Abstract

This guide introduces Red Hat Integration - Camel K, explains how to install on OpenShift, and how to get started deploying Camel K integrations and tutorials with OpenShift Serverless. This guide also explains how to configure and monitor Camel K integrations, and provides reference details on Camel K traits that you can configure for advanced features.
# Table of Contents

## CHAPTER 1. INTRODUCTION TO CAMEL K
- 1. CAMEL K OVERVIEW .......................................................... 5
- 1.2. CAMEL K TECHNOLOGY PREVIEW FEATURES ...................... 6
  - 1.2.1. Platform and component versions ................................. 6
  - 1.2.2. Technology Preview features ...................................... 6
- 1.3. CAMEL K CLOUD-NATIVE ARCHITECTURE ......................... 6
  - 1.3.1. Kamelets ............................................................... 7
- 1.4. CAMEL K DEVELOPMENT TOOLING .................................. 8
- 1.5. CAMEL K DISTRIBUTIONS ............................................. 8

## CHAPTER 2. INSTALLING CAMEL K
- 2.1. INSTALLING CAMEL K FROM THE OPENSHIFT OPERATORHUB .... 10
- 2.2. INSTALLING OPENSSHIFT SERVERLESS FROM THE OPERATORHUB ................................................................. 11
- 2.3. INSTALLING THE CAMEL K AND OPENSHEET COMMAND LINE TOOLS ...................................................... 12

## CHAPTER 3. GETTING STARTED WITH CAMEL K
- 3.1. SETTING UP YOUR CAMEL K DEVELOPMENT ENVIRONMENT .... 14
- 3.2. DEVELOPING CAMEL K INTEGRATIONS IN JAVA ................ 16
- 3.3. DEVELOPING CAMEL K INTEGRATIONS IN XML ................. 16
- 3.4. DEVELOPING CAMEL K INTEGRATIONS IN YAML ............... 17
- 3.5. RUNNING CAMEL K INTEGRATIONS .................................. 18
- 3.6. RUNNING CAMEL K INTEGRATIONS IN DEVELOPMENT MODE .... 20
- 3.7. RUNNING CAMEL K INTEGRATIONS USING MODELINE .......... 22

## CHAPTER 4. CAMEL K QUICK START DEVELOPER TUTORIALS
- 4.1. DEPLOYING A BASIC CAMEL K JAVA INTEGRATION ............... 25
- 4.2. DEPLOYING A CAMEL K SERVERLESS INTEGRATION WITH KNATIVE ................................................................. 26
- 4.3. DEPLOYING A CAMEL K TRANSFORMATIONS INTEGRATION .... 27
- 4.4. DEPLOYING A CAMEL K SERVERLESS EVENT STREAMING INTEGRATION ................................................................. 28
- 4.5. DEPLOYING A CAMEL K SERVERLESS API-BASED INTEGRATION .. 29
- 4.6. DEPLOYING A CAMEL K SAAS INTEGRATION ...................... 30

## CHAPTER 5. MANAGING CAMEL K INTEGRATIONS
- 5.1. MANAGING CAMEL K INTEGRATIONS ................................ 32
- 5.2. MANAGING CAMEL K INTEGRATION LOGGING LEVELS ........ 34

## CHAPTER 6. MONITORING CAMEL K INTEGRATIONS
- 6.1. ENABLING USER WORKLOAD MONITORING IN OPENSHEET .... 36
- 6.2. CONFIGURING CAMEL K INTEGRATION METRICS ................ 37
- 6.3. ADDING CUSTOM CAMEL K INTEGRATION METRICS ............. 37

## CHAPTER 7. CONFIGURING CAMEL K INTEGRATIONS
- 7.1. CONFIGURING CAMEL K INTEGRATIONS USING PROPERTIES ... 41
- 7.2. CONFIGURING CAMEL K INTEGRATIONS USING PROPERTY FILES ................. 42
- 7.3. CONFIGURING CAMEL K PROPERTIES USING AN OPENSHEET CONFIGMAP ................................................................. 43
- 7.4. CONFIGURING CAMEL K PROPERTIES USING AN OPENSHEET SECRET ................................................................. 44
- 7.5. CONFIGURING CAMEL INTEGRATION COMPONENTS ............. 45
- 7.6. CONFIGURING CAMEL K INTEGRATION DEPENDENCIES ...... 46

