Red Hat Fuse 7.4

Deploying into Spring Boot

Building and running Spring Boot applications in standalone mode

Red Hat Fuse 7.4 Deploying into Spring Boot

Building and running 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

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 runs directly inside the Java Virtual Machine (JVM). That is, 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.

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 container is also minimal, being just the JVM itself.

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 that only dependencies defined in the Fuse BOM are supported by Red Hat.

- *The Spring Boot Maven Plug-In* — implements the build process for a standalone Spring Boot application in Maven. This plug-in is responsible for packaging your Spring Boot application as an executable Jar file.

1.1.1. Deploying into Spring Boot 2

Currently, Spring Boot 2 is not supported on OpenShift.

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

> **NOTE**
>
> For more information on all of the differences between Spring Boot 1 and Spring Boot 2, see Spring Boot's Migration Guide and Spring Boot 2 Release Notes.

1.1.1.1. New Camel Components for Spring Boot 2

Spring Boot 2 supports Camel version **2.23** and thus supports some new camel components that are not available for Spring Boot 1:

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 2.2.1, "Circuit Breaker booster" – An example of enabling a distributed application to cope with interruptions to network connectivity and temporary unavailability of backend services.

- Section 2.2.2, "Externalized Configuration booster" – An example of how to externalize configuration for an Apache Camel route.

- Section 2.2.3, "REST API booster" – An example that introduces the mechanics of interacting with a remote (exposed by Apache Camel) service using the HTTP protocol.

2.1. 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.

2.2. 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 Apache Maven 3.3.x or later.

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. Click the **Log in or register** button and then log in.

4. On the **Launcher** page, click the **Deploy an Example Application** button.

5. On the **Create Example Application** page, type the name, **fuse-circuit-breaker**, in the **Create Example Application as** field.

6. Click **Select an Example**.

7. 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 **7.3.0 (Red Hat Fuse)** (do not select the **2.21.2 (Community)** version).

   c. Click **Save**.

8. On the **Create Example Application** page, click **Download**.

9. When you see the **Your Application is Ready** dialog, click **Download.zip**. Your browser downloads the generated booster project (packaged as a ZIP file).

10. Use an archive utility to extract the generated project to a convenient location on your local file system.

2.2.1. 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.

2.2.1.1. Build and run the Circuit Breaker booster

After you follow the Section 2.2, "Generating your booster project" steps for the **Circuit Breaker** mission, 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:

Greeting service

Stop   Start   Clear

Results:

```
{"greetings":"Hello, Jacopo"}
{"greetings":"Hello, Jacopo"}
{"greetings":"Hello, Jacopo"}
{"greetings":"Hello, Jacopo"}
{"greetings":"Hello, Jacopo"}
{"greetings":"Hello, Jacopo"}
{"greetings":"Hello, Jacopo"}
{"greetings":"Hello, Jacopo"}
{"greetings":"Hello, Jacopo"}
```

