Red Hat build of Apache Camel 4.0

Developing Applications with Red Hat build of Apache Camel for Quarkus

Developing Applications with Red Hat build of Apache Camel for Quarkus

Last Updated: 2023-12-04

Red Hat build of Apache Camel 4.0 Developing Applications with Red Hat build of Apache Camel for Quarkus

Developing Applications with Red Hat build of Apache Camel for Quarkus

Legal Notice

Abstract

This guide is for developers writing Camel applications on top of 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. INTRODUCTION TO DEVELOPING APPLICATIONS WITH RED HAT BUILD OF APACHE CAMEL FOR QUARKUS

This guide is for developers writing Camel applications on top of Red Hat build of Apache Camel for Quarkus.

Camel components which are supported in Red Hat build of Apache Camel for Quarkus have an associated Red Hat build of Apache Camel for Quarkus extension. For more information about the Red Hat build of Apache Camel for Quarkus extensions supported in this distribution, see the Red Hat build of Apache Camel for Quarkus Reference reference guide.

CHAPTER 2. DEPENDENCY MANAGEMENT

A specific Red Hat build of Apache Camel for Quarkus release is supposed to work only with a specific Quarkus release.

2.1. QUARKUS TOOLING FOR STARTING A NEW PROJECT

The easiest and most straightforward way to get the dependency versions right in a new project is to use one of the Quarkus tools:

- code.quarkus.redhat.com - an online project generator,
- Quarkus Maven plugin

These tools allow you to select extensions and scaffold a new Maven project.

TIP

The universe of available extensions spans over Quarkus Core, Camel Quarkus and several other third party participating projects, such as Hazelcast, Cassandra, Kogito and OptaPlanner.

The generated **pom.xml** will look similar to the following:

```
<project>
  ...
  <properties>
    <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>
    ...
  </properties>
  <dependencyManagement>
    <dependencies>
      <!-- The BOMs managing the dependency versions -->
      <dependency>
        <groupId>${quarkus.platform.group-id}</groupId>
        <artifactId>quarkus-bom</artifactId>
        <version>${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>
    </dependencies>
  </dependencyManagement>
```

```
  <dependencies>
    <!-- The extensions you chose in the project generator tool -->
    <dependency>
      <groupId>org.apache.camel.quarkus</groupId>
      <artifactId>camel-quarkus-sql</artifactId>
      <!-- No explicit version required here and below -->
    </dependency>
    ...
  </dependencies>
  ...
</project>
```

NOTE

BOM stands for "Bill of Materials" - it is a **pom.xml** whose main purpose is to manage the versions of artifacts so that end users importing the BOM in their projects do not need to care which particular versions of the artifacts are supposed to work together. In other words, having a BOM imported in the **<depependencyManagement>** section of your **pom.xml** allows you to avoid specifying versions for the dependencies managed by the given BOM.

Which particular BOMs end up in the **pom.xml** file depends on extensions you have selected in the generator tool. The generator tools take care to select a minimal consistent set.

If you choose to add an extension at a later point that is not managed by any of the BOMs in your **pom.xml** file, you do not need to search for the appropriate BOM manually.

With the **quarkus-maven-plugin** you can select the extension, and the tool adds the appropriate BOM as required. You can also use the **quarkus-maven-plugin** to upgrade the BOM versions.

The **com.redhat.quarkus.platform** BOMs are aligned with each other which means that if an artifact is managed in more than one BOM, it is always managed with the same version. This has the advantage that application developers do not need to care for the compatibility of the individual artifacts that may come from various independent projects.

2.2. COMBINING WITH OTHER BOMS

When combining **camel-quarkus-bom** with any other BOM, think carefully in which order you import them, because the order of imports defines the precedence.

I.e. if **my-foo-bom** is imported before **camel-quarkus-bom** then the versions defined in **my-foo-bom** will take the precedence. This might or might not be what you want, depending on whether there are any overlaps between **my-foo-bom** and **camel-quarkus-bom** and depending on whether those versions with higher precedence work with the rest of the artifacts managed in **camel-quarkus-bom**.

CHAPTER 3. DEFINING CAMEL ROUTES

Red Hat build of Apache Camel for Quarkus supports the Java DSL language to define Camel Routes.

