745/org/jboss/redhat-fuse/fuse-springboot-patch-metadata/maven-
metadata.xml
Downloaded from fuse-patch: zip:file:/tmp/patch-3.zip-
1742974214598205745/org/jboss/redhat-fuse/fuse-springboot-patch-metadata/maven-
metadata.xml (406 B at 16 kB/s)
Downloading from fuse-patch: zip:file:/tmp/patch-3.zip-
1742974214598205745/org/jboss/redhat-fuse/fuse-springboot-patch-metadata/7.8.0.fuse-
sb2-781023/fuse-springboot-patch-metadata-7.8.0.fuse-sb2-781023.xml
Downloaded from fuse-patch: zip:file:/tmp/patch-3.zip-
1742974214598205745/org/jboss/redhat-fuse/fuse-springboot-patch-metadata/7.8.0.fuse-
sb2-781023/fuse-springboot-patch-metadata-7.8.0.fuse-sb2-781023.xml (926 B at 309 kB/s)
[INFO] [PATCH] Resolved patch descriptor: /home/user/.m2/repository/org/jboss/redhat-
fuse/fuse-springboot-patch-metadata/7.8.0.fuse-sb2-781023/fuse-springboot-patch-
metadata-7.8.0.fuse-sb2-781023.xml
...
```

9. メタデータがリモートリポジトリー、ローカルリポジトリー、または ZIP ファイルからであるかどうかに関わらず、**patch-maven-plugin** によって分析されます。フェッチされたメタデータには CVE の一覧が含まれ、各 CVE には、影響を受ける Maven アーティファクトのリスト (glob パターンおよびバージョン範囲で指定) と、指定の CVE の修正が含まれるバージョンがあります。以下に例を示します。

```
<?xml version="1.0" encoding="UTF-8" ?>

<metadata xmlns="urn:redhat:fuse:patch-metadata:1">
   <product-bom groupId="org.jboss.redhat-fuse" artifactId="fuse-springboot-bom" versions="
[7.8,7.9)" />
   <cves>
      <cve id="CVE-2020-xyz" description="Jetty can be configured to listen on port 8080"
           cve-link="https://nvd.nist.gov/vuln/detail/CVE-2020-xyz"
           bz-link="https://bugzilla.redhat.com/show_bug.cgi?id=42">
         <affects groupId="org.eclipse.jetty" artifactId="jetty-*" versions="[9.4,9.4.32)"
fix="9.4.32.v20200930" />
         <affects groupId="org.eclipse.jetty.http2" artifactId="http2-*" versions="[9.4,9.4.32)"
fix="9.4.32.v20200930" />
      </cve>
   </cves>
   <fixes />
</metadata>
```

10. 最後に、現在のプロジェクトの管理された依存関係に繰り返し処理が行われるときに、パッチメタデータに指定された修正リストが参照されます。一致するこれらの依存関係 (および管理された依存関係) は、固定バージョンに変更されます。以下は例になります。

```
$ mvn clean install -U
[INFO] Scanning for projects...
[INFO

========== Red Hat Fuse Maven patching ==========

[INFO] [PATCH] Reading patch metadata and artifacts from 2 project repositories
[INFO] [PATCH]  - local-nexus: http://everfree.forest:8081/repository/maven-releases/
[INFO] [PATCH]  - central: https://repo.maven.apache.org/maven2
Downloading from local-nexus: http://everfree.forest:8081/repository/maven-
releases/org/jboss/redhat-fuse/fuse-springboot-patch-metadata/maven-metadata.xml
Downloading from central: https://repo.maven.apache.org/maven2/org/jboss/redhat-
fuse/fuse-springboot-patch-metadata/maven-metadata.xml
Downloaded from local-nexus: http://everfree.forest:8081/repository/maven-
releases/org/jboss/redhat-fuse/fuse-springboot-patch-metadata/maven-metadata.xml (363 B
at 4.3 kB/s)
[INFO] [PATCH] Resolved patch descriptor: /home/user/.m2/repository/org/jboss/redhat-
fuse/fuse-springboot-patch-metadata/7.8.0.fuse-sb2-780032/fuse-springboot-patch-
metadata-7.8.0.fuse-sb2-780032.xml
[INFO] [PATCH] Patch metadata found for org.jboss.redhat-fuse/fuse-springboot-
bom/[7.8,7.9)
[INFO] [PATCH]  - patch contains 1 CVE fix
[INFO] [PATCH] Processing managed dependencies to apply CVE fixes...
(https://nvd.nist.gov/vuln/detail/CVE-2020-xyz, https://bugzilla.redhat.com/show_bug.cgi?
id=42_
[INFO] [PATCH] - CVE-2020-xyz: Jetty can be configured to expose itself on port 8080
[INFO] [PATCH]   Applying change org.eclipse.jetty/jetty-*/[9.4,9.4.32) -> 9.4.32.v20200930
[INFO] [PATCH]    - managed dependency: org.eclipse.jetty/jetty-alpn-
client/9.4.30.v20200611 -> 9.4.32.v20200930
...
[INFO] [PATCH]    - managed dependency: org.eclipse.jetty/jetty-openid/9.4.30.v20200611 ->
9.4.32.v20200930
[INFO] [PATCH]   Applying change org.eclipse.jetty.http2/http2-*/[9.4,9.4.32) ->
```

