Red Hat Fuse 7.9

Deploying into Spring Boot

Build and run Spring Boot applications in standalone mode

Last Updated: 2022-02-11

Red Hat Fuse 7.9 Deploying into Spring Boot

Build and run Spring Boot applications in standalone mode

Legal Notice

Abstract

This guide explains how to build Spring Boot applications that are packaged as Jar files and run directly in a JVM (standalone mode).

Table of Contents

MAKING OPEN SOURCE MORE INCLUSIVE

Red Hat is committed to replacing problematic language in our code, documentation, and web properties. We are beginning with these four terms: master, slave, blacklist, and whitelist. Because of the enormity of this endeavor, these changes will be implemented gradually over several upcoming releases. For more details, see our CTO Chris Wright's message.

CHAPTER 1. GETTING STARTED WITH SPRING BOOT STANDALONE

1.1. ABOUT SPRING BOOT STANDALONE DEPLOYMENT MODE

In standalone deployment mode, a Spring Boot application is packaged as a Jar file and run directly inside the Java Virtual Machine (JVM). This approach to packaging and running the application is consistent with the microservices philosophy, where a service is packaged with the minimum set of requirements. The Spring Boot application can be run directly using the **java** command with the **-jar** option. For example:

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

Where Spring Boot provides the main class for the executable Jar. The following elements are required for building a Spring Boot standalone application in Fuse:

- *The Fuse Bill of Materials (BOM)* – defines a carefully curated set of dependencies from the Red Hat Maven repository. The BOM exploits Maven's *dependency management* mechanism to define the appropriate versions of Maven dependencies.
  Note: Only the dependencies defined in the Fuse BOM are supported by Red Hat.

- *The Spring Boot Maven Plugin* – implements the build process for a standalone Spring Boot application in Maven. This plugin is responsible for packaging your Spring Boot application as an executable Jar file.

1.2. DEPLOYING INTO SPRING BOOT 2

In standalone deployment mode, you have the option of deploying into Spring Boot 2.

NOTE

For details about the OpenShift mode of deployment, see the Fuse on OpenShift Guide.

1.3. NEW CAMEL COMPONENTS FOR SPRING BOOT 2

Spring Boot 2 supports Camel version **2.23** and supports some new camel components that are listed below:

New Camel Components for Spring Boot 2

- 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

CHAPTER 2. USING FUSE BOOSTERS

Red Hat Fuse provides the following boosters to help you get started with Fuse applications and demonstrate useful components:

- Section 2.2, “Build and run the Circuit Breaker booster” - An example of enabling a distributed application to cope with interruptions to network connectivity and temporary unavailability of backend services.
- Section 2.3, “Build and run the Externalized Configuration booster” - An example of how to externalize configuration for an Apache Camel route.
- Section 2.4, “Build and run the REST API booster” - An example that introduces the mechanics of interacting with a remote (exposed by Apache Camel) service using the HTTP protocol.

Prerequisites To build and run the booster demonstrations, install the following prerequisites:

- A supported version of the Java Developer Kit (JDK). See the Supported Configurations page for details.
- Apache Maven 3.3.x or later. See the Maven Download page.

2.1. GENERATING YOUR BOOSTER PROJECT

Fuse booster projects exist to help developers get started with running standalone applications. The instructions provided here guide you through generating one of those booster projects, the Circuit Breaker booster. This exercise demonstrates useful components of the Fuse on Spring Boot.

The Netflix/Hystrix circuit breaker enables distributed applications to handle interruptions to network connectivity and temporary unavailability of backend services. The basic idea of the circuit breaker pattern is that the loss of a dependent service is detected automatically and an alternative behavior can be programmed, in case the backend service is temporarily unavailable.

The Fuse circuit breaker booster consists of two related services:

- A **name** service, the backend service that returns a name to greet.
- A **greetings** service, the frontend service that invokes the **name** service to get a name and then returns the string, **Hello, NAME**.

In this booster demonstration, the Hystrix circuit breaker is inserted between the **greetings** service and the **name** service. If the backend **name** service becomes unavailable, the **greetings** service can fall back to an alternative behavior and respond to the client immediately, instead of being blocked while it waits for the **name** service to restart.

Prerequisites

- You must have access to the Red Hat Developer Platform.
- You must have a supported version of the Java Developer Kit (JDK). See the Supported Configurations page for details.
- You must have installed and configured Apache Maven 3.3.x or later as described in Setting up Maven locally).

Procedure

1. Navigate to https://developers.redhat.com/launch.
2. Click **START**.
   The launcher wizard prompts you to log in to your Red Hat account.
3. On the **Launcher** page, click the **Deploy an Example Application** button.
4. On the **Create Example Application** page, type the name, **fuse-circuit-breaker**, in the **Create Example Application as** field.
5. Click **Select an Example**.
6. In the **Example** dialog, select the **Circuit Breaker** option. A **Select a Runtime** dropdown menu appears.
   a. From the **Select a Runtime** dropdown, select **Fuse**.
   b. From the version dropdown menu, select **7.9 (Red Hat Fuse)** (do not select the **2.21.2 (Community)** version).
   c. Click **Save**.
7. On the **Create Example Application** page, click **Download**.
8. When you see the **Your Application is Ready** dialog, click **Download.zip**. Your browser downloads the generated booster project (packaged as a ZIP file).
9. Use an archive utility to extract the generated project to a convenient location on your local file system.

2.2. BUILD AND RUN THE CIRCUIT BREAKER BOOSTER

The Netflix/Hystrix circuit breaker component enables distributed applications to cope with interruptions to network connectivity and temporary unavailability of backend services. The basic idea of the circuit breaker pattern is that the loss of a dependent service is detected automatically and an alternative behavior can be programmed, in case the backend service is temporarily unavailable.

The Fuse circuit breaker booster consists of two related services:

- A *name service*, which returns a name to greet
- A *greetings service*, which invokes the name service to get a name and then returns the string, **Hello, NAME**.

In this demonstration, the Hystrix circuit breaker is inserted between the greetings service and the name service. If the name service becomes unavailable, the greetings service can fall back to an alternative behavior and respond to the client immediately, instead of blocking or timing out while it waits for the name service to restart.

Prerequisites

- you have completed the steps mentioned in the Section 2.1, “Generating your booster project” section.

Procedure

Follow these steps to build and run the Circuit breaker booster project:

1. Open a shell prompt and build the project from the command line, using Maven:

```
cd PROJECT_DIR
mvn clean package
```

