Red Hat build of Apache Camel 4.0

Getting Started with Red Hat build of Apache Camel for Quarkus

Getting Started with Red Hat build of Apache Camel for Quarkus

Last Updated: 2023-12-04

Red Hat build of Apache Camel 4.0 Getting Started with Red Hat build of Apache Camel for Quarkus

Getting Started with Red Hat build of Apache Camel for Quarkus

Legal Notice

Abstract

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

Table of Contents

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 QUARKUS

This guide introduces Red Hat build of Apache Camel for Quarkus, the various ways to create a project and how to get started building an application using Red Hat build of Apache Camel for Quarkus:

- Section 1.1, “Red Hat build of Apache Camel for Quarkus overview”
- Section 1.2, “Tooling”
- Section 1.4, “Building your first project with Red Hat build of Apache Camel for Quarkus”

NOTE

Red Hat provides Maven repositories that host the content we ship with our products. These repositories are available to download from the software downloads page.

For Red Hat build of Apache Camel for Quarkus the following repositories are required:

- rhi-camel-extensions-for-quarkus

In this release, running Red Hat build of Apache Camel for Quarkus in a disconnected environment (offline mode) is supported, but building Red Hat build of Apache Camel for Quarkus in offline mode is not supported.

For information on using the Apache Maven repository for Camel Quarkus, see Reconfiguring your Maven project to Red Hat build of Quarkus

Chapter 2.2. “Downloading and configuring the Quarkus Maven repository” in the *Developing and compiling your Quarkus applications with Apache Maven* guide

1.1. RED HAT BUILD OF APACHE CAMEL FOR QUARKUS OVERVIEW

Red Hat build of Apache Camel for Quarkus brings the integration capabilities of Apache Camel and its vast component library to the Quarkus runtime.

The benefits of using Red Hat build of Apache Camel for Quarkus include the following:

- Enables users to take advantage of the performance benefits, developer joy and the container first ethos which Quarkus provides.
- Provides Quarkus extensions for many of the Apache Camel components.
- Takes advantage of the many performance improvements made in Camel 3, which results in a lower memory footprint, less reliance on reflection and faster startup times.
- You can define Camel routes using the Java DSL.

1.2. TOOLING

IMPORTANT

Red Hat does not provide support for these developer tools.

1.3. FIRST STEPS

1.3.1. IDE plugins

Quarkus has plugins for most of the popular development IDEs which provide Quarkus language support, code/configuration completion, project creation wizards and much more. The plugins are available at each respective IDE marketplace.

- Eclipse plugin
- IntelliJ plugin
- VS Code extension

Check the plugin documentation to discover how to create projects for your preferred IDE.

1.3.2. Camel content assist

The following plugins provide support for content assist when editing Camel routes and **application.properties**:

- VS Code Language support for Camel - a part of the Camel extension pack
- Debug Adapter for Apache Camel to debug Camel integrations written in Java, YAML or XML locally.
    - For more information about scope of development support, see Development Support Scope of Coverage
- Eclipse Desktop Language Support for Camel - a part of Jboss Tools and CodeReady Studio
- Apache Camel IDEA plugin (not always up to date)
- Users of other IDEs supporting Language Server Protocol may choose to install and configure Camel Language Server manually

1.4. BUILDING YOUR FIRST PROJECT WITH RED HAT BUILD OF APACHE CAMEL FOR QUARKUS

1.4.1. Overview

You can use code.quarkus.redhat.com to generate a Quarkus Maven project which automatically adds and configures the extensions that you want to use in your application.

This section walks you through the process of creating a Quarkus Maven project with Red Hat build of Apache Camel for Quarkus including:

- Creating the skeleton application using code.quarkus.redhat.com
- Adding a simple Camel route
- Exploring the application code
- Compiling the application in development mode

- Testing the application

1.4.2. Generating the skeleton application with code.quarkus.redhat.com

You can bootstrap and generate projects on code.quarkus.redhat.com.

The Red Hat build of Apache Camel for Quarkus extensions are located under the 'Integration' heading.

If you need additional extensions, use the 'search' field to find them.

Select the component extensions that you want to work with and click 'Generate your application' to download a basic skeleton project.

You can also push the project directly to GitHub.