```
9.4.32.v20200930
[INFO] [PATCH]    - managed dependency: org.eclipse.jetty.http2/http2-
client/9.4.30.v20200611 -> 9.4.32.v20200930
...
[INFO] [PATCH] Done in 635ms

================================================
```

パッチのスキップ

特定のパッチをプロジェクトに適用したくない場合、**patch-maven-plugin** は **skip** オプションを提供します。すでに **patch-maven-plugin** をプロジェクトの **pom.xml** ファイルに追加済みで、バージョンを変更したくない場合は、以下のいずれかの方法を使用してパッチをスキップできます。

- 以下のように、プロジェクトの **pom.xml** ファイルに skip オプションを追加します。

```
<build>
   <plugins>
      <plugin>
         <groupId>org.jboss.redhat-fuse</groupId>
         <artifactId>patch-maven-plugin</artifactId>
         <version>${version.org.jboss-redhat-fuse}</version>
         <extensions>true</extensions>
         <configuration>
            <skip>true</skip>
         </configuration>
      </plugin>
   </plugins>
</build>
```

- または、以下のように **mvn** コマンドの実行時に **-DskipPatch** オプションを使用します。

```
$ mvn dependency:tree -DskipPatch
[INFO] Scanning for projects...
[INFO]
[INFO] ------< org.jboss.redhat-fuse:cve-dependency-management-module1 >-------
[INFO] Building cve-dependency-management-module1 7.8.0.fuse-sb2-780033
[INFO] --------------------------------[ jar ]---------------------------------
...
```

上記の出力にあるように、**patch-maven-plugin** は呼び出されず、パッチはアプリケーションに適用されません。

付録A MAVEN を使用する準備

ここでは、Red Hat Fuse プロジェクトをビルドするために Maven を準備する方法の概要を説明し、Maven アーティファクトの検索に使用される Maven コーディネートの概念を紹介します。

A.1. MAVEN 設定の準備

Maven は、Apache の無料のオープンソースビルドツールです。通常は、Maven を使用して Fuse アプリケーションをビルドします。

手順

1. Maven ダウンロードページ から最新バージョンの Maven をダウンロードします。
2. システムがインターネットに接続していることを確認します。
   プロジェクトのビルド中、Maven が外部リポジトリーを探し、必要なアーティファクトをダウンロードするのがデフォルトの動作になります。Maven はインターネット上でアクセス可能なリポジトリーを探します。

   このデフォルト動作を変更し、Maven によってローカルネットワーク上のリポジトリーのみが検索されるようにすることができます。これは Maven をオフラインモードで実行できることを意味します。オフラインモードでは、Maven によってローカルリポジトリーのアーティファクトが検索されます。「ローカル Maven リポジトリーの追加」 を参照してください。

A.2. RED HAT リポジトリーを MAVEN へ追加

Red Hat Maven リポジトリーあるアーティファクトにアクセスするには、Red Hat Maven リポジトリーを Maven の **settings.xml** ファイルに追加する必要があります。Maven は、**.m2** ディレクトリーで **settings.xml** ファイルを探します。ユーザー指定の **settings.xml** ファイルがない場合、Maven は **M2_HOME/conf/settings.xml** のシステムレベルの **settings.xml** ファイルを使用します。

前提条件

Red Hat リポジトリーを追加する **settings.xml** ファイルがある場所を知っている必要があります。

手順

以下の例のように、**settings.xml** ファイルに Red Hat リポジトリーの **repository** 要素を追加します。

```
<?xml version="1.0"?>
<settings>

  <profiles>
    <profile>
      <id>extra-repos</id>
      <activation>
        <activeByDefault>true</activeByDefault>
      </activation>
      <repositories>
       <repository>
          <id>redhat-ga-repository</id>
          <url>https://maven.repository.redhat.com/ga</url>
          <releases>
             <enabled>true</enabled>
          </releases>
```

