Red Hat build of Apache Camel for Spring Boot 3.20

Camel Spring Boot User Guide

Camel Spring Boot User Guide

Last Updated: 2024-03-18

Red Hat build of Apache Camel for Spring Boot 3.20 Camel Spring Boot User Guide

Camel Spring Boot User Guide

Legal Notice

Abstract

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

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. USING LANGUAGE SUPPORT FOR APACHE CAMEL EXTENSION

> **IMPORTANT**
>
> The VS Code extensions for Apache Camel are listed as development support. For more information about scope of development support, see Development Support Scope of Coverage

The Visual Studio Code language support extension adds the language support for Apache Camel for XML DSL and Java DSL code.

1.1. ABOUT LANGUAGE SUPPORT FOR APACHE CAMEL EXTENSION

This extension provides completion, validation and documentation features for Apache Camel URI elements directly in your Visual Studio Code editor. It works as a client using the Microsoft Language Server Protocol which communicates with Camel Language Server to provide all functionalities.

1.2. FEATURES OF LANGUAGE SUPPORT FOR APACHE CAMEL EXTENSION

The important features of the language support extension are listed below:

- Language service support for Apache Camel URIs.

- Quick reference documentation when you hover the cursor over a Camel component.

- Diagnostics for Camel URIs.

- Navigation for Java and XML langauges.

- Creating a Camel Route specified with Yaml DSL using Camel JBang.

1.3. REQUIREMENTS

Following points must be considered when using the Apache Camel Language Server:

- Java 11 is currently required to launch the Apache Camel Language Server. The **java.home** VS Code option is used to use a different version of JDK than the default one installed on the machine.

- For some features, JBang must be available on a system command line.

- For an XML DSL files:

  - Use an **.xml** file extension.

  - Specify the Camel namespace, for reference, see http://camel.apache.org/schema/blueprint or http://camel.apache.org/schema/spring.

- For a Java DSL files:

  - Use a **.java** file extension.

- Specify the Camel package(usually from an imported package), for example, **import org.apache.camel.builder.RouteBuilder**.

- To reference the Camel component, use from or to and a string without a space. The string cannot be a variable. For example, **from("timer:timerName")** works, but **from("timer:timerName")** and **from(aVariable)** do not work.

1.4. INSTALLING LANGUAGE SUPPORT FOR APACHE CAMEL EXTENSION

You can download the Language support for Apache Camel extension from the VS Code Extension Marketplace and the Open VSX Registry. You can also install the Language Support for Apache Camel extension directly in the Microsoft VS Code.

Procedure

1. Open the VS Code editor.

2. In the **VS Code** editor, select **View > Extensions**.

3. In the search bar, type **Camel**. Select the **Language Support for Apache Camel** option from the search results and then click Install.

This installs the language support extension in your editor.

Additional resources

- [Language Support for Apache Camel by Red Hat](#)

CHAPTER 2. USING VS CODE DEBUG ADAPTER FOR APACHE CAMEL EXTENSION
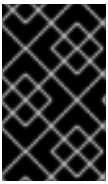
> **IMPORTANT**
>
> The VS Code extensions for Apache Camel are listed as development support. For more information about scope of development support, see Development Support Scope of Coverage

This is the Visual Studio Code extension that adds Camel Debugger power by attaching to a running Camel route written in Java, Yaml or XML DSL.

2.1. FEATURES OF DEBUG ADAPTER

The VS Code Debug Adapter for Apache Camel extension supports the following features:

- Camel Main mode for XML only.

- The use of Camel debugger by attaching it to a running Camel route written in Java, Yaml or XML using the JMX url.

- The local use of Camel debugger by attaching it to a running Camel route written in Java, Yaml or XML using the PID.

- You can use it for a single Camel context.

- Add or remove the breakpoints.

- The conditional breakpoints with simple language.

- Inspecting the variable values on suspended breakpoints.

- Resume a single route instance and resume all route instances.

- Stepping when the route definition is in the same file.

- Allow to update variables in scope Debugger, in the message body, in a message header of type String, and an exchange property of type String

- Supports the command **Run Camel Application with JBang and Debug**.

  - This command allows a one-click start and Camel debug in simple cases. This command is available through:

    - Command Palette. It requires a valid Camel file opened in the current editor.

    - Contextual menu in File explorer. It is visible to all ***.xml**, ***.java**, ***.yaml** and ***.yml**.

    - Codelens at the top of a Camel file (the heuristic for the codelens is checking that there is a from and a to or a log on **java**, **xml**, and **yaml** files).

- Supports the command **Run Camel application with JBang**.

  - It requires a valid Camel file defined in Yaml DSL (.yaml|.yml) opened in editor.

- Configuration snippets for Camel debugger launch configuration

- Configuration snippets to launch a Camel application ready to accept a Camel debugger connection using JBang, or a Maven with Camel maven plugin

2.2. REQUIREMENTS

Following points must be considered when using the VS Code Debug Adapter for Apache Camel extension:

- Java Runtime Environment 11 or later with **com.sun.tools.attach.VirtualMachine** (available in most JVMs such as Hotspot and OpenJDK) must be installed.

- The Camel instance to debug must follow these requirements:

  - Camel 3.16 or later

  - Have camel-debug on the classpath.

  - Have JMX enabled.
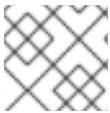
> **NOTE**
>
> For some features, The JBang must be available on a system commandline.

2.3. INSTALLING VS CODE DEBUG ADAPTER FOR APACHE CAMEL

You can download the VS Code Debug Adapter for Apache Camel extension from the VS Code Extension Marketplace and the Open VSX Registry. You can also install the Debug Adapter for Apache Camel extension directly in the Microsoft VS Code.

Procedure

1. Open the VS Code editor.

2. In the **VS Code** editor, select **View > Extensions**.

3. In the search bar, type **Camel Debug**. Select the **Debug Adapter for Apache Camel** option from the search results and then click Install.

This installs the Debug Adapter for Apache Camel in the VS Code editor.

2.4. USING DEBUG ADAPTER

Following procedure explains how to debug a camel application using the debug adapter.

Procedure

1. Ensure that the **jbang** binary is available on the system commandline.

2. Open a Camel route which can be started with Camel JBang.

3. Call the **command Palette** using the keys **Ctrl + Alt + P**, and select the **Run Camel Application with JBang and Debug** command or click on the codelens **Camel Debug with JBang** that appears on top of the file.

4. Wait until the route is started and debugger is connected.

5. Put a breakpoint on the Camel route.

6. Debug.

Additional resources

- Debug Adapter for Apache Camel by Red Hat

CHAPTER 3. USING CAMEL JBANG

Camel Jbang is a JBang-based Camel application for running Camel routes.