3.1. JAVA DSL

Extending **org.apache.camel.builder.RouteBuilder** and using the fluent builder methods available there is the most common way of defining Camel Routes. Here is a simple example of a route using the timer component:

```
import org.apache.camel.builder.RouteBuilder;

public class TimerRoute extends RouteBuilder {

    @Override
    public void configure() throws Exception {
        from("timer:foo?period=1000")
                .log("Hello World");
    }
}
```

3.1.1. Endpoint DSL

Since Camel 3.0, you can use fluent builders also for defining Camel endpoints. The following is equivalent with the previous example:

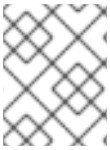
```
import org.apache.camel.builder.RouteBuilder;
import static org.apache.camel.builder.endpoint.StaticEndpointBuilders.timer;

public class TimerRoute extends RouteBuilder {

    @Override
    public void configure() throws Exception {
        from(timer("foo").period(1000))
                .log("Hello World");
    }
}
```

NOTE

Builder methods for all Camel components are available via **camel-quarkus-core**, but you still need to add the given component's extension as a dependency for the route to work properly. In case of the above example, it would be **camel-quarkus-timer**.

CHAPTER 4. CONFIGURATION

Camel Quarkus automatically configures and deploys a Camel Context bean which by default is started/stopped according to the Quarkus Application lifecycle. The configuration step happens at build time during Quarkus' augmentation phase and it is driven by the Camel Quarkus extensions which can be tuned using Camel Quarkus specific **quarkus.camel.*** properties.



NOTE

quarkus.camel.* configuration properties are documented on the individual extension pages - for example see Camel Quarkus Core.

After the configuration is done, a minimal Camel Runtime is assembled and started in the RUNTIME_INIT phase.

4.1. CONFIGURING CAMEL COMPONENTS

4.1.1. application.properties

To configure components and other aspects of Apache Camel through properties, make sure that your application depends on **camel-quarkus-core** directly or transitively. Because most Camel Quarkus extensions depend on **camel-quarkus-core**, you typically do not need to add it explicitly.

camel-quarkus-core brings functionalities from Camel Main to Camel Quarkus.

In the example below, you set a specific **ExchangeFormatter** configuration on the **LogComponent** via **application.properties**:

```
camel.component.log.exchange-formatter =
#class:org.apache.camel.support.processor.DefaultExchangeFormatter
camel.component.log.exchange-formatter.show-exchange-pattern = false
camel.component.log.exchange-formatter.show-body-type = false
```

4.1.2. CDI

You can also configure a component programmatically using CDI.

The recommended method is to observe the **ComponentAddEvent** and configure the component before the routes and the **CamelContext** are started:

```
import jakarta.enterprise.context.ApplicationScoped;
import jakarta.enterprise.event.Observes;
import org.apache.camel.quarkus.core.events.ComponentAddEvent;
import org.apache.camel.component.log.LogComponent;
import org.apache.camel.support.processor.DefaultExchangeFormatter;

@ApplicationScoped
public static class EventHandler {
    public void onComponentAdd(@Observes ComponentAddEvent event) {
        if (event.getComponent() instanceof LogComponent) {
            /* Perform some custom configuration of the component */
            LogComponent logComponent = ((LogComponent) event.getComponent());
            DefaultExchangeFormatter formatter = new DefaultExchangeFormatter();
```

```
            formatter.setShowExchangePattern(false);
            formatter.setShowBodyType(false);
            logComponent.setExchangeFormatter(formatter);
        }
    }
}
```

4.1.2.1. Producing a @Named component instance

Alternatively, you can create and configure the component yourself in a **@Named** producer method. This works as Camel uses the component URI scheme to look-up components from its registry. For example, in the case of a **LogComponent** Camel looks for a **log** named bean.

WARNING

While producing a **@Named** component bean will usually work, it may cause subtle issues with some components.

Camel Quarkus extensions may do one or more of the following:

- Pass custom subtype of the default Camel component type. See the Vert.x WebSocket extension example.
- Perform some Quarkus specific customization of the component. See the JPA extension example.