This page also provides a link to the Hystrix dashboard, which monitors the state of the circuit breaker.
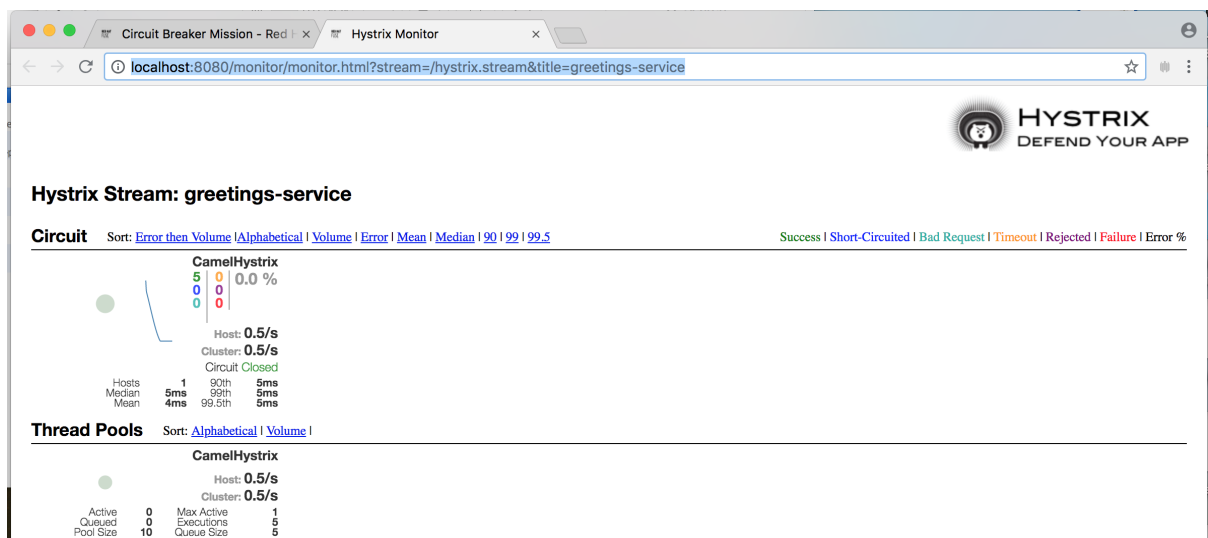
5. To demonstrate the circuit breaker functionality provided by Camel Hystrix, kill the backend name service by pressing **Ctrl+C** while in the window of the shell prompt where the name service is running.

   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:

Greeting service

[Stop] [Start] [Clear]

Results:

```
{"greetings":"Hello, default fallback"}
{"greetings":"Hello, default fallback"}
{"greetings":"Hello, default fallback"}
{"greetings":"Hello, default fallback"}
{"greetings":"Hello, default fallback"}
{"greetings":"Hello, default fallback"}
{"greetings":"Hello, default fallback"}
{"greetings":"Hello, default fallback"}
{"greetings":"Hello, default fallback"}
{"greetings":"Hello, default fallback"}
{"greetings":"Hello, Jacopo"}
{"greetings":"Hello, Jacopo"}
{"greetings":"Hello, Jacopo"}
{"greetings":"Hello, Jacopo"}
{"greetings":"Hello, Jacopo"}
{"greetings":"Hello, Jacopo"}
```

2.2.2. 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.

2.2.2.1. Build and run the Externalized Configuration booster

After you follow the Section 2.2, "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:

Greeting Service

Clear

Results:

```
{"greetings":"Hello, default"}
{"greetings":"Hello, default"}
{"greetings":"Hello, default"}
{"greetings":"Hello, default"}
{"greetings":"Hello, default"}
{"greetings":"Hello, default"}
{"greetings":"Hello, default"}
{"greetings":"Hello, default"}
{"greetings":"Hello, default"}
{"greetings":"Hello, default"}
```

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.

Greeting Service

`Clear`

Results:

```
{"greetings":"Hello, Thomas"}
{"greetings":"Hello, Thomas"}
{"greetings":"Hello, Thomas"}
{"greetings":"Hello, Thomas"}
{"greetings":"Hello, default"}
{"greetings":"Hello, default"}
{"greetings":"Hello, default"}
{"greetings":"Hello, default"}
```

2.2.3. 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.

This 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.