3.1. INSTALLING CAMEL JBANG

Prerequisites

1. JBang must be installed on your machine. See instructions on how to download and install the JBang.

After the JBang is installed, you can verify JBang is working by executing the following command from a command shell:

```
jbang version
```

This outputs the version of installed JBang.

Procedure

1. Run the following command to install the Camel JBang application:

```
jbang app install camel@apache/camel
```

This installs the Apache Camel as the **camel** command within JBang. This means that you can run Camel from the command line by just executing **camel** command.

3.2. USING CAMEL JBANG

The Camel JBang supports multiple commands. The **camel help** command can display all the available commands.

```
camel --help
```

> **NOTE**
>
> The first time you run this command, it may cause dependencies to be cached, therefore taking a few extra seconds to run. If you are already using JBang and you get errors such as **Exception in thread "main" java.lang.NoClassDefFoundError: "org/apache/camel/dsl/jbang/core/commands/CamelJBangMain"**, try clearing the JBang cache and re-install again.

All the commands support the **--help** and will display the appropriate help if that flag is provided.

3.2.1. Enable shell completion

Camel JBang provides shell completion for bash and zsh out of the box. To enable shell completion for Camel JBang, run:

```
source <(camel completion)
```

To make it permanent, run:

```
echo 'source <(camel completion)' >> ~/.bashrc
```

3.3. CREATING AND RUNNING CAMEL ROUTES

You can create a new basic routes with the **init** command. For example to create an XML route, run the following command:

```
camel init cheese.xml
```

This creates the file **cheese.xml** (in the current directory) with a sample route.

To run the file, run:

```
camel run cheese.xml
```
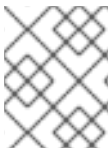
> **NOTE**
>
> You can create and run any of the supported DSLs in Camel such as YAML, XML, Java, Groovy.

To create a new **.java** route, run:

```
camel init foo.java
```

When you use the init command, Camel by default creates the file in the current directory. However, you can use the **--directory** option to create the file in the specified directory. For example to create in a folder named **foobar**, run:

```
camel init foo.java --directory=foobar
```

> **NOTE**
>
> When you use the **--directory** option, Camel automatically cleans this directory if already exists.

3.3.1. Running routes from multiple files

You can run routes from more than one file, for example to run two YAML files:

```
camel run one.yaml two.yaml
```

You can run routes from two different files such as yaml and Java:

```
camel run one.yaml hello.java
```

You can use wildcards (i.e. *****) to match multiple files, such as running all the yaml files:

```
camel run *.yaml
```

You can run all files starting with foo*:

```
camel run foo*
```

To run all the files in the directory, use:

```
camel run *
```

> **NOTE**
>
> The **run** goal can also detect files that are **properties**, such as **application.properties**.

3.3.2. Running routes from input parameter

For very small Java routes, it is possible to provide the route as CLI argument, as shown below:

```
camel run --code='from("kamelet:beer-source").to("log:beer")'
```

This is very limited as the CLI argument is a bit cumbersome to use than files. When you run the routes from input parameter, remember that:

- Only Java DSL code is supported.

- Code is wrapped in single quote, so you can use double quote in Java DSL.

- Code is limited to what literal values possible to provide from the terminal and JBang.

- All route(s) must be defined in a single **--code** parameter.

> **NOTE**
>
> Using **--code** is only usable for very quick and small prototypes.

3.3.3. Dev mode with live reload

You can enable the dev mode that comes with live reload of the route(s) when the source file is updated (saved), using the **--dev** options as shown:

```
camel run foo.yaml --dev
```

Then while the Camel integration is running, you can update the YAML route and update when saving. This option works for all DLS including **java**, for example:

```
camel run hello.java --dev
```
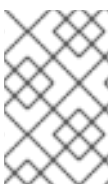
> **NOTE**
>
> The live reload option is meant for development purposes only, and if you encounter problems with reloading such as JVM class loading issues, then you may need to restart the integration.

3.3.4. Developer Console

You can enable the developer console, which presents a variety of information to the developer. To enable the developer console, run:

```
camel run hello.java --console
```

The console is then accessible from a web browser at http://localhost:8080/q/dev (by default). The link is also displayed in the log when the Camel is starting up.

The console can give you insights into your running Camel integration, such as reporting the top routes that takes the longest time to process messages. You can then identify the slowest individual EIPs in these routes.

The developer console can also output the data in **JSON** format, that can be used by 3rd-party tooling to capture the information. For example, to output the top routes via curl, run:

```
curl -s -H "Accept: application/json" http://0.0.0.0:8080/q/dev/top/
```

If you have **jq** installed, that can format and output the JSON data in colour, run:

```
curl -s -H "Accept: application/json" http://0.0.0.0:8080/q/dev/top/ | jq
```

3.3.5. Using profiles

A **profile** in Camel JBang is a name (id) that refers to the configuration that is loaded automatically with Camel JBang. The default profile is named as the **application** which is a (smart default) to let Camel JBang automatic load **application.properties** (if present). This means that you can create profiles that match to a specific properties file with the same name.

For example, running with a profile named **local** means that Camel JBang will load **local.properties** instead of **application.properties**. To use a profile, specify the command line option **--profile** as shown:

```
camel run hello.java --profile=local
```

You can only specify one profile name at a time, for example, **--profile=local,two** is not valid.

In the **properties** files you can configure all the configurations from Camel Main. To turn off and enable log masking run the following command:

```
camel.main.streamCaching=false
camel.main.logMask=true
```

You can also configure Camel components such as **camel-kafka** to declare the URL to the brokers:

```
camel.component.kafka.brokers=broker1:9092,broker2:9092,broker3:9092
```

> **NOTE**
>
> Keys starting with **camel.jbang** are reserved keys that are used by Camel JBang internally, and allow for pre-configuring arguments for Camel JBang commands.

3.3.6. Downloading JARs over the internet

By default, Camel JBang automatically resolves the dependencies needed to run Camel, this is done by JBang and Camel respectively. Camel itself detects at runtime if a component has a need for the JARs that are not currently available on the classpath, and can then automatically download the JARs.

Camel downloads these JARs in the following order:

1. from the local disk in ~/**.m2/repository**

2. from the internet in Maven Central

3. from internet in the custom 3rd-party Maven repositories

4. from all the repositories found in active profiles of ~/**.m2/settings.xml** or a settings file specified using **--maven-settings** option.