These actions are not performed when you produce your own component instance, therefore, configuring components in an observer method is the recommended method.

```
import jakarta.enterprise.context.ApplicationScoped;
import jakarta.inject.Named;

import org.apache.camel.component.log.LogComponent;
import org.apache.camel.support.processor.DefaultExchangeFormatter;

@ApplicationScoped
public class Configurations {
    /**
     * Produces a {@link LogComponent} instance with a custom exchange formatter set-up.
     */
    @Named("log") 1
    LogComponent log() {
        DefaultExchangeFormatter formatter = new DefaultExchangeFormatter();
        formatter.setShowExchangePattern(false);
        formatter.setShowBodyType(false);

        LogComponent component = new LogComponent();
        component.setExchangeFormatter(formatter);
```

```
        return component;
    }
}
```

1 The **"log"** argument of the **@Named** annotation can be omitted if the name of the method is the same.

4.2. CONFIGURATION BY CONVENTION

In addition to support configuring Camel through properties, **camel-quarkus-core** allows you to use conventions to configure the Camel behavior. For example, if there is a single **ExchangeFormatter** instance in the CDI container, then it will automatically wire that bean to the **LogComponent**.

Additional resources

- Configuring and using Metering in OpenShift Container Platform

CHAPTER 5. CONTEXTS AND DEPENDENCY INJECTION (CDI) IN CAMEL QUARKUS

CDI plays a central role in Quarkus and Camel Quarkus offers a first class support for it too.

You may use **@Inject**, **@ConfigProperty** and similar annotations e.g. to inject beans and configuration values to your Camel **RouteBuilder**, for example:

```
import jakarta.enterprise.context.ApplicationScoped;
import jakarta.inject.Inject;
import org.apache.camel.builder.RouteBuilder;
import org.eclipse.microprofile.config.inject.ConfigProperty;

@ApplicationScoped (1)
public class TimerRoute extends RouteBuilder {

    @ConfigProperty(name = "timer.period", defaultValue = "1000") (2)
    String period;

    @Inject
    Counter counter;

    @Override
    public void configure() throws Exception {
        fromF("timer:foo?period=%s", period)
                .setBody(exchange -> "Incremented the counter: " + counter.increment())
                .to("log:cdi-example?showExchangePattern=false&showBodyType=false");
    }
}
```

(1) The **@ApplicationScoped** annotation is required for **@Inject** and **@ConfigProperty** to work in a **RouteBuilder**. Note that the **@ApplicationScoped** beans are managed by the CDI container and their life cycle is thus a bit more complex than the one of the plain **RouteBuilder**. In other words, using **@ApplicationScoped** in **RouteBuilder** comes with some boot time penalty and you should therefore only annotate your **RouteBuilder** with **@ApplicationScoped** when you really need it.

(2) The value for the **timer.period** property is defined in **src/main/resources/application.properties** of the example project.

TIP

Refer to the Quarkus Dependency Injection guide for more details.

5.1. ACCESSING CAMELCONTEXT

To access **CamelContext** just inject it into your bean:

```
import jakarta.inject.Inject;
import jakarta.enterprise.context.ApplicationScoped;
import java.util.stream.Collectors;
import java.util.List;
import org.apache.camel.CamelContext;
```

```
@ApplicationScoped
public class MyBean {

    @Inject
    CamelContext context;

    public List<String> listRouteIds() {
        return context.getRoutes().stream().map(Route::getId).sorted().collect(Collectors.toList());
    }
}
```

5.2. @ENDPOINTINJECT AND @PRODUCE

If you are used to **@org.apache.camel.EndpointInject** and **@org.apache.camel.Produce** from plain Camel or from Camel on SpringBoot, you can continue using them on Quarkus too.

The following use cases are supported by **org.apache.camel.quarkus:camel-quarkus-core**:

```
import jakarta.enterprise.context.ApplicationScoped;
import org.apache.camel.EndpointInject;
import org.apache.camel.FluentProducerTemplate;
import org.apache.camel.Produce;
import org.apache.camel.ProducerTemplate;

@ApplicationScoped
class MyBean {

    @EndpointInject("direct:myDirect1")
    ProducerTemplate producerTemplate;

    @EndpointInject("direct:myDirect2")
    FluentProducerTemplate fluentProducerTemplate;

    @EndpointInject("direct:myDirect3")
    DirectEndpoint directEndpoint;

    @Produce("direct:myDirect4")
    ProducerTemplate produceProducer;

    @Produce("direct:myDirect5")
    FluentProducerTemplate produceProducerFluent;

}
```

You can use any other Camel producer endpoint URI instead of **direct:myDirect***.