```
        <snapshots>
          <enabled>false</enabled>
        </snapshots>
      </repository>
      <repository>
        <id>redhat-ea-repository</id>
        <url>https://maven.repository.redhat.com/earlyaccess/all</url>
        <releases>
          <enabled>true</enabled>
        </releases>
        <snapshots>
          <enabled>false</enabled>
        </snapshots>
      </repository>
      <repository>
        <id>jboss-public</id>
        <name>JBoss Public Repository Group</name>
        <url>https://repository.jboss.org/nexus/content/groups/public/</url>
      </repository>
    </repositories>
    <pluginRepositories>
      <pluginRepository>
        <id>redhat-ga-repository</id>
        <url>https://maven.repository.redhat.com/ga</url>
        <releases>
          <enabled>true</enabled>
        </releases>
        <snapshots>
          <enabled>false</enabled>
        </snapshots>
      </pluginRepository>
      <pluginRepository>
        <id>redhat-ea-repository</id>
        <url>https://maven.repository.redhat.com/earlyaccess/all</url>
        <releases>
          <enabled>true</enabled>
        </releases>
        <snapshots>
          <enabled>false</enabled>
        </snapshots>
      </pluginRepository>
      <pluginRepository>
        <id>jboss-public</id>
        <name>JBoss Public Repository Group</name>
        <url>https://repository.jboss.org/nexus/content/groups/public</url>
      </pluginRepository>
    </pluginRepositories>
  </profile>
 </profiles>

 <activeProfiles>
  <activeProfile>extra-repos</activeProfile>
 </activeProfiles>

</settings>
```

A.3. ローカル MAVEN リポジトリーの追加

インターネットへ接続せずにコンテナーを実行し、オフライン状態では使用できない依存関係を持つアプリケーションをデプロイする場合、Maven 依存関係プラグインを使用してアプリケーションの依存関係を Maven オフラインリポジトリーにダウンロードすることができます。ダウンロード後、このカスタマイズされた Maven オフラインリポジトリーをインターネットに接続していないマシンに提供することができます。

手順

1. **pom.xml** ファイルが含まれるプロジェクトディレクトリーで、以下のようなコマンドを実行し、Maven プロジェクトのリポジトリーをダウンロードします。

```
mvn org.apache.maven.plugins:maven-dependency-plugin:3.1.0:go-offline -
Dmaven.repo.local=/tmp/my-project
```

   この例では、プロジェクトのビルドに必要な Maven 依存関係とプラグインは **/tmp/my-project** ディレクトリーにダウンロードされます。

2. このカスタマイズされた Maven オフラインリポジトリーを、インターネットに接続していない内部のマシンに提供します。

A.4. MAVEN アーティファクトおよびコーディネート

Maven ビルドシステムでは、**アーティファクト** が基本のビルドブロックです。ビルド後のアーティファクトの出力は、通常 JAR や WAR ファイルなどのアーカイブになります。

Maven の主な特徴として、アーティファクトを検索し、検索したアーティファクト間で依存関係を管理できる機能が挙げられます。**Maven コーディネート** は、特定のアーティファクトの場所を特定する値のセットです。基本的なコーディネートには、以下の形式の 3 つの値があります。

groupId:artifactId:version

基本的なコーディネートに **packaging** の値、または **packaging** と **classifier** の値の両方を追加することがあります。Maven コーディネートには以下の形式のいずれかを使用できます。

```
groupId:artifactId:version
groupId:artifactId:packaging:version
groupId:artifactId:packaging:classifier:version
```

値の説明は次のとおりです。

groupdId

アーティファクトの名前の範囲を定義します。通常、パッケージ名のすべてまたは一部をグループ ID として使用します。例: **org.fusesource.example**

artifactId

グループ名に関連するアーティファクト名を定義します。

version

アーティファクトのバージョンを指定します。バージョン番号には **n.n.n.n** のように最大 4 つの部分を含めることができ、最後の部分には数字以外の文字を含めることができます。たとえば **1.0-SNAPSHOT** の場合、最後の部分は英数字のサブ文字列である **0-SNAPSHOT** になります。

packaging

プロジェクトのビルド時に生成されるパッケージ化されたエンティティーを定義します。OSGi プロジェクトでは、パッケージングは **bundle** になります。デフォルト値は **jar** です。

classifier

同じ POM からビルドされた内容が異なるアーティファクトを区別できるようにします。

アーティファクトの POM ファイルの要素は、以下のようにアーティファクトのグループ ID、アーティファクト ID、パッケージング、およびバージョンを定義します。