## CHAPTER 8. CAMEL K TRAIT CONFIGURATION REFERENCE
- Camel K feature traits ......................................................... 48
- Camel K core platform traits ................................................ 48
- 8.1. CAMEL K TRAIT AND PROFILE CONFIGURATION ............ 49
8.2. CAMEL K FEATURE TRAITS
8.2.1. 3scale Trait
8.2.1.1. Configuration
8.2.2. Affinity Trait
8.2.2.1. Configuration
8.2.2.2. Examples
8.2.3. Cron Trait
8.2.3.1. Configuration
8.2.4. Gc Trait
8.2.4.1. Configuration
8.2.5. Istio Trait
8.2.5.1. Configuration
8.2.6. Jolokia Trait
8.2.6.1. Configuration
8.2.7. Knative Trait
8.2.7.1. Configuration
8.2.8. Knative Service Trait
8.2.8.1. Configuration
8.2.9. Master Trait
8.2.9.1. Configuration
8.2.10. Prometheus Trait
8.2.10.1. Configuration
8.2.11. Quarkus Trait
8.2.11.1. Configuration
8.2.11.2. Supported Camel Components
8.2.12. Route Trait
8.2.12.1. Configuration
8.2.13. Service Trait
8.2.13.1. Configuration

8.3. CAMEL K PLATFORM TRAITS
8.3.1. Builder Trait
8.3.1.1. Configuration
8.3.2. Container Trait
8.3.2.1. Configuration
8.3.3. Camel Trait
8.3.3.1. Configuration
8.3.4. Dependencies Trait
8.3.4.1. Configuration
8.3.5. Deployer Trait
8.3.5.1. Configuration
8.3.6. Deployment Trait
8.3.6.1. Configuration
8.3.7. Environment Trait
8.3.7.1. Configuration
8.3.8.Jvm Trait
8.3.8.1. Configuration
8.3.9. Openapi Trait
8.3.9.1. Configuration
8.3.10. Owner Trait
8.3.10.1. Configuration
8.3.11. Platform Trait
8.3.11.1. Configuration
<table>
<thead>
<tr>
<th>CHAPTER 9. CAMEL K COMMAND REFERENCE</th>
<th>72</th>
</tr>
</thead>
<tbody>
<tr>
<td>9.1. CAMEL K COMMAND LINE</td>
<td>72</td>
</tr>
<tr>
<td>9.2. CAMEL K MODELINE OPTIONS</td>
<td>73</td>
</tr>
</tbody>
</table>
CHAPTER 1. INTRODUCTION TO CAMEL K

This chapter introduces the concepts, features, and cloud-native architecture provided by Red Hat Integration - Camel K:

- Section 1.1, “Camel K overview”
- Section 1.2, “Camel K Technology Preview features”
- Section 1.3, “Camel K cloud-native architecture”
- Section 1.4, “Camel K development tooling”
- Section 1.5, “Camel K distributions”

IMPORTANT

Red Hat Integration - Camel K 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.

1.1. CAMEL K OVERVIEW

Red Hat Integration - Camel K is a lightweight integration framework built from Apache Camel K that runs natively in the cloud on OpenShift. Camel K is specifically designed for serverless and microservice architectures. You can use Camel K to instantly run your integration code written in Camel Domain Specific Language (DSL) directly on OpenShift. Camel K is a subproject of the Apache Camel open source community: https://github.com/apache/camel-k.

Camel K is implemented in the Go programming language and uses the Kubernetes Operator SDK to automatically deploy integrations in the cloud. For example, this includes automatically creating services and routes on OpenShift. This provides much faster turnaround times when deploying and redeploying integrations in the cloud, such as a few seconds or less instead of minutes.