WARNING

@EndpointInject and **@Produce** are not supported on setter methods - see #2579

The following use case is supported by **org.apache.camel.quarkus:camel-quarkus-bean**:

```
import jakarta.enterprise.context.ApplicationScoped;
import org.apache.camel.Produce;

@ApplicationScoped
class MyProduceBean {

    public interface ProduceInterface {
        String sayHello(String name);
    }

    @Produce("direct:myDirect6")
    ProduceInterface produceInterface;

    void doSomething() {
        produceInterface.sayHello("Kermit")
    }

}
```

5.3. CDI AND THE CAMEL BEAN COMPONENT

5.3.1. Refer to a bean by name

To refer to a bean in a route definition by name, just annotate the bean with **@Named("myNamedBean")** and **@ApplicationScoped** (or some other supported scope). The **@RegisterForReflection** annotation is important for the native mode.

```
import jakarta.enterprise.context.ApplicationScoped;
import jakarta.inject.Named;
import io.quarkus.runtime.annotations.RegisterForReflection;

@ApplicationScoped
@Named("myNamedBean")
@RegisterForReflection
public class NamedBean {
    public String hello(String name) {
        return "Hello " + name + " from the NamedBean";
    }
}
```

Then you can use the **myNamedBean** name in a route definition:

```
import org.apache.camel.builder.RouteBuilder;
public class CamelRoute extends RouteBuilder {
    @Override
    public void configure() {
        from("direct:named")
                .bean("myNamedBean", "hello");
        /* ... which is an equivalent of the following: */
        from("direct:named")
                .to("bean:myNamedBean?method=hello");
    }
}
```

As an alternative to **@Named**, you may also use **io.smallrye.common.annotation.Identifier** to name and identify a bean.

```
import jakarta.enterprise.context.ApplicationScoped;
import io.quarkus.runtime.annotations.RegisterForReflection;
import io.smallrye.common.annotation.Identifier;

@ApplicationScoped
@Identifier("myBeanIdentifier")
@RegisterForReflection
public class MyBean {
    public String hello(String name) {
        return "Hello " + name + " from MyBean";
    }
}
```

Then refer to the identifier value within the Camel route:

```
import org.apache.camel.builder.RouteBuilder;
public class CamelRoute extends RouteBuilder {
    @Override
    public void configure() {
        from("direct:start")
                .bean("myBeanIdentifier", "Camel");
    }
}
```

NOTE

We aim at supporting all use cases listed in Bean binding section of Camel documentation. Do not hesitate to file an issue if some bean binding scenario does not work for you.

5.3.2. @Consume

Since Camel Quarkus 2.0.0, the **camel-quarkus-bean** artifact brings support for **@org.apache.camel.Consume** - see the Pojo consuming section of Camel documentation.

Declaring a class like the following

```
import org.apache.camel.Consume;
public class Foo {
```

```
  @Consume("activemq:cheese")
  public void onCheese(String name) {
    ...
  }
}
```

will automatically create the following Camel route

```
from("activemq:cheese").bean("foo1234", "onCheese")
```

for you. Note that Camel Quarkus will implicitly add **@jakarta.inject.Singleton** and **jakarta.inject.Named("foo1234")** to the bean class, where **1234** is a hash code obtained from the fully qualified class name. If your bean has some CDI scope (such as **@ApplicationScoped**) or **@Named("someName")** set already, those will be honored in the auto-created route.

CHAPTER 6. OBSERVABILITY

6.1. HEALTH & LIVENESS CHECKS

Health & liveness checks are supported via the MicroProfile Health extension. They can be configured via the Camel Health API or via Quarkus MicroProfile Health.

All configured checks are available on the standard MicroProfile Health endpoint URLs:

- http://localhost:8080/q/health
- http://localhost:8080/q/health/live
- http://localhost:8080/q/health/ready

6.2. METRICS

We provide MicroProfile Metrics for exposing metrics.

Some basic Camel metrics are provided for you out of the box, and these can be supplemented by configuring additional metrics in your routes.

Metrics are available on the standard Quarkus metrics endpoint:

- http://localhost:8080/q/metrics

CHAPTER 7. NATIVE MODE

For additional information about compiling and testing application in native mode, see Producing a native executable in the *Compiling your Quarkus applications to native executables* guide.