```
<project ... >
  ...
  <groupId>org.fusesource.example</groupId>
  <artifactId>bundle-demo</artifactId>
  <packaging>bundle</packaging>
  <version>1.0-SNAPSHOT</version>
  ...
</project>
```

前述のアーティファクトの依存関係を定義するには、以下の **dependency** 要素を POM ファイルに追加します。

```
<project ... >
  ...
  <dependencies>
    <dependency>
      <groupId>org.fusesource.example</groupId>
      <artifactId>bundle-demo</artifactId>
      <version>1.0-SNAPSHOT</version>
    </dependency>
  </dependencies>
  ...
</project>
```

注記

バンドルは特定タイプの JAR ファイルで、**jar** はデフォルトの Maven パッケージタイプであるため、前述の依存関係に **bundle** パッケージを指定する必要はありません。依存関係でパッケージタイプを明示的に指定する必要がある場合は、**type** 要素を使用できます。

付録B SPRING BOOT MAVEN プラグイン

Spring Boot Maven プラグインによって Maven で Spring Boot のサポートが提供され、実行可能な **jar** または **war** アーカイブをパッケージ化や、アプリケーション **in-place** の実行を可能にします。

B.1. SPRING BOOT MAVEN プラグインのゴール

Spring Boot Maven プラグインには以下のゴールが含まれます。

- **spring-boot:run** は Spring Boot アプリケーションを実行します。
- **spring-boot:repackage** は、**.jar** および **.war** ファイルを再パッケージして実行可能にします。
- **spring-boot:start** および **spring-boot:stop** の両方は、Spring Boot アプリケーションのライフサイクルを管理するために使用されます。
- **spring-boot:build-info** は、Actuator が使用できるビルド情報を生成します。

B.2. SPRING BOOT MAVEN プラグインの使用

Spring Boot プラグインの使用方法に関する一般的な手順は、**https://docs.spring.io/spring-boot/docs/current/maven-plugin/reference/htmlsingle/#using** を参照してください。以下の例は Spring Boot の **spring-boot-maven-plugin** の使用方法を示しています。

- Spring Boot 2 の例

注記

Spring Boot Maven プラグインの詳細は、**https://docs.spring.io/spring-boot/docs/current/maven-plugin/reference/htmlsingle/** を参照してください。

B.2.1. Spring Boot 2 の Spring Boot Maven プラグインの使用

以下の例は Spring Boot 2 の **spring-boot-maven-plugin** の使用方法を示しています。

例

```
<project>
  <modelVersion>4.0.0</modelVersion>

  <groupId>com.redhat.fuse</groupId>
  <artifactId>spring-boot-camel</artifactId>
  <version>1.0-SNAPSHOT</version>

    <properties>
      <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
      <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>

      <!-- configure the Fuse version you want to use here -->
      <fuse.bom.version>7.9.0.fuse-sb2-790065-redhat-00001</fuse.bom.version>

      <!-- maven plugin versions -->
      <maven-compiler-plugin.version>3.7.0</maven-compiler-plugin.version>
      <maven-surefire-plugin.version>2.19.1</maven-surefire-plugin.version>
```

```
</properties>


<build>
  <defaultGoal>spring-boot:run</defaultGoal>

  <plugins>
    <plugin>
      <groupId>org.jboss.redhat-fuse</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
      <version>${fuse.bom.version}</version>
      <executions>
        <execution>
          <goals>
            <goal>repackage</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>

<repositories>
  <repository>
    <id>redhat-ga-repository</id>
    <url>https://maven.repository.redhat.com/ga</url>
    <releases>
      <enabled>true</enabled>
    </releases>
    <snapshots>
      <enabled>false</enabled>
    </snapshots>
  </repository>
  <repository>
    <id>redhat-ea-repository</id>
    <url>https://maven.repository.redhat.com/earlyaccess/all</url>
    <releases>
      <enabled>true</enabled>
    </releases>
    <snapshots>
      <enabled>false</enabled>
    </snapshots>
  </repository>
</repositories>

<pluginRepositories>
  <pluginRepository>
    <id>redhat-ga-repository</id>
    <url>https://maven.repository.redhat.com/ga</url>
    <releases>
      <enabled>true</enabled>
    </releases>
    <snapshots>
      <enabled>false</enabled>
    </snapshots>
  </pluginRepository>
```

```
        <pluginRepository>
          <id>redhat-ea-repository</id>
          <url>https://maven.repository.redhat.com/earlyaccess/all</url>
          <releases>
            <enabled>true</enabled>
          </releases>
          <snapshots>
            <enabled>false</enabled>
          </snapshots>
        </pluginRepository>
      </pluginRepositories>
    </project>
```