For more information about using **code.quarkus.redhat.com** to generate Quarkus Maven projects, see Creating a Quarkus Maven project using code.quarkus.redhat.com in the *Getting started with Red Hat build of Quarkus* guide.

Procedure

1. In the code.quarkus.redhat.com website, select the following extensions:
   - **camel-quarkus-rest**
   - **camel-quarkus-jackson**
   - **camel-quarkus-direct**

   NOTE

   Do not compile the application on code.quarkus.redhat.com (in the final step of the procedure). Instead, use the compile command described in the Section 1.4.5, "Development mode" section below.

2. Navigate to the directory where you extracted the generated project files from the previous step:

```
$ cd <directory_name>
```

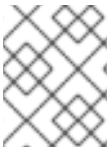
1.4.3. Explore the application code

The application has two compile dependencies which are managed within the **com.redhat.quarkus.platform:quarkus-camel-bom** that is imported in **<dependencyManagement>**.:

pom.xml

```
<quarkus.platform.artifact-id>quarkus-bom</quarkus.platform.artifact-id>
<quarkus.platform.group-id>com.redhat.quarkus.platform</quarkus.platform.group-id>
<quarkus.platform.version>
    <!-- The latest 3.2.x version from
https://maven.repository.redhat.com/ga/com/redhat/quarkus/platform/quarkus-bom -->
</quarkus.platform.version>
```

```
...

<dependency>
    <groupId>${quarkus.platform.group-id}</groupId>
    <artifactId>${quarkus.platform.artifact-id}</artifactId>
    <version>${quarkus.platform.version}</version>
    <type>pom</type>
    <scope>import</scope>
</dependency>
<dependency>
    <groupId>${quarkus.platform.group-id}</groupId>
    <artifactId>quarkus-camel-bom</artifactId>
    <version>${quarkus.platform.version}</version>
    <type>pom</type>
    <scope>import</scope>
</dependency>
```



NOTE

For more information about BOM dependency management, see Developing Applications with Red Hat build of Apache Camel for Quarkus

The application is configured by properties defined within **src/main/resources/application.properties**, for example, the **camel.context.name** can be set there.

1.4.4. Adding a simple Camel route

Procedure

1. Create a file named **Routes.java** in the **src/main/java/org/acme/** subfolder.
2. Add a Camel Rest route as shown in the following code snippet:

Routes.java

```
package org.acme;

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

import org.apache.camel.builder.RouteBuilder;
import org.apache.camel.model.rest.RestBindingMode;

import io.quarkus.runtime.annotations.RegisterForReflection;

public class Routes extends RouteBuilder {
    private final List<Fruit> fruits = new CopyOnWriteArrayList<>(Arrays.asList(new
Fruit("Apple")));

    @Override
    public void configure() throws Exception {
        restConfiguration().bindingMode(RestBindingMode.json);
```

```
        rest("/customers/")
            .get("/{id}").to("direct:customerDetail")
            .get("/{id}/orders").to("direct:customerOrders")
            .post("/neworder").to("direct:customerNewOrder");

    }

    @RegisterForReflection // Let Quarkus register this class for reflection during the native
build
    public static class Fruit {
        private String name;

        public Fruit() {
        }

        public Fruit(String name) {
            this.name = name;
        }

        public String getName() {
            return name;
        }

        public void setName(String name) {
            this.name = name;
        }

        @Override
        public int hashCode() {
            return Objects.hash(name);
        }

        @Override
        public boolean equals(Object obj) {
            if (this == obj)
                return true;
            if (obj == null)
                return false;
            if (getClass() != obj.getClass())
                return false;
            Fruit other = (Fruit) obj;
            return Objects.equals(name, other.name);
        }

    }

}
```

1.4.5. Development mode

```
$ mvn clean compile quarkus:dev
```

This command compiles the project, starts your application, and lets the Quarkus tooling watch for changes in your workspace. Any modifications you make to your project will automatically take effect in the running application.

You can check the application in your browser. (For example, for the **rest-json** sample application, access **http://localhost:8080/fruits**)

If you change the application code, for example, change 'Apple' to 'Orange', your application automatically updates. To see the changes applied, refresh your browser.

Refer to Quarkus documentation Development mode section for more details about the development mode.