7.1. CHARACTER ENCODINGS

By default, not all **Charsets** are available in native mode.

```
Charset.defaultCharset(), US-ASCII, ISO-8859-1, UTF-8, UTF-16BE, UTF-16LE, UTF-16
```

If you expect your application to need any encoding not included in this set or if you see an **UnsupportedCharsetException** thrown in the native mode, please add the following entry to your **application.properties**:

```
quarkus.native.add-all-charsets = true
```

See also quarkus.native.add-all-charsets in Quarkus documentation.

7.2. LOCALE

By default, only the building JVM default locale is included in the native image. Quarkus provides a way to set the locale via **application.properties**, so that you do not need to rely on **LANG** and **LC_*** environement variables:

```
quarkus.native.user-country=US
quarkus.native.user-language=en
```

There is also support for embedding multiple locales into the native image and for selecting the default locale via Mandrel command line options **-H:IncludeLocales=fr,en**, **H:+IncludeAllLocales** and **-H:DefaultLocale=de**. You can set those via the Quarkus **quarkus.native.additional-build-args** property.

7.3. EMBEDDING RESOURCES IN THE NATIVE EXECUTABLE

Resources accessed via **Class.getResource()**, **Class.getResourceAsStream()**, **ClassLoader.getResource()**, **ClassLoader.getResourceAsStream()**, etc. at runtime need to be explicitly listed for including in the native executable.

This can be done using Quarkus **quarkus.native.resources.includes** and **quarkus.native.resources.excludes** properties in **application.properties** file as demonstrated below:

```
quarkus.native.resources.includes = docs/*,images/*
quarkus.native.resources.excludes = docs/ignored.adoc,images/ignored.png
```

In the example above, resources named **docs/included.adoc** and **images/included.png** would be embedded in the native executable while **docs/ignored.adoc** and **images/ignored.png** would not.

resources.includes and **resources.excludes** are both lists of comma separated Ant-path style glob patterns.

Refer to Red Hat build of Apache Camel for Quarkus Reference Reference for more details.

7.4. USING THE ONEXCEPTION CLAUSE IN NATIVE MODE

When using Camel **onException** handling in native mode, it is your responsibility to register the exception classes for reflection.

For instance, having a camel context with **onException** handling:

```
onException(MyException.class).handled(true);
from("direct:route-that-could-produce-my-exception").throw(MyException.class);
```

The class **mypackage.MyException** should be registered for reflection. For more information, see Registering classes for reflection.

7.5. REGISTERING CLASSES FOR REFLECTION

By default, dynamic reflection is not available in native mode. Classes for which reflective access is needed, have to be registered for reflection at compile time.

In many cases, application developers do not need to care because Quarkus extensions are able to detect the classes that require the reflection and register them automatically.

However, in some situations, Quarkus extensions may miss some classes and it is up to the application developer to register them. There are two ways to do that:

1. The **@io.quarkus.runtime.annotations.RegisterForReflection** annotation can be used to register classes on which it is used, or it can also register third party classes via its **targets** attribute.

```
import io.quarkus.runtime.annotations.RegisterForReflection;

@RegisterForReflection
class MyClassAccessedReflectively {
}

@RegisterForReflection(
    targets = {
        org.third-party.Class1.class,
        org.third-party.Class2.class
    }
)
class ReflectionRegistrations {
}
```

2. The **quarkus.camel.native.reflection** options in **application.properties**:

```
quarkus.camel.native.reflection.include-patterns = org.apache.commons.lang3.tuple.*
quarkus.camel.native.reflection.exclude-patterns = org.apache.commons.lang3.tuple.*Triple
```

For these options to work properly, the artifacts containing the selected classes must either contain a Jandex index ('META-INF/jandex.idx') or they must be registered for indexing using the 'quarkus.index-dependency.*' options in 'application.properties' - for example:

```
quarkus.index-dependency.commons-lang3.group-id = org.apache.commons
quarkus.index-dependency.commons-lang3.artifact-id = commons-lang3
```

7.6. REGISTERING CLASSES FOR SERIALIZATION

If serialization support is requested via **quarkus.camel.native.reflection.serialization-enabled**, the classes listed in CamelSerializationProcessor.BASE_SERIALIZATION_CLASSES are automatically registered for serialization.

You can register more classes using **@RegisterForReflection(serialization = true)**.