2. Open a new shell prompt and start the name service, as follows:

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

   As Spring Boot starts up, you should see some output like the following:

```
...
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. Open a new shell prompt and start the greetings service, as follows:

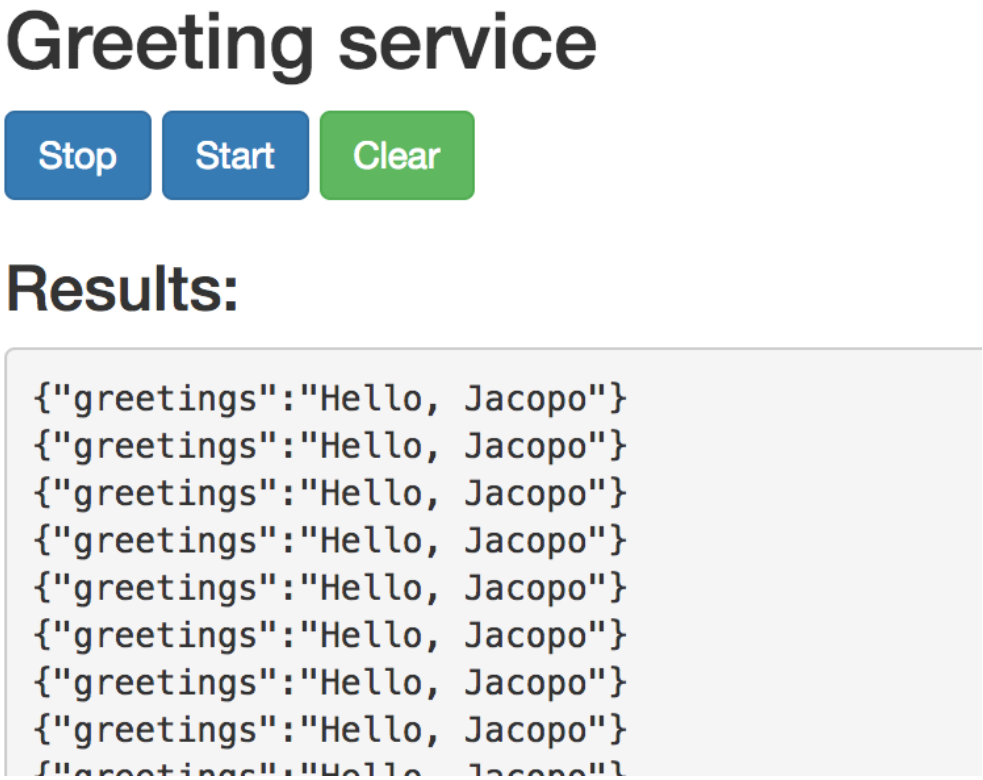
```
cd greetings-service
mvn spring-boot:run -DskipTests
```

   As Spring Boot starts up, you should see some output like the following:

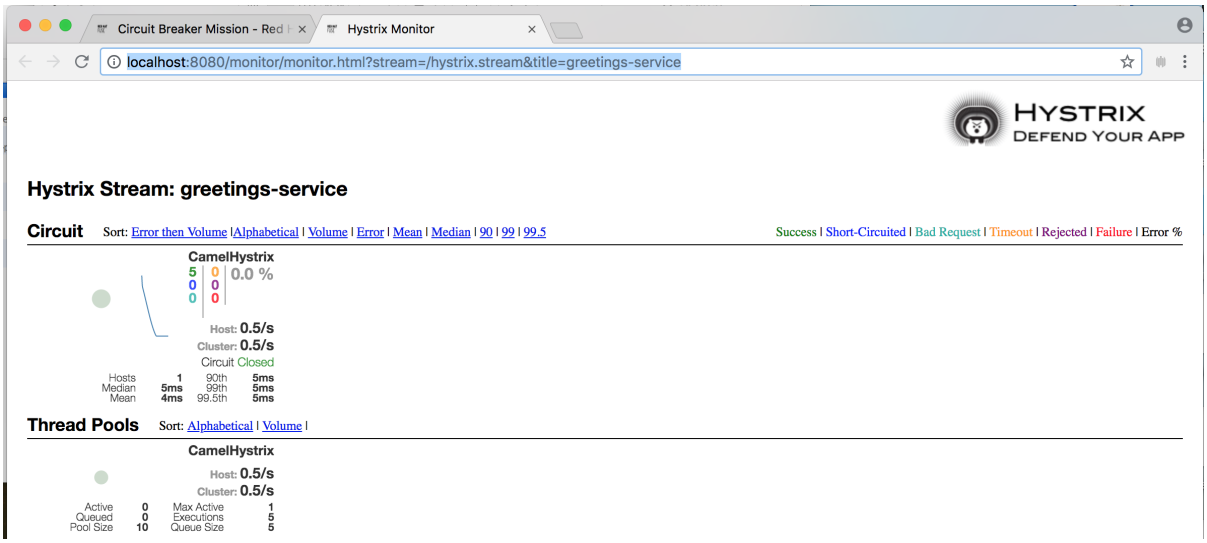
```
...
2017-12-08 15:46:58.521  INFO 22887 --- [           main] o.a.c.c.s.CamelHttpTransportServlet
: Initialized CamelHttpTransportServlet[name=CamelServlet, contextPath=]
2017-12-08 15:46:58.524  INFO 22887 --- [           main]
s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat started on port(s): 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)
```

   The greetings service exposes a REST endpoint at the URL, **http://localhost:8080/camel/greetings**.

4. Go to http://localhost:8080
   When you open this page, it invokes the Greeting Service:

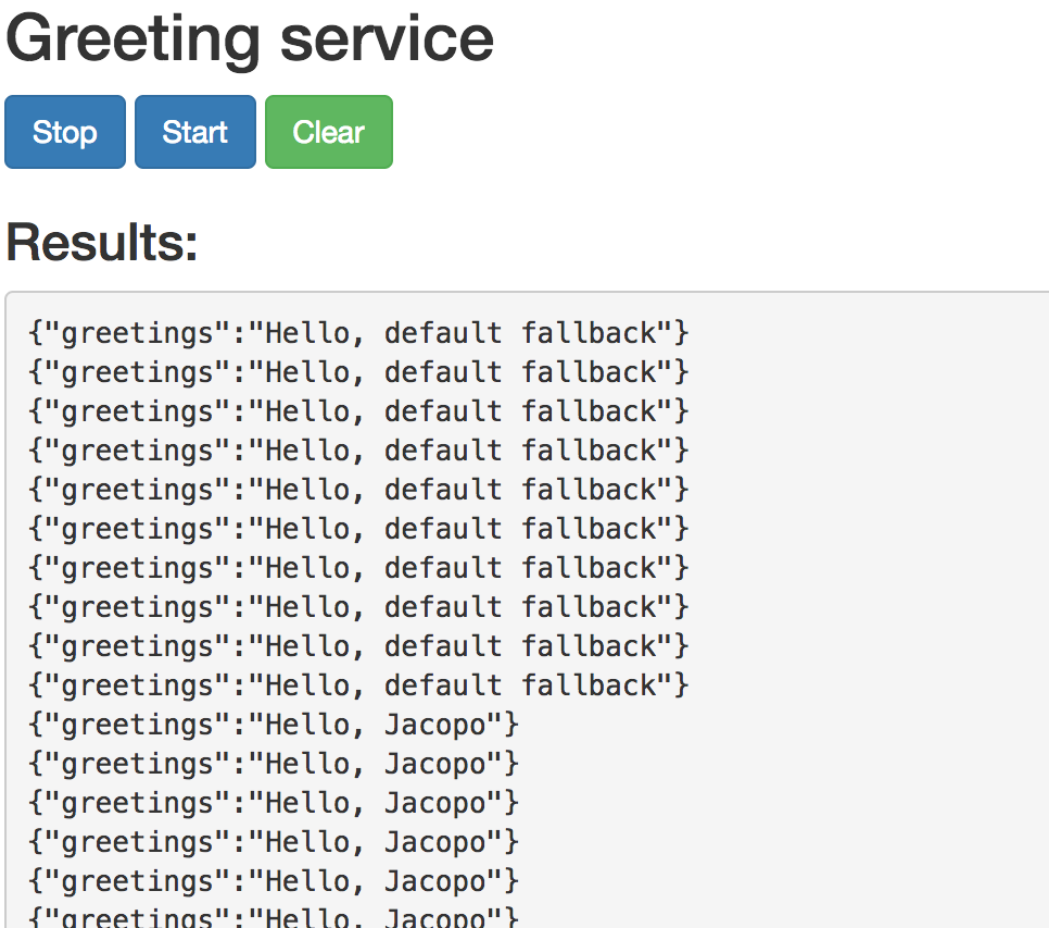


This page also provides a link to the Hystrix dashboard, which monitors the state of the circuit breaker.



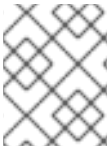
5. To demonstrate the circuit breaker functionality provided by Camel Hystrix, kill the backend name service by pressing **Ctrl+C** while in the window of the shell prompt where the name service is running.
   Now that the name service is unavailable, the circuit breaker kicks in to prevent the greetings service from hanging when it is invoked.

6. Observe the changes in the Hystrix Monitor dashboard and in the Greeting Service output:



2.3. BUILD AND RUN THE EXTERNALIZED CONFIGURATION BOOSTER

The Externalized Configuration booster provides an example of how to externalize configuration for an Apache Camel route. For Spring Boot standalone deployments, the configuration data is stored in an **application.properties** file.



NOTE

For Fuse on OpenShift deployments, the configuration data is stored in a ConfigMap object.

Prerequisites

- you have completed the steps mentioned in the Section 2.1, “Generating your booster project” section.

Procedure

After you follow the Section 2.1, “Generating your booster project” steps for the **Externalized Configuration** mission, follow these steps to build and run the Externalized Configuration booster as a standalone project on your local machine:

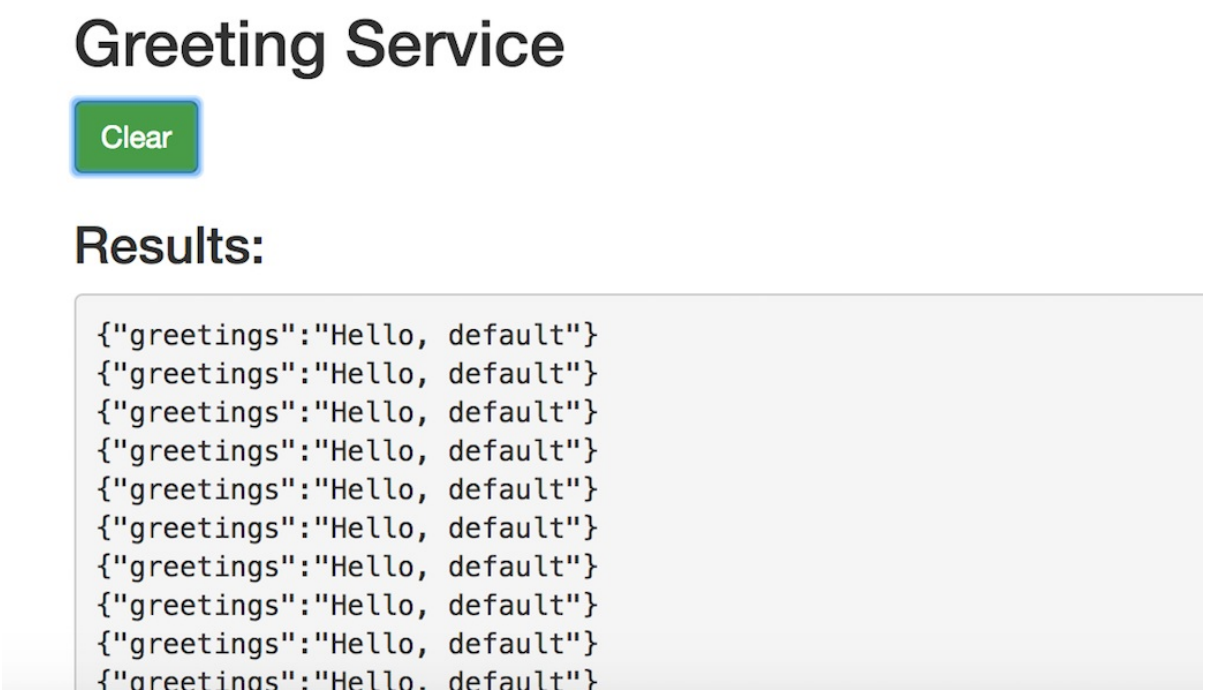
1. Download the project and extract the archive on your local filesystem.
2. Build the project:

```
cd PROJECT_DIR
mvn clean package
```

3. Run the service:

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

4. Open a web browser to http://localhost:8080. This page invokes a Greeting Service every 5 seconds. The Greetings Service responds by using the **booster.nameToGreetvalue** from the **target/classes/application.properties** file.
The Greeting Service prints a greeting to "default" every five seconds:



5. Modify the **booster.nameToGreet** value:

   a. Open the **target/classes/application.properties** file in an editor.

   b. Change the value of the **booster.nameToGreetvalue** from **default** to another value, for example **Thomas**:

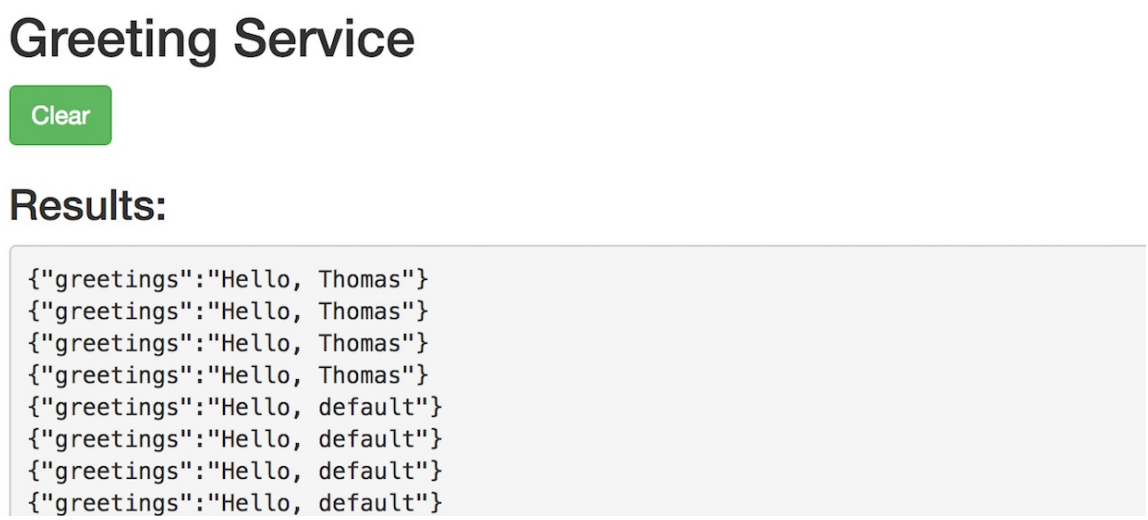
```
booster.nameToGreetvalue=Thomas
```

6. In the Terminal window, stop the service by pressing **CTRL+C**.

7. Run the service again:

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

8. In the web browser, return to the http://localhost:8080 page to view the changed value in the Greeting Service's results window.



2.4. BUILD AND RUN THE REST API BOOSTER

The REST API Level 0 mission shows how to map business operations to a remote procedure call endpoint over HTTP by using a REST framework. This mission corresponds to Level 0 in the Richardson Maturity Model.

The REST API booster introduces the mechanics of interacting with a remote (exposed by Apache Camel) service using the HTTP protocol. By using this Fuse booster, you can quickly prototype and flexibly design a REST API.

Use this booster to:

- Execute an HTTP GET request on the **camel/greetings/{name}** endpoint. This request produces a response in JSON format with a payload of **Hello, $name!** (where **$name** is replaced by the value of the URL parameter from the HTTP GET request).
- Change the value of the URL **{name}** parameter to see the changed value reflected in the response.
- View the REST API's Swagger page.

Prerequisites

- you have completed the steps mentioned in the Section 2.1, "Generating your booster project" section.

Procedure

Follow these steps to build and run the REST API booster as a standalone project on your local machine:

1. Download the project and extract the archive on your local filesystem.
2. Build the project:

```
cd PROJECT_DIR
mvn clean package
```

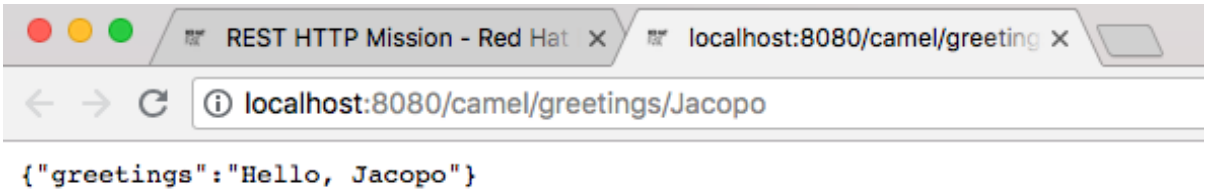
3. Run the service:

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

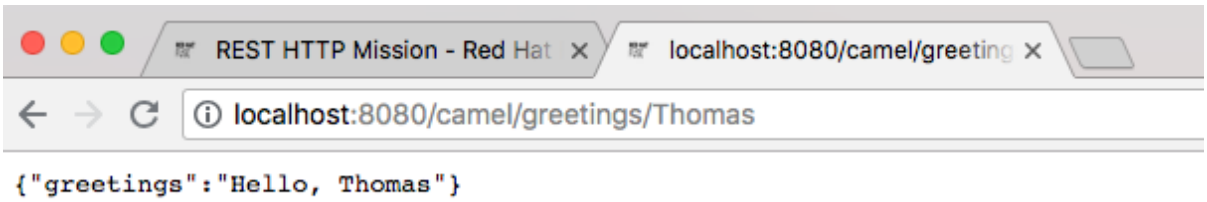
4. Open a web browser to: http://localhost:8080

5. To execute the example HTTP GET request, click the **camel/greetings/{name}** button.
   A new web browser window opens with the **localhost:8080/camel/greetings/Jacopo** URL. The default value of the URL **{name}** parameter is **Jacopo**.

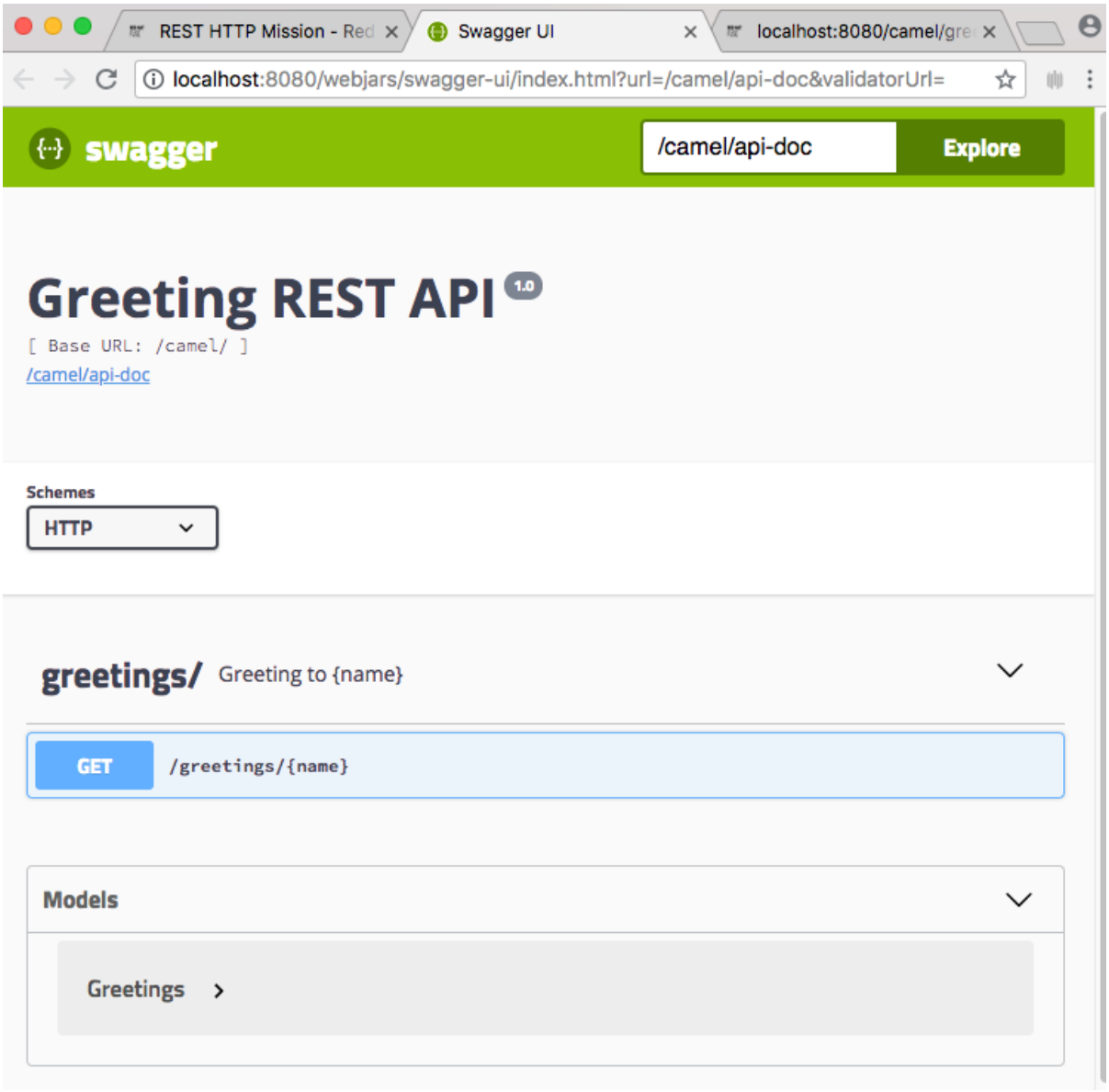
   The JSON response appears in the browser window:



6. To change the value of the **{name}** parameter, change the URL. For example, to change the name to **Thomas**, use this URL: **localhost:8080/camel/greetings/Thomas**.
   The updated JSON response appears in the browser window:



7. To view the REST API's Swagger page, click the API Swagger page button.
   The API swagger page opens in a browser window.


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

CHAPTER 3. USING RED HAT SINGLE SIGN-ON WITH SPRING BOOT

Red Hat Single Sign-On client adapters are libraries that make it very easy to secure applications and services with Red Hat Single Sign-On. You can use the Keycloak Spring Boot adapter to secure your Spring Boot project.

3.1. USING RED HAT SINGLE SIGN-ON WITH SPRING BOOT CONTAINER

To secure a Spring Boot application, add the Keycloak Spring Boot adapter JAR to your project. The Keycloak Spring Boot adapter takes advantage of Spring Boot's autoconfiguration feature so all you need to do is add the Keycloak Spring Boot starter to your project.

Procedure

1. To manually add the Keycloak Spring Boot starter, add the following to your project's **pom.xml**.

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

2. Add the Adapter BOM dependency.