If you do not want the Camel JBang to download over the internet, you can turn this off with the **--download** option, as shown:

```
camel run foo.java --download=false
```

3.3.7. Adding custom JARs

Camel JBang automatically detects the dependencies for the Camel components, languages, and data formats from its own release. This means that it is not necessary to specify which JARs to use. However, if you need to add 3rd-party custom JARs then you can specify these with the **--deps** as CLI argument in Maven GAV syntax (**groupId:artifactId:version**), such as:

```
camel run foo.java --deps=com.foo:acme:1.0
```

```
To add a Camel dependency explicitly you can use a shorthand syntax (starting with `camel:` or `camel-`):
```

```
camel run foo.java --deps=camel-saxon
```

You can specify multiple dependencies separated by comma:

```
camel run foo.java --deps=camel-saxon,com.foo:acme:1.0
```

3.3.8. Using 3rd-party Maven repositories

Camel JBang downloads from the local repository first, and then from the online Maven Central repository. To download from the 3rd-party Maven repositories, you must specify this as CLI argument, or in the **application.properties** file.

```
camel run foo.java --repos=https://packages.atlassian.com/maven-external
```

> **NOTE**
>
> You can specify multiple repositories separated by comma.

The configuration for the 3rd-party Maven repositories is configured in the **application.properties** file with the key **camel.jbang.repos** as shown:

```
camel.jbang.repos=https://packages.atlassian.com/maven-external
```

When you run Camel route, the **application.properties** is automatically loaded:

```
camel run foo.java
```

You can also explicitly specify the properties file to use:

```
camel run foo.java application.properties
```

Or you can specify this as a profile:

```
camel run foo.java --profile=application
```

Where the profile id is the name of the properties file.

3.3.9. Configuration of Maven usage

By default, the existing ~/**.m2**/**settings.xml** file is loaded, so it is possible to alter the behavior of the Maven resolution process. Maven settings file provides the information about the Maven mirrors, credential configuration (potentially encrypted) or active profiles and additional repositories.

Maven repositories can use authentication and the Maven-way to configure credentials is through **<server>** elements:

```xml
<server>
    <id>external-repository</id>
    <username>camel</username>
    <password>{SSVqy/PexxQHvubrWhdguYuG7HnTvHlaNr6g3dJn7nk=}</password>
</server>
```

While the password may be specified using plain text, it si recommended to configure the maven master password first and then use it to configure repository password:

```
$ mvn -emp
Master password: camel
{hqXUuec2RowH8dA8vdqkF6jn4NU9ybOsDjuTmWvYj4U=}
```

The above password must be added to ~/**.m2**/**settings-security.xml** file as shown:

```xml
<settingsSecurity>
  <master>{hqXUuec2RowH8dA8vdqkF6jn4NU9ybOsDjuTmWvYj4U=}</master>
</settingsSecurity>
```

Then you can configure a normal password:

```
$ mvn -ep
Password: camel
{SSVqy/PexxQHvubrWhdguYuG7HnTvHlaNr6g3dJn7nk=}
```

Then you can use this password in the **<server>/<password>** configuration.

By default, Maven reads the master password from ~/**.m2**/**settings-security.xml** file, but you can override it. Location of the **settings.xml** file itself can be specified as shown:

```
camel run foo.java --maven-settings=/path/to/settings.xml --maven-settings-
security=/path/to/settings-security.xml
```

If you want to run Camel application without assuming any location (even ~/**.m2**/**settings.xml**), use this option:

```
camel run foo.java --maven-settings=false
```

3.3.10. Running routes hosted on GitHub

You can run a route that is hosted on the GitHub using the Camels resource loader. For example, to run one of the Camel K examples, use:

```
camel run github:apache:camel-kamelets-examples:jbang/hello-java/Hey.java
```

You can also use the **https** URL for the GitHub. For example, you can browse the examples from a web-browser and then copy the URL from the browser window and run the example with Camel JBang:

```
camel run https://github.com/apache/camel-kamelets-examples/tree/main/jbang/hello-java
```

You can also use wildcards (i.e. *) to match multiple files, such as running all the groovy files:

```
camel run https://github.com/apache/camel-kamelets-examples/tree/main/jbang/languages/*.groovy
```

Or you can run all files starting with rou*:

```
camel run https://github.com/apache/camel-kamelets-examples/tree/main/jbang/languages/rou*
```

3.3.10.1. Running routes from the GitHub gists

Using the gists from the GitHub is a quick way to share the small Camel routes that you can easily run. For example to run a gist, use:

```
camel run https://gist.github.com/davsclaus/477ddff5cdeb1ae03619aa544ce47e92
```

A gist can contain one or more files, and Camel JBang will gather all relevant files, so a gist can contain multiple routes, properties files, and Java beans.

3.3.11. Downloading routes hosted on the GitHub

You can use Camel JBang to download the existing examples from GitHub to local disk, which allows to modify the example and to run locally. For example, you can download the **dependency injection** example by running the following command:

```
camel init https://github.com/apache/camel-kamelets-examples/tree/main/jbang/dependency-injection
```

Then the files (not sub folders) are downloaded to the current directory. You can then run the example locally with:

> camel run *

You can also download to the files to a new folder using the **--directory** option, for example to download the files to a folder named **myproject**, run:

> camel init https://github.com/apache/camel-kamelets-examples/tree/main/jbang/dependency-injection --directory=myproject

> **NOTE**
>
> When using **--directory** option, Camel will automatically clean this directory if already exists.

You can run the example in dev mode, to hot-deploy on the source code changes.

> camel run * --dev

You can download a single file, for example, to download one of the Camel K examples, run:

> camel init https://github.com/apache/camel-k-examples/blob/main/generic-examples/languages/simple.groovy

This is a groovy route, which you can run with (or use *****):

> camel run simple.groovy

3.3.11.1. Downloading routes form GitHub gists

You can download the files from the gists as shown:

> camel init https://gist.github.com/davsclaus/477ddff5cdeb1ae03619aa544ce47e92

This downloads the files to local disk, which you can run afterwards:

> camel run *

You can download to a new folder using the **--directory** option, for example, to download to a folder named **foobar**, run:

> camel init https://gist.github.com/davsclaus/477ddff5cdeb1ae03619aa544ce47e92 --directory=foobar

> **NOTE**
>
> When using **--directory** option, Camel automatically cleans this directory if already exists.

3.3.12. Using a specific Camel version

You can specify which Camel version to run as shown:

```
jbang run -Dcamel.jbang.version=3.20.1 camel@apache/camel [command]
```

NOTE

Older versions of Camel may not work as well with Camel JBang as the newest versions. It is recommended to use the versions starting from Camel 3.18 onwards.