The Camel K runtime provides significant performance optimizations. The Quarkus cloud-native Java framework is enabled by default to provide faster start up times, and lower memory and CPU footprints. When running Camel K in developer mode, you can make live updates to your integration DSL and view results instantly in the cloud on OpenShift, without waiting for your integration to redeploy.

Using Camel K with OpenShift Serverless and Knative Serving, containers are created only as needed and are autoscaled under load up and down to zero. This reduces cost by removing the overhead of server provisioning and maintenance and enables you to focus on application development instead.

Using Camel K with OpenShift Serverless and Knative Eventing, you can manage how components in your system communicate in an event-driven architecture for serverless applications. This provides flexibility and creates efficiencies through decoupled relationships between event producers and consumers using a publish-subscribe or event-streaming model.

Additional resources
1.2. CAMEL K TECHNOLOGY PREVIEW FEATURES

The Camel K Technology Preview includes the following main platforms and features:

1.2.1. Platform and component versions

- OpenShift Container Platform 4.6
- OpenShift Serverless 1.7
- Quarkus 1.7 Java runtime
- Camel 3.5
- Java 11

1.2.2. Technology Preview features

- Knative Serving for autoscaling and scale-to-zero
- Knative Eventing for event-driven architectures
- Performance optimizations using Quarkus runtime by default
- Camel integrations written in Java, XML, or YAML DSL
- Development tooling with Visual Studio Code
- Monitoring of integrations using Prometheus in OpenShift
- Quickstart tutorials, including new Transformations and SaaS examples
- Kamelet catalog of connectors to external systems such as AWS, Jira, and Salesforce

**NOTE**

The Technology Preview includes building Camel K integration images with OpenShift only. Installing Camel K with the Buildah or Kaniko image builder is not included in the Technology Preview and has community-only support.

1.3. CAMEL K CLOUD-NATIVE ARCHITECTURE

The following diagram shows a simplified view of the Camel K cloud-native architecture:
Camel K automatically wraps the Camel integration in a Kubernetes custom resource and uploads it to the cloud. This architecture provides the following benefits:

- Cloud-native integration and developer experience on OpenShift for faster development cycles
- Automatic installation of Camel K and deployment of integrations using the Camel K Operator
- Live code updates using Camel K developer mode, without needing to redeploy
- Autoscaling up and down to zero with Knative using the OpenShift Serverless Operator
- Performance optimizations and cost savings using the Quarkus Java runtime:
  - Pre-compilation and pre-initialization of code at build time
  - Fast start up, deploy, and redeploy times
  - Low memory and CPU footprint
- Automatic dependency resolution of Camel integration code
- Configuring advanced features using Camel K traits on the command line and modeline

Additional resources

- Apache Camel architecture

1.3.1. Kamelets

Kamelets hide the complexity of connecting to external systems behind a simple interface, which contains all the information needed to instantiate them, even for users who are not familiar with Camel.

Kamelets are implemented as custom resources that you can install on an OpenShift cluster and use in Camel K integrations. They contain high-level connectors in the form of route templates. Kamelets abstract the details of connecting to external systems. You can also combine Kamelets to create complex Camel integrations, just like using standard Camel components.

Additional resources

- Apache Camel Kamelets
1.4. CAMEL K DEVELOPMENT TOOLING

The Camel K Technology Preview provides development tooling extensions for Visual Studio (VS) Code, Red Hat CodeReady WorkSpaces, and Eclipse Che. The Camel-based tooling extensions include features such as automatic completion of Camel DSL code, Camel K modeline configuration, and Camel K traits. While Didact tutorial tooling extensions provide automatic execution of Camel K quick start tutorial commands.

The following VS Code development tooling extensions are available:

- VS Code Extension Pack for Apache Camel by Red Hat
  - Tooling for Apache Camel K extension
  - Language Support for Apache Camel extension
  - Additional extensions for OpenShift, Java, XML, and more
- Didact Tutorial Tools for VS Code extension

For details on how to set up these VS Code extensions for Camel K, see Section 3.1, “Setting up your Camel K development environment”.

Red Hat CodeReady Workspaces and Eclipse Che also provide these features using the vscode-camelk plug-in.

