Red Hat build of Apache Camel 4.4

Getting Started with Red Hat build of Apache Camel for Spring Boot

Last Updated: 2024-07-02

Legal Notice

Copyright © 2024 Red Hat, Inc.

The text of and illustrations in this document are licensed by Red Hat under a Creative Commons Attribution–Share Alike 3.0 Unported license ("CC-BY-SA"). An explanation of CC-BY-SA is available at
http://creativecommons.org/licenses/by-sa/3.0/
. In accordance with CC-BY-SA, if you distribute this document or an adaptation of it, you must provide the URL for the original version.

Red Hat, as the licensor of this document, waives the right to enforce, and agrees not to assert, Section 4d of CC-BY-SA to the fullest extent permitted by applicable law.

Red Hat, Red Hat Enterprise Linux, the Shadowman logo, the Red Hat logo, JBoss, OpenShift, Fedora, the Infinity logo, and RHCE are trademarks of Red Hat, Inc., registered in the United States and other countries.

Linux ® is the registered trademark of Linus Torvalds in the United States and other countries.

Java ® is a registered trademark of Oracle and/or its affiliates.

XFS ® is a trademark of Silicon Graphics International Corp. or its subsidiaries in the United States and/or other countries.

MySQL ® is a registered trademark of MySQL AB in the United States, the European Union and other countries.

Node.js ® is an official trademark of Joyent. Red Hat is not formally related to or endorsed by the official Joyent Node.js open source or commercial project.

The OpenStack ® Word Mark and OpenStack logo are either registered trademarks/service marks or trademarks/service marks of the OpenStack Foundation, in the United States and other countries and are used with the OpenStack Foundation's permission. We are not affiliated with, endorsed or sponsored by the OpenStack Foundation, or the OpenStack community.

All other trademarks are the property of their respective owners.

Abstract

This guide introduces Red Hat build of Apache Camel and explains the various ways to create and deploy an application using Red Hat build of Apache Camel.

Table of Contents

PREFACE 4
MAKING OPEN SOURCE MORE INCLUSIVE 4

CHAPTER 1. GETTING STARTED WITH RED HAT BUILD OF APACHE CAMEL FOR SPRING BOOT 5
1.1. RED HAT BUILD OF APACHE CAMEL FOR SPRING BOOT STARTERS 5
1.1.1. Spring Boot configuration support 6
1.1.2. Adding Camel routes 6
1.1.3. Using Domain Specific Languages 6
1.1.3.1. Advantages of DSLs 6
1.1.3.2. Comparing different DSLs 7
1.2. SPRING BOOT 11
1.2.1. Camel Spring Boot Starter 11
1.2.2. Spring Boot Auto-configuration 12
1.2.3. Auto-configured Camel context 12
1.2.4. Auto-detecting Camel routes 12
1.2.5. Camel properties 13
1.2.6. Custom Camel context configuration 14
1.2.7. Auto-configured consumer and producer templates 14
1.2.8. Auto-configured TypeConverter 15
1.2.8.1. Spring type conversion API bridge 15
1.2.9. Keeping the application alive 15
1.2.10. Adding XML routes 16
1.2.11. Testing the JUnit 5 way 16
1.3. COMPONENT STARTERS 17
1.4. STARTER CONFIGURATION 31
1.4.1. Using External Configuration 31
1.4.2. Using Beans 32
1.5. GENERATING A CAMEL FOR SPRING BOOT APPLICATION USING MAVEN 32
1.6. DEPLOYING A CAMEL SPRING BOOT APPLICATION TO OPENSHIFT 33
1.7. APPLYING PATCH TO RED HAT BUILD OF APACHE CAMEL FOR SPRING BOOT 34
1.8. CAMEL REST DSL OPENAPI MAVEN PLUGIN 38
1.8.1. Adding plugin to Maven pom.xml 38
1.8.2. camel-restdsl-openapi:generate 39
1.8.3. Options 39
1.8.4. Spring Boot Project with Servlet component 40
1.8.5. camel-restdsl-openapi:generate-with-dto 41
1.8.6. Options 41
1.8.7. camel-restdsl-openapi:generate-xml 42
1.8.8. Options 42
1.8.9. camel-restdsl-openapi:generate-xml-with-dto 44
1.8.10. Options 44
1.8.11. camel-restdsl-openapi:generate-yaml 45
1.8.12. Options 45
1.8.13. camel-restdsl-openapi:generate-yaml-with-dto 46
1.8.14. Options 47
1.9. SUPPORT FOR FIPS COMPLIANCE 48
1.9.1. FIPS validation in OpenShift Container Platform 48

CHAPTER 2. SETTING UP MAVEN LOCALLY 49
2.1. PREPARING TO SET UP MAVEN 49
2.2. ADDING RED HAT REPOSITORIES TO MAVEN 49

2.3. BUILDING AN OFFLINE MAVEN REPOSITORY 51
2.4. USING LOCAL MAVEN REPOSITORIES 52
2.5. SETTING MAVEN MIRROR USING ENVIRONMENTAL VARIABLES OR SYSTEM PROPERTIES 53
2.5.1. About Maven mirror 53
2.5.2. Adding Maven mirror to settings.xml 53
2.5.3. Setting Maven mirror using environmental variable or system property 53
2.5.4. Using Maven options to specify Maven mirror url 53
2.6. ABOUT MAVEN ARTIFACTS AND COORDINATES 54

CHAPTER 3. MONITORING CAMEL SPRING BOOT INTEGRATIONS **56**
3.1. ENABLING USER WORKLOAD MONITORING IN OPENSHIFT 56
3.2. DEPLOYING A CAMEL SPRING BOOT APPLICATION 57

CHAPTER 4. USING CAMEL WITH SPRING XML **62**
4.1. USING JAVA DSL WITH SPRING XML FILES 62
4.1.1. Configure Spring Boot Application 62
4.2. SPECIFYING CAMEL ROUTES USING SPRING XML 63
4.3. CONFIGURING COMPONENTS AND ENDPOINTS 63
4.4. USING PACKAGE SCANNING 64
4.5. USING CONTEXT SCANNING 64

CHAPTER 5. XML IO DSL **66**
5.1. EXAMPLE 66
5.2. USING BEANS WITH CONSTRUCTORS 68
5.3. CREATING BEANS FROM FACTORY METHOD 68
5.4. CREATING BEANS FROM BUILDER CLASSES 69
5.5. CREATING BEANS FROM FACTORY BEAN 69
5.6. CREATING BEANS USING SCRIPT LANGUAGE 70
5.7. USING INIT AND DESTROY METHODS ON BEANS 71
5.8. REST AND ROUTES IN THE SAME XML IO DSL FILE 71

PREFACE

MAKING OPEN SOURCE MORE INCLUSIVE

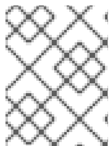
Red Hat is committed to replacing problematic language in our code, documentation, and web properties. We are beginning with these four terms: master, slave, blacklist, and whitelist. Because of the enormity of this endeavor, these changes will be implemented gradually over several upcoming releases.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

CHAPTER 1. GETTING STARTED WITH RED HAT BUILD OF APACHE CAMEL FOR SPRING BOOT

This guide introduces Red Hat build of Apache Camel for Spring Boot and demonstrates how to get started building an application using Red Hat build of Apache Camel for Spring Boot.

1.1. RED HAT BUILD OF APACHE CAMEL FOR SPRING BOOT STARTERS

Camel support for Spring Boot provides auto-configuration of the Camel and starters for many Camel components. The opinionated 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.



NOTE

For information about using a Maven archtype to generate a Camel for Spring Boot application see Generating a Camel for Spring Boot application using Maven .

To get started, you must add the Camel Spring Boot BOM to your Maven **pom.xml** file.

```
<dependencyManagement>

    <dependencies>
        <!-- Camel BOM -->
        <dependency>
            <groupId>com.redhat.camel.springboot.platform</groupId>
            <artifactId>camel-spring-boot-bom</artifactId>
            <version>4.4.0.redhat-00014</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
        <!-- ... other BOMs or dependencies ... -->
    </dependencies>

</dependencyManagement>
```

The **camel-spring-boot-bom** is a basic BOM that contains the list of Camel Spring Boot starter JARs.

Next, add the Camel Spring Boot starter to startup the Camel Context.

```
    <dependencies>
        <!-- Camel Starter -->
        <dependency>
            <groupId>org.apache.camel.springboot</groupId>
            <artifactId>camel-spring-boot-starter</artifactId>
        </dependency>
        <!-- ... other dependencies ... -->
    </dependencies>
```

You must also add the component starters that your Spring Boot application requires. The following example shows how to add the auto-configuration starter to the MQTT5 component.

```
<dependencies>
  <!-- ... other dependencies ... -->
  <dependency>
    <groupId>org.apache.camel.springboot</groupId>
    <artifactId>camel-paho-mqtt5</artifactId>
  </dependency>
</dependencies>
```

1.1.1. Spring Boot configuration support

Each starter lists configuration parameters you can configure in the standard **application.properties** or **application.yml** files. These parameters have the form of **camel.component.[component-name].[parameter]**. For example to configure the URL of the MQTT5 broker you can set:

```
camel.component.paho-mqtt5.broker-url=tcp://localhost:61616
```

1.1.2. Adding Camel routes

Camel routes are detected in the Spring application context, for example a route annotated with **org.springframework.stereotype.Component** will be loaded, added to the Camel context and run.