2.2.3.1. Build and run the REST API booster

After you follow the Section 2.2, "Generating your booster project" steps for the **REST API** mission, 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:
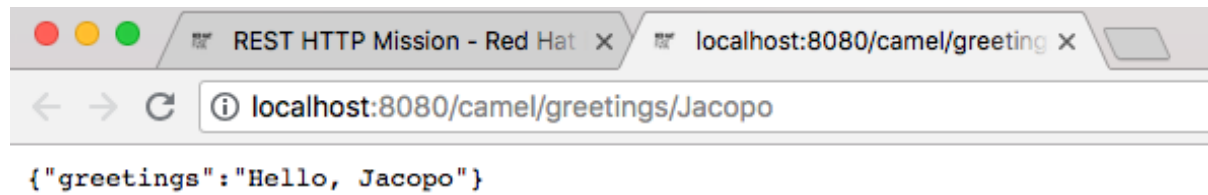
6. To change the value of the **{name} `parameter, change the URL. For example, to change
   the name to Thomas, use this URL: `localhost:8080/camel/greetings/Thomas**.
   The updated JSON response appears in the browser window:
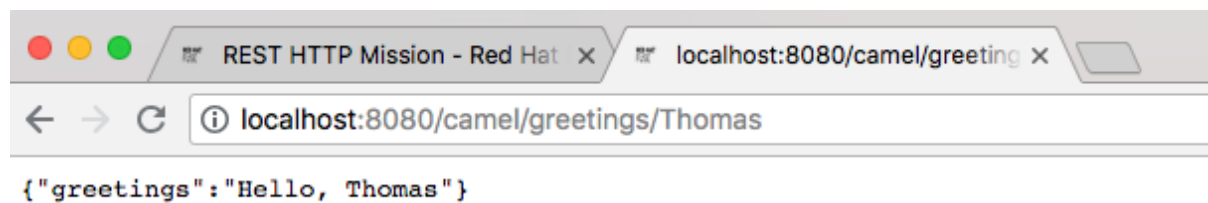
7. To view the REST API's Swagger page, click the API Swagger page button.
   The API swagger page opens in a browser window.
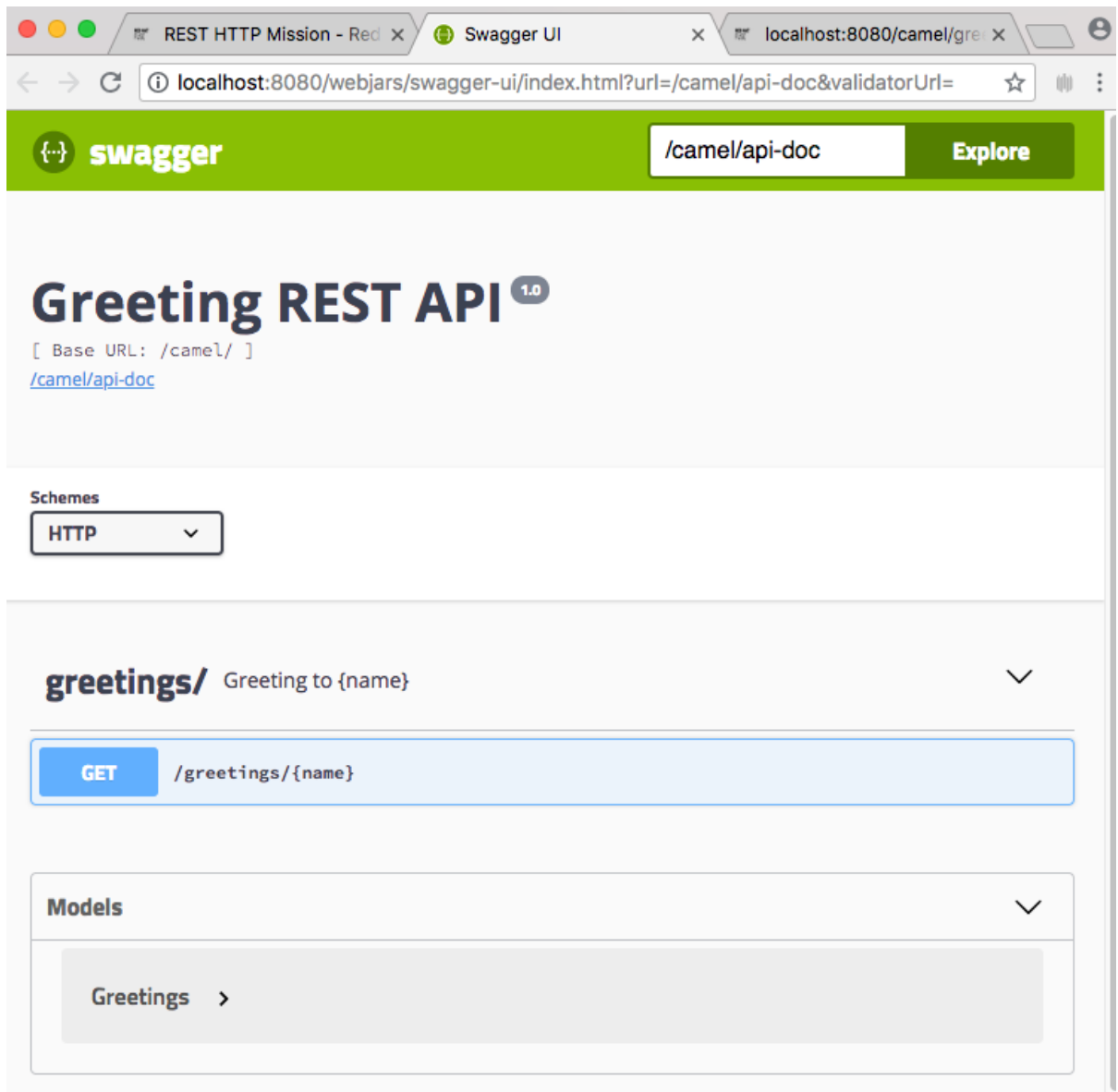
CHAPTER 3. BUILDING WITH MAVEN