Additional resources

- VS Code Tooling for Apache Camel K by Red Hat extension
- VS Code tooling for Apache Camel K example
- Eclipse Che tooling for Apache Camel K
- Red Hat CodeReady WorkSpaces cloud tooling

1.5. CAMEL K DISTRIBUTIONS

Table 1.1. Red Hat Integration - Camel K distributions

<table>
<thead>
<tr>
<th>Distribution</th>
<th>Description</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maven repository</td>
<td>Maven artifacts for Red Hat Integration - Camel K</td>
<td>Software Downloads for Red Hat Integration</td>
</tr>
<tr>
<td>Source code</td>
<td>Source code for Red Hat Integration - Camel K</td>
<td>Software Downloads for Red Hat Integration</td>
</tr>
</tbody>
</table>
### Quickstarts

- **Quick start tutorials:**
  - Basic Java integration
  - Event streaming integration
  - Kamelet catalog
  - Knative integration
  - SaaS integration
  - Serverless API integration
  - Transformations integration

<table>
<thead>
<tr>
<th>Distribution</th>
<th>Description</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quickstarts</td>
<td>Quick start tutorials:</td>
<td><a href="https://github.com/openshift-integration">https://github.com/openshift-integration</a></td>
</tr>
</tbody>
</table>

**NOTE**

You must have a subscription for Red Hat Integration and be logged into the Red Hat Customer Portal to access the Red Hat Integration - Camel K distributions.
CHAPTER 2. INSTALLING CAMEL K

This chapter explains how to install Red Hat Integration – Camel K and OpenShift Serverless on OpenShift, and how to install the required Camel K and OpenShift client tools in your development environment.

- Section 2.1, “Installing Camel K from the OpenShift OperatorHub”
- Section 2.2, “Installing OpenShift Serverless from the OperatorHub”
- Section 2.3, “Installing the Camel K and OpenShift command line tools”

2.1. INSTALLING CAMEL K FROM THE OPENSHIFT OPERATORHUB

You can install the Red Hat Integration – Camel K Operator on your OpenShift cluster from the OperatorHub. The OperatorHub is available from the OpenShift Container Platform web console and provides an interface for cluster administrators to discover and install Operators. For more details on the OperatorHub, see the OpenShift documentation.

Prerequisites

- You must have cluster administrator access to an OpenShift 4.6 cluster

NOTE

You do not need to create a pull secret when installing Camel K from the OpenShift OperatorHub. The Camel K Operator automatically reuses the OpenShift cluster-level authentication to pull the Camel K image from registry.redhat.io.

Procedure

1. In the OpenShift Container Platform web console, log in using an account with cluster administrator privileges.

2. Create a new OpenShift project:
   a. In the left navigation menu, click Home > Project > Create Project.
   b. Enter a project name, for example, my-camel-k-project, and click Create.

3. In the left navigation menu, click Operators > OperatorHub.

4. In the Filter by keyword text box, enter Camel K to find the Red Hat Integration – Camel K Operator.

5. Read the information about the Operator, and click Install to display the Operator subscription page.

6. Select the following subscription settings:
   - Update Channel > techpreview
   - Installation Mode > A specific namespace on the cluster > my-camel-k-project
   - Approval Strategy > Automatic
NOTE
The Installation mode > All namespaces on the cluster and Approval Strategy > Manual settings are also available if required by your environment.

7. Click Install, and wait a few moments until the Camel K Operator is ready for use.

Additional resources
- Adding Operators to an OpenShift cluster

2.2. INSTALLING OPENSIFT SERVERLESS FROM THE OPERATORHUB

You can install the OpenShift Serverless Operator on your OpenShift cluster from the OperatorHub. The OperatorHub is available from the OpenShift Container Platform web console and provides an interface for cluster administrators to discover and install Operators. For more details on the OperatorHub, see the OpenShift documentation.

The OpenShift Serverless Operator supports both Knative Serving and Knative Eventing features. For more details, see Getting started with OpenShift Serverless.

Prerequisites
- You must have cluster administrator access to an OpenShift 4.6 cluster

Procedure
1. In the OpenShift Container Platform web console, log in using an account with cluster administrator privileges.