```
<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. Configure your Spring Boot project to use Keycloak. Instead of a **keycloak.json** file, you can configure the realm for the Spring Boot Keycloak adapter using the normal Spring Boot configuration. For example, add following configuration to **src/main/resources/application.properties** file.

```
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
```

You can disable the Keycloak Spring Boot Adapter (for example in tests) by setting **keycloak.enabled = false**. To configure a Policy Enforcer, unlike **keycloak.json**, **policy-enforcer-config** must be used instead of just **policy-enforcer**.

4. Specify the Java EE security configuration in the **web.xml**. The Spring Boot Adapter will set the **login-method** to KEYCLOAK and configure the **security-constraints** at the time of startup. An example configuration is given below.

```
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
```

Note: If you plan to deploy your Spring Application as a WAR then do not use the Spring Boot Adapter. Use the dedicated adapter for the application server or servlet container you are using. Your Spring Boot should also contain a **web.xml** file.

3.2. BUILD AND DEPLOY SPRING BOOT CXF JAXRS KEYCLOAK QUICKSTART

This example demonstrates how you can use Apache CXF JAXRS which is secured by Keycloak with Spring Boot. The quickstart uses Spring Boot to configure an application that includes a CXF JAXRS endpoint with Swagger enabled, which is secured by Keycloak. You can run this quickstart in the standalone mode.

NOTE

This is an upstream demo with no support from Red Hat. See section Using Spring Boot BOM in the Deploying into Spring Boot guide

Procedure

To run this quickstart as a standalone project on your local machine:

1. Download the Spring Boot CXF JAXRS Keycloak quickstart[https://github.com/ffang/spring-boot-cxf-keycloak] and extract the archive on your local filesystem.

2. Navigate to the quickstart directory and build the project.

```
cd PROJECT_DIR
mvn clean package
```

3. Run the following command to build and deploy the Spring Boot CXF JAXRS Keycloak quickstart.

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

This starts the Keycloak auth server with predefined configuration (./src/main/resources/keycloak-config/realm-export-new.json) along with CXF JAXRS SB2 endpoint.

4. You can then access the CXF JAXRS endpoint directly from your web browser, for example, open **http://localhost:8080/services/helloservice/sayHello/FIS** to access the endpoint. Since

the CXF JAXRS endpoint is secured by Keycloak, this will redirect request to Keycloak auth server.

5. Enter **admin** as the username and **passw0rd** as the password. This fetches the OAuth2 JWT token and redirects to the CXF JAXRS endpoint. You can see **Hello FIS, Welcome to CXF RS Spring Boot World!!!** message on the browser.

CHAPTER 4. HOW TO USE ENCRYPTED PROPERTY PLACEHOLDERS IN SPRING BOOT

When securing a container it is not recommended to use the plain text passwords in configuration files. One way to avoid using plain text passwords is to use encrypted property placeholders whenever possible.

4.1. ABOUT THE MASTER PASSWORD FOR ENCRYPTING VALUES

To use Jasypt to encrypt a value, a master password is required. It is up to you or an administrator to choose the master password. Jasypt provides several ways to set the master password. Jasypt can be integrated into the Spring configuration framework so that property values are decrypted as the configuration file is loaded. One way is to specify the master password in plain text in a Spring boot configuration.

Spring uses the **PropertyPlaceholder** framework to replace tokens with values from a properties file, and Jasypt's approach replaces the **PropertyPlaceholderConfigurer** class with one that recognizes encrypted strings and decrypts them.

Example

```
<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>
```

Instead of specifying the master password in plain text, you can use an environment variable to set your master password. In the Spring Boot configuration file, specify this environment variable as the value of the **passwordEnvName** property. For example, if you set the **MASTER_PW** environment variable to your master password, then you would have this entry in your Spring Boot configuration file:

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

4.2. USING ENCRYPTED PROPERTY PLACEHOLDERS IN SPRING BOOT

By using Jasypt, you can provide encryption for the property sources and the application can decrypt the encrypted properties and retrieve the original values. Following procedure explains how to encrypt and decrypt the property sources in Spring Boot.

Procedure

1. Add **jasypt** dependency to your project's **pom.xml** file.

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

2. Add Maven repository to your project's pom.xml.

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

3. Add the Jasypt Maven plugin to your project as well as it allows you to use the Maven commands for encryption and decryption.

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

4. Add the plugin repository to **pom.xml**.

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

5. To encrypt the username and password listed in the **application.properties** file, wrap these values inside **DEC()** as shown below.

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

6. Run the following command to encrypt the username and password.

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

This replaces the DEC() placeholders in the **application.properties** file with the encrypted value, for example,

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

7. To decrypt the credentials in the Spring application configuration file, run following command.

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

This prints out the content of the **application.properties** file as it was before the encryption. However, this does not update the configuration file.

CHAPTER 5. BUILDING WITH MAVEN

The standard approach to developing applications for Spring Boot in Fuse is to use the Apache Maven build tool and to structure your source code as a Maven project. Fuse provides Maven quickstarts to get you started quickly and many of the Fuse build tools are provided as Maven plug-ins. For this reason, it is highly recommended that you adopt Maven as the build tool for Spring Boot projects in Fuse.

5.1. GENERATING A MAVEN PROJECT

Fuse provides a selection of quickstarts, based on Maven archetypes, which you can use to generate an initial Maven project for a Spring Boot application. To prevent you from having to remember the location information and versions for various Maven archetypes, Fuse provides tooling to help you generate Maven projects for standalone Spring Boot projects.

5.1.1. Project generator at developers.redhat.com/launch

The quickest way to get started with Spring Boot standalone in Fuse is to navigate to developers.redhat.com/launch and follow the instructions for the Spring Boot standalone runtime, to generate a new Maven project. After following the on-screen instructions, you will be prompted to download an archive file, which contains a complete Maven project that you can build and run locally.

5.1.2. Fuse tooling wizard in Developer Studio

Alternatively, you can download and install Red Hat JBoss Developer Studio (which includes Fuse Tooling). Using the **Fuse New Integration Project** wizard, you can generate a new Spring Boot standalone project and continue to develop inside the Eclipse-based IDE.

5.2. USING SPRING BOOT BOM

After creating and building your first Spring Boot project, you will soon want to add more components. But how do you know which versions of the Maven dependencies to add to your project? The simplest (and recommended) approach is to use the relevant Bill of Materials (BOM) file, which automatically defines all of the version dependencies for you.

5.2.1. BOM file for Spring Boot

The purpose of a Maven Bill of Materials (BOM) file is to provide a curated set of Maven dependency versions that work well together, preventing you from having to define versions individually for every Maven artifact.

IMPORTANT

Ensure you are using the correct Fuse BOM based on the version of Spring Boot you are using.

The Fuse BOM for Spring Boot offers the following advantages:

- Defines versions for Maven dependencies, so that you do not need to specify the version when you add a dependency to your POM.
- Defines a set of curated dependencies that are fully tested and supported for a specific version of Fuse.

- Simplifies upgrades of Fuse.

IMPORTANT

Only the set of dependencies defined by a Fuse BOM are supported by Red Hat.