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.

3.1. 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.

3.1.1. 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.

3.1.2. 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.

3.2. DEPEND ON THE BOM FOR SPRING BOOT

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.

3.2.1. 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.
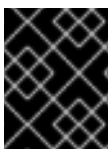
> **IMPORTANT**
>
> Ensure you are using the correct Fuse BOM based on the version of Spring Boot you are using (Spring Boot 1 or Spring Boot 2).

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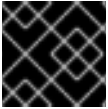
> **IMPORTANT**
>
> Only the set of dependencies defined by a Fuse BOM are supported by Red Hat.

3.2.1.1. 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 and Spring Boot 1 below:

- Spring Boot 2 BOM

- Spring Boot 1 BOM

```xml
<?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>7.4.0.fuse-sb2-740019-redhat-00005</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>
```

```xml
<?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>7.4.0.fuse-740036-redhat-00002</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 becomes 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.

3.2.2. 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:

```
<?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>7.4.0.fuse-740036-redhat-00002</fuse.version>

  </properties>
  ...
  <build>
    <plugins>
      <plugin>
        <groupId>org.jboss.redhat-fuse</groupId>
```

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

CHAPTER 4. APACHE CAMEL IN SPRING BOOT

4.1. INTRODUCTION TO CAMEL SPRING BOOT

The Camel Spring Boot component provides auto configuration for Apache Camel. Auto-configuration of the Camel context auto-detects 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.

Every Camel Spring Boot application should use **dependencyManagement** with productized versions, see quickstart pom. Versions that are tagged later can be omitted to not override the versions from BOM.

```
<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>
```
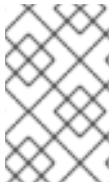
NOTE

camel-spring-boot jar comes with the **spring.factories** file which allows you to add that dependency into your classpath and hence Spring Boot will automatically auto-configure Camel.

4.2. INTRODUCTION TO CAMEL SPRING BOOT STARTER

Apache Camel includes a Spring Boot starter module that allows you to develop Spring Boot applications using starters.
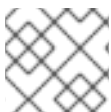
NOTE

For more details, see sample application in the source code.

To use the starter, add the following snippet to your Spring Boot **pom.xml** file:

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

The starter allows you to add 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");
    }
}
```

You can customize the Camel application in the **application.properties** or **application.yml** file.