You can also try bleeding edge development by using SNAPSHOT such as:

```
jbang run --fresh -Dcamel.jbang.version=3.20.1-SNAPSHOT camel@apache/camel [command]
```

3.3.13. Running the Camel K integrations or bindings

Camel supports running the Camel K integrations and binding files, that are in the CRD format (Kubernetes Custom Resource Definitions).For example, to run a kamelet binding file named **joke.yaml**:

```
#!/usr/bin/env jbang camel@apache/camel run
apiVersion: camel.apache.org/v1alpha1
kind: KameletBinding
metadata:
  name: joke
spec:
  source:
    ref:
      kind: Kamelet
      apiVersion: camel.apache.org/v1
      name: chuck-norris-source
    properties:
      period: 2000
  sink:
    ref:
      kind: Kamelet
      apiVersion: camel.apache.org/v1
      name: log-sink
    properties:
      show-headers: false
```

```
camel run joke.yaml
```

3.3.14. Run from the clipboard

You can run the Camel routes directly from the OS clipboard. This allows to copy some code, and then quickly run the route.

```
camel run clipboard.<extension>
```

Where **<extension>** is the type of the content of the clipboard is, such as **java**, **xml**, or **yaml**.

For example, you can copy this to your clipboard and then run the route:

```
<route>
  <from uri="timer:foo"/>
  <log message="Hello World"/>
</route>
```

```
camel run clipboard.xml
```

3.3.15. Controlling the local Camel integrations

To list the Camel integrations that are currently running, use the **ps** option:

```
camel ps
 PID   NAME                READY  STATUS   AGE
 61818 sample.camel.MyCamelApplica…  1/1  Running  26m38s
 62506 test1               1/1   Running  4m34s
```

This lists the PID, the name and age of the integration.

You can use the **stop** command to stop any of these running Camel integrations. For example to stop the **test1**, run:

```
camel stop test1
Stopping running Camel integration (pid: 62506)
```

You can use the PID to stop the integration:

```
camel stop 62506
Stopping running Camel integration (pid: 62506)
```
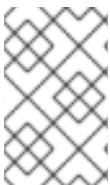
> **NOTE**
>
> You do not have to type the full name, as the stop command will match the integrations that starts with the input, for example you can type **camel stop t** to stop all integrations starting with **t**.

To stop all integrations, use the **--all** option as follows:

```
camel stop --all
Stopping running Camel integration (pid: 61818)
Stopping running Camel integration (pid: 62506)
```

3.3.16. Controlling the Spring Boot and Quarkus integrations

The Camel JBang CLI by default only controls the Camel integrations that are running using the CLI, for example, **camel run foo.java**.

For the CLI to be able to control and manage the Spring Boot or Quarkus applications, you need to add a dependency to these projects to integrate with the Camel CLI.

Spring Boot

In the Spring Boot application, add the following dependency:

```
<dependency>
    <groupId>org.apache.camel.springboot</groupId>
    <artifactId>camel-cli-connector-starter</artifactId>
</dependency>
```

Quarkus

In the Quarkus application, add the following dependency:

```
<dependency>
    <groupId>org.apache.camel.quarkus</groupId>
    <artifactId>camel-quarkus-cli-connector</artifactId>
</dependency>
```

3.3.17. Getting the status of Camel integrations

The **get** command in the Camel JBang is used for getting the Camel specific status for one or all of the running Camel integrations. To display the status of the running Camel integrations, run:

```
camel get
 PID  NAME    CAMEL         PLATFORM       READY STATUS   AGE   TOTAL  FAILED
INFLIGHT  SINCE-LAST
 61818  MyCamel  3.20.1-SNAPSHOT  Spring Boot v2.7.3  1/1  Running 28m34s  854    0
0    0s/0s/-
 63051  test1    3.20.1-SNAPSHOT  JBang        1/1  Running   18s   14   0     0   0s/0s/-
 63068  mygroovy 3.20.1-SNAPSHOT  JBang        1/1  Running   5s   2   0    0
0s/0s/-
```

The **camel get** command displays the default integrations, which is equivalent to typing the **camel get integrations** or the **camel get int** commands.

This displays the overall information for the every Camel integration, where you can see the total number of messages processed. The column **Since Last** shows how long time ago the last processed message for three stages (started/completed/failed).

The value of **0s/0s/-** means that the last started and completed message just happened (0 seconds ago), and that there has not been any failed message yet. In this example, **9s/9s/1h3m** means that last started and completed message is 9 seconds ago, and last failed is 1 hour and 3 minutes ago.

You can also see the status of every routes, from all the local Camel integrations with **camel get route**:

```
camel get route
 PID  NAME    ID    FROM           STATUS   AGE   TOTAL FAILED INFLIGHT MEAN
MIN  MAX  SINCE-LAST
 61818  MyCamel  hello  timer://hello?period=2000  Running 29m2s  870   0     0   0  0  14
0s/0s/-
 63051  test1    java   timer://java?period=1000  Running   46s   46   0     0   0  0  9
0s/0s/-
 63068  mygroovy  groovy timer://groovy?period=1000 Running   34s   34   0     0   0  0  5
0s/0s/-
```
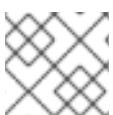
> **NOTE**
>
> Use **camel get --help** to display all the available commands.

3.3.17.1. Top status of the Camel integrations

The **camel top** command is used for getting top utilization statistics (highest to lowest heap used memory) of the running Camel integrations.

```
camel top
 PID  NAME     JAVA     CAMEL          PLATFORM          STATUS   AGE       HEAP         NON-
HEAP   GC    THREADS  CLASSES
 22104  chuck    11.0.13  3.20.1-SNAPSHOT  JBang              Running   2m10s  131/322/4294 MB
70/73 MB  17ms (6)     7/8  7456/7456
 14242  MyCamel  11.0.13  3.20.1-SNAPSHOT  Spring Boot v2.7.3  Running  33m40s  115/332/4294
MB  62/66 MB  37ms (6)    16/16  8428/8428
 22116  bar      11.0.13  3.20.1-SNAPSHOT  JBang              Running   2m7s   33/268/4294 MB
54/58 MB  20ms (4)     7/8  6104/6104
```

The **HEAP** column shows the heap memory (used/committed/max) and the non–heap (used/committed). The **GC** column shows the garbage collection information (time and total runs). The **CLASSES** column shows the number of classes (loaded/total).