2. In the left navigation menu, click Operators > OperatorHub.

3. In the Filter by keyword text box, enter Serverless to find the OpenShift Serverless Operator.

4. Read the information about the Operator, and click Install to display the Operator subscription page.

5. Select the default subscription settings:
   - Update Channel > Select the channel that matches your OpenShift version, for example, 4.6
   - Installation Mode > All namespaces on the cluster
   - Approval Strategy > Automatic

NOTE
The Approval Strategy > Manual setting is also available if required by your environment.
6. Click **Install**, and wait a few moments until the Operator is ready for use.

7. Install the required Knative components using the steps in the OpenShift documentation:
   - Installing Knative Serving
   - Installing Knative Eventing

**Additional resources**
- Installing OpenShift Serverless in the OpenShift documentation

### 2.3. INSTALLING THE CAMEL K AND OPENSSHIFT COMMAND LINE TOOLS

Camel K and OpenShift provide command line tools to deploy and manage your integrations in the cloud. This section explains how to install the following Command Line Interface (CLI) tools:

- **kamel** - Camel K CLI
- **oc** - OpenShift Container Platform CLI
- **kn** - OpenShift Serverless CLI

These command line tools are all available on Linux, Windows, and Mac.

#### Prerequisites

- You must have access to an OpenShift cluster on which the Camel K Operator and OpenShift Serverless Operator are installed:
  - Section 2.1, "Installing Camel K from the OpenShift OperatorHub"
  - Section 2.2, “Installing OpenShift Serverless from the OperatorHub”

#### Procedure

1. In the OpenShift Container Platform web console, log in using an account with developer or administrator privileges.

2. Click the help icon in the toolbar, and select **Command Line Tools**

3. Download and extract the **oc** - OpenShift CLI archive if this tool is not already installed. For more details, see the OpenShift CLI documentation.

4. Download and extract the **kn** - OpenShift Serverless CLI archive if this tool is not already installed. For more details, see the OpenShift Serverless CLI documentation.

5. Download and extract the **kamel** - Camel K CLI archive to install.

6. Add the **kamel** binary file to your system path. For example, on Linux, you can put **kamel** in `/usr/bin`.

7. Log into your OpenShift cluster using the **oc** client tool, for example:
8. Enter the following command to verify the installation of the **kamel** client tool:

```
$ oc login --token=my-token --server=https://my-cluster.example.com:6443
$ kamel --help
```

**Additional resources**

- [OpenShift Container Platform CLI documentation](#)
- [OpenShift Serverless CLI documentation](#)
CHAPTER 3. GETTING STARTED WITH CAMEL K

This chapter explains how to set up your development environment and how to develop and deploy simple Camel K integrations written in Java, XML, and YAML. It also shows how to use the `kamel` command line to manage Camel K integrations at runtime. For example, this includes running, describing, logging, and deleting integrations,

- Section 3.1, “Setting up your Camel K development environment”
- Section 3.2, “Developing Camel K integrations in Java”
- Section 3.3, “Developing Camel K integrations in XML”
- Section 3.4, “Developing Camel K integrations in YAML”
- Section 3.5, “Running Camel K integrations”
- Section 3.6, “Running Camel K integrations in development mode”
- Section 3.7, “Running Camel K integrations using modeline”

3.1. SETTING UP YOUR CAMEL K DEVELOPMENT ENVIRONMENT

You must set up your environment with the recommended development tooling before you can automatically deploy the Camel K quick start tutorials. This section explains how to install the recommended Visual Studio (VS) Code IDE and the extensions that it provides for Camel K.

NOTE

VS Code is recommended for ease of use and the best developer experience of Camel K. This includes automatic completion of Camel DSL code and Camel K traits, and automatic execution of tutorial commands. However, you can manually enter your code and tutorial commands using your chosen IDE instead of VS Code.