1.4.6. Testing

1.4.6.1. JVM mode

To test the Camel Rest route that we have created in JVM mode, add a test class as follows:

Procedure

1. Create a file named **RoutesTest.java** in the **src/test/java/org/acme/** subfolder.
2. Add the **RoutesTest** class as shown in the following code snippet:

RoutesTest.java

```
package org.acme;

import io.quarkus.test.junit.QuarkusTest;
import org.junit.jupiter.api.Test;

import static io.restassured.RestAssured.given;
import org.hamcrest.Matchers;

@QuarkusTest
public class RoutesTest {

    @Test
    public void testFruitsEndpoint() {

        /* Assert the initial fruit is there */
        given()
                .when().get("/fruits")
                .then()
                .statusCode(200)
                .body(
                        "$.size()", Matchers.is(1),
                        "name", Matchers.contains("Orange"));

        /* Add a new fruit */
        given()
                .body("{\"name\": \"Pear\"}")
                .header("Content-Type", "application/json")
                .when()
```

```
                .post("/fruits")
                .then()
                .statusCode(200);

        /* Assert that pear was added */
        given()
                .when().get("/fruits")
                .then()
                .statusCode(200)
                .body(
                        "$.size()", Matchers.is(2),
                        "name", Matchers.contains("Orange", "Pear"));
    }

}
```

The JVM mode tests are run by **maven-surefire-plugin** in the **test** Maven phase:

```
$ mvn clean test
```

1.4.6.2. Native mode

To test the Camel Rest route that we have created in Native mode, add a test class as follows:

Procedure

1. Create a file named **NativeRoutesIT.java** in the **src/test/java/org/acme/** subfolder.
2. Add the **NativeRoutesIT** class as shown in the following code snippet:

   NativeRoutesIT.java

   ```
   package org.acme;

   import io.quarkus.test.junit.NativeImageTest;

   @NativeImageTest
   public class NativeRoutesIT extends RoutesTest {

       // Execute the same tests but in native mode.
   }
   ```

   The native mode tests are verified by **maven-failsafe-plugin** in the **verify** phase.

3. Pass the **native** property to activate the profile that runs them:

   ```
   $ mvn clean verify -Pnative
   ```

TIP

For more details, and how to use the **CamelTestSupport** style of testing, see Testing Camel Quarkus Extensions.

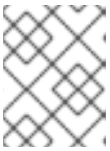
1.4.7. Packaging and running the application

1.4.7.1. JVM mode

Procedure

1. Run **mvn package** to prepare a thin **jar** for running on a stock JVM:

```
$ mvn clean package
$ ls -lh target/quarkus-app
...
-rw-r--r--. 1 user user 238K Oct 11 18:55  quarkus-run.jar
...
```

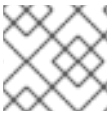


NOTE

The thin **jar** contains just the application code. You also need the dependencies in **target/quarkus-app/lib** to run it.

2. Run the jar as follows:

```
$ java -jar target/quarkus-app/quarkus-run.jar
...
[io.quarkus] (main) Quarkus started in 1.163s. Listening on: http://[::]:8080
```



NOTE

The boot time should be around a second.

1.4.7.2. Native mode

Procedure

To prepare a native executable, do as follows:

1. Run the command **mvn clean package -Pnative**:

```
$ mvn clean package -Pnative
$ ls -lh target
...
-rwxr-xr-x. 1 user user  46M Oct 11 18:57  code-with-quarkus-1.0.0-SNAPSHOT-runner
...
```

NOTE

The **runner** has no **.jar** extension and has the **x** (executable) permission set. You can run it directly:

```
$ ./target/*-runner
...
[io.quarkus] (main) Quarkus started in 0.013s. Listening on: http://[::]:8080
```

```
...
```

The application started in 13 milliseconds.

2. View the memory usage with the **ps -o rss,command -p $(pgrep code-with)** command :

```
$ ps -o rss,command -p $(pgrep code-with)
  RSS COMMAND
65852 ./target/code-with-quarkus-1.0.0-SNAPSHOT-runner
```

The application uses 65 MB of memory.

TIP

See Producing a native executable in the *Compiling your Quarkus applications to native executables* guide for additional information about preparing a native executable.

TIP

Quarkus Native executable guide contains more details, including steps for creating a container image .