You can also see the top performing routes (highest to lowest mean processing time) of every routes, from all the local Camel integrations with **camel top route**:

```
camel top route
 PID  NAME     ID                 FROM                        STATUS   AGE   TOTAL  FAILED
INFLIGHT  MEAN  MIN  MAX  SINCE-LAST
 22104  chuck    chuck-norris-source-1  timer://chuck?period=10000          Started   10s    1     0
0  163  163  163      9s
 22116  bar      route1                 timer://yaml2?period=1000          Started   7s    7     0      0   1
0  11        0s
 22104  chuck    chuck                  kamelet://chuck-norris-source      Started   10s    1     0      0
0   0   0        9s
 22104  chuck    log-sink-2             kamelet://source?routeId=log-sink-2 Started   10s    1     0
0   0   0   0        9s
 14242  MyCamel  hello                  timer://hello?period=2000          Started  31m41s  948    0
0   0   0   4        0s
```
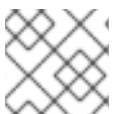
> **NOTE**
>
> Use **camel top --help** to display all the available commands.

3.3.17.2. Starting and Stopping the routes

The **camel cmd** is used for executing the miscellaneous commands in the running Camel integrations, for example, the commands to start and stop the routes.

To stop all the routes in the **chuck** integration, run:

```
camel cmd stop-route chuck
```

The status will be then changed to **Stopped** for the **chuck** integration:

```
camel get route
 PID  NAME    ID              FROM                        STATUS   AGE   TOTAL  FAILED
INFLIGHT  MEAN  MIN  MAX  SINCE-LAST
```

```
 81663  chuck   chuck                   kamelet://chuck-norris-source       Stopped         600   0      0
 0   0   1       4s
 81663  chuck   chuck-norris-source-1  timer://chuck?period=10000           Stopped         600   0
 0  65  52 290      4s
 81663  chuck   log-sink-2              kamelet://source?routeId=log-sink-2  Stopped         600   0
 0   0   0   1       4s
 83415  bar     route1                  timer://yaml2?period=1000           Started  5m30s  329   0      0
 0   0  10       0s
 83695  MyCamel hello                   timer://hello?period=2000           Started  3m52s  116   0      0
 0   0   9       1s
```

To start the route, run:

```
camel cmd start-route chuck
```

To stop **all** the routes in every the Camel integration, use the **--all** flag as follows:

```
camel cmd stop-route --all
```

To start **all** the routes, use:

```
camel cmd start-route --all
```

> **NOTE**
>
> You can stop one or more route by their ids by separating them using comma, for example, **camel cmd start-route --id=route1,hello**. Use the **camel cmd start-route --help** command for more details.

3.3.17.3. Configuring the logging levels

You can see the current logging levels of the running Camel integrations by:

```
camel cmd logger
 PID  NAME  AGE  LOGGER  LEVEL
 90857  bar  2m48s  root  INFO
 91103  foo    20s  root  INFO
```

The logging level can be changed at a runtime. For example, to change the level for the **foo** to DEBUG, run:

```
camel cmd logger --level=DEBUG foo
```
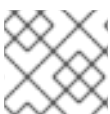
> **NOTE**
>
> You can use **--all** to change logging levels for all running integrations.

3.3.17.4. Listing services

Some Camel integrations may host a service which clients can call, such as REST, or SOAP-WS, or socket-level services using TCP protocols. You can list the available services as shown in the example below:

```
camel get service
 PID  NAME     COMPONENT    PROTOCOL  SERVICE
 1912 netty    netty        tcp       tcp:localhost:4444
 2023 greetings platform-http rest      http://0.0.0.0:7777/camel/greetings/{name} (GET)
 2023 greetings platform-http http      http://0.0.0.0:7777/q/dev
```

Here, you can see the two Camel integrations. The netty integration hosts a TCP service that is available on port 4444. The other Camel integration hosts a REST service that can be called via GET only. The third integration comes with embedded web console (started with the **--console** option).

> **NOTE**
>
> For a service to be listed the Camel components must be able to advertise the services using Camel Console.

3.3.17.5. Listing state of Circuit Breakers

If your Camel integration uses the link:https://camel.apache.org/components/3.22.x/eips/circuitBreaker-eip.html [Circuit Breaker], then you can output the status of the breakers with Camel JBang as follows:

```
camel get circuit-breaker
 PID  NAME COMPONENT    ROUTE  ID            STATE      PENDING  SUCCESS FAIL
REJECT
 56033 mycb resilience4j route1 circuitBreaker1 HALF_OPEN     5       2    3      0
```

Here we can see the circuit breaker is in **half open** state, that is a state where the breaker is attempting to transition back to closed, if the failures start to drop.

> **NOTE**
>
> You can run the command with **watch** option to show the latest state, for example, **watch camel get circuit-breaker**.

3.3.18. Using Jolokia and Hawtio

The web console allows inspecting running the Camel integrations, such as all the JMX management information, and not but least to visualize the Camel routes with live performance metrics.

To allow Hawtio to inspect the Camel integrations, the Jolokia JVM Agent must be installed in the running integration. This is done explicitly as follows:

```
camel ps
 PID  NAME                   READY  STATUS   AGE
 61818 sample.camel.MyCamelApplica…  1/1   Running  26m38s
 62506 test1.java             1/1   Running  4m34s
```

With the PID, you can then attach Jolokia:

```
camel jolokia 62506
Started Jolokia for PID 62506
http://127.0.0.1:8778/jolokia/
```

Instead of using the PID you can also attach by the name pattern. In this example, the two Camel integrations have unique names (foo and test1), you can attach Jolokia without the PID as follows:

```
camel jolokia te
Started Jolokia for PID 62506
http://127.0.0.1:8778/jolokia/
```

Then you can launch the Hawtio using Camel JBang:

```
camel hawtio
```

This will automatically download and start the Hawtio, and then open in the web browser.
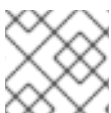
> **NOTE**
>
> See **camel hawtio --help** for more options.

When the Hawtio launches in the web browser, click the **Discover** tab which lists the all local available Jolokia Agents. You can use **camel jolokia PID** to connect to multiple different Camel integrations and from this list select which to load.

Click the green **lightning** icon to connect to the specific running Camel integration.

You can uninstall the Jolokia JVM Agent in a running Camel integration when no longer needed:

```
camel jolokia 62506 --stop
Stopped Jolokia for PID 62506
```

It is also possible to achieve this with only one command, as follows:

```
camel hawtio test1
```

Where **test1** is the name of the running Camel integration. When you stop Hawtio (using **ctrl** + **c**), then Camel will attempt to uninstall the Jolokia JVM Agent, however this is not successful sometimes, because the JVM is being terminated which can prevent camel–jbang from doing JVM process communication to the running Camel integration.

3.3.19. Scripting from the terminal using pipes

You can execute a Camel JBang file as a script that is used for terminal scripting with pipes and filters.

> **NOTE**
>
> Every time the script is executed a JVM is started with Camel. This is not very fast or low on memory usage, so use the Camel JBang terminal scripting, for example, to use the many Camel components or Kamelets to more easily send or receive data from disparate IT systems.