Prerequisites

- You must have access to an OpenShift cluster on which the Camel K Operator and OpenShift Serverless Operator are installed:
  - Section 2.1, “Installing Camel K from the OpenShift OperatorHub”
  - Section 2.2, “Installing OpenShift Serverless from the OperatorHub”
  - Section 2.3, “Installing the Camel K and OpenShift command line tools”

Procedure

1. Install VS Code on your development platform. For example, on Red Hat Enterprise Linux:
   a. Install the required key and repository:

   ```bash
   $ sudo rpm --import https://packages.microsoft.com/keys/microsoft.asc
   $ sudo sh -c 'echo -e "[code]\nname=Visual Studio Code\nbaseurl=https://packages.microsoft.com/yumrepos/vscode\nenabled=1\ngpgcheck=1" >> /etc/yum.repos.d/microsoft.gpg'
   ```
b. Update the cache and install the VS Code package:

```
$ yum check-update
$ sudo yum install code
```

For details on installing on other platforms, see the VS Code installation documentation.

2. Enter the **code** command to launch the VS Code editor. For more details, see the VS Code command line documentation.

3. Install the VS Code Camel Extension Pack, which includes the extensions required for Camel K. For example, in VS Code:

   a. In the left navigation bar, click **Extensions**.

   b. In the search box, enter **Apache Camel**.

   c. Select the **Extension Pack for Apache Camel by Red Hat** and click **Install**.

   ![Extension Pack for Apache Camel by Red Hat](image)

   For more details, see the instructions for the **Extension Pack for Apache Camel by Red Hat**.

4. Install the VS Code Didact extension, which you can use to automatically run quick start tutorial commands by clicking links in the tutorial. For example, in VS Code:

   a. In the left navigation bar, click **Extensions**.

   b. In the search box, enter **Didact**.

   c. Select the extension, and click **Install**.

   For more details, see the instructions for the **Didact extension**.

Additional resources

- VS Code Getting Started documentation
- VS Code Tooling for Apache Camel K by Red Hat extension
- VS Code Language Support for Apache Camel by Red Hat extension
3.2. DEVELOPING CAMEL K INTEGRATIONS IN JAVA

This section shows how to develop a simple Camel K integration in Java DSL. Writing an integration in Java to be deployed using Camel K is the same as defining your routing rules in Camel. However, you do not need to build and package the integration as a JAR when using Camel K.

You can use any Camel component directly in your integration routes. Camel K automatically handles the dependency management and imports all the required libraries from the Camel catalog using code inspection.

Prerequisites

- Section 3.1, “Setting up your Camel K development environment”

Procedure

1. Enter the kamel init command to generate a simple Java integration file. For example:

   ```bash
   $ kamel init HelloCamelK.java
   ```

2. Open the generated integration file in your IDE and edit as appropriate. For example, the HelloCamelK.java integration automatically includes the Camel timer and log components to help you get started:

   ```java
   // camel-k: language=java
   import org.apache.camel.builder.RouteBuilder;

   public class HelloCamelK extends RouteBuilder {
   @Override
   public void configure() throws Exception {

     // Write your routes here, for example:
     from("timer:java?period=1s")
     .routeId("java")
     .setBody()
     .simple("Hello Camel K from ${routeId}")
     .to("log:info");
   }
   }
   ```

Next steps

- Section 3.5, “Running Camel K integrations”

3.3. DEVELOPING CAMEL K INTEGRATIONS IN XML

This section explains how to develop a simple Camel K integration in classic XML DSL. Writing an integration in XML to be deployed using Camel K is the same as defining your routing rules in Camel.
You can use any Camel component directly in your integration routes. Camel K automatically handles the dependency management and imports all the required libraries from the Camel catalog using code inspection.

**Prerequisites**

- Section 3.1, "Setting up your Camel K development environment"

**Procedure**

1. Enter the **kamel init** command to generate a simple XML integration file. For example:

   ```
   $ kamel init hello-camel-k.xml
   ```

2. Open the generated integration file in your IDE and edit as appropriate. For example, the **hello-camel-k.xml** integration automatically includes the Camel timer and log components to help you get started:

   `<?xml version="1.0" encoding="UTF-8"?>

   <!-- camel-k: language=xml -->

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

   <!-- Write your routes here, for example: -->

   <route id="xml">
     <from uri="timer:xml?period=1s"/>
     <setBody>
       <simple>Hello Camel K from ${routeId}</simple>
     </setBody>
     <to uri="log:info"/>
   </route>

   </routes>`

**Next steps**

- Section 3.5, "Running Camel K integrations"

### 3.4. DEVELOPING CAMEL K INTEGRATIONS IN YAML

This section explains how to develop a simple Camel K integration in YAML DSL. Writing an integration in YAML to be deployed using Camel K is the same as defining your routing rules in Camel.

