Red Hat Integration 2021.Q3

Developing Applications with Camel Quarkus

TECHNOLOGY PREVIEW - Developing applications with Camel Quarkus

Last Updated: 2021-08-16

Red Hat Integration 2021.Q3 Developing Applications with Camel Quarkus

TECHNOLOGY PREVIEW - Developing applications with Camel Quarkus

Legal Notice

Abstract

This guide is for developers writing Camel applications on top of Camel 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 CAMEL QUARKUS

This guide is for developers writing Camel applications on top of Quarkus.

Camel components which are supported on Camel Quarkus have an associated Camel Quarkus extension. For more information about the Camel Quarkus extensions supported in this distribution, see the Camel Extensions for Quarkus reference guide.

IMPORTANT

Camel Quarkus is a Technology Preview feature only. Technology Preview features are not supported with Red Hat production service level agreements (SLAs) and might not be functionally complete. Red Hat does not recommend using them in production.

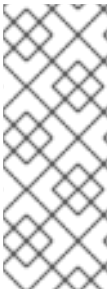
These features provide early access to upcoming product features, enabling customers to test functionality and provide feedback during the development process. For more information about the support scope of Red Hat Technology Preview features, see https://access.redhat.com/support/offerings/techpreview.

CHAPTER 2. DEPENDENCY MANAGEMENT

A specific Camel Quarkus release is supposed to work only with a specific Quarkus release.

2.1. QUARKUS BOM

The core Quarkus dependencies are managed via **com.redhat.quarkus.quarkus:quarkus-bom**. This BOM does not manage the additional Quarkus extensions (for example Camel Quarkus or Kubernetes), so there are other BOMs available for you to use in your projects that contain import **quarkus-bom**.



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.

2.2. CAMEL QUARKUS BOM

As long as you do not plan to use any dependencies beyond those from Quarkus and Camel Quarkus, you should use **org.apache.camel.quarkus:camel-quarkus-bom** which manages all supported Camel artifacts and imports **com.redhat.quarkus.quarkus:quarkus-bom**.

WARNING

For this technology preview, do not use **com.redhat.quarkus:quarkus-universe-bom** as it does not contain the supported Camel extensions included in this release.

2.3. 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.

For example, if **my-foo-bom** is imported before **camel-quarkus-bom** then the artifact versions defined in **my-foo-bom** will take the precedence. This may or may not be what you intend, depending on whether or not there are any overlapping actifacts in your **my-foo-bom** and **camel-quarkus-bom**. Additionally, you may also need to consider if those artifact versions in **my-foo-bom** with a higher precedence are compatible with the rest of the artifacts managed in **camel-quarkus-bom**.

CHAPTER 3. DEFINING CAMEL ROUTES

Camel 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");
    }
}
```

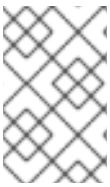
3.1.1. Endpoint DSL

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 - see e.g. 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 javax.enterprise.context.ApplicationScoped;
import javax.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

Please note that 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 javax.enterprise.context.ApplicationScoped;
import javax.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**.

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 javax.enterprise.context.ApplicationScoped;
import javax.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

Please refer to the Quarkus Dependency Injection guide for more details.

5.1. ACCESSING CAMELCONTEXT

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

```
import javax.inject.Inject;
import javax.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. CDI AND THE CAMEL BEAN COMPONENT

5.2.1. 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**. The **@RegisterForReflection** annotation is important for the native mode.

```
import javax.enterprise.context.ApplicationScoped;
import javax.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")
                .to("bean:namedBean?method=hello");
    }
}
```