Camel Spring Boot now supports referring to bean by the id name in the configuration files (application.properties or yaml file) when you configure any of the Camel starter components. In the **src/main/resources/application.properties** (or yaml) file you can now easily configure the options on the Camel that refers to other beans by referring to the beans ID name. For example, the xslt component can refer to a custom bean using the bean ID as follows:

Refer to a custom bean by the id myExtensionFactory as follows:

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

Which you can then create using Spring Boot @Bean annotation as follows:

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

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

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

4.3. AUTO-CONFIGURED CAMEL CONTEXT

Camel 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.

For example, you can access the **camelContext** as shown below:

```
@Configuration
public class MyAppConfig {

  @Autowired
  CamelContext camelContext;

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

}
```

4.4. AUTO-DETECTING CAMEL ROUTES

Camel auto configuration collects all the **RouteBuilder** instances from the Spring context and automatically injects them into the **CamelContext**. It simplifies the process of creating new Camel route with the Spring Boot starter. You can create the routes by adding 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");
  }

}
```

To create a new route **RouteBuilder** bean in your **@Configuration** class, see below:

```
@Configuration
public class MyRouterConfiguration {

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

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

    };
  }

}
```

4.5. CAMEL PROPERTIES

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

These properties are defined in **application.properties** file:

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

Use as system property

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

Use as placeholders in Camel route:

```
@Component
public class MyRouter extends RouteBuilder {
```

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

}
```

4.6. CUSTOM CAMEL CONTEXT CONFIGURATION

To perform operations on **CamelContext** bean created by Camel auto configuration, you need to register **CamelContextConfiguration** instance in your 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
      }
    };
  }

}
```

> **NOTE**
>
> The method **CamelContextConfiguration** and **beforeApplicationStart(CamelContext)** will be called before the Spring context is started, so the **CamelContext** instance 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.

4.7. DISABLING JMX

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

For example, you could add the following property to your **application.properties** file:

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

4.8. AUTO-CONFIGURED CONSUMER AND PRODUCER TEMPLATES

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

```
@Component
```

```java
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 those values using the following Spring properties:

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

4.9. AUTO-CONFIGURED TYPECONVERTER

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

```java
@Component
public class InvoiceProcessor {

  @Autowired
  private TypeConverter typeConverter;

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

}
```

4.10. SPRING TYPE CONVERSION API BRIDGE

Spring consist of 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. That 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:

```java
@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**.

4.11. DISABLING TYPE CONVERSIONS FEATURES

To disable registering type conversion features of Camel Spring Boot such as **TypeConverter** instance or Spring bridge, set the **camel.springboot.typeConversion** property to **false** as shown below:

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

4.12. ADDING XML ROUTES

By default, you can put Camel XML routes in the classpath under the directory camel, which **camel-spring-boot** will auto detect and include. From **Camel version 2.17** onwards you can configure the directory name or disable this feature using the configuration option, as shown below:

```
// 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 be Camel XML routes and not **CamelContext** such as:

```
<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>
```

When using Spring XML files with <camelContext>, you can configure Camel in the Spring XML file as well as in the application.properties file. For example, to set a name on Camel and turn On the stream caching, add:

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

4.13. ADDING XML REST-DSL

By default, you can put Camel Rest-DSL XML routes in the classpath under the directory **camel-rest**, which **camel-spring-boot** will auto detect and include. You can configure the directory name or disable this feature using the configuration option, as shown below:

```
// 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 be Camel XML rests and not **CamelContext** such as:

```xml
<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>
```

4.14. TESTING WITH CAMEL SPRING BOOT

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

When you have multiple Camel routes in different RouteBuilder classes, Camel Spring Boot will include all these routes. 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.

You can specify these 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 double asterisk, which matches with any leading package name. **/Foo*** means the class name must start with Foo, for example, FooRoute. You can run a test using following maven command:

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

4.15. SEE ALSO

- Configuring Camel

- Component

- Endpoint

- Getting Started

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

4.16.1. Using an external messaging broker.

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);
```

APPENDIX A. PREPARING TO USE MAVEN

A.1. OVERVIEW

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.2. PREREQUISITES

In order to build a project using Maven, you must have the following prerequisites:

- **Maven installation** — Maven is a free, open source build tool from Apache. You can download the latest version from the Maven download page .