1.5. TESTING CAMEL QUARKUS EXTENSIONS

Testing offers a good way to ensure Camel routes behave as expected over time. If you haven't already, read the Camel Quarkus user guide First Steps and the Quarkus documentation link: Testing your application section.

The easiest way of testing a route in Quarkus is to write local integration tests. This has the advantage of covering both JVM and native mode.

In JVM mode, you can use the **CamelTestSupport** style of testing.

1.5.1. Running in JVM mode

In JVM mode, use the **@QuarkusTest** annotation to bootstrap Quarkus and start Camel routes *before* the **@Test** logic executes.

For example:

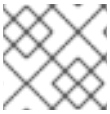
```
import io.quarkus.test.junit.QuarkusTest;
import org.junit.jupiter.api.Test;

@QuarkusTest
class MyTest {
    @Test
    public void test() {
        // Use any suitable code that sends test data to the route and then assert outcomes
        ...
    }
}
```

TIP

You can find a sample implementation in the Camel Quarkus source:

- MessageTest.java

1.5.2. Running in native mode



NOTE

Always test that your application works in native mode for all supported extensions.

You can reuse the test logic defined for JVM mode by inheriting the logic from the respective JVM mode class.

Add the **@QuarkusIntegrationTest** annotation to tell the Quarkus JUnit extension to compile the application under test to native image and start it before running the tests.

```
import io.quarkus.test.junit.QuarkusIntegrationTest;

@QuarkusIntegrationTest
class MyIT extends MyTest {
    ...
}
```

TIP

You can find a sample implementation in the Camel Quarkus source:

- MessageRecordIT.java

1.5.3. Differences between @QuarkusTest and @QuarkusIntegrationTest

A native executable does not need a JVM to run, and cannot run in a JVM, because it is native code, not bytecode.

There is no point in compiling tests to native code so they run using a traditional JVM.

This means that communication between tests and the application must go over the network (HTTP/REST, or any other protocol your application speaks), through watching filesystems (log files for example), or any other interprocess communication.

1.5.3.1. @QuarkusTest in JVM mode

In JVM mode, tests annotated with **@QuarkusTest** execute in the same JVM as the application under test.

This means you can use **@Inject** to add beans from the application into the test code.

You can also define new beans or even override the beans from the application using **@javax.enterprise.inject.Alternative** and **@javax.annotation.Priority**.

1.5.3.2. @QuarkusIntegrationTest in native mode

In native mode, tests annotated with **@QuarkusIntegrationTest** execute in a JVM hosted in a process separate from the running native application.

QuarkusIntegrationTest provides additional features that are not available through **@QuarkusTest**:

- In JVM mode, you can launch and test the runnable application JAR produced by the Quarkus build.
- In native mode, you can launch and test the native application produced by the Quarkus build.
- If you add a container image to the build, a container starts, and tests execute against it.

For more information about **QuarkusIntegrationTest**, see the Quarkus testing guide.

1.5.4. Testing with external services

1.5.4.1. Testcontainers

Sometimes your application needs to access some external resource, such as a messaging broker, a database, or other service.

If a container image is available for the service of interest, you can use Testcontainers to start and configure the services during testing.

1.5.4.1.1. Passing configuration data with QuarkusTestResourceLifecycleManager

For the application to work properly, it is often essential to pass the connection configuration data (host, port, user, password of the remote service) to the application before it starts.

In the Quarkus ecosystem, **QuarkusTestResourceLifecycleManager** serves this purpose.

You can start one or more Testcontainers in the **start()** method and return the connection configuration from the method in the form of a **Map**.

The entries of this map are then passed to the application in different ways depending on the mode:

- Native mode: a command line (**-Dkey=value**)
- JVM Mode: a special MicroProfile configuration provider

NOTE

These settings have a higher precedence than the settings in the **application.properties** file.