You can use any Camel component directly in your integration routes. Camel K automatically handles the dependency management and imports all the required libraries from the Camel catalog using code inspection.

**Prerequisites**

- Section 3.1, "Setting up your Camel K development environment"
**Procedure**

1. Enter the `kamel init` command to generate a simple XML integration file. For example:

   ```bash
   $ kamel init hello.camelk.yaml
   ```

2. Open the generated integration file in your IDE and edit as appropriate. For example, the `hello.camelk.yaml` integration automatically includes the Camel `timer` and `log` components to help you get started:

   ```yaml
   # camel-k: language=yaml
   # Write your routes here, for example:
   - from:
     uri: "timer:yaml"
     parameters:
       period: "1s"
     steps:
       - set-body:
         constant: "Hello Camel K from yaml"
       - to: "log:info"
   ```

   **IMPORTANT**

   Integrations written in YAML must have a file name with the pattern `*.camelk.yaml` or a first line of `# camel-k: language=yaml`.

**Additional resources**

- [Writing Apache Camel integrations in YAML](#)

### 3.5. RUNNING CAMEL K INTEGRATIONS

You can run Camel K integrations in the cloud on your OpenShift cluster from the command line using the `kamel run` command.

**Prerequisites**

- [Section 3.1, “Setting up your Camel K development environment”](#).
- You must already have a Camel integration written in Java, XML, or YAML DSL.

**Procedure**

1. Log into your OpenShift cluster using the `oc` client tool, for example:

   ```bash
   $ oc login --token=my-token --server=https://my-cluster.example.com:6443
   ```

2. Ensure that the Camel K Operator is running, for example:

   ```bash
   $ oc get pod
   NAME                               READY   STATUS    RESTARTS   AGE
   camel-k-operator-86b8d94b4-pk7d6   1/1     Running   0          6m28s
   ```
3. Enter the **kamel run** command to run your integration in the cloud on OpenShift. For example:

**Java example**

```
$ kamel run HelloCamelK.java
integration "hello-camel-k" created
```

**XML example**

```
$ kamel run hello-camel-k.xml
integration "hello-camel-k" created
```

**YAML example**

```
$ kamel run hello.camelk.yaml
integration "hello" created
```

4. Enter the **kamel get** command to check the status of the integration:

```
$ kamel get
NAME       PHASE           KIT
hello      Building Kit    kit-bq666mjej725sk8sn12g
```

When the integration runs for the first time, Camel K builds the integration kit for the container image, which downloads all the required Camel modules and adds them to the image classpath.

5. Enter **kamel get** again to verify that the integration is running:

```
$ kamel get
NAME       PHASE   KIT
hello      Running kit-bq666mjej725sk8sn12g
```

6. Enter the **kamel log** command to print the log to **stdout**:

```
$ kamel log hello
```
7. Press **Ctrl-C** to terminate logging in the terminal.

**Additional resources**

- For more details on the **kamel run** command, enter **kamel run --help**
- For faster deployment turnaround times, see Section 3.6, “Running Camel K integrations in development mode”
- For details of development tools to run integrations, see **VS Code Tooling for Apache Camel K by Red Hat**
- See also Section 5.1, “Managing Camel K integrations”

### 3.6. RUNNING CAMEL K INTEGRATIONS IN DEVELOPMENT MODE

You can run Camel K integrations in development mode on your OpenShift cluster from the command line. Using development mode, you can iterate quickly on integrations in development and get fast feedback on your code.