5.2.2. Incorporate the BOM file

To incorporate a BOM file into your Maven project, specify a **dependencyManagement** element in your project's **pom.xml** file (or, possibly, in a parent POM file), as shown in the examples for both Spring Boot 2:

- Spring Boot 2 BOM

Spring Boot 2 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>
```

After specifying the BOM using the dependency management mechanism, it is possible to add Maven dependencies to your POM *without* specifying the version of the artifact. For example, to add a dependency for the **camel-hystrix** component, you would add the following XML fragment to the **dependencies** element in your POM:

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

Note how the Camel artifact ID is specified with the **-starter** suffix – that is, you specify the Camel Hystrix component as **camel-hystrix-starter**, not as **camel-hystrix**. The Camel starter components are packaged in a way that is optimized for the Spring Boot environment.

5.2.3. Spring Boot Maven plugin

The Spring Boot Maven plugin is provided by Spring Boot and it is a developer utility for building and running a Spring Boot project:

- *Building* – create an executable Jar package for your Spring Boot application by entering the command **mvn package** in the project directory. The output of the build is placed in the **target/** subdirectory of your Maven project.
- *Running* – for convenience, you can run the newly-built application with the command, **mvn spring-boot:start**.

To incorporate the Spring Boot Maven plugin into your project POM file, add the plugin configuration to the **project/build/plugins** section of your **pom.xml** file, as shown in the following example.

Example

```
<?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>
```

CHAPTER 6. RUNNING APACHE CAMEL APPLICATION IN SPRING BOOT

The Apache Camel Spring Boot component automatically configures Camel context for Spring Boot. Auto-configuration of the Camel context automatically detects the Camel routes available in the Spring context and registers the key Camel utilities such as producer template, consumer template, and the type converter as beans. The Apache Camel component includes a Spring Boot starter module that allows you to develop Spring Boot applications by using starters.

6.1. INTRODUCTION TO THE CAMEL SPRING BOOT COMPONENT

Every Camel Spring Boot application must use the **dependencyManagement** element in the project's **pom.xml** to specify the productized versions of the dependencies. These dependencies are defined in the Red Hat Fuse BOM and are supported for the specific version of Red Hat Fuse. You can omit the version number attribute for the additional starters so as not to override the versions from BOM. See quickstart pom for more information.

Example

```
<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>
```

NOTE

The **camel-spring-boot** jar contains with the **spring.factories** file which allows you to add that dependency to your classpath so Spring Boot will automatically configure Camel context.

6.2. INTRODUCTION TO THE CAMEL SPRING BOOT STARTER MODULE

Starters are the Apache Camel modules that are intended to be used in Spring Boot applications. There is a **camel-xxx-starter** module for each Camel component (with a few exceptions listed in the Section 6.3, "List of the Camel components that do not have starter modules" section).

Starters meet the following requirements:

- Allow auto-configuration of the component by using the native Spring Boot configuration system which is compatible with IDE tooling.
- Allow auto-configuration of data formats and languages.
- Manage transitive logging dependencies to integrate with the Spring Boot logging system.

- Include additional dependencies and align transitive dependencies to minimize the effort of creating a working Spring Boot application.

Each starter has its own integration test in **tests/camel-itest-spring-boot**, that verifies the compatibility with the current release of Spring Boot.

6.3. LIST OF THE CAMEL COMPONENTS THAT DO NOT HAVE STARTER MODULES

The following components do not have starter modules because of compatibility issues:

- **camel-blueprint** (intended for OSGi only)
- **camel-cdi** (intended for CDI only)
- **camel-core-osgi** (intended for OSGi only)
- **camel-ejb** (intended for JEE only)
- **camel-eventadmin** (intended for OSGi only)
- **camel-ibatis** (**camel-mybatis-starter** is included)
- **camel-jclouds**
- **camel-mina** (**camel-mina2-starter** is included)
- **camel-paxlogging** (intended for OSGi only)
- **camel-quartz** (**camel-quartz2-starter** is included)
- **camel-spark-rest**
- **camel-openapi-java** (**camel-openapi-java-starter** is included)

6.4. USING CAMEL SPRING BOOT STARTER

Apache Camel provides a starter module that allows you to quickly get started developing Spring Boot applications.

Procedure

1. Add the following dependency to your Spring Boot pom.xml file:

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

2. Add the classes with your Camel routes as shown in the snippet below. Once these routes are added to the class path the routes are started automatically.

```
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. Optional. To keep the main thread blocked so that Camel stays up, do one of the following.

    a. Include the **spring-boot-starter-web** dependency,

    b. Or add **camel.springboot.main-run-controller=true** to your **application.properties** or **application.yml** file.
    You can customize the Camel application in the **application.properties** or **application.yml** file with **camel.springboot.* properties**.

4. Optional. To refer to a custom bean by using the bean's ID name, configure the options in the **src/main/resources/application.properties** (or the **application.yml**) file. The following example shows how the xslt component refers to a custom bean by using the bean ID.

    a. Refer to a custom bean by the id **myExtensionFactory**.

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

    b. Then create the custom bean using Spring Boot @Bean annotation.

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

    Or, for a Jackson ObjectMapper, in the **camel-jackson** data-format:

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

6.5. ABOUT CAMEL CONTEXT AUTO-CONFIGURATION FOR SPRING BOOT

Camel Spring Boot auto-configuration provides a **CamelContext** instance and creates a **SpringCamelContext**. It also initializes and performs shutdown of that context. This Camel context is registered in the Spring application context under **camelContext** bean name and you can access it like other Spring bean. You can access the **camelContext** as shown below.

Example

```
@Configuration
public class MyAppConfig {

  @Autowired
  CamelContext camelContext;
```

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

}
```

6.6. AUTO-DETECTING CAMEL ROUTES IN SPRING BOOT APPLICATIONS

Camel auto-configuration collects all the **RouteBuilder** instances from the Spring context and automatically injects them into the **CamelContext**. This simplifies the process of creating a new Camel route with the Spring Boot starter. You can create the routes as follows:

Example

Add the **@Component** annotated class to your classpath.

```
@Component
public class MyRouter extends RouteBuilder {

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

}
```

Or create a new route **RouteBuilder** bean in your **@Configuration** class.

```
@Configuration
public class MyRouterConfiguration {

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

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

    };
  }

}
```

6.7. CONFIGURING CAMEL PROPERTIES FOR CAMEL SPRING BOOT AUTO-CONFIGURATION

Spring Boot auto-configuration connects to the Spring Boot external configuration such as properties placeholders, OS environment variables, or system properties with Camel properties support.

Procedure

1. Define the properties either in the **application.properties** file:

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

Or set the Camel properies as the system properties, for example:

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

2. Use the configured properties as placeholders in Camel route as follows.

```
@Component
public class MyRouter extends RouteBuilder {

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

}
```

6.8. CONFIGURING CUSTOM CAMEL CONTEXT

To perform operations on the **CamelContext** bean created by Camel Spring Boot auto-configuration, register a **CamelContextConfiguration** instance in your Spring context.

Procedure

- Register an instance of **CamelContextConfiguration** in the Spring context as shown below.

```
@Configuration
public class MyAppConfig {

  ...