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

@Component
public class MyRoute extends RouteBuilder {

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

}
```

1.1.3. Using Domain Specific Languages

Apache Camel uses a Java Domain Specific Language or DSL for creating Enterprise Integration Patterns or Routes in a variety of domain-specific languages (DSL) as listed below:

- Java DSL: Java based DSL using the fluent builder style.
- XML DSL: XML based DSL in Camel XML files only.
- Yaml DSL for creating routes using YAML format.

1.1.3.1. Advantages of DSLs

The advantages of using a DSL over general-purpose languages are the following:

- Easier to learn and easier to work with. You can see where the main logic begins and ends.

- Safer code. DSL in Apache Camel has the solid building blocks which binds all the steps together.
- Errors are domain-specific. In case of failures, error descriptions are more explicit and explanatory. Simpler code also means less error-prone code.
- DSLs are designed to be platform-independent. In case of code changes, its impact is delegated to lower layers.

1.1.3.2. Comparing different DSLs

Following section describes the differences between the DSLs and different scenarios where you may use these DSLs.

	Java DSL	XML DSL	YAML DSL

	Java DSL	XML DSL	YAML DSL
Developer tools	• You can use every IDE with Java support. • Red Hat provides the Extension Pack for Apache Camel in VS Code. This pack contains all the necessary extensions to work with Red Hat build of Apache Camel in VS Code. This includes language support for Camel K Java standalone, support for Camel URI completion and diagnostics, and running and debugging Camel routes from the source editor. • Language support and basic Camel textual route debugging. • It provides code assistance and offers a route debugger.	• You can use every IDE with XML support. • Red Hat provides the Extension Pack for Apache Camel in VS Code. This pack contains all the necessary extensions to work with Red Hat build of Apache Camel in VS Code. This includes language support for Camel K Java standalone, support for Camel URI completion and diagnostics, and running and debugging Camel routes from the source editor. • Language support and basic Camel textual route debugging • It provides code assistance and offers a route debugger.	• You can use every IDE with YAML support. • Red Hat provides the Extension Pack for Apache Camel in VS Code. This pack contains all the necessary extensions to work with Red Hat build of Apache Camel in VS Code. This includes language support for Camel K Java standalone, support for Camel URI completion and diagnostics, and running and debugging Camel routes from the source editor. • Language support and basic Camel textual route debugging. • It provides code assistance and offers a route debugger. • It also includes the Kaoto VS Code extension, which offers a visual integration designer.

	Java DSL	XML DSL	YAML DSL
Hawtio / Fuse Console	Hawtio retrieves the routes from the runtime as XML and display the routes regardless of which DSL was used to create the routes.	Hawtio retrieves the routes from the runtime as XML and display the routes regardless of which DSL was used to create the routes.	Hawtio retrieves the routes from the runtime as XML and display the routes regardless of which DSL was used to create the routes.
Software development model	The DSL adopts a fluent builder API.	• Modeling development approach with graphical editor is possible (Eclipse Desktop). • Allows drag-and-drop based development. • Textual-based development is also possible with very mature IDE support.	Harder to write from scratch. A modelling development approach with a graphical editor is possible.
Debugging code	• There are IDE plug-ins that provide step by step DSL debugging over the EIPs. You can step into the RouteBuilder, but it is called only at startup and not during processing. • Breakpoints can be put in Java code of the core Camel classes. • It is possible to add temporary Processors and use the Java debugger.	• There are IDE plug-ins that provide step by step DSL debugging over the EIPs. • Breakpoints can be put in Java code of the core Camel classes.	• There are IDE plug-ins that provide step by step DSL debugging over the EIPs. • Breakpoints can be put in Java code of the core Camel classes.

	Java DSL	XML DSL	YAML DSL
Integration with dependency injection (DI) frameworks	Easier to integrate with any DI framework.	While it is possible to refer to existing beans from DI frameworks in XML DSL, declaring new beans in XML makes these beans exclusive to Camel itself, and not part of the DI framework (for example, Quarkus or Spring Boot).	While it is possible to refer to existing beans from DI frameworks in YAML DSL, declaring new beans in YAML makes these beans exclusive to Camel itself, and not part of the DI framework (for example, Quarkus or Spring Boot).
Team size	More flexible, but harder to read code. Good for small co-located teams that work and support code for a long period.	• Beneficial for large and disparate teams. • Less flexible, making it challenging to create complicated routes.	• Beneficial for large and disparate teams. • Less flexible, making it challenging to create complicated routes.
Team structure	Requires the team to have Java developers for developing Camel integrations. Other team members also required to understand Java in order to read the integration flow.	• XML is a widespread language, and all developers can reuse existing skills when developing with Camel. • It offers a higher level of abstraction and makes it easy to communicate with business developers and support teams.	• YAML is a widespread language, and all developers can reuse existing skills when developing with Camel. • It offers a higher level of abstraction and makes it easy to communicate with business developers and support teams.

	Java DSL	XML DSL	YAML DSL
Developer experience and preference	• More suited to experienced developers as Java is more concise than XML, with inner classes and functional aspects. • Java developers tend to prefer pure Java and annotations rather than XML.	Ideal for new users, as it offers a graphical approach for designing routes.	Ideal for new users, as it offers a graphical approach for designing routes.

1.2. SPRING BOOT

Spring Boot automatically configures Camel for you. The opinionated auto-configuration of the Camel context auto-detects Camel routes available in the Spring context and registers the key Camel utilities (like producer template, consumer template and the type converter) as beans.

Maven users will need to add the following dependency to their **pom.xml** in order to use this component:

```
<dependency>
    <groupId>org.apache.camel.springboot</groupId>
    <artifactId>camel-spring-boot</artifactId>
    <version>4.4.0.redhat-00014</version> <!-- use the same version as your Camel core version -->
</dependency>
```

camel-spring-boot jar comes with the **spring.factories** file, so as soon as you add that dependency into your classpath, Spring Boot will automatically auto-configure Camel for you.

1.2.1. Camel Spring Boot Starter

Apache Camel ships a Spring Boot Starter module that allows you to develop Spring Boot applications using starters. There is a sample application in the source code also.

To use the starter, add the following to your spring boot pom.xml file:

```
<dependency>
    <groupId>org.apache.camel.springboot</groupId>
    <artifactId>camel-spring-boot-bom</artifactId>
    <version>4.4.0.redhat-00014</version> <!-- use the same version as your Camel core version -->
</dependency>
```

Then you can just add classes with your Camel routes such as:

```
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");
    }
}
```

Then these routes will be started automatically.

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

1.2.2. Spring Boot Auto-configuration

When using spring-boot with Spring Boot make sure to use the following Maven dependency to have support for auto configuration:

```
<dependency>
  <groupId>org.apache.camel.springboot</groupId>
  <artifactId>camel-spring-boot-starter</artifactId>
  <version>4.4.0.redhat-00014</version> <!-- use the same version as your Camel core version -->
</dependency>
```

1.2.3. Auto-configured Camel context

The most important piece of functionality provided by the Camel auto-configuration is the **CamelContext** instance. Camel auto-configuration creates a **SpringCamelContext** for you and takes care of the proper initialization and shutdown of that context. The created Camel context is also registered in the Spring application context (under the **camelContext** bean name), so you can access it like any other Spring bean.

```
@Configuration
public class MyAppConfig {

  @Autowired
  CamelContext camelContext;

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

}
```

1.2.4. Auto-detecting Camel routes

Camel auto-configuration collects all the **RouteBuilder** instances from the Spring context and automatically injects them into the provided **CamelContext**. This means that creating new Camel routes with the Spring Boot starter is as simple as 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");
  }

}
```

Or creating 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");
      }

    };
  }

}
```

1.2.5. Camel properties

Spring Boot auto-configuration automatically connects to Spring Boot external configuration (which may contain properties placeholders, OS environment variables or system properties) with the Camel properties support. It basically means that any property defined in **application.properties** file:

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

Or set via system property:

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

can be used as placeholders in Camel route:

```
@Component
public class MyRouter extends RouteBuilder {

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

```
    }

}
```

1.2.6. Custom Camel context configuration

If you want to perform some operations on **CamelContext** bean created by Camel auto-configuration, register **CamelContextConfiguration** instance in your Spring context:

```
@Configuration
public class MyAppConfig {

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

}
```

The method **beforeApplicationStart** will be called just before the Spring context is started, so the **CamelContext** instance passed to this callback is fully auto-configured. If you add multiple instances of **CamelContextConfiguration** into your Spring context, each instance is executed.

1.2.7. Auto-configured consumer and producer templates

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

```
@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 modifying the following Spring properties:

```
camel.springboot.consumer-template-cache-size = 100
camel.springboot.producer-template-cache-size = 200
```

1.2.8. Auto-configured TypeConverter

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

```
@Component
public class InvoiceProcessor {

  @Autowired
  private TypeConverter typeConverter;

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

}
```

1.2.8.1. Spring type conversion API bridge

Spring comes with the powerful type conversion API. The Spring API is similar to the Camel type converter API. As both APIs are so similar, 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 like Camel ones. With this approach you can use both Camel and Spring converters accessed via Camel **TypeConverter** API:

```
@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());
  }

}
```

Under the hood Camel 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 will create one for you.

1.2.9. Keeping the application alive

Camel applications which have this feature enabled launch a new thread on startup for the sole purpose of keeping the application alive by preventing JVM termination. This means that after you start a Camel application with Spring Boot, your application waits for a **Ctrl+C** signal and does not exit immediately.

The controller thread can be activated using the **camel.springboot.main-run-controller** to **true**.

–

```
camel.springboot.main-run-controller = true
```

Applications using web modules (for example, applications that import the **org.springframework.boot:spring-boot-web-starter** module), usually don't need to use this feature because the application is kept alive by the presence of other non-daemon threads.

1.2.10. 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. You can configure the directory name or turn this off using the configuration option:

```
# turn off
camel.springboot.routes-include-pattern = false
```

```
# scan only in the com/foo/routes classpath
camel.springboot.routes-include-pattern = classpath:com/foo/routes/*.xml
```

The XML files should be Camel XML routes (**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>
```

1.2.11. Testing the JUnit 5 way

For testing, Maven users will need to add the following dependencies to their **pom.xml**:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <version>3.2.5</version> <!-- Use the same version as your Spring Boot version -->
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-test-spring-junit5</artifactId>
    <version>4.4.0.redhat-00019</version> <!-- use the same version as your Camel core version -->
    <scope>test</scope>
</dependency>
```

To test a Camel Spring Boot application, annotate your test class(es) with **@CamelSpringBootTest**. This brings Camel's Spring Test support to your application, so that you can write tests using Spring Boot test conventions.

To get the **CamelContext** or **ProducerTemplate**, you can inject them into the class in the normal Spring manner, using **@Autowired**.

You can also use camel-test-spring-junit5 to configure tests declaratively. This example uses the **@MockEndpoints** annotation to auto-mock an endpoint:

```
@CamelSpringBootTest
@SpringBootApplication
@MockEndpoints("direct:end")
public class MyApplicationTest {

    @Autowired
    private ProducerTemplate template;

    @EndpointInject("mock:direct:end")
    private MockEndpoint mock;

    @Test
    public void testReceive() throws Exception {
        mock.expectedBodiesReceived("Hello");
        template.sendBody("direct:start", "Hello");
        mock.assertIsSatisfied();
    }