This requires to add the following line in top of the file, for example, as in the **upper.yaml** file below:

```
///usr/bin/env jbang --quiet camel@apache/camel pipe "$0" "$@" ; exit $?
```

```
# Will upper-case the input
- from:
    uri: "stream:in"
    steps:
      - setBody:
          simple: "${body.toUpperCase()}"
      - to: "stream:out"
```

To execute this as a script, you need to set the execute file permission:

```
chmod +x upper.yaml
```

Then you can then execute this as a script:

```
echo "Hello\nWorld" | ./upper.yaml
```

This outputs:

```
HELLO
WORLD
```

You can turn on the logging using **--logging=true** which then logs to **.camel-jbang/camel-pipe.log** file. The name of the logging file cannot be configured.

```
echo "Hello\nWorld" | ./upper.yaml --logging=true
```

3.3.19.1. Using **stream:in** with line vs raw mode

When using **stream:in** to read data from **System in** then the Stream Component works in two modes:

- line mode (default) – reads input as single lines (separated by line breaks). Message body is a **String**.

- raw mode – reads the entire stream until *end of stream*. Message body is a **byte[]**.

> **NOTE**
>
> The default mode is due to historically how the stream component was created. Therefore, you may want to set **stream:in?readLine=false** to use raw mode.

3.3.20. Running local Kamelets

You can use Camel JBang to try local Kamelets, without the need to publish them on GitHub or package them in a jar.

```
camel run --local-kamelet-dir=/path/to/local/kamelets earthquake.yaml
```

> **NOTE**
>
> When the kamelets are from local file system, then they can be live reloaded, if they are updated, when you run Camel JBang in **--dev** mode.

You can also point to a folder in a GitHub repository. For example:

```
camel run --local-kamelet-dir=https://github.com/apache/camel-kamelets-examples/tree/main/custom-kamelets user.java
```
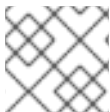
> **NOTE**
>
> If a kamelet is loaded from GitHub, then they cannot be live reloaded.

3.3.21. Using the `platform-http` component

When a route is started from **platform-http** then the Camel JBang automatically includes a VertX HTTP server running on port 8080. following example shows the route in a file named **server.yaml**:

```
- from:
    uri: "platform-http:/hello"
    steps:
      - set-body:
          constant: "Hello World"
```

You can run this example with:

```
camel run server.yaml
```

And then call the HTTP service with:

```
$ curl http://localhost:8080/hello
Hello World%
```

3.3.22. Using Java beans and processors

There is basic support for including regular Java source files together with Camel routes, and let the Camel JBang runtime compile the Java source. This means you can include smaller utility classes, POJOs, Camel Processors that the application needs.

> **NOTE**
>
> The Java source files cannot use package names.

3.3.23. Dependency Injection in Java classes

When running the Camel integrations with **camel-jbang**, the runtime is **camel-main** based. This means there is no Spring Boot, or Quarkus available. However, there is a support for using annotation based dependency injection in Java classes.

3.3.23.1. Using Spring Boot dependency injection

You can use the following Spring Boot annotations:

- **@org.springframework.stereotype.Component** or **@org.springframework.stereotype.Service** on class level to create an instance of the class and register in the Registry.

- **@org.springframework.beans.factory.annotation.Autowired** to dependency inject a bean on a class field. **@org.springframework.beans.factory.annotation.Qualifier** can be used to specify the bean id.

- **@org.springframework.beans.factory.annotation.Value** to inject a property placeholder. Such as a property defined in **application.properties**.

- **@org.springframework.context.annotation.Bean** on a method to create a bean by invoking the method.

3.3.24. Debugging

There are two kinds of debugging available:

- **Java debugging** – Java code debugging (Standard Java)

- **Camel route debugging** – Debugging Camel routes (requires Camel tooling plugins)

3.3.24.1. Java debugging

You can debug your integration scripts by using the **--debug** flag provided by JBang. However, to enable the Java debugging when starting the JVM, use the **jbang** command, instead of **camel** as shown:

```
jbang --debug  camel@apache/camel run hello.yaml
Listening for transport dt_socket at address: 4004
```

As you can see the default listening port is 4004 but can be configured as described in JBang debugging.

This is a standard Java debug socket. You can then use the IDE of your choice. You can add a **Processor** to put breakpoints hit during route execution (as opposed to route definition creation).

3.3.24.2. Camel route debugging

The Camel route debugger is available by default (the **camel-debug** component is automatically added to the classpath). By default, it can be reached through JMX at the URL **service:jmx:rmi:///jndi/rmi://localhost:1099/jmxrmi/camel**. You can then use the Integrated Development Environment (IDE) of your choice.

3.3.25. Health Checks

The status of health checks is accessed using the Camel JBang from the CLI as follows:

```
camel get health
 PID  NAME   AGE ID         RL STATE RATE   SINCE  MESSAGE
 61005 mybind  8s camel/context  R  UP   2/2/-  1s/3s/-
```

Here you can see the Camel is **UP**. The application has been running for 8 seconds, and there are two health checks invoked.

The output shows the **default** level of checks as:

- **CamelContext** health check

- Component specific health checks (such as from **camel-kafka** or **camel-aws**)

- Custom health checks

- Any check which are not **UP**

The **RATE** column shows three numbers separated by /. So **2/2/-** means 2 checks in total, 2 successful and no failures. The two last columns will reset when a health check changes state as this number is the number of consecutive checks that was successful or failure. So if the health check starts to fail then the numbers could be:

```
camel get health
 PID   NAME    AGE   ID         RL  STATE  RATE    SINCE    MESSAGE
 61005  mybind  3m2s  camel/context  R   UP    77/-/3  1s/-/17s  some kind of error
```

Here you can see the numbers is changed to **77/-/3**. This means the total number of checks is 77. There is no success, but the check has been failing 3 times in a row. The **SINCE** column corresponds to the **RATE**. So in this case you can see the last check was 1 second ago, and that the check has been failing for 17 second in a row.

You can use **--level=full** to output every health checks that will include consumer and route level checks as well.