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

}
```

The **CamelContextConfiguration** and **beforeApplicationStart(CamelContext)** methods are called before the Spring context is started, so the **CamelContext** instance that is passed to this callback is fully auto-configured. You can add many instances of **CamelContextConfiguration** into your Spring context and all of them will be executed.

6.9. DISABLING JMX IN THE AUTO-CONFIGURED CAMELCONTEXT

To disable JMX in the auto-configured **CamelContext**, you can use the **camel.springboot.jmxEnabled** property as JMX is enabled by default.

Procedure

- Add the following property to your **application.properties** file and set it to **false**:

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

6.10. INJECTING AUTO-CONFIGURED CONSUMER AND PRODUCER TEMPLATES INTO SPRING-MANAGED BEANS

Camel auto configuration provides pre-configured **ConsumerTemplate** and **ProducerTemplate** instances. You can inject them into your Spring-managed beans.

Example

```
@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());
  }

}
```

By default consumer templates and producer templates come with the endpoint cache sizes set to 1000. You can change these values by setting the following Spring properties to the desired cache size, for example:

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

6.11. ABOUT THE AUTO-CONFIGURED TYPECONVERTER IN THE SPRING CONTEXT

Camel auto configuration registers a **TypeConverter** instance named **typeConverter** in the Spring context.

Example

```
@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 TYPE CONVERSION API BRIDGE

Spring consist of a powerful type conversion API. Spring API is similar to the Camel type converter API. Due to the similarities between the two APIs Camel Spring Boot automatically registers a bridge converter (**SpringTypeConverter**) that delegates to the Spring conversion API. This means that out-of-the-box Camel will treat Spring Converters similar to Camel.

This allows you to access both Camel and Spring converters using the Camel **TypeConverter** API, as shown below:

Example

```
@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());
  }

}
```

Here, Spring Boot delegates conversion to the Spring's **ConversionService** instances available in the application context. If no **ConversionService** instance is available, Camel Spring Boot auto configuration creates an instance of **ConversionService**.

6.13. DISABLING TYPE CONVERSIONS FEATURES

To disable the Camel Spring Boot type conversion features, set the **camel.springboot.typeConversion** property to **false**. When this property is set to **false**, the auto-configuration does not register a type converter instance and does not enable the delegation of type conversion to the Spring Boot type conversion API.

Procedure

- To disable the type conversion features of Camel Spring Boot component, set the **camel.springboot.typeConversion** property to **false** as shown below:

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

6.14. ADDING XML ROUTES TO THE CLASSPATH FOR AUTO-CONFIGURATION

By default, the Camel Spring Boot component auto-detects and includes the Camel XML routes that are in the classpath in the **camel** directory. You can configure the directory name or disable this feature using the configuration option.

Procedure

- Configure the Camel Spring Boot XML routes in the classpath as follows.

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

NOTE

The XML files should define the Camel XML route elements and not **CamelContext** elements, for example:

```
<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>
```

Using Spring XML files

To use Spring XML files with the <camelContext>, you can configure a Camel context in the Spring XML file or in the **application.properties** file. To set the name of the Camel context and turn on the stream caching, add the following in the **application.properties** file:

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

6.15. ADDING XML REST-DSL ROUTES FOR AUTO-CONFIGURATION

The Camel Spring Boot component auto-detects and embeds the Camel Rest-DSL XML routes that are added in the classpath under the **camel-rest** directory. You can configure the directory name or disable this feature using the configuration option.

Procedure

- Configure the Camel Spring Boot Rest-DSL XML routes in the classpath as follows.

```
// turn off
camel.springboot.xmlRests = false
```

```
// scan in the com/foo/routes classpath
camel.springboot.xmlRests = classpath:com/foo/rests/*.xml
```

NOTE

The Rest-DSL XML files should define the Camel XML REST elements and not **CamelContext** elements, for example:

```
<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. TESTING WITH CAMEL SPRING BOOT

When Camel runs on the Spring Boot, Spring Boot automatically embeds Camel and all its routes, which are annotated with **@Component**. When testing with Spring Boot use **@SpringBootTest** instead of **@ContextConfiguration** to specify which configuration class to use.

When you have multiple Camel routes in different RouteBuilder classes, the Camel Spring Boot component automatically embeds all these routes when running the application. Hence, when you wish to test routes from only one RouteBuilder class you can use the following patterns to include or exclude which RouteBuilders to enable:

- java-routes-include-pattern: Used for including RouteBuilder classes that match the pattern.
- java-routes-exclude-pattern: Used for excluding RouteBuilder classes that match the pattern. Exclude takes precedence over include.

Procedure

1. Specify the **include** or **exclude** patterns in your unit test classes as properties to **@SpringBootTest** annotation, as shown below:

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

In the **FooTest** class, the include pattern is **/**Foo***, which represents an Ant style pattern. Here, the pattern starts with a double asterisk, which matches with any leading package name. /**Foo*** means the class name must start with Foo, for example, FooRoute.

2. Run the test using the following maven command:

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

Additional Resources

- Configuring Camel
- Component
- Endpoint
- Getting Started

6.17. USING SPRING BOOT, APACHE CAMEL AND EXTERNAL MESSAGING BROKERS

Fuse uses external messaging brokers. See Supported Configurations for more information about the supported broker, client and Camel component combinations.

The Camel component must be connected to the JMS connection-factory. The example below shows how to connect the **camel-amqp** component to a JMS connection-factory.

```
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);
```

CHAPTER 7. PATCHING RED HAT FUSE APPLICATION

Using the new **patch-maven-plugin** mechanism you can apply a patch to your Red Hat Fuse application. This mechanism allows you to change the individual versions provided by different Red Hat Fuse BOMS, for example, **fuse-springboot-bom** and **fuse-karaf-bom**.

7.1. ABOUT PATCH-MAVEN-PLUGIN

The **patch-maven-plugin** performs following operations:

- Retrieve the patch metadata related to current Red Hat Fuse BOMs.
- Apply the version changes to **<dependencyManagement>** imported from the BOMs.

After the **patch-maven-plugin** fetches the metadata, it iterates through all managed and direct dependencies of the project where the plugin was declared and replaces the dependency versions (if they match) using CVE/patch metadata. After versions are replaced, Maven build continues and progresses through standard Maven project stages.

7.2. APPLYING PATCH TO RED HAT FUSE APPLICATIONS

The purpose of **patch-maven-plugin** is to update the versions of the dependencies listed in the Red Hat Fuse BOM to the versions specified in the patch metadata of the patch that you wish to apply to your applications.

Procedure

Following procedure explains how to apply the patch to your application.

1. Add **patch-maven-plugin** to your project's **pom.xml** file. The version of the **patch-maven-plugin** must be the same as the version of the Fuse BOM.

```
<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. When you run any one of the **mvn clean deploy** or **mvn validate** or **mvn dependency:tree** commands, the plugin searches through the project modules to check whether one of Red Hat Fuse BOMs is used. Only two are considered as the supported BOMs:
   - **org.jboss.redhat-fuse:fuse-karaf-bom**: for Fuse Karaf BOM
   - **org.jboss.redhat-fuse:fuse-springboot-bom**: for Fuse Spring Boot BOM
3. If none of the above BOMs are found, the plugin will display following messages:

```
$ 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. If both Fuse BOMs are found, the **patch-maven-plugin** stops with the following warning:

```
$ 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. The **patch-maven-plugin** attempts to fetch one of the following Maven metadata values.

   - For the projects with Fuse Karaf BOM, the **org.jboss.redhat-fuse/fuse-karaf-patch-metadata/maven-metadata.xml** is resolved. This is the metadata for the artifact with the **org.jboss.redhat-fuse:fuse-karaf-patch-metadata:RELEASE** coordinates.