- **Network connection** — whilst performing a build, Maven dynamically searches external repositories and downloads the required artifacts on the fly. By default, Maven looks for repositories that are accessed over the Internet. You can change this behavior so that Maven will prefer searching repositories that are on a local network.

> **NOTE**
>
> Maven can run in an offline mode. In offline mode Maven only looks for artifacts in its local repository.

A.3. ADDING THE RED HAT MAVEN REPOSITORIES

In order to access artifacts from the Red Hat Maven repositories, you need to add them to Maven's **settings.xml** file. Maven looks for your **settings.xml** file in the **.m2** directory of the user's home directory. If there is not a user specified **settings.xml** file, Maven will use the system-level **settings.xml** file at **M2_HOME/conf/settings.xml**.

To add the Red Hat repositories to Maven's list of repositories, you can either create a new **.m2/settings.xml** file or modify the system-level settings. In the **settings.xml** file, add **repository** elements for the Red Hat repositories as shown in Adding the Red Hat Fuse Repositories to Maven .

Adding the Red Hat Fuse Repositories to Maven

```xml
<?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.4. ARTIFACTS

The basic building block in the Maven build system is an *artifact*. The output of an artifact, after performing a Maven build, is typically an archive, such as a JAR or a WAR.

A.5. MAVEN COORDINATES

A key aspect of Maven functionality is the ability to locate artifacts and manage the dependencies between them. Maven defines the location of an artifact using the system of *Maven coordinates*, which uniquely define the location of a particular artifact. A basic coordinate tuple has the form, **{groupId, artifactId, version}**. Sometimes Maven augments the basic set of coordinates with the additional coordinates, *packaging* and *classifier*. A tuple can be written with the basic coordinates, or with the additional *packaging* coordinate, or with the addition of both the *packaging* and *classifier* coordinates, as follows:

> *groupdId*:*artifactId*:*version*
> *groupdId*:*artifactId*:*packaging*:*version*
> *groupdId*:*artifactId*:*packaging*:*classifier*:*version*

Each coordinate can be explained as follows:

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.

The group ID, artifact ID, packaging, and version are defined by the corresponding elements in an artifact's POM file. For example:

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

For example, to define a dependency on the preceding artifact, you could add the following **dependency** element to a POM:

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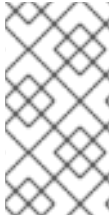
NOTE

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 PLUG-IN

B.1. SPRING BOOT MAVEN PLUGIN OVERVIEW

This appendix describes the Spring Boot Maven Plugin. It 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.2. GOALS

The Spring Boot Plugin includes the following goals:

1. **spring-boot:run** runs your Spring Boot application.

2. **spring-boot:repackage** repackages your **.jar** and **.war** files to be executable.

3. **spring-boot:start** and **spring-boot:stop** both are used to manage the lifecycle of your Spring Boot application.

4. **spring-boot:build-info** generates build information that can be used by the Actuator.

B.3. USAGE

You can find general instructions on how to use the Spring Boot Plugin at: **http://docs.spring.io/spring-boot/docs/current/maven-plugin/usage.html**. The following example illustrates the usage of the **spring-boot-maven-plugin** for Spring Boot.

- Spring Boot 2 Example

- Spring Boot 1 Example

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

B.3.1. Spring Boot Maven Plugin for Spring Boot 2

```xml
<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>7.4.0.fuse-sb2-740019-redhat-00005</fuse.bom.version>

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

```xml
      <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>
```

B.3.2. Spring Boot Maven Plugin for Spring Boot 1

```
<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>7.4.0.fuse-740036-redhat-00002</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>
```

```xml
    <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>
```