A health check may often be failed due to an exception was thrown which can be shown using **--trace** flag:

```
camel get health --trace
 PID   NAME     AGE   ID                         RL  STATE  RATE      SINCE     MESSAGE
 61038  mykafka  6m19s  camel/context                     R   UP    187/187/-  1s/6m16s/-
 61038  mykafka  6m19s  camel/kafka-consumer-kafka-not-secure…   R  DOWN   187/-/187  1s/-
 /6m16s  KafkaConsumer is not ready - Error: Invalid url in bootstrap.servers: value


---------------------------------------------------------------------------------------------------------
                                       STACK-TRACE
---------------------------------------------------------------------------------------------------------
   PID: 61038
   NAME: mykafka
   AGE: 6m19s
   CHECK-ID: camel/kafka-consumer-kafka-not-secured-source-1
   STATE: DOWN
   RATE: 187
   SINCE: 6m16s
   METADATA:
      bootstrap.servers = value
      group.id = 7d8117be-41b4-4c81-b4df-cf26b928d38a
      route.id = kafka-not-secured-source-1
      topic = value
   MESSAGE: KafkaConsumer is not ready - Error: Invalid url in bootstrap.servers: value
   org.apache.kafka.common.KafkaException: Failed to construct kafka consumer
      at org.apache.kafka.clients.consumer.KafkaConsumer.<init>(KafkaConsumer.java:823)
      at org.apache.kafka.clients.consumer.KafkaConsumer.<init>(KafkaConsumer.java:664)
      at org.apache.kafka.clients.consumer.KafkaConsumer.<init>(KafkaConsumer.java:645)
      at org.apache.kafka.clients.consumer.KafkaConsumer.<init>(KafkaConsumer.java:625)
      at
org.apache.camel.component.kafka.DefaultKafkaClientFactory.getConsumer(DefaultKafkaClientFactory
.java:34)
      at
```

```
org.apache.camel.component.kafka.KafkaFetchRecords.createConsumer(KafkaFetchRecords.java:241
)
    at
org.apache.camel.component.kafka.KafkaFetchRecords.createConsumerTask(KafkaFetchRecords.java
:201)
    at org.apache.camel.support.task.ForegroundTask.run(ForegroundTask.java:123)
    at org.apache.camel.component.kafka.KafkaFetchRecords.run(KafkaFetchRecords.java:125)
    at java.base/java.util.concurrent.Executors$RunnableAdapter.call(Executors.java:515)
    at java.base/java.util.concurrent.FutureTask.run(FutureTask.java:264)
    at
java.base/java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1128)
    at
java.base/java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:628)
    at java.base/java.lang.Thread.run(Thread.java:829)
  Caused by: org.apache.kafka.common.config.ConfigException: Invalid url in bootstrap.servers:
value
    at org.apache.kafka.clients.ClientUtils.parseAndValidateAddresses(ClientUtils.java:59)
    at org.apache.kafka.clients.ClientUtils.parseAndValidateAddresses(ClientUtils.java:48)
    at org.apache.kafka.clients.consumer.KafkaConsumer.<init>(KafkaConsumer.java:730)
    ... 13 more
```

Here you can see that the health check fails because of the
org.apache.kafka.common.config.ConfigException which is due to invalid configuration: **Invalid url
in bootstrap.servers: value**.

> **NOTE**
>
> Use **camel get health --help** to see all the various options.

3.4. LISTING WHAT CAMEL COMPONENTS IS AVAILABLE

Camel comes with a lot of artifacts out of the box which are:

- components

- data formats

- expression languages

- miscellaneous components

- kamelets

You can use the Camel CLI to list what Camel provides using the **camel catalog** command. For
example, to list all the components:

```
camel catalog components
```

To see which Kamelets are available:

```
camel catalog kamelets
```

> **NOTE**
>
> Use **camel catalog --help** to see all possible commands.

3.4.1. Displaying component documentation

The **doc** goal can show quick documentation for every component, dataformat, and kamelets. For example, to see the kafka component run:

```
camel doc kafka
```

> **NOTE**
>
> The documentation is not the full documentation as shown on the website, as the Camel CLI does not have direct access to this information and can only show a basic description of the component, but include tables for every configuration option.

To see the documentation for jackson dataformat:

```
camel doc jackson
```

In some rare cases then there may be a component and dataformat with the same name, and the **doc** goal prioritizes components. In such a situation you can prefix the name with dataformat, for example:

```
camel doc dataformat:thrift
```

You can also see the kamelet documentation such as shown:

```
camel doc aws-kinesis-sink
```

3.4.1.1. Browsing online documentation from the Camel website

You can use the **doc** command to quickly open the url in the web browser for the online documentation. For example to browse the kafka component, you use **--open-url**:

```
camel doc kafka --open-url
```

This also works for data formats, languages, kamelets.

```
camel doc aws-kinesis-sink --open-url
```

> **NOTE**
>
> To just get the link to the online documentation, then use **camel doc kafka --url**.

3.4.1.2. Filtering options listed in the tables

Some components may have many options, and in such cases you can use the **--filter** option to only list the options that match the filter either in the name, description, or the group (producer, security, advanced).

For example, to list only security related options:

```
camel doc kafka --filter=security
```

To list only something about **timeout**:

```
camel doc kafka --filter=timeout
```

3.5. OPEN API

Camel JBang allows to quickly expose an Open API service using **contract first** approach, where you have an existing OpenAPI specification file. Camel JBang bridges each API endpoints from the OpenAPI specification to a Camel route with the naming convention **direct:<operationId>**. This make it quicker to implement a Camel route for a given operation.

See the OpenAPI example for more details.

3.6. GATHERING LIST OF DEPENDENCIES

The dependencies are automatically resolved when you work with Camel JBang. This means that you do not have to use a build system like Maven or Gradle to add every Camel components as a dependency.

However, you may want to know what dependencies are required to run the Camel integration. You can use the **dependencies** command to see the dependencies required. The command output does not output a detailed tree, such as **mvn dependencies:tree**, as the output is intended to list which Camel components, and other JARs needed (when using Kamelets).

The dependency output by default is **vanilla** Apache Camel with the **camel-main** as runtime, as shown:

```
camel dependencies
org.apache.camel:camel-dsl-modeline:3.20.0
org.apache.camel:camel-health:3.20.0
org.apache.camel:camel-kamelet:3.20.0
org.apache.camel:camel-log:3.20.0
org.apache.camel:camel-rest:3.20.0
org.apache.camel:camel-stream:3.20.0
org.apache.camel:camel-timer:3.20.0
org.apache.camel:camel-yaml-dsl:3.20.0
org.apache.camel.kamelets:camel-kamelets-utils:0.9.3
org.apache.camel.kamelets:camel-kamelets:0.9.3
```

The output is by default a line per maven dependency in GAV format (groupId:artifactId:version).