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

import io.quarkus.test.common.QuarkusTestResourceLifecycleManager;
import org.testcontainers.containers.GenericContainer;
import org.testcontainers.containers.wait.strategy.Wait;

public class MyTestResource implements QuarkusTestResourceLifecycleManager {

    private GenericContainer myContainer;
```

```
    @Override
    public Map<String, String> start() {
        // Start the needed container(s)
        myContainer = new GenericContainer(...)
                .withExposedPorts(1234)
                .waitingFor(Wait.forListeningPort());

        myContainer.start();

        // Pass the configuration to the application under test
        return new HashMap<>() {{
                put("my-container.host", container.getContainerIpAddress());
                put("my-container.port", "" + container.getMappedPort(1234));
        }};
    }

    @Override
    public void stop() {
        // Stop the needed container(s)
        myContainer.stop();
        ...
    }
```

Reference the defined test resource from the test classes with **@QuarkusTestResource**:

```
import io.quarkus.test.common.QuarkusTestResource;
import io.quarkus.test.junit.QuarkusTest;

@QuarkusTest
@QuarkusTestResource(MyTestResource.class)
class MyTest {
    ...
}
```

TIP

You can find a sample implementation in the Camel Quarkus source:

- NatsTestResource.java

1.5.4.2. WireMock

Instead of having the tests connect to live endpoints, for example, if they are unavailable, unreliable, or expensive, you can stub HTTP interactions with third-party services & APIs.

You can use WireMock for mocking & recording HTTP interactions. It is used extensively throughout the Camel Quarkus test suite for various component extensions.

1.5.4.2.1. Setting up WireMock

Procedure

1. Set up the WireMock server.

NOTE

It is important to configure the Camel component under test to pass any HTTP interactions through the WireMock proxy. You can achieve this by configuring a component property that determines the API endpoint URL.

```
import static com.github.tomakehurst.wiremock.client.WireMock.aResponse;
import static com.github.tomakehurst.wiremock.client.WireMock.get;
import static com.github.tomakehurst.wiremock.client.WireMock.urlEqualTo;
import static
com.github.tomakehurst.wiremock.core.WireMockConfiguration.wireMockConfig;

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

import com.github.tomakehurst.wiremock.WireMockServer;

import io.quarkus.test.common.QuarkusTestResourceLifecycleManager;

public class WireMockTestResource implements QuarkusTestResourceLifecycleManager {

    private WireMockServer server;

    @Override
    public Map<String, String> start() {
        // Setup & start the server
        server = new WireMockServer(
            wireMockConfig().dynamicPort()
        );
        server.start();

        // Stub a HTTP endpoint. Note that WireMock also supports a record and playback mode
        // http://wiremock.org/docs/record-playback/
        server.stubFor(
            get(urlEqualTo("/api/greeting"))
                .willReturn(aResponse()
                    .withHeader("Content-Type", "application/json")
                    .withBody("{\"message\": \"Hello World\"}")));

        // Ensure the camel component API client passes requests through the WireMock proxy
        Map<String, String> conf = new HashMap<>();
        conf.put("camel.component.foo.server-url", server.baseUrl());
        return conf;
    }

    @Override
    public void stop() {
        if (server != null) {
            server.stop();
        }
    }
}
```

NOTE

Sometimes things are less straightforward, and some extra work is required to configure the API client library. For example, for Twilio.

2. Ensure your test class has the **@QuarkusTestResource** annotation with the appropriate test resource class specified as the value.

```
import io.quarkus.test.common.QuarkusTestResource;
import io.quarkus.test.junit.QuarkusTest;

@QuarkusTest
@QuarkusTestResource(WireMockTestResource.class)
class MyTest {
    ...
}
```

The WireMock server starts before all tests execute and shuts down when all tests finish.

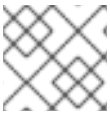
TIP

You can find a sample implementation in the Camel Quarkus integration test source tree:

- Geocoder.

1.5.5. Using CamelQuarkusTestSupport

Since Camel Quarkus 2.13.0, you can use **CamelQuarkusTestSupport** for testing. It is a replacement for **CamelTestSupport**.



NOTE

This will only work in JVM mode.

1.5.5.1. Testing with CamelQuarkusTestSupport in JVM mode

Add the following dependency into your module (preferably in the **test** scope):

```
<dependency>
    <groupId>org.apache.camel.quarkus</groupId>
    <artifactId>camel-quarkus-junit5</artifactId>
    <scope>test</scope>
</dependency>
```

You can use **CamelQuarkusTestSupport** in your test like this:

```
@QuarkusTest
@TestProfile(SimpleTest.class) //necessary only if "newly created" context is required for the test (worse performance)
public class SimpleTest extends CamelQuarkusTestSupport {
    ...
}
```

1.5.5.2. Limitations when using CamelQuarkusTestSupport

When using `CamelQuarkusTestSupport, there are several limitations:

1.5.5.2.1. Methods

Some methods do not execute. Use the new methods starting with **do** instead:

Not executed	Use instead
afterAll	**doAfterAll**
afterEach	**doAfterEach**
afterTestExecution	**doAfterTestExecution**
beforeAll	**doBeforeAll**
beforeEach	**doBeforeEach**

NOTE

If you use **@TestInstance(TestInstance.Lifecycle.PER_METHOD)**, **doAfterConstruct** means a callback before each test. This is different from **beforeAll**.

1.5.5.2.2. Annotations

You must annotate the test class with **@io.quarkus.test.junit.QuarkusTest** and extend **org.apache.camel.quarkus.test.CamelQuarkusTestSupport**.

1.5.5.2.3. Starting and stopping

- You cannot stop and restart the same **CamelContext** instance within the life cycle of a single application. You can call **CamelContext.stop()**, but **CamelContext.start()** won't work.
- **CamelContext** is generally bound to starting and stopping the application, also when testing.
- The application under test starts once for all test classes of the given Maven/Gradle module. Quarkus JUnit Extension controls the start and stop of the application. You must explicitly tell the application to stop.

1.5.5.2.4. Restarting the application

To force Quarkus JUnit Extension to restart the application and **CamelContext** for a given test class, you need to assign a unique **@io.quarkus.test.junit.TestProfile** to that class.

For instructions, see testing different profiles in the Quarkus documentation.

For a similar effect, you can also use **@io.quarkus.test.common.QuarkusTestResource**.

1.5.5.2.5. Beans production

Camel Quarkus executes the production of beans during the build phase. Because the tests are built together, exclusion behavior is implemented into **CamelQuarkusTestSupport**. If a producer of the specific type and name is used in one test, the instance will be the same for the rest of the tests.

1.5.5.2.6. JUnit Jupiter callbacks may not work

These JUnit Jupiter callbacks and annotations may not work:

Callbacks	Annotations
BeforeEachCallback	**@BeforeEach**
AfterEachCallback	**@AfterEach**
AfterAllCallback	**@AfterAll**
BeforeAllCallback	**@BeforeAll**
BeforeTestExecutionCallback	
AfterTestExecutionCallback	

For more information, see the Enrichment via QuarkusTest*Callback documentation.

1.5.5.2.7. Using adviceWith

When **adviceWith** is set to true, all unadvised routes do not start. You must execute the method **CamelQuarkusTestSupport.startRouteDefinitions()** for those routes to start them.

1.5.5.2.8. Using @Produces

Use **@Produces** with the overridden method **createRouteBuilder()**. The combination of **@Produces** and **RouteBuilder()** may not work correctly.

1.5.5.2.9. Configuring routes

To configure which routes from the application (**src/main/java**) to include or exclude, you can use the following:

- **quarkus.camel.routes-discovery.exclude-patterns**
- **quarkus.camel.routes-discovery.include-patterns**

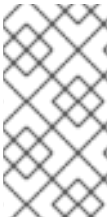
For more details, see the Core documentation.

CHAPTER 2. DEPLOYING QUARKUS APPLICATIONS

You can deploy your Quarkus application on OpenShift by using any of the the following build strategies:

- Docker build
- S2I Binary
- Source S2I

For more details about each of these build strategies, see *Chapter 1. OpenShift build strategies and Quarkus* of the *Deploying your Quarkus applications to OpenShift Container Platform* guide.



NOTE

The OpenShift Docker build strategy is the preferred build strategy that supports Quarkus applications targeted for JVM as well as Quarkus applications compiled to native executables. You can configure the deployment strategy using the **quarkus.openshift.build-strategy** property.

CHAPTER 3. SETTING UP MAVEN LOCALLY

Typical Red Hat build of Apache Camel application development uses Maven to build and manage projects.

The following topics describe how to set up Maven locally:

3.1. PREPARING TO SET UP MAVEN

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

Procedure

1. Download Maven {ceq-maven-version} 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 {ceq-maven-version} or newer, and is using {ceq-jdk-versions}.

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 3.3, “Using local Maven repositories” .

3.2. ADDING RED HAT REPOSITORIES TO MAVEN

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

Prerequisite

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

Procedure

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

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

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

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

</settings>
```

3.3. USING LOCAL MAVEN REPOSITORIES

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

Procedure

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

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

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

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

3.4. 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.