}
```

1.3. COMPONENT STARTERS

Camel Spring Boot supports the following Camel artifacts as Spring Boot Starters:

- Table 1.1, “Camel Components”
- Table 1.2, “Camel Data Formats”
- Table 1.3, “Camel Languages”
- Table 1.4, “Miscellaneous Extensions”

Table 1.1. Camel Components

Component	Artifact	Description	Support on IBM Power and IBM Z
AMQP	camel-amqp-starter	Messaging with AMQP protocol using Apache QPid Client.	Yes
AWS Cloudwatch	camel-aws2-cw-starter	Sending metrics to AWS CloudWatch using AWS SDK version 2.x.	Yes
AWS DynamoDB	camel-aws2-ddb-starter	Store and retrieve data from AWS DynamoDB service using AWS SDK version 2.x.	Yes

Component	Artifact	Description	Support on IBM Power and IBM Z
AWS Kinesis	camel-aws2-kinesis-starter	Consume and produce records from and to AWS Kinesis Streams using AWS SDK version 2.x.	Yes
AWS Lambda	camel-aws2-lambda-starter	Manage and invoke AWS Lambda functions using AWS SDK version 2.x.	Yes
AWS S3 Storage Service	camel-aws2-s3-starter	Store and retrieve objects from AWS S3 Storage Service using AWS SDK version 2.x.	Yes
AWS Simple Notification System (SNS)	camel-aws2-sns-starter	Send messages to an AWS Simple Notification Topic using AWS SDK version 2.x.	Yes
AWS Simple Queue Service (SQS)	camel-aws2-sqs-starter	Send and receive messages to/from AWS SQS service using AWS SDK version 2.x.	Yes
Azure ServiceBus	camel-azure-servicebus-starter	Send and receive messages to/from Azure Event Bus.	Yes
Azure Storage Blob Service	camel-azure-storage-blob-starter	Store and retrieve blobs from Azure Storage Blob Service using SDK v12.	Yes
Azure Storage Queue Service	camel-azure-storage-queue-starter	The azure-storage-queue component is used for storing and retrieving the messages to/from Azure Storage Queue using Azure SDK v12.	Yes
Bean	camel-bean-starter	Invoke methods of Java beans stored in Camel registry.	Yes

Component	Artifact	Description	Support on IBM Power and IBM Z
Bean Validator	camel-bean-validator-starter	Validate the message body using the Java Bean Validation API.	Yes
Browse	camel-browse-starter	Inspect the messages received on endpoints supporting BrowsableEndpoint.	Yes
Cassandra CQL	camel-cassandraql-starter	Integrate with Cassandra 2.0 using the CQL3 API (not the Thrift API). Based on Cassandra Java Driver provided by DataStax.	Yes
CICS	camel-cics-starter	Interact with CICS® general-purpose transaction processing subsystem.	No
Control Bus	camel-controlbus-starter	Manage and monitor Camel routes.	Yes
Cron	camel-cron-starter	A generic interface for triggering events at times specified through the Unix cron syntax.	Yes
Crypto (JCE)	camel-crypto-starter	Sign and verify exchanges using the Signature Service of the Java Cryptographic Extension (JCE).	Yes
CXF	camel-cxf-soap-starter	Expose SOAP WebServices using Apache CXF or connect to external WebServices using CXF WS client.	Yes
CXF-RS	camel-cxf-rest-starter	Expose JAX-RS REST services using Apache CXF or connect to external REST services using CXF REST client.	Yes

Component	Artifact	Description	Support on IBM Power and IBM Z
Data Format	camel-dataformat-starter	Use a Camel Data Format as a regular Camel Component.	Yes
Dataset	camel-dataset-starter	Provide data for load and soak testing of your Camel application.	Yes
Direct	camel-direct-starter	Call another endpoint from the same Camel Context synchronously.	Yes
Elastic Search	camel-elasticsearch-starter	Send requests to ElasticSearch via Java Client API.	No
FHIR	camel-fhir-starter	Exchange information in the healthcare domain using the FHIR (Fast Healthcare Interoperability Resources) standard.	No
File	camel-file-starter	Read and write files.	Yes
Flink	camel-flink-starter	Send DataSet jobs to an Apache Flink cluster.	Yes
FTP	camel-ftp-starter	Upload and download files to/from FTP servers.	Yes
Google BigQuery	camel-google-bigquery-starter	Google BigQuery data warehouse for analytics.	Yes
Google Pubsub	camel-google-pubsub-starter	Send and receive messages to/from Google Cloud Platform PubSub Service.	Yes
gRPC	camel-grpc-starter	Expose gRPC endpoints and access external gRPC endpoints.	Yes
HTTP	camel-http-starter	Send requests to external HTTP servers using Apache HTTP Client 4.x.	Yes

Component	Artifact	Description	Support on IBM Power and IBM Z
Infinispan	camel-infinispan-starter	Read and write from/to Infinispan distributed key/value store and data grid.	No
Infinispan Embedded	camel-infinispan-embedded-starter	Read and write from/to Infinispan distributed key/value store and data grid.	Yes
JDBC	camel-jdbc-starter	Access databases through SQL and JDBC.	Yes
Jira	camel-jira-starter	Interact with JIRA issue tracker.	Yes
JMS	camel-jms-starter	Sent and receive messages to/from a JMS Queue or Topic.	Yes
JPA	camel-jpa-starter	Store and retrieve Java objects from databases using Java Persistence API (JPA).	Yes
JSLT	camel-jslt-starter	Query or transform JSON payloads using an JSLT.	Yes
Kafka	camel-kafka-starter	Sent and receive messages to/from an Apache Kafka broker.	Yes
Kamelet	camel-kamelet-starter	To call Kamelets	Yes
Kubernetes ConfigMap	camel-kubernetes-starter	Perform operations on Kubernetes ConfigMaps and get notified on ConfigMaps changes.	Yes
Kubernetes Custom Resources	camel-kubernetes-starter	Perform operations on Kubernetes Custom Resources and get notified on Deployment changes.	Yes

Component	Artifact	Description	Support on IBM Power and IBM Z
Kubernetes Deployments	camel-kubernetes-starter	Perform operations on Kubernetes Deployments and get notified on Deployment changes.	Yes
Kubernetes Event	camel-kubernetes-starter	Perform operations on Kubernetes Events and get notified on Events changes.	Yes
Kubernetes HPA	camel-kubernetes-starter	Perform operations on Kubernetes Horizontal Pod Autoscalers (HPA) and get notified on HPA changes.	Yes
Kubernetes Job	camel-kubernetes-starter	Perform operations on Kubernetes Jobs.	Yes
Kubernetes Namespaces	camel-kubernetes-starter	Perform operations on Kubernetes Namespaces and get notified on Namespace changes.	Yes
Kubernetes Nodes	camel-kubernetes-starter	Perform operations on Kubernetes Nodes and get notified on Node changes.	Yes
Kubernetes Persistent Volume	camel-kubernetes-starter	Perform operations on Kubernetes Persistent Volumes and get notified on Persistent Volume changes.	Yes
Kubernetes Persistent Volume Claim	camel-kubernetes-starter	Perform operations on Kubernetes Persistent Volumes Claims and get notified on Persistent Volumes Claim changes.	Yes
Kubernetes Pods	camel-kubernetes-starter	Perform operations on Kubernetes Pods and get notified on Pod changes.	Yes

Component	Artifact	Description	Support on IBM Power and IBM Z
Kubernetes Replication Controller	camel-kubernetes-starter	Yes Perform operations on Kubernetes Replication Controllers and get notified on Replication Controllers changes.	Yes
Kubernetes Resources Quota	camel-kubernetes-starter	Perform operations on Kubernetes Resources Quotas.	Yes
Kubernetes Secrets	camel-kubernetes-starter	Perform operations on Kubernetes Secrets.	Yes
Kubernetes Service Account	camel-kubernetes-starter	Perform operations on Kubernetes Service Accounts.	Yes
Kubernetes Services	camel-kubernetes-starter	Perform operations on Kubernetes Services and get notified on Service changes.	Yes
Kudu	camel-kudu-starter	Interact with Apache Kudu, a free and open source column-oriented data store of the Apache Hadoop ecosystem.	No
Language	camel-language-starter	Execute scripts in any of the languages supported by Camel.	Yes
LDAP	camel-ldap-starter	Perform searches on LDAP servers.	Yes
Log	camel-log-starter	Log messages to the underlying logging mechanism.	Yes
LRA	camel-lra-starter	Camel saga binding for Long-Running-Action framework.	Yes
Mail	camel-mail-starter	Send and receive emails using imap, pop3 and smtp protocols.	Yes

Component	Artifact	Description	Support on IBM Power and IBM Z
Mail Microsoft OAuth	camel-mail-microsoft-oauth-starter	Camel Mail OAuth2 Authenticator for Microsoft Exchange Online.	Yes
MapStruct	camel-mapstruct-starter	Type Conversion using Mapstruct.	Yes
Master	camel-master-starter	Have only a single consumer in a cluster consuming from a given endpoint; with automatic failover if the JVM dies.	Yes
Micrometer	camel-micrometer-starter	Collect various metrics directly from Camel routes using the Micrometer library.	Yes
Minio	camel-minio-starter	Store and retrieve objects from Minio Storage Service using Minio SDK.	Yes
MLLP	camel-mllp-starter	Communicate with external systems using the MLLP protocol.	Yes
Mock	camel-mock-starter	Test routes and mediation rules using mocks.	Yes
MongoDB	camel-mongodb-starter	Perform operations on MongoDB documents and collections.	Yes
MyBatis	camel-mybatis-starter	Performs a query, poll, insert, update or delete in a relational database using MyBatis.	Yes
Netty	camel-netty-starter	Socket level networking using TCP or UDP with Netty 4.x.	Yes

Component	Artifact	Description	Support on IBM Power and IBM Z
Olingo4	camel-olingo4-starter	Communicate with OData 4.0 services using Apache Olingo OData API.	Yes
Openshift Build Config	camel-kubernetes-starter	Perform operations on OpenShift Build Configs.	Yes
Openshift Builds	camel-kubernetes-starter	Perform operations on OpenShift Builds.	Yes
Openshift Deployment Configs	camel-kubernetes-starter	Perform operations on Openshift Deployment Configs and get notified on Deployment Config changes.	Yes
Netty HTTP	camel-netty-http-starter	Netty HTTP server and client using the Netty 4.x.	Yes
Paho	camel-paho-starter	Communicate with MQTT message brokers using Eclipse Paho MQTT Client.	Yes
Paho MQTT 5	camel-paho-mqtt5-starter	Communicate with MQTT message brokers using Eclipse Paho MQTT v5 Client.	Yes
Platform HTTP	camel-platform-http-starter	Expose HTTP endpoints using the HTTP server available in the current platform.	Yes
Quartz	camel-quartz-starter	Schedule sending of messages using the Quartz 2.x scheduler.	Yes
Ref	camel-ref-starter	Route messages to an endpoint looked up dynamically by name in the Camel Registry.	Yes

Component	Artifact	Description	Support on IBM Power and IBM Z
REST	camel-rest-starter	Expose REST services or call external REST services.	Yes
Saga	camel-saga-starter	Execute custom actions within a route using the Saga EIP.	Yes
Salesforce	camel-salesforce-starter	Communicate with Salesforce using Java DTOs.	Yes
SAP	camel-sap-starter	Uses the SAP Java Connector (SAP JCo) library to facilitate bidirectional communication with SAP and the SAP IDoc library to facilitate the transmission of documents in the Intermediate Document (IDoc) format.	Yes
Scheduler	camel-scheduler-starter	Generate messages in specified intervals using java.util.concurrent.ScheduledExecutorService.	Yes
SEDA	camel-seda-starter	Asynchronously call another endpoint from any Camel Context in the same JVM.	Yes
Servlet	camel-servlet-starter	Serve HTTP requests by a Servlet.	Yes
Slack	camel-slack-starter	Send and receive messages to/from Slack.	Yes
SMB	camel-smb-starter	Receive files from SMB (Server Message Block) shares.	Yes
SNMP	camel-snmp-starter	Receive traps and poll SNMP (Simple Network Management Protocol) capable devices.	Yes

Component	Artifact	Description	Support on IBM Power and IBM Z
Splunk	camel-splunk-starter	Publish or search for events in Splunk.	No
Spring Batch	camel-spring-batch-starter	Send messages to Spring Batch for further processing.	Yes
Spring JDBC	camel-spring-jdbc-starter	Access databases through SQL and JDBC with Spring Transaction support.	Yes
Spring LDAP	camel-spring-ldap-starter	Perform searches in LDAP servers using filters as the message payload.	Yes
Spring RabbitMQ	camel-spring-rabbitmq-starter	Send and receive messages from RabbitMQ using Spring RabbitMQ client.	Yes
Spring Redis	camel-spring-redis-starter	Send and receive messages from Redis.	Yes
Spring Webservice	camel-spring-ws-starter	You can use this component to integrate with Spring Web Services. It offers client-side support for accessing web services and server-side support for creating your contract-first web services.	Yes
SQL	camel-sql-starter	Perform SQL queries using Spring JDBC.	Yes
SQL Stored Procedure	camel-sql-starter	Perform SQL queries as a JDBC Stored Procedures using Spring JDBC.	Yes
SSH	camel-ssh-starter	Execute commands on remote hosts using SSH.	Yes

Component	Artifact	Description	Support on IBM Power and IBM Z
Stub	camel-stub-starter	Stub out any physical endpoints while in development or testing.	Yes
Telegram	camel-telegram-starter	Send and receive messages acting as a Telegram Bot Telegram Bot API.	Yes
Timer	camel-timer-starter	Generate messages in specified intervals using java.util.Timer.	Yes
Validator	camel-validator-starter	Validate the payload using XML Schema and JAXP Validation.	Yes
Velocity	camel-velocity-starter	Transform messages using a Velocity template.	Yes
Vert.x HTTP Client	camel-vertx-http-starter	Send requests to external HTTP servers using Vert.x.	Yes
Vert.x WebSocket	camel-vertx-websocket-starter	Expose WebSocket endpoints and connect to remote WebSocket servers using Vert.x.	Yes
Webhook	camel-webhook-starter	Expose webhook endpoints to receive push notifications for other Camel components.	Yes
XJ	camel-xj-starter	Transform JSON and XML message using a XSLT.	Yes
XSLT	camel-xslt-starter	Transforms XML payload using an XSLT template.	Yes

Component	Artifact	Description	Support on IBM Power and IBM Z
XSLT Saxon	camel-xslt-saxon-starter	Transform XML payloads using an XSLT template using Saxon.	Yes

Table 1.2. Camel Data Formats

Component	Artifact	Description	Support on IBM Power and IBM Z
Avro	camel-avro-starter	Serialize and deserialize messages using Apache Avro binary data format.	Yes
Avro Jackson	camel-jackson-avro-starter	Marshal POJOs to Avro and back using Jackson.	Yes
Bindy	camel-bindy-starter	Marshal and unmarshal between POJOs and key-value pair (KVP) format using Camel Bindy.	Yes
HL7	camel-hl7-starter	Marshal and unmarshal HL7 (Health Care) model objects using the HL7 MLLP codec.	Yes
JacksonXML	camel-jacksonxml-starter	Unmarshal a XML payloads to POJOs and back using XMLMapper extension of Jackson.	Yes
JAXB	camel-jaxb-starter	Unmarshal XML payloads to POJOs and back using JAXB2 XML marshalling standard.	Yes
JSON Gson	camel-gson-starter	Marshal POJOs to JSON and back using Gson	Yes
JSON Jackson	camel-jackson-starter	Marshal POJOs to JSON and back using Jackson	Yes

Component	Artifact	Description	Support on IBM Power and IBM Z
Protobuf Jackson	camel-jackson-protobuf-starter	Marshal POJOs to Protobuf and back using Jackson.	Yes
SOAP	camel-soap-starter	Marshal Java objects to SOAP messages and back.	Yes
Zip File	camel-zipfile-starter	Compression and decompress streams using java.util.zip.ZipStream.	Yes

Table 1.3. Camel Languages

Language	Artifact	Description	Support on IBM Power and IBM Z
Constant	camel-core-starter	A fixed value set only once during the route startup.	Yes
CSimple	camel-core-starter	Evaluate a compiled simple expression.	Yes
ExchangeProperty	camel-core-starter	Gets a property from the Exchange.	Yes
File	camel-core-starter	File related capabilities for the Simple language.	Yes
Header	camel-core-starter	Gets a header from the Exchange.	Yes
JQ	camel-jq-starter	Evaluates a JQ expression against a JSON message body.	Yes
JSONPath	camel-jsonpath-starter	Evaluates a JSONPath expression against a JSON message body.	Yes
Ref	camel-core-starter	Uses an existing expression from the registry.	Yes

Language	Artifact	Description	Support on IBM Power and IBM Z
Simple	camel-core-starter	Evaluates a Camel simple expression.	Yes
Tokenize	camel-core-starter	Tokenize text payloads using delimiter patterns.	Yes
XML Tokenize	camel-xml-jaxp-starter	Tokenize XML payloads.	Yes
XPath	camel-xpath-starter	Evaluates an XPath expression against an XML payload.	Yes
XQuery	camel-saxon-starter	Query and/or transform XML payloads using XQuery and Saxon.	Yes

Table 1.4. Miscellaneous Extensions

Extensions	Artifact	Description	Support on IBM Power and IBM Z
Jasypt	camel-jasypt-starter	Security using Jasypt	Yes
Kamelet Main	camel-kamelet-main-starter	Main to run Kamelet standalone	Yes
Openapi Java	camel-openapi-java-starter	Rest-dsl support for using openapi doc	Yes
OpenTelemetry	camel-opentelemetry-starter	Distributed tracing using OpenTelemetry	Yes
Spring Security	camel-spring-security-starter	Security using Spring Security	Yes
YAML DSL	camel-yaml-dsl-starter	Camel DSL with YAML	Yes

1.4. STARTER CONFIGURATION

Clear and accessible configuration is a crucial part of any application. Camel starters fully support Spring Boot's external configuration mechanism. You can also configure them through Spring Beans for more complex use cases.

1.4.1. Using External Configuration

Internally, every starter is configured through Spring Boot's ConfigurationProperties. Each configuration parameter can be set in various ways (**application.[properties|json|yaml]** files, command line arguments, environments variables etc.). Parameters have the form of **camel.[component|language|dataformat].[name].[parameter]**

For example to configure the URL of the MQTT5 broker you can set:

```
camel.component.paho-mqtt5.broker-url=tcp://localhost:61616
```

Or to configure the **delimeter** of the CSV dataformat to be a semicolon(;) you can set:

```
camel.dataformat.csv.delimiter=;
```

Camel will use the Type Converter mechanism when setting properties to the desired type.

You can refer to beans in the Registry using the **#bean:name**:

```
camel.component.jms.transactionManager=#bean:myjtaTransactionManager
```

The **Bean** would be typically created in Java:

```
@Bean("myjtaTransactionManager")
public JmsTransactionManager myjtaTransactionManager(PooledConnectionFactory pool) {
    JmsTransactionManager manager = new JmsTransactionManager(pool);
    manager.setDefaultTimeout(45);
    return manager;
}
```

Beans can also be created in configuration files but this is not recommended for complex use cases.

1.4.2. Using Beans

Starters can also be created and configured via Spring Beans. Before creating a starter , Camel will first lookup it up in the Registry by it's name if it already exists. For example to configure a Kafka component:

```
@Bean("kafka")
public KafkaComponent kafka(KafkaConfiguration kafkaconfiguration){
    return ComponentsBuilderFactory.kafka()
                        .brokers("{{kafka.host}}:{{kafka.port}}")
                        .build();
}
```

The **Bean** name has to be equal to that of the Component, Dataformat or Language that you are configuring. If the **Bean** name isn't specified in the annotation it will be set to the method name.

Typical Camel Spring Boot projects will use a combination of external configuration and Beans to configure an application. For more examples on how to configure your Camel Spring Boot project, please see the example repository.

1.5. GENERATING A CAMEL FOR SPRING BOOT APPLICATION USING MAVEN

You can generate a Red Hat build of Apache Camel for Spring Boot application using the Maven archetype **org.apache.camel.archetypes:camel-archetype-spring-boot:4.4.0.redhat-00014**.

Procedure

1. Run the following command:

```
mvn archetype:generate \
 -DarchetypeGroupId=org.apache.camel.archetypes \
 -DarchetypeArtifactId=camel-archetype-spring-boot \
 -DarchetypeVersion=4.4.0.redhat-00014 \
 -DgroupId=com.redhat \
 -DartifactId=csb-app \
 -Dversion=1.0-SNAPSHOT \
 -DinteractiveMode=false
```

2. Build the application:

```
mvn package -f csb-app/pom.xml
```

3. Run the application:

```
java -jar csb-app/target/csb-app-1.0-SNAPSHOT.jar
```

4. Verify that the application is running by examining the console log for the *Hello World* output which is generated by the application.

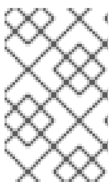
```
com.redhat.MySpringBootApplication       : Started MySpringBootApplication in 3.514
seconds (JVM running for 4.006)
Hello World
Hello World
```

1.6. DEPLOYING A CAMEL SPRING BOOT APPLICATION TO OPENSHIFT

This guide demonstrates how to deploy a Camel Spring Boot application to OpenShift.

Prerequisites

- You have access to the OpenShift cluster.
- The OpenShift **oc** CLI client is installed or you have access to the OpenShift Container Platform web console.



NOTE

The certified OpenShift Container platforms are listed in the Camel for Spring Boot Supported Configurations. The Red Hat OpenJDK 11 (ubi8/openjdk-11) container image is used in the following example.

Procedure

1. Generate a Camel for Spring Boot application using Maven by following the instructions in section 1.5 Generating a Camel for Spring Boot application using Maven of this guide.

2. Under the directory which the modified pom.xml exists, execute the following command.

```
mvn clean -DskipTests oc:deploy -Popenshift
```

3. Verify that the CSB application is running on the pod.

```
oc logs -f dc/csb-app
```

1.7. APPLYING PATCH TO RED HAT BUILD OF APACHE CAMEL FOR SPRING BOOT

Using the new **patch-maven-plugin** mechanism, you can apply a patch to your Red Hat Red Hat build of Apache Camel for Spring Boot application. This mechanism allows you to change the individual versions provided by different Red Hat application BOMS, for example, **camel-spring-boot-bom**.

The purpose of the **patch-maven-plugin** is to update the versions of the dependencies listed in the Camel on Spring Boot BOM to the versions specified in the patch metadata that you wish to apply to your applications.

The patch-maven-plugin performs the following operations:

- Retrieve the patch metadata related to current Red Hat application 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 the versions are replaced, the Maven build continues and progresses through standard Maven project stages.

Procedure

The 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 Camel on Spring Boot BOM.

```
<build>
   <plugins>
      <<plugin>
         <groupId>com.redhat.camel.springboot.platform</groupId>
         <artifactId>patch-maven-plugin</artifactId>
         <version>${camel-spring-boot-version}</version>
         <extensions>true</extensions>
      </plugin>
   </plugins>
</build>
```

2. When you run any of the **mvn clean deploy**, **mvn validate**, or **mvn dependency:tree** commands, the plugin searches through the project modules to check if the modules use the Red Hat Red Hat build of Apache Camel for Spring Boot BOM. Only the following is the supported BOM:

- **com.redhat.camel.springboot.platform:camel-spring-boot-bom**: for Red Hat build of Apache Camel for Spring Boot BOM

3. If the plugin does not find the above BOM, the plugin displays the following messages:

```
$ mvn clean install

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

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

[INFO] [PATCH] No project in the reactor uses Camel on Spring Boot product BOM. Skipping patch processing.
[INFO] [PATCH] Done in 7ms

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

4. If the correct BOM is used, the patch metadata is found, but without any patches.

```
$ mvn clean install

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

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

[INFO] [PATCH] Reading patch metadata and artifacts from 2 project repositories
[INFO] [PATCH]  - redhat-ga-repository: http://maven.repository.redhat.com/ga/
[INFO] [PATCH]  - central: https://repo.maven.apache.org/maven2
Downloading from redhat-ga-repository:
http://maven.repository.redhat.com/ga/com/redhat/camel/springboot/platform/redhat-camel-spring-boot-patch-metadata/maven-metadata.xml
Downloading from central:
https://repo.maven.apache.org/maven2/com/redhat/camel/springboot/platform/redhat-camel-spring-boot-patch-metadata/maven-metadata.xml
[INFO] [PATCH] Resolved patch descriptor:
/path/to/.m2/repository/com/redhat/camel/springboot/platform/redhat-camel-spring-boot-patch-metadata/3.20.1.redhat-00043/redhat-camel-spring-boot-patch-metadata-3.20.1.redhat-00043.xml
[INFO] [PATCH] Patch metadata found for com.redhat.camel.springboot.platform/camel-spring-boot-bom/[3.20,3.21)
[INFO] [PATCH] Done in 938ms

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

5. The **patch-maven-plugin** attempts to fetch this Maven metadata.

- For the projects with Camel Spring Boot BOM, the **com.redhat.camel.springboot.platform:redhat-camel-spring-boot-patch-metadata/maven-metadata.xml** is resolved. This XML data is the metadata for the artifact with the **com.redhat.camel.springboot.platform:redhat-camel-spring-boot-patch-metadata:RELEASE** coordinates.

Example metadata generated by Maven

```
<?xml version="1.0" encoding="UTF-8"?>
<metadata>
  <groupId>com.redhat.camel.springboot.platform</groupId>
  <artifactId>redhat-camel-spring-boot-patch-metadata</artifactId>
  <versioning>
    <release>3.20.1.redhat-00041</release>
    <versions>
      <version>3.20.1.redhat-00041</version>
    </versions>
    <lastUpdated>20230322103858</lastUpdated>
  </versioning>
</metadata>
```

6. The **patch-maven-plugin** parses the metadata to select the version which applies to the current project. This action is possible only for the Maven projects using Camel on Spring Boot BOM with the specific version. Only the metadata that matches the version range or later is applicable, and it fetches only the latest version of the metadata.

7. The **patch-maven-plugin** collects a list of remote Maven repositories for downloading the patch metadata identified by **groupId**, **artifactId**, and **version** found in previous steps. These Maven repositories are listed in the project's **<repositories>** elements in the active profiles, and also the repositories from the **settings.xml** file.

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

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

[INFO] [PATCH] Reading patch metadata and artifacts from 2 project repositories
[INFO] [PATCH]  - MRRC-GA: https://maven.repository.redhat.com/ga
[INFO] [PATCH]  - central: https://repo.maven.apache.org/maven2
```

8. Whether the metadata comes from a remote repository, local repository, or ZIP file, it is analyzed by the **patch-maven-plugin**. The fetched metadata contains a list of CVEs, and for each CVE, we have a list of the 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:patch-metadata:1">
    <product-bom groupId="com.redhat.camel.springboot.platform" artifactId="camel-spring-
boot-bom" versions="[3.20,3.21)" />
    <cves>
    </cves>
    <fixes>
        <fix id="HF0-1" description="logback-classic (Example) - Version Bump">
            <affects groupId="ch.qos.logback" artifactId="logback-classic" versions="[1.0,1.3.0)"
fix="1.3.0" />
        </fix>
    </fixes>
</metadata>
```

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