   - For the projects with Fuse Spring Boot BOM project, the **org.jboss.redhat-fuse/fuse-springboot-patch-metadata/maven-metadata.xml** is resolved. This is the metadata for the artifact with the **org.jboss.redhat-fuse:fuse-springboot-patch-metadata:RELEASE** coordinates.

     Example metadata generated by Maven

```
<?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. The **patch-maven-plugin** parses the metadata to select the version which is applicable to the current project. This is possible only for the Maven projects using Fuse BOM with version **7.8.xxx**. Only the metadata that matches the version range 7.8, 7.9 or later is applicable and only the latest version of the metadata is fetched.

7. The **patch-maven-plugin** collects a list of remote Maven repositories to be used when

downloading the patch metadata identified by **groupId**, **artifactId** and **version** found in previous steps. These Maven repositories are the ones that are listed in the project's **<repositories>** elements in the active profiles and also the repositories from **settings.xml** file.

```
$ 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. Optionally, if you wish to use a offline repository, you can use **-Dpatch** option to specify a ZIP file which is produced by **fuse-karaf/fuse-karaf-patch-repository** or **fuse-springboot/fuse-springboot-patch-repository** modules of **jboss-fuse/redhat-fuse** project. These ZIP files have the same internal structure as the Maven repository structure. For example,

```
$ 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. Whether the metadata comes from remote repository, local repository or ZIP file, it is analyzed by the **patch-maven-plugin**. The fetched metadata contains list of CVEs and for each CVE we have a list of affected Maven artifacts (specified by glob patterns and version ranges) together with a version that contains a fix for a given CVE. For example,

```
<?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. Finally a list of fixes specified in patch metadata is consulted when iterating over all managed dependencies in current project. These dependencies (and managed dependencies) that match are changed to fixed versions. For example:

```
$ 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

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

Skipping the patch

In case when you do not wish to apply a specific patch to your project, the **patch-maven-plugin** provides a **skip** option. Assuming that you have already added **patch-maven-plugin** to project's **pom.xml** file, and you don't wish for the versions to be altered, you can use one of the following method to skip the patch.

- Add the skip option to your project's **pom.xml** file as follows.

```
<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>
```

- Or use **-DskipPatch** option when running the **mvn** command as follows.

```
$ 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 ]---------------------------------
...
```

As shown in the above output, the **patch-maven-plugin** was not invoked, which results in the patch not being applied to the application.

APPENDIX A. PREPARING TO USE MAVEN

This section gives a brief overview of how to prepare Maven for building Red Hat Fuse projects and introduces the concept of Maven coordinates, which are used to locate Maven artifacts.

A.1. PREPARING TO SET UP MAVEN

Maven is a free, open source, build tool from Apache. Typically, you use Maven to build Fuse applications.

Procedure

1. Download the latest version of Maven from the Maven download page .
2. Ensure that your system is connected to the Internet.
   While building a project, the default behavior is that Maven searches external repositories and downloads the required artifacts. Maven looks for repositories that are accessible over the Internet.

   You can change this behavior so that Maven searches only repositories that are on a local network. That is, Maven can run in an offline mode. In offline mode, Maven looks for artifacts in its local repository. See Section A.3, “Using local Maven repositories” .

A.2. ADDING RED HAT REPOSITORIES TO MAVEN

To access artifacts that are in Red Hat Maven repositories, you need to add those repositories to Maven’s **settings.xml** file. Maven looks for the **settings.xml** file in the **.m2** directory of the user’s home directory. If there is not a user specified **settings.xml** file, Maven uses the system-level **settings.xml** file at **M2_HOME/conf/settings.xml**.

Prerequisite

You know the location of the **settings.xml** file in which you want to add the Red Hat repositories.

Procedure

In the **settings.xml** file, add **repository** elements for the Red Hat repositories as shown in this example:

```
<?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. USING LOCAL MAVEN REPOSITORIES

If you are running a container without an Internet connection, and you need to deploy an application that has dependencies that are not available offline, you can use the Maven dependency plug-in to download the application's dependencies into a Maven offline repository. You can then distribute this customized Maven offline repository to machines that do not have an Internet connection.

Procedure

1. In the project directory that contains the **pom.xml** file, download a repository for a Maven project by running a command such as the following:

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

   In this example, Maven dependencies and plug-ins that are required to build the project are downloaded to the **/tmp/my-project** directory.

2. Distribute this customized Maven offline repository internally to any machines that do not have an Internet connection.

A.4. ABOUT MAVEN ARTIFACTS AND COORDINATES

In the Maven build system, the basic building block is an *artifact*. After a build, the output of an artifact is typically an archive, such as a JAR or WAR file.

A key aspect of Maven is the ability to locate artifacts and manage the dependencies between them. A *Maven coordinate* is a set of values that identifies the location of a particular artifact. A basic coordinate has three values in the following form:

groupId:artifactId:version

Sometimes Maven augments a basic coordinate with a *packaging* value or with both a *packaging* value and a *classifier* value. A Maven coordinate can have any one of the following forms:

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

Here are descriptions of the values:

groupdId

Defines a scope for the name of the artifact. You would typically use all or part of a package name as a group ID. For example, **org.fusesource.example**.

artifactId

Defines the artifact name relative to the group ID.

version

Specifies the artifact's version. A version number can have up to four parts: **n.n.n.n**, where the last part of the version number can contain non-numeric characters. For example, the last part of **1.0-SNAPSHOT** is the alphanumeric substring, **0-SNAPSHOT**.

packaging

Defines the packaged entity that is produced when you build the project. For OSGi projects, the packaging is **bundle**. The default value is **jar**.

classifier

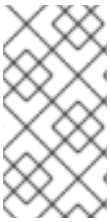
Enables you to distinguish between artifacts that were built from the same POM, but have different content.

Elements in an artifact's POM file define the artifact's group ID, artifact ID, packaging, and version, as shown here:

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

To define a dependency on the preceding artifact, you would add the following **dependency** element to a POM file:

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



NOTE

It is not necessary to specify the **bundle** package type in the preceding dependency, because a bundle is just a particular kind of JAR file and **jar** is the default Maven package type. If you do need to specify the packaging type explicitly in a dependency, however, you can use the **type** element.

APPENDIX B. SPRING BOOT MAVEN PLUGIN

Spring Boot Maven plugin provides the Spring Boot support in Maven and allows you to package the executable **jar** or **war** archives and run an application **in-place**.

B.1. SPRING BOOT MAVEN PLUGIN GOALS

The Spring Boot Maven plugin includes the following goals:

- **spring-boot:run** runs your Spring Boot application.
- **spring-boot:repackage** repackages your **.jar** and **.war** files to be executable.
- **spring-boot:start** and **spring-boot:stop** both are used to manage the lifecycle of your Spring Boot application.
- **spring-boot:build-info** generates build information that can be used by the Actuator.

B.2. USING SPRING BOOT MAVEN PLUGIN

You can find general instructions on how to use the Spring Boot Plugin at: **https://docs.spring.io/spring-boot/docs/current/maven-plugin/reference/htmlsingle/#using**. The following examples illustrates the usage of the **spring-boot-maven-plugin** for Spring Boot.

- Spring Boot 2 Example

NOTE

For more information on Spring Boot Maven Plugin, refer the **https://docs.spring.io/spring-boot/docs/current/maven-plugin/reference/htmlsingle/** link.

B.2.1. Using Spring Boot Maven plugin for Spring Boot 2

The following example illustrates the usage of the **spring-boot-maven-plugin** for Spring Boot 2.

Example

```
<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>
```