3.4.1. 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.

3.4.2. 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.

3.4.3. 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

3.4.4. 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.

3.5. 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

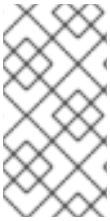
Enables you to distinguish between artifacts that were built from the same POM, but have different content.

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 4. EXAMPLES

The quickstart examples listed in the following table can be cloned or downloaded from the Camel Quarkus Examples Git repository.

Number of Examples: 1

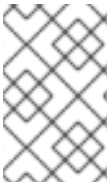
Example	Description
File consumer with Bindy and FTP	Shows how to consume CSV files, marshal & unmarshal the data and send it onwards via FTP

4.1. GETTING STARTED WITH THE FILE CONSUMER QUICKSTART EXAMPLE

You can download or clone the quickstarts from the Camel Quarkus Examples Git repository. The example is in the **file-bindy-ftp** directory.

Extract the contents of the zip file or clone the repository to a local folder, for example a new folder named **quickstarts**.

You can run this example in development mode on your local machine from the command line. Using development mode, you can iterate quickly on integrations in development and get fast feedback on your code. Refer to the Development mode section of the Camel Quarkus User guide for more details.



NOTE

If you need to configure container resource limits or enable the Quarkus Kubernetes client to trust self signed certificates, you can find these configuration options in the **src/main/resources/application.properties** file.

Prerequisites

- You have **cluster admin** access to the OpenShift cluster.
- You have access to an SFTP server and you have set the server properties (which are prefixed by **ftp**) in the application properties configuration file: **src/main/resources/application.properties**.

Procedure

1. Use Maven to build the example application in development mode:

```
$ cd quickstarts/file-bindy-ftp
$ mvn clean compile quarkus:dev
```

The application triggers the timer component every 10 seconds, generates some random "books" data and creates a CSV file in a temporary directory with 100 entries. The following message is displayed in the console:

```
[route1] (Camel (camel-1) thread #3 - timer://generateBooks) Generating randomized books CSV data
```

Next, the CSV file is read by a file consumer and Bindy is used to marshal the individual data rows into Book objects:

```
[route2] (Camel (camel-1) thread #1 - file:///tmp/books) Reading books CSV data from 89A0EE24CB03A69-0000000000000000
```

Next the collection of Book objects is split into individual items and is aggregated based on the genre property:

```
[route3] (Camel (camel-1) thread #0 - AggregateTimeoutChecker) Processed 34 books for genre 'Action'
[route3] (Camel (camel-1) thread #0 - AggregateTimeoutChecker) Processed 31 books for genre 'Crime'
[route3] (Camel (camel-1) thread #0 - AggregateTimeoutChecker) Processed 35 books for genre 'Horror'
```

Finally, the aggregated book collections are unmarshalled back to CSV format and uploaded to the test FTP server.

```
[route4] (Camel (camel-1) thread #2 - seda://processed) Uploaded books-Action-89A0EE24CB03A69-0000000000000069.csv
[route4] (Camel (camel-1) thread #2 - seda://processed) Uploaded books-Crime-89A0EE24CB03A69-0000000000000069.csv
[route4] (Camel (camel-1) thread #2 - seda://processed) Uploaded books-Horror-89A0EE24CB03A69-0000000000000069.csv
```

2. To run the application in JVM mode, enter the following commands:

```
$ mvn clean package -DskipTests
$ java -jar target/*-runner.jar
```

3. You can build and deploy the example application to OpenShift, by entering the following command:

```
$ mvn clean package -DskipTests -Dquarkus.kubernetes.deploy=true
```

4. Check that the pods are running:

```
$oc get pods

NAME                                          READY   STATUS      RESTARTS   AGE
camel-quarkus-examples-file-bindy-ftp-1-d72mb   1/1     Running     0          5m15s
ssh-server-deployment-5f6f685658-jtr9n          1/1     Running     0          5m28s
```

5. Optional: Enter the following command to monitor the application log:

```
oc logs -f camel-quarkus-examples-file-bindy-ftp-5d48f4d85c-sjl8k
```

Additional resources

- Developing Applications with Red Hat build of Apache Camel for Quarkus
- Camel Quarkus User guide