```
$ mvn dependency:tree

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

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

[INFO] [PATCH] Reading patch metadata and artifacts from 3 project repositories
[INFO] [PATCH]  - redhat-ga-repository: http://maven.repository.redhat.com/ga/
[INFO] [PATCH]  - local: file:///path/to/.m2/repository
[INFO] [PATCH]  - central: https://repo.maven.apache.org/maven2
[INFO] [PATCH] Resolved patch
descriptor:/path/to/.m2/repository/com/redhat/camel/springboot/platform/redhat-camel-spring-boot-patch-metadata/3.20.1.redhat-00043/redhat-camel-spring-boot-patch-metadata-3.20.1.redhat-00043.xml
[INFO] [PATCH] Patch metadata found for com.redhat.camel.springboot.platform/camel-spring-boot-bom/[3.20,3.21)
[INFO] [PATCH]  - patch contains 1 patch fix
[INFO] [PATCH] Processing managed dependencies to apply patch fixes...
[INFO] [PATCH] - HF0-1: logback-classic (Example) - Version Bump
[INFO] [PATCH]   Applying change ch.qos.logback/logback-classic/[1.0,1.3.0) -> 1.3.0
[INFO] [PATCH]   Project com.test:yaml-routes
[INFO] [PATCH]    - managed dependency: ch.qos.logback/logback-classic/1.2.11 -> 1.3.0
[INFO] [PATCH] Done in 39ms

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

Skipping the patch

If 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 the **patch-maven-plugin** to the project's **pom.xml** file, and you do not wish to alter the versions, 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>com.redhat.camel.springboot.platform</groupId>
      <artifactId>patch-maven-plugin</artifactId>
      <version>${camel-spring-boot-version}</version>
      <extensions>true</extensions>
      <configuration>
        <skip>true</skip>
      </configuration>
    </plugin>
  </plugins>
</build>
```

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

```
$ mvn clean install -DskipPatch
```

```
[INFO] Scanning for projects...
[INFO]
[INFO] -------------------------< com.example:test-csb >-------------------------
[INFO] Building A Camel Spring Boot Route 1.0-SNAPSHOT
...
```

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

1.8. CAMEL REST DSL OPENAPI MAVEN PLUGIN

The Camel REST DSL OpenApi Maven Plugin supports the following goals.

- camel-restdsl-openapi:generate - To generate consumer REST DSL RouteBuilder source code from OpenApi specification
- camel-restdsl-openapi:generate-with-dto - To generate consumer REST DSL RouteBuilder source code from OpenApi specification and with DTO model classes generated via the swagger-codegen-maven-plugin.
- camel-restdsl-openapi:generate-xml - To generate consumer REST DSL XML source code from OpenApi specification
- camel-restdsl-openapi:generate-xml-with-dto - To generate consumer REST DSL XML source code from OpenApi specification and with DTO model classes generated via the swagger-codegen-maven-plugin.
- camel-restdsl-openapi:generate-yaml - To generate consumer REST DSL YAML source code from OpenApi specification
- camel-restdsl-openapi:generate-yaml-with-dto - To generate consumer REST DSL YAML source code from OpenApi specification and with DTO model classes generated via the swagger-codegen-maven-plugin.

1.8.1. Adding plugin to Maven pom.xml

This plugin can be added to your Maven **pom.xml** file by adding it to the **plugins** section, for example in a Spring Boot application:

```
<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
    </plugin>

    <plugin>
      <groupId>org.apache.camel</groupId>
      <artifactId>camel-restdsl-openapi-plugin</artifactId>
      <version>{CamelCommunityVersion}</version>
    </plugin>

  </plugins>
</build>
```

The plugin can then be executed using its prefix **camel-restdsl-openapi** as shown below.

```
$mvn camel-restdsl-openapi:generate
```

1.8.2. camel-restdsl-openapi:generate

The goal of the Camel REST DSL OpenApi Maven Plugin is used to generate REST DSL RouteBuilder implementation source code from Maven.

1.8.3. Options

The plugin supports the following options which can be configured from the command line (use **-D** syntax), or defined in the **pom.xml** file in the **configuration** tag.

Parameter	Default Value	Description
skip	**false**	Set to **true** to skip code generation
filterOperation		Used for including only the operation ids specified. Multiple ids can be separated by comma. Wildcards can be used, eg **find*** to include all operations starting with **find**.
specificationUri	**src/spec/openapi.json**	URI of the OpenApi specification, supports filesystem paths, HTTP and classpath resources, by default **src/spec/openapi.json** within the project directory. Supports JSON and YAML.
auth		Adds authorization headers when fetching the OpenApi specification definitions remotely. Pass in a URL-encoded string of name:header with a comma separating multiple values.
className	from **title** or **RestDslRoute**	Name of the generated class, taken from the OpenApi specification title or set to **RestDslRoute** by default
packageName	from **host** or **rest.dsl.generated**	Name of the package for the generated class, taken from the OpenApi specification host value or **rest.dsl.generated** by default

Parameter	Default Value	Description
indent	**" "**	Which indenting character(s) to use, by default four spaces, you can use **\t** to signify tab character
outputDirectory	**generated-sources/restdsl-openapi**	Where to place the generated source file, by default **generated-sources/restdsl-openapi** within the project directory
destinationGenerator		Fully qualified class name of the class that implements **org.apache.camel.generator.openapi.DestinationGenerator** interface for customizing destination endpoint
destinationToSyntax	**direct:${operationId}**	The default to syntax for the to uri, which is to use the direct component.
restConfiguration	**true**	Whether to include generation of the rest configuration with detected rest component to be used.
apiContextPath		Define openapi endpoint path if **restConfiguration** is set to true.
clientRequestValidation	**false**	Whether to enable request validation.
basePath		Overrides the api base path as defined in the OpenAPI specification.
requestMappingValues	**/****	Allows generation of custom **RequestMapping** mapping values. Multiple mapping values can be passed as: **<requestMappingValues> <param>/my-api-path/</param> <param>/my-other-path/</param> </requestMappingValues>**

1.8.4. Spring Boot Project with Servlet component

If the Maven project is a Spring Boot project and **restConfiguration** is enabled and the servlet component is being used as REST component, then this plugin will autodetect the package name (if packageName has not been explicitly configured) where the **@SpringBootApplication** main class is located, and use the same package name for generating Rest DSL source code and a needed **CamelRestController** support class.

1.8.5. camel-restdsl-openapi:generate-with-dto

Works as **generate** goal but also generates DTO model classes by automatic executing the swagger-codegen-maven-plugin to generate java source code of the DTO model classes from the OpenApi specification.

This plugin has been scoped and limited to only support a good effort set of defaults for using the swagger-codegen-maven-plugin to generate the model DTOs. If you need more power and flexibility then use the Swagger Codegen Maven Plugin directly to generate the DTO and not this plugin.

The DTO classes may require additional dependencies such as:

```
<dependency>
  <groupId>com.google.code.gson</groupId>
  <artifactId>gson</artifactId>
  <version>2.10.1</version>
</dependency>
<dependency>
  <groupId>io.swagger.core.v3</groupId>
  <artifactId>swagger-core</artifactId>
  <version>2.2.8</version>
</dependency>
<dependency>
  <groupId>org.threeten</groupId>
  <artifactId>threetenbp</artifactId>
  <version>1.6.8</version>
</dependency>
```

1.8.6. Options

The plugin supports the following **additional** options

Parameter	Default Value	Description
swaggerCodegenMavenPluginVersion	3.0.36	The version of the **io.swagger.codegen.v3:swagger-codegen-maven-plugin** maven plugin to be used.
modelOutput		Target output path (default is ${project.build.directory}/generated-sources/openapi)
modelPackage	**io.swagger.client.model**	The package to use for generated model objects/classes

Parameter	Default Value	Description
modelNamePrefix		Sets the pre- or suffix for model classes and enums
modelNameSuffix		Sets the pre- or suffix for model classes and enums
modelWithXml	false	Enable XML annotations inside the generated models (only works with libraries that provide support for JSON and XML)
configOptions		Pass a map of language-specific parameters to **swagger-codegen-maven-plugin**

1.8.7. camel-restdsl-openapi:generate-xml

The **camel-restdsl-openapi:generate-xml** goal of the Camel REST DSL OpenApi Maven Plugin is used to generate REST DSL XML implementation source code from Maven.

1.8.8. Options

The plugin supports the following options which can be configured from the command line (use **-D** syntax), or defined in the **pom.xml** file in the **<configuration>** tag.

Parameter	Default Value	Description
skip	**false**	Set to **true** to skip code generation.
filterOperation		Used for including only the operation ids specified. Multiple ids can be separated by comma. Wildcards can be used, eg **find*** to include all operations starting with **find**.
specificationUri	**src/spec/openapi.json**	URI of the OpenApi specification, supports filesystem paths, HTTP and classpath resources, by default **src/spec/openapi.json** within the project directory. Supports JSON and YAML.

Parameter	Default Value	Description
auth		Adds authorization headers when fetching the OpenApi specification definitions remotely. Pass in a URL-encoded string of name:header with a comma separating multiple values.
outputDirectory	**generated-sources/restdsl-openapi**	Where to place the generated source file, by default **generated-sources/restdsl-openapi** within the project directory
fileName	**camel-rest.xml**	The name of the XML file as output.
blueprint	**false**	If enabled generates OSGi Blueprint XML instead of Spring XML.
destinationGenerator		Fully qualified class name of the class that implements **org.apache.camel.generator.openapi.DestinationGenerator** interface for customizing destination endpoint
destinationToSyntax	**direct:${operationId}**	The default to syntax for the to uri, which is to use the direct component.
	restConfiguration	**true**
Whether to include generation of the rest configuration with detected rest component to be used.	**apiContextPath**	
Define openapi endpoint path if **restConfiguration** is set to **true**.	**clientRequestValidation**	**false**
Whether to enable request validation.	**basePath**	

Parameter	Default Value	Description
Overrides the api base path as defined in the OpenAPI specification.	**requestMappingValues**	/**

1.8.9. camel-restdsl-openapi:generate-xml-with-dto

Works as **generate-xml** goal but also generates DTO model classes by automatic executing the swagger-codegen-maven-plugin to generate java source code of the DTO model classes from the OpenApi specification.

This plugin has been scoped and limited to only support a good effort set of defaults for using the swagger-codegen-maven-plugin to generate the model DTOs. If you need more power and flexibility then use the Swagger Codegen Maven Plugin directly to generate the DTO and not this plugin.

The DTO classes may require additional dependencies such as:

```
<dependency>
  <groupId>com.google.code.gson</groupId>
  <artifactId>gson</artifactId>
  <version>2.10.1</version>
</dependency>
<dependency>
  <groupId>io.swagger.core.v3</groupId>
  <artifactId>swagger-core</artifactId>
  <version>2.2.8</version>
</dependency>
<dependency>
  <groupId>org.threeten</groupId>
  <artifactId>threetenbp</artifactId>
  <version>1.6.8</version>
</dependency>
```

1.8.10. Options

The plugin supports the following **additional** options

Parameter	Default Value	Description
swaggerCodegenMavenPluginVersion	3.0.36	The version of the **io.swagger.codegen.v3:swagger-codegen-maven-plugin** maven plugin to be used.
modelOutput		Target output path (default is ${project.build.directory}/generated-sources/openapi)
modelPackage	**io.swagger.client.model**	The package to use for generated model objects/classes

Parameter	Default Value	Description
modelNamePrefix		Sets the pre- or suffix for model classes and enums
modelNameSuffix		Sets the pre- or suffix for model classes and enums
modelWithXml	false	Enable XML annotations inside the generated models (only works with libraries that provide support for JSON and XML)
configOptions		Pass a map of language-specific parameters to **swagger-codegen-maven-plugin**

1.8.11. camel-restdsl-openapi:generate-yaml

The **camel-restdsl-openapi:generate-yaml** goal of the Camel REST DSL OpenApi Maven Plugin is used to generate REST DSL YAML implementation source code from Maven.

1.8.12. Options

The plugin supports the following options which can be configured from the command line (use **-D** syntax), or defined in the **pom.xml** file in the **<configuration>** tag.

Parameter	Default Value	Description
skip	**false**	Set to **true** to skip code generation.
filterOperation		Used for including only the operation ids specified. Multiple ids can be separated by comma. Wildcards can be used, eg **find*** to include all operations starting with **find**.
specificationUri	**src/spec/openapi.json**	URI of the OpenApi specification, supports filesystem paths, HTTP and classpath resources, by default **src/spec/openapi.json** within the project directory. Supports JSON and YAML.

Parameter	Default Value	Description
auth		Adds authorization headers when fetching the OpenApi specification definitions remotely. Pass in a URL-encoded string of name:header with a comma separating multiple values.
outputDirectory	**generated-sources/restdsl-openapi**	Where to place the generated source file, by default **generated-sources/restdsl-openapi** within the project directory
fileName	**camel-rest.xml**	The name of the XML file as output.
destinationGenerator		Fully qualified class name of the class that implements **org.apache.camel.generator.openapi.DestinationGenerator** interface for customizing destination endpoint
destinationToSyntax	**direct:${operationId}**	The default to syntax for the to uri, which is to use the direct component.
	restConfiguration	**true**
Whether to include generation of the rest configuration with detected rest component to be used.	**apiContextPath**	
Define openapi endpoint path if **restConfiguration** is set to **true**.	**clientRequestValidation**	**false**
Whether to enable request validation.	**basePath**	
Overrides the api base path as defined in the OpenAPI specification.	**requestMappingValues**	/**

1.8.13. camel-restdsl-openapi:generate-yaml-with-dto

Works as **generate-yaml** goal but also generates DTO model classes by automatic executing the swagger-codegen-maven-plugin to generate java source code of the DTO model classes from the OpenApi specification.

This plugin has been scoped and limited to only support a good effort set of defaults for using the **swagger-codegen-maven-plugin** to generate the model DTOs. If you need more power and flexibility then use the Swagger Codegen Maven Plugin directly to generate the DTO and not this plugin.

The DTO classes may require additional dependencies such as:

```
<dependency>
  <groupId>com.google.code.gson</groupId>
  <artifactId>gson</artifactId>
  <version>2.10.1</version>
</dependency>
<dependency>
  <groupId>io.swagger.core.v3</groupId>
  <artifactId>swagger-core</artifactId>
  <version>2.2.8</version>
</dependency>
<dependency>
  <groupId>org.threeten</groupId>
  <artifactId>threetenbp</artifactId>
  <version>1.6.8</version>
</dependency>
```

1.8.14. Options

The plugin supports the following **additional** options

Parameter	Default Value	Description
swaggerCodegenMavenPluginVersion	3.0.36	The version of the **io.swagger.codegen.v3:swagger-codegen-maven-plugin** maven plugin to be used.
modelOutput		Target output path (default is ${project.build.directory}/generated-sources/openapi)
modelPackage	**io.swagger.client.model**	The package to use for generated model objects/classes
modelNamePrefix		Sets the pre- or suffix for model classes and enums
modelNameSuffix		Sets the pre- or suffix for model classes and enums

Parameter	Default Value	Description
modelWithXml	false	Enable XML annotations inside the generated models (only works with libraries that provide support for JSON and XML)
configOptions		Pass a map of language-specific parameters to **swagger-codegen-maven-plugin**

1.9. SUPPORT FOR FIPS COMPLIANCE

You can install an OpenShift Container Platform cluster that uses FIPS Validated / Modules in Process cryptographic libraries on the x86_64 architecture.

For the Red Hat Enterprise Linux CoreOS (RHCOS) machines in your cluster, this change applies when the machines deploy based on the status of an option in the install-config.yaml file, which governs the cluster options that users can change during cluster deployment. With Red Hat Enterprise Linux (RHEL) machines, you must enable FIPS mode when installing the operating system on the machines you plan to use as worker machines. These configuration methods ensure that your cluster meets the requirements of a FIPS compliance audit. Only FIPS Validated / Modules in Process cryptography packages are enabled before the initial system boot.

Because you must enable FIPS before your cluster's operating system boots for the first time, you cannot enable FIPS after you deploy a cluster.

1.9.1. FIPS validation in OpenShift Container Platform

OpenShift Container Platform uses certain FIPS Validated / Modules in Process modules within RHEL and RHCOS for its operating system components. For example, when users SSH into OpenShift Container Platform clusters and containers, those connections are properly encrypted.

OpenShift Container Platform components are written in Go and built with Red Hat's Golang compiler. When you enable FIPS mode for your cluster, all OpenShift Container Platform components that require cryptographic signing call RHEL and RHCOS cryptographic libraries.

For more details about FIPS, see FIPS mode attributes and limitations

For details on deploying Camel Spring Boot on OpenShift, see How to deploy a Camel Spring Boot application to OpenShift?

Details about supported configurations can be found at, Camel for Spring Boot Supported Configurations

CHAPTER 2. SETTING UP MAVEN LOCALLY

Maven is the typical choice for Red Hat build of Apache Camel application development and project management.

2.1. PREPARING TO SET UP MAVEN

Maven is a free, open source, build tool from Apache.

Procedure

1. Download Maven 3.8.6 or later from the Maven download page .

   TIP

   To verify that you have the correct Maven and JDK version installed, open a command terminal and enter the following command:

   ```
   mvn --version
   ```

   Check the output to verify that Maven is version 3.8.6 or newer, and is using OpenJDK 17.

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 2.4, “Using local Maven repositories” .

2.2. 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.

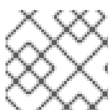
Maven looks for the **settings.xml** file in the **.m2** directory of the user’s home directory. If there is not a user specified **settings.xml** file, Maven uses the system-level **settings.xml** file at **M2_HOME/conf/settings.xml**.

Prerequisite

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

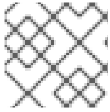
Procedure

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



NOTE

If you are using the **camel-jira** component, also add the atlassian repository.



NOTE

If you want to use technology preview builds, also add the **earlyaccess** repository.

```
<?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>atlassian</id>
            <url>https://packages.atlassian.com/maven-external/</url>
            <name>atlassian external repo</name>
            <snapshots>
                <enabled>false</enabled>
            </snapshots>
            <releases>
                <enabled>true</enabled>
            </releases>
        </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>
  </profile>
</profiles>

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

</settings>
```

2.3. BUILDING AN OFFLINE MAVEN REPOSITORY

Red Hat build of Apache Camel for Spring Boot users can build their own offline Maven repository which is used in a restricted environment. For each release of Red Hat build of Apache Camel for Spring Boot users can download the zip file from the Red Hat Customer Portal.

Procedure

1. Download the offile Maven repository builder from the customer portal. For example, for Red Hat build of Camel Spring Boot version 4.4, use the Offline Maven builder.

2. The downloaded file is a zip file that contains everything to build an offline Maven repository for this specific release.

3. Unzip the downloaded zip file. The directory structure of the archive is as follows:

```
├── README
├── build-offline-repo.sh
├── errors.log
├── logback.xml
├── maven-repositories.txt
├── offliner-2.0-sources.jar
├── offliner-2.0-sources.jar.md5
├── offliner-2.0.jar
├── offliner-2.0.jar.md5
├── offliner.log
├── rhaf-camel-offliner-4.4.0.txt
└── rhaf-camel-spring-boot-offliner-4.4.0.txt
```

This zip contains the following files:

- build-offline-repo.sh - A wrapper script around the Offliner tool.

- offliner-2.0.jar - Downloads the artifacts in the manifest.
- redhat-camel-4.4.0-offline-manifest.txt
  - Lists the required artifacts that need to be downloaded.
- redhat-camel-spring-boot-4.4.0-offline-manifest.txt
  - Lists the required artifacts that need to be downloaded.
- README - Explains the steps and commands required for building the offline Maven repository.

4. To build an offline repository, run the **build-offline-repo.sh** script as per instructions given in the **README** file. Optionally you can specify a directory where the artifacts should be downloaded to. If not specified, a directory called 'repository' is created in the current working directory.

If needed, you can configure the tool to use additional Maven repositories, by adding them to file **maven-repositories.txt**. This is generally not necessary as the tool is pre-configured with the right set of Maven repositories.

In case of a HTTP proxy and any HTTP calls that need to go via this proxy, you may need to change the script. Add the arguments **--proxy <proxy-host> --proxy-user <proxy-user> --proxy-pass <proxy-pass>** in the line that invokes the JVM in the script.

You can use the option **-v** to print the version number of the script. This version is the version number of the script and not related to the Red Hat build of Apache Camel product version.

Troubleshooting

You can configure the logging via the provided **logback.xml** file. When the shell script is executed, any download activity will be written to the log file **offliner.log** and any download failures are listed in **errors.log**. At the end of the execution the offliner tool displays a summary of the downloaded and failed artifacts, but we also recommend to scan through **errors.log** for any download failures.

If any artifacts are failed to be downloaded, re-run the tool against the same target folder. The tool will avoid to download artifacts that it already downloaded and only attempt those that it failed on previously.

2.4. 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.

2.5. SETTING MAVEN MIRROR USING ENVIRONMENTAL VARIABLES OR SYSTEM PROPERTIES

When running the applications you need access to the artifacts that are in the Red Hat Maven repositories. These repositories are added to Maven's **settings.xml** file. Maven checks the following locations for **settings.xml** file:

- looks for the specified url
- if not found looks for **${user.home}/.m2/settings.xml**
- if not found looks for **${maven.home}/conf/settings.xml**
- if not found looks for **${M2_HOME}/conf/settings.xml**
- if no location is found, empty **org.apache.maven.settings.Settings** instance is created.

2.5.1. About Maven mirror

Maven uses a set of remote repositories to access the artifacts, which are currently not available in local repository. The list of repositories almost always contains Maven Central repository, but for Red Hat Fuse, it also contains Maven Red Hat repositories. In some cases where it is not possible or allowed to access different remote repositories, you can use a mechanism of Maven mirrors. A mirror replaces a particular repository URL with a different one, so all HTTP traffic when remote artifacts are being searched for can be directed to a single URL.

2.5.2. Adding Maven mirror to settings.xml

To set the Maven mirror, add the following section to Maven's **settings.xml**:

```
<mirror>
      <id>all</id>
      <mirrorOf>*</mirrorOf>
      <url>http://host:port/path</url>
</mirror>
```

No mirror is used if the above section is not found in the **settings.xml** file. To specify a global mirror without providing the XML configuration, you can use either system property or environmental variables.

2.5.3. Setting Maven mirror using environmental variable or system property

To set the Maven mirror using either environmental variable or system property, you can add:

- Environmental variable called **MAVEN_MIRROR_URL** to **bin/setenv** file
- System property called **mavenMirrorUrl** to **etc/system.properties** file

2.5.4. Using Maven options to specify Maven mirror url

To use an alternate Maven mirror url, other than the one specified by environmental variables or system property, use the following maven options when running the application:

- **-DmavenMirrorUrl=mirrorId::mirrorUrl**
  for example, **-DmavenMirrorUrl=my-mirror::http://mirror.net/repository**

- **-DmavenMirrorUrl=mirrorUrl**
  for example, **-DmavenMirrorUrl=http://mirror.net/repository**. In this example, the <id> of the <mirror> is just a mirror.

2.6. 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.

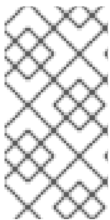
Elements in an artifact's POM file define the artifact's group ID, artifact ID, packaging, and version, as shown here:

```
<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

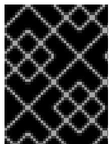
It is not necessary to specify the **bundle** package type in the preceding dependency, because a bundle is just a particular kind of JAR file and **jar** is the default Maven package type. If you do need to specify the packaging type explicitly in a dependency, however, you can use the **type** element.

CHAPTER 3. MONITORING CAMEL SPRING BOOT INTEGRATIONS

This chapter explains how to monitor integrations on Red Hat build of Camel Spring Boot at runtime. You can use the Prometheus Operator that is already deployed as part of OpenShift Monitoring to monitor your own applications.