You can specify the **Maven** format for the the output as shown:

```
camel dependencies --output=maven
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-main</artifactId>
    <version>3.20.0</version>
</dependency>
<dependency>
    <groupId>org.apache.camel</groupId>
```

```xml
    <artifactId>camel-dsl-modeline</artifactId>
    <version>3.20.0</version>
</dependency>
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-health</artifactId>
    <version>3.20.0</version>
</dependency>
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-kamelet</artifactId>
    <version>3.20.0</version>
</dependency>
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-log</artifactId>
    <version>3.20.0</version>
</dependency>
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-rest</artifactId>
    <version>3.20.0</version>
</dependency>
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-stream</artifactId>
    <version>3.20.0</version>
</dependency>
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-timer</artifactId>
    <version>3.20.0</version>
</dependency>
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-yaml-dsl</artifactId>
    <version>3.20.0</version>
</dependency>
<dependency>
    <groupId>org.apache.camel.kamelets</groupId>
    <artifactId>camel-kamelets-utils</artifactId>
    <version>0.9.3</version>
</dependency>
<dependency>
    <groupId>org.apache.camel.kamelets</groupId>
    <artifactId>camel-kamelets</artifactId>
    <version>0.9.3</version>
</dependency>
```

You can also choose the target runtime as either `quarkus` or **spring-boot** as shown:

```
camel dependencies --runtime=spring-boot
org.springframework.boot:spring-boot-starter-actuator:2.7.5
org.springframework.boot:spring-boot-starter-web:2.7.5
org.apache.camel.springboot:camel-spring-boot-engine-starter:3.20.0
org.apache.camel.springboot:camel-dsl-modeline-starter:3.20.0
```

```
org.apache.camel.springboot:camel-kamelet-starter:3.20.0
org.apache.camel.springboot:camel-log-starter:3.20.0
org.apache.camel.springboot:camel-rest-starter:3.20.0
org.apache.camel.springboot:camel-stream-starter:3.20.0
org.apache.camel.springboot:camel-timer-starter:3.20.0
org.apache.camel.springboot:camel-yaml-dsl-starter:3.20
org.apache.camel.kamelets:camel-kamelets-utils:0.9.3
org.apache.camel.kamelets:camel-kamelets:0.9.3
```

3.7. CREATING PROJECTS

You can **export** your Camel JBang integration to a traditional Java based project such as Spring Boot or Quarkus. You may want to do this after you have built a prototype using Camel JBang, and are in the need of a traditional Java based project with more need for Java coding, or to use the powerful runtimes of Spring Boot, Quarkus or vanilla Camel Main.

3.7.1. Exporting to Camel Spring Boot

The command **export --runtime=spring-boot** exports your current Camel JBang file(s) to a Maven based Spring Boot project with files organized in **src/main/** folder structure.

For example, to export to the Spring Boot using the Maven groupId **com.foo** and the artifactId **acme** and with version **1.0-SNAPSHOT**, run:

```
camel export --runtime=spring-boot --gav=com.foo:acme:1.0-SNAPSHOT
```

> **NOTE**
>
> This will export to the **current** directory, this means that files are moved into the needed folder structure.

To export to another directory, run:

```
camel export --runtime=spring-boot --gav=com.foo:acme:1.0-SNAPSHOT --directory=../myproject
```

When exporting to the Spring Boot, the Camel version defined in the **pom.xml** or **build.gradle** is the same version as Camel JBang uses. However, you can specify the different Camel version as shown:

```
camel export --runtime=spring-boot --gav=com.foo:acme:1.0-SNAPSHOT --directory=../myproject --camel-spring-boot-version=3.20.1.redhat-00097
```

> **NOTE**
>
> See the possible options by running the **camel export --help** command for more details.

3.7.2. Exporting with Camel CLI included

When exporting to Spring Boot, Quarkus or Camel Main, the Camel JBang CLI is not included out of the box. To continue to use the Camel CLI (that is **camel**), you need to add **camel:cli-connector** in the **--deps** option, as shown:

```
camel export --runtime=quarkus --gav=com.foo:acme:1.0-SNAPSHOT --deps=camel:cli-connector --
directory=../myproject
```

3.7.3. Configuring exporting

The export command by default loads the configuration from **application.properties** file which is used for exporting specific parameters such as selecting the runtime and java version.

The following options related to **exporting**, can be configured in the **application.properties** file:

Option	Description
camel.jbang.runtime	Runtime (spring-boot, quarkus, or camel-main)
camel.jbang.gav	The Maven group:artifact:version
camel.jbang.dependencies	Additional dependencies (Use commas to separate multiple dependencies). See more details at Adding custom JARs.
camel.jbang.classpathFiles	Additional files to add to classpath (Use commas to separate multiple files). See more details at Adding custom JARs.
camel.jbang.javaVersion	Java version (11 or 17)
camel.jbang.kameletsVersion	Apache Camel Kamelets version
camel.jbang.localKameletDir	Local directory for loading Kamelets
camel.jbang.camelSpringBootVersion	Camel version to use with Spring Boot
camel.jbang.springBootVersion	Spring Boot version
camel.jbang.quarkusGroupId	Quarkus Platform Maven groupId
camel.jbang.quarkusArtifactId	Quarkus Platform Maven artifactId
camel.jbang.quarkusVersion	Quarkus Platform version
camel.jbang.mavenWrapper	Include Maven Wrapper files in exported project
camel.jbang.gradleWrapper	Include Gradle Wrapper files in exported project
camel.jbang.buildTool	Build tool to use (maven or gradle)

Option	Description
camel.jbang.repos	Additional maven repositories for download on-demand (Use commas to separate multiple repositories)
camel.jbang.mavenSettings	Optional location of maven setting.xml file to configure servers, repositories, mirrors and proxies. If set to false, not even the default ~/.m2/settings.xml will be used.
camel.jbang.mavenSettingsSecurity	Optional location of maven settings-security.xml file to decrypt settings.xml
camel.jbang.exportDir	Directory where the project will be exported.
camel.jbang.platform-http.port	HTTP server port to use when running standalone Camel, such as when --console is enabled (port 8080 by default).
camel.jbang.console	Developer console at /q/dev on local HTTP server (port 8080 by default) when running standalone Camel.
camel.jbang.health	Health check at /q/health on local HTTP server (port 8080 by default) when running standalone Camel.

NOTE

These are the options from the export command. You can see more details and default values using **camel export --help**.

3.8. TROUBLESHOOTING

When you use JBang, it stores the state in ~/**.jbang** directory. This is also the location where JBang stores downloaded JARs. Camel JBang also downloads the needed dependencies while running. However, these dependencies are downloaded to your local Maven repository ~/**.m2**. So when you troubleshoot the problems such as an outdated JAR while running the Camel JBang, try to delete these directories, or parts of it.

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

```xml
<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. SPECIFYING CAMEL ROUTES USING SPRING XML

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

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

4.2. CONFIGURING COMPONENTS AND ENDPOINTS

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

```xml
<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.

4.3. 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"/>
```

4.4. 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.