When you specify the **kamel run** command with the **--dev** option, this deploys the integration in the cloud immediately and shows the integration logs in the terminal. You can then change the code and see the changes automatically applied instantly to the remote integration Pod on OpenShift. The terminal automatically displays all redeployments of the remote integration in the cloud.

**NOTE**

The artifacts generated by Camel K in development mode are identical to those that you run in production. The purpose of development mode is faster development.

**Prerequisites**

- Section 3.1, “Setting up your Camel K development environment”.
- You must already have a Camel integration written in Java, XML, or YAML DSL.
Procedure

1. Log into your OpenShift cluster using the oc client tool, for example:

   $ oc login --token=my-token --server=https://my-cluster.example.com:6443

2. Ensure that the Camel K Operator is running, for example:

   $ oc get pod
   NAME                               READY   STATUS    RESTARTS   AGE
   camel-k-operator-86b8d94b4-pk7d6   1/1     Running   0          6m28s

3. Enter the kamel run command with --dev to run your integration in development mode on OpenShift in the cloud. The following shows a simple Java example:

   $ kamel run HelloCamelK.java --dev
   integration "hello-camel-k" created
   Progress: integration "hello-camel-k" in phase Initialization
   Progress: integration "hello-camel-k" in phase Building Kit
   Progress: integration "hello-camel-k" in phase Deploying
   Progress: integration "hello-camel-k" in phase Running
   IntegrationPlatformAvailable for Integration hello-camel-k: camel-k
   Integration hello-camel-k in phase Initialization
   No IntegrationKitAvailable for Integration hello-camel-k: creating a new integration kit
   Integration hello-camel-k in phase Building Kit
   IntegrationKitAvailable for Integration hello-camel-k: kit-bq8t5cudeam3u3sj13tg
   Integration hello-camel-k in phase Deploying
   No CronJobAvailable for Integration hello-camel-k: different controller strategy used
   (deployment)
   DeploymentAvailable for Integration hello-camel-k: deployment name is hello-camel-k
   No ServiceAvailable for Integration hello-camel-k: no http service required
   No ExposureAvailable for Integration hello-camel-k: no target service found
   Integration hello-camel-k in phase Running
   ConfigureRoutes
4. Edit the content of your integration DSL file, save your changes, and see the changes displayed instantly in the terminal. For example:

```
... integration "hello-camel-k" updated ...
```

5. Press **Ctrl-C** to terminate logging in the terminal.

Additional resources

- For more details on the `kamel run` command, enter **kamel run --help**
- For details of development tools to run integrations, see [VS Code Tooling for Apache Camel K by Red Hat](#)
- Section 5.1, "Managing Camel K integrations"
- Section 7.6, "Configuring Camel K integration dependencies"

### 3.7. RUNNING CAMEL K INTEGRATIONS USING MODELINE
You can use the Camel K modeline to specify multiple configuration options in a Camel K integration source file, which are executed at runtime. This creates efficiencies by saving you the time of re-entering multiple command line options and helps to prevent input errors.

The following example shows a modeline entry from a Java integration file that configures traits for Prometheus monitoring and 3scale API Management, and includes a dependency on an external Maven library:

```
// camel-k: language=java trait=prometheus.enabled=true trait=3scale.enabled=true
dependency=mvn:org.my/application:1.0
```

Prerequisites

- Section 3.1, “Setting up your Camel K development environment”.
- You must already have a Camel integration written in Java, XML, or YAML DSL.

Procedure

1. Add a Camel K modeline entry to your integration file. For example:

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

public class Hello extends RouteBuilder {

    @Override
    public void configure() throws Exception {

        from("timer:java?period=1000")
            .bean(org.my.BusinessLogic)
            .log("${body}");

    }
}
```

   1 The modeline entry enables monitoring in Prometheus, API management with 3scale, and specifies a dependency on an external Maven library.
   2 This bean uses a business logic class from the external Maven library configured in the modeline.

2. Enter the following command to run the integration:

```bash
$ kamel run Hello.java
Modeline options have been loaded from source files
Full command: kamel run Hello.java --trait=prometheus.enabled=true --dependency mvn:org.my/application:1.0
```

The `kamel run` command outputs any modeline