3.1. ENABLING USER WORKLOAD MONITORING IN OPENSHIFT

You can enable the monitoring for user-defined projects by setting the **enableUserWorkload: true** field in the cluster monitoring ConfigMap object.



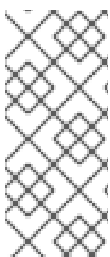
IMPORTANT

In OpenShift Container Platform 4.13 you must remove any custom Prometheus instances before enabling monitoring for user-defined projects.

Prerequisites

You must have access to the cluster as a user with the cluster-admin cluster role access to enable monitoring for user-defined projects in OpenShift Container Platform. Cluster administrators can then optionally grant users permission to configure the components that are responsible for monitoring user-defined projects.

- You have cluster admin access to the OpenShift cluster.
- You have installed the OpenShift CLI (oc).
- You have created the **cluster-monitoring-config** ConfigMap object.
- Optional: You have created and configured the **user-workload-monitoring-config** ConfigMap object in the **openshift-user-workload-monitoring** project. You can add configuration options to this ConfigMap object for the components that monitor user-defined projects.



NOTE

Every time you save configuration changes to the user-workload-monitoring-config ConfigMap object, the pods in the openshift-user-workload-monitoring project are redeployed. It can sometimes take a while for these components to redeploy. You can create and configure the ConfigMap object before you first enable monitoring for user-defined projects, to prevent having to redeploy the pods often.

Procedure

1. Login to OpenShift with administrator permissions.

```
oc login --user system:admin --token=my-token --server=https://my-
cluster.example.com:6443
```

2. Edit the **cluster-monitoring-config** ConfigMap object.

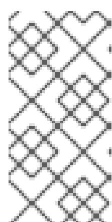
```
$ oc -n openshift-monitoring edit configmap cluster-monitoring-config
```

3. Add **enableUserWorkload: true** in the data/config.yaml section.

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: cluster-monitoring-config
  namespace: openshift-monitoring
data:
  config.yaml: |
    enableUserWorkload: true
```

When it is set to true, the **enableUserWorkload** parameter enables monitoring for user-defined projects in a cluster.

4. Save the file to apply the changes. The monitoring for the user-defined projects is then enabled automatically.



NOTE

When the changes are saved to the **cluster-monitoring-config** ConfigMap object, the pods and other resources in the **openshift-monitoring** project might be redeployed. The running monitoring processes in that project might also be restarted.

5. Verify that the **prometheus-operator**, **prometheus-user-workload** and **thanos-ruler-user-workload** pods are running in the **openshift-user-workload-monitoring** project.

```
$ oc -n openshift-user-workload-monitoring get pod

  Example output

  NAME                                   READY   STATUS        RESTARTS   AGE
  prometheus-operator-6f7b748d5b-t7nbg   2/2     Running       0          3h
  prometheus-user-workload-0             4/4     Running       1          3h
  prometheus-user-workload-1             4/4     Running       1          3h
  thanos-ruler-user-workload-0           3/3     Running       0          3h
  thanos-ruler-user-workload-1           3/3     Running       0          3h
```

3.2. DEPLOYING A CAMEL SPRING BOOT APPLICATION

After you enable the monitoring for your project, you can deploy and monitor the Camel Spring Boot application. This section uses the **monitoring-micrometrics-grafana-prometheus** example listed in the Camel Spring Boot Examples.

Procedure

1. Add the openshift-maven-plugin to the **pom.xml** file of the **monitoring-micrometrics-grafana-prometheus** example. In the **pom.xml**, add an **openshift** profile to allow deployment to openshift through the openshift-maven-plugin.

```
<profiles>
    <profile>
        <id>openshift</id>
        <build>
```

```
            <plugins>
                <plugin>
                    <groupId>org.eclipse.jkube</groupId>
                    <artifactId>openshift-maven-plugin</artifactId>
                    <version>1.13.1</version>
                    <executions>
                        <execution>
                            <goals>
                                <goal>resource</goal>
                                <goal>build</goal>
                            </goals>
                        </execution>
                    </executions>
                </plugin>
            </plugins>
        </build>
    </profile>
</profiles>
```

2. Add the Prometheus support. In order to add the Prometheus support to your Camel Spring Boot application, expose the Prometheus statistics on an actuator endpoint.

    a. Edit your **src/main/resources/application.properties** file. If you have a **management.endpoints.web.exposure.include** entry, add prometheus, metrics, and health. If you do not have a **management.endpoints.web.exposure.include** entry, please add one.

    ```
    # expose actuator endpoint via HTTP
    management.endpoints.web.exposure.include=mappings,metrics,health,shutdown,jolokia,prometheus
    ```

3. Add the following to the **<dependencies/>** section of your pom.xml to add some starter support to your application.

    ```
    <dependency>
        <groupId>io.micrometer</groupId>
        <artifactId>micrometer-registry-prometheus</artifactId>
    </dependency>

    <dependency>
        <groupId>org.jolokia</groupId>
        <artifactId>jolokia-core</artifactId>
        <version>${jolokia-version}</version>
    </dependency>

    <dependency>
        <groupId>io.prometheus.jmx</groupId>
        <artifactId>collector</artifactId>
        <version>${prometheus-version}</version>
    </dependency>
    ```

4. Add the following to the **Application.java** of your Camel Spring Boot application.

    ```
    import org.springframework.context.annonation.Bean;
    import org.apache.camel.component.micrometer.MicrometerConstants;
    ```

```
import
org.apache.camel.component.micrometer.eventnotifier.MicrometerExchangeEventNotifier;
import org.apache.camel.component.micrometer.eventnotifier.MicrometerRouteEventNotifier;
import
org.apache.camel.component.micrometer.messagehistory.MicrometerMessageHistoryFactory;

import org.apache.camel.component.micrometer.routepolicy.MicrometerRoutePolicyFactory;
```

5. The updated **Application.java** is shown below.

```
@SpringBootApplication
public class SampleCamelApplication {

@Bean(name = {MicrometerConstants.METRICS_REGISTRY_NAME,
"prometheusMeterRegistry"})
public PrometheusMeterRegistry prometheusMeterRegistry(
        PrometheusConfig prometheusConfig, CollectorRegistry collectorRegistry, Clock clock)
throws MalformedObjectNameException, IOException {

    InputStream resource = new
ClassPathResource("config/prometheus_exporter_config.yml").getInputStream();

    new JmxCollector(resource).register(collectorRegistry);
    new BuildInfoCollector().register(collectorRegistry);
    return new PrometheusMeterRegistry(prometheusConfig, collectorRegistry, clock);
}

@Bean
public CamelContextConfiguration camelContextConfiguration(@Autowired
PrometheusMeterRegistry registry) {

    return new CamelContextConfiguration() {
        @Override
        public void beforeApplicationStart(CamelContext camelContext) {
            MicrometerRoutePolicyFactory micrometerRoutePolicyFactory = new
MicrometerRoutePolicyFactory();
            micrometerRoutePolicyFactory.setMeterRegistry(registry);
            camelContext.addRoutePolicyFactory(micrometerRoutePolicyFactory);

            MicrometerMessageHistoryFactory micrometerMessageHistoryFactory = new
MicrometerMessageHistoryFactory();
            micrometerMessageHistoryFactory.setMeterRegistry(registry);
            camelContext.setMessageHistoryFactory(micrometerMessageHistoryFactory);

            MicrometerExchangeEventNotifier micrometerExchangeEventNotifier =  new
MicrometerExchangeEventNotifier();
            micrometerExchangeEventNotifier.setMeterRegistry(registry);

camelContext.getManagementStrategy().addEventNotifier(micrometerExchangeEventNotifier);


            MicrometerRouteEventNotifier micrometerRouteEventNotifier = new
MicrometerRouteEventNotifier();
            micrometerRouteEventNotifier.setMeterRegistry(registry);

camelContext.getManagementStrategy().addEventNotifier(micrometerRouteEventNotifier);
```

```
        }

        @Override
        public void afterApplicationStart(CamelContext camelContext) {
        }
    };
}
```

6. Deploy the application to OpenShift.

```
mvn -Popenshift oc:deploy
```

7. Verify if your application is deployed.

```
oc get pods -n myapp

NAME                                       READY   STATUS      RESTARTS   AGE
camel-example-spring-boot-xml-2-deploy     0/1     Completed   0          13m
camel-example-spring-boot-xml-2-x78rk      1/1     Running     0          13m
camel-example-spring-boot-xml-s2i-2-build  0/1     Completed   0          14m
```

8. Add the Service Monitor for this application so that Openshift's prometheus instance can start scraping from the / actuator/prometheus endpoint.

    a. Create the following YAML manifest for a Service monitor. In this example, the file is named as **servicemonitor.yaml**.

    ```
    apiVersion: monitoring.coreos.com/v1
    kind: ServiceMonitor
    metadata:
      labels:
        k8s-app: csb-demo-monitor
      name: csb-demo-monitor
    spec:
      endpoints:
      - interval: 30s
        port: http
        scheme: http
        path: /actuator/prometheus
      selector:
        matchLabels:
          app: camel-example-spring-boot-xml
    ```

    b. Add a Service Monitor for this application.

    ```
    oc apply -f servicemonitor.yml
    servicemonitor.monitoring.coreos.com/csb-demo-monitor "myapp" created
    ```

9. Verify that the service monitor was successfully deployed.

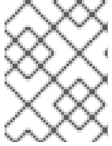
```
oc get servicemonitor

NAME                   AGE
```

```
csb-demo-monitor       9m17s
```

10. Verify that you can see the service monitor in the list of scrape targets. In the Administrator view, navigate to Observe → Targets. You can find **csb-demo-monitor** within the list of scrape targets.

11. Wait about ten minutes after deploying the servicemonitor. Then navigate to the Observe → Metrics in the Developer view. Select **Custom query** in the drop-down menu and type **camel** to view the Camel metrics that are exposed through the /actuator/prometheus endpoint.



NOTE

Red Hat does not offer support for installing and configuring Prometheus and Grafana on non-OCP environments.

CHAPTER 4. USING CAMEL WITH SPRING XML

Using Camel with Spring XML files is a way of using XML DSL with Camel. Camel has historically been using Spring XML for a long time. The Spring framework started with XML files as a popular and common configuration for building Spring applications.

Example of Spring application

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="
       http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
       http://camel.apache.org/schema/spring http://camel.apache.org/schema/spring/camel-spring.xsd
    ">

    <camelContext xmlns="http://camel.apache.org/schema/spring">
        <route>
            <from uri="direct:a"/>
            <choice>
                <when>
                    <xpath>$foo = 'bar'</xpath>
                    <to uri="direct:b"/>
                </when>
                <when>
                    <xpath>$foo = 'cheese'</xpath>
                    <to uri="direct:c"/>
                </when>
                <otherwise>
                    <to uri="direct:d"/>
                </otherwise>
            </choice>
        </route>
    </camelContext>

</beans>
```

4.1. USING JAVA DSL WITH SPRING XML FILES

You can use Java Code to define your RouteBuilder implementations. These are defined as beans in spring and then referenced in your camel context, as shown:

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
  <routeBuilder ref="myBuilder"/>
</camelContext>

<bean id="myBuilder" class="org.apache.camel.spring.example.test1.MyRouteBuilder"/>
```

4.1.1. Configure Spring Boot Application

To use Spring Boot Autoconfigure XML routes for beans, you musy import the XML resource. To do this, you can use a **Configuration** class.

For example, given that the Spring XML file is located to **src/main/resources/camel-context.xml** you can use the following configuration class to load the camel-context:

Example: using a Configuration class

```
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.ImportResource;


/**
 * A Configuration class that import the Spring XML resource
 */
@Configuration
// load the spring xml file from classpath
@ImportResource("classpath:camel-context.xml")
public class CamelSpringXMLConfiguration {
}
```

4.2. SPECIFYING CAMEL ROUTES USING SPRING XML

You can use Spring XML files to specify Camel routes using XML DSL as shown:

```
<camelContext id="camel-A" xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="seda:start"/>
    <to uri="mock:result"/>
  </route>
</camelContext>
```

4.3. CONFIGURING COMPONENTS AND ENDPOINTS

You can configure your Component or Endpoint instances in your Spring XML as follows in this example.

```
<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
</camelContext>

<bean id="jmsConnectionFactory"
class="org.apache.activemq.artemis.jms.client.ActiveMQConnectionFactory">
  <property name="brokerURL" value="tcp:someserver:61616"/>
</bean>
<bean id="jms" class="org.apache.camel.component.jms.JmsComponent">
  <property name="connectionFactory">
    <bean class="org.apache.activemq.artemis.jms.client.ActiveMQConnectionFactory">
  <property name="brokerURL" value="tcp:someserver:61616"/>
      </bean>
  </property>
</bean>
```

This allows you to configure a component using any name, but its common to use the same name, for example, **jms**. Then you can refer to the component using **jms:destinationName**.

This works by the Camel fetching components from the Spring context for the scheme name you use for Endpoint URIs.

4.4. USING PACKAGE SCANNING

Camel also provides a powerful feature that allows for the automatic discovery and initialization of routes in given packages. This is configured by adding tags to the camel context in your spring context definition, specifying the packages to be recursively searched for **RouteBuilder** implementations. To use this feature add a <package></package> tag specifying a comma separated list of packages that should be searched. For example,

```
<camelContext>
  <packageScan>
    <package>com.foo</package>
    <excludes>**.*Excluded*</excludes>
    <includes>**.*</includes>
  </packageScan>
</camelContext>
```

This scans for RouteBuilder classes in the **com.foo** and the sub-packages.

You can also filter the classes with includes or excludes such as:

```
<camelContext>
  <packageScan>
    <package>com.foo</package>
    <excludes>**.*Special*</excludes>
  </packageScan>
</camelContext>
```

This skips the classes that has Special in the name. Exclude patterns are applied before the include patterns. If no include or exclude patterns are defined then all the Route classes discovered in the packages are returned.

? matches one character, ***** matches zero or more characters, ****** matches zero or more segments of a fully qualified name.

4.5. USING CONTEXT SCANNING

You can allow Camel to scan the container context, for example, the Spring **ApplicationContext** for route builder instances. This allows you to use the Spring **<component-scan>** feature and have Camel pickup any RouteBuilder instances which was created by Spring in its scan process.

```
<!-- enable Spring @Component scan -->
<context:component-scan base-package="org.apache.camel.spring.issues.contextscan"/>

<camelContext xmlns="http://camel.apache.org/schema/spring">
    <!-- and then let Camel use those @Component scanned route builders -->
    <contextScan/>
</camelContext>
```

This allows you to just annotate your routes using the Spring **@Component** and have those routes included by Camel:

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

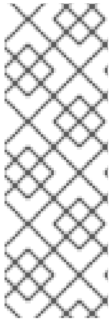
```
    @Override
    public void configure() throws Exception {
        from("direct:start")
            .to("mock:result");
    }
}
```

You can also use the ANT style for inclusion and exclusion, as mentioned above in the package scan section.

CHAPTER 5. XML IO DSL

The **xml-io-dsl** is the Camel optimized XML DSL with a very fast and low overhead XML parser. It is a source code generated parser that is Camel specific and can only parse Camel **.xml** route files (not classic Spring **<beans>** XML files).

We recommend that you use **xml-io-dsl** instead of **xml-jaxb-dsl** for Camel XML DSL. It works with all Camel runtimes.



NOTE

When you are using XML IO DSL, the **camel-spring-boot** application will by default look for xml files in **src/main/resources/camel/*.xml**.

You can configure this behavior by providing a different path in the **camel.springboot.routes-include-pattern** property:

camel.springboot.routes-include-pattern=/path/to/*.xml

5.1. EXAMPLE

The following **my-route.xml** source file can be loaded and run with Camel CLI or Camel K:

my-route.xml

```
<routes xmlns="http://camel.apache.org/schema/spring">
    <route>
        <from uri="timer:tick"/>
        <setBody>
            <constant>Hello Camel K!</constant>
         </setBody>
        <to uri="log:info"/>
    </route>
</routes>
```

TIP

You can omit the **xmlns** namespace.

If there is only a single route, you can use **<route>** as the root XML tag instead of **<routes>**.

Running with Camel K

```
kamel run my-route.xml
```

Running with Camel CLI

```
camel run my-route.xml
```

You can use **xml-io-dsl** to declare some beans to be bound to the Camel Registry.

You can declare and Beans define their properties (including **nested** properties) in XML. For example:

Bean declaration and definition

```
<camel>

 <bean name="beanFromProps" type="com.acme.MyBean">
  <properties>
   <property key="field1" value="f1_p" />
   <property key="field2" value="f2_p" />
   <property key="nested.field1" value="nf1_p" />
   <property key="nested.field2" value="nf2_p" />
  </properties>
 </bean>

</camel>
```

While keeping all the benefits of fast XML parser used by **xml-io-dsl**, Camel can also process XML elements declared in other XML namespaces and process them separately. With this mechanism it is possible to include XML elements using Spring's **http://www.springframework.org/schema/beans** namespace.

This brings the flexibility of Spring Beans into Camel main without actually running any Spring Application Context (or Spring Boot).

When elements from Spring namespace are found, they are used to populate and configure an instance of **org.springframework.beans.factory.support.DefaultListableBeanFactory** and leverage Spring dependency injection to wire the beans together.

These beans are then exposed through normal Camel Registry and may be used by Camel routes.

Here's an example **camel.xml** file, which defines both the routes and beans used (referred to) by the route definition:

camel.xml

```
<camel>

    <beans xmlns="http://www.springframework.org/schema/beans">
        <bean id="messageString" class="java.lang.String">
            <constructor-arg index="0" value="Hello"/>
        </bean>

        <bean id="greeter" class="org.apache.camel.main.app.Greeter">
            <description>Spring Bean</description>
            <property name="message">
                <bean class="org.apache.camel.main.app.GreeterMessage">
                    <property name="msg" ref="messageString"/>
                </bean>
            </property>
        </bean>
    </beans>

    <route id="my-route">
        <from uri="direct:start"/>
        <bean ref="greeter"/>
        <to uri="mock:finish"/>
```

```
    </route>

</camel>
```

A **my-route** route is referring to **greeter** bean which is defined using Spring **<bean>** element.

More examples can be found on the Apache Camel JBang page.

5.2. USING BEANS WITH CONSTRUCTORS

When you want to create beans with constructor arguments, from Camel 4.1 onwards you can add them as XML tags. For example:

Camel 4.1+: Beans with constructor tags

```
<camel>

 <bean name="beanFromProps" type="com.acme.MyBean">
       <constructors>
         <constructor index="0" value="true"/>
         <constructor index="1" value="Hello World"/>
       </constructors>
       <!-- and you can still have properties -->
  <properties>
   <property key="field1" value="f1_p" />
   <property key="field2" value="f2_p" />
   <property key="nested.field1" value="nf1_p" />
   <property key="nested.field2" value="nf2_p" />
  </properties>
 </bean>

</camel>
```

If you use Camel 4.0, you must put then constructor arguments in the **type** attribute:

Camel 4.0: Beans with constructor arguments in the type attribute

```
<bean name="beanFromProps" type="com.acme.MyBean(true, 'Hello World')">
    <properties>
       <property key="field1" value="f1_p" />
       <property key="field2" value="f2_p" />
       <property key="nested.field1" value="nf1_p" />
       <property key="nested.field2" value="nf2_p" />
    </properties>
</bean>
```

5.3. CREATING BEANS FROM FACTORY METHOD

A bean can also be created from a **public static** factory method:

Factory method XML

```
 <bean name="myBean" type="com.acme.MyBean" factoryMethod="createMyBean">
```

```
        <constructors>
          <constructor index="0" value="true"/>
          <constructor index="1" value="Hello World"/>
        </constructors>
    </bean>
```

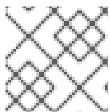
When you use a **factoryMethod**, you must provide **constructor** tags for the arguments.

For example, this means that the class **com.acme.MyBean** should be as follows:

Factory method

```
public class MyBean {

    public static MyBean createMyBean(boolean important, String message) {
        MyBean answer = ...
        // create and configure the bean
        return answer;
    }
}
```



NOTE

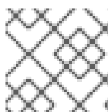
You must make the factory method **public static** in the created class.

5.4. CREATING BEANS FROM BUILDER CLASSES

You can create a bean created from another builder class as shown below:

Builder XML

```
<bean name="myBean" type="com.acme.MyBean"
        builderClass="com.acme.MyBeanBuilder" builderMethod="createMyBean">
      <properties>
        <property key="id" value="123"/>
        <property key="name" value="Acme"/>
      </constructors>
</bean>
```



NOTE

You must make the builder class **public** with a no-arg default constructor.

You can then use the builder class to create the actual bean by using fluent builder style configuration.

Set the properties on the builder class, and create the bean by invoking the **builderMethod** at the end.

You invocate this method via Java reflection.

5.5. CREATING BEANS FROM FACTORY BEAN

You can create a bean from a factory bean as shown below:

Factory XML

```
<bean name="myBean" type="com.acme.MyBean"
      factoryBean="com.acme.MyHelper" factoryMethod="createMyBean">
    <constructors>
      <constructor index="0" value="true"/>
      <constructor index="1" value="Hello World"/>
    </constructors>
</bean>
```

TIP

You can also use **factoryBean** to refer to an existing bean by bean id instead of the FQN classname.

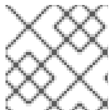
When you use a **factoryBean** the, you must provide arguments as **constructor** tags.

For example, the class **com.acme.MyHelper** should be as follows:

Factory bean

```
public class MyHelper {

    public static MyBean createMyBean(boolean important, String message) {
        MyBean answer = ...
        // create and configure the bean
        return answer;
    }
}
```



NOTE

You must make the factory method **public static**.

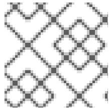
5.6. CREATING BEANS USING SCRIPT LANGUAGE

If you have advanced use-cases, you can inline a script language, such as groovy, java, javascript, and so on, to create the bean.

With scripting, you can be more flexible and use a bit of programming to create and configure the bean:

Scripting

```
<bean name="myBean" type="com.acme.MyBean" scriptLanguage="groovy">
    <script>
  // some groovy script here to create the bean
  bean = ...
  ...
  return bean
    </script>
</bean>
```



NOTE

When you use **script**, the constructors, factory bean, and factory method are not used.

5.7. USING INIT AND DESTROY METHODS ON BEANS

If you need to do initialization and cleanup work before you use a bean, you can use the **initMethod** and **destroyMethod** which are triggered as appropriate by Camel.

Those methods must be **public void** and have no arguments, as shown below:

Initialization and cleanup methods

```
public class MyBean {

    public void initMe() {
        // do init work here
    }

    public void destroyMe() {
        // do cleanup work here
    }

}
```

You also have to declare those methods in the XML DSL as follows:

Initialization and cleanup XML

```
<bean name="myBean" type="com.acme.MyBean"
        initMethod="initMe" destroyMethod="destroyMe">
      <constructors>
        <constructor index="0" value="true"/>
        <constructor index="1" value="Hello World"/>
      </constructors>
</bean>
```

Both **initMethod** and **destroyMethod** are optional, so a bean does not have to have both.

5.8. REST AND ROUTES IN THE SAME XML IO DSL FILE

You can have both REST and routes in the same DSL file:

REST and routes in the same XML IO DSL file

```
<camel xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns="http://camel.apache.org/schema/spring"
       xsi:schemaLocation="
           http://camel.apache.org/schema/spring
           https://camel.apache.org/schema/spring/camel-spring.xsd">
  <rest id="rest">
      <post id="post" path="start">
          <to uri="direct:start"/>
      </post>
```

```
    </rest>

    <route>
        <from uri="direct:start"/>
        <to uri="amqp:queue:Test.Broker.StreamMessage?
jmsMessageType=Stream&amp;disableReplyTo=true"/>
    </route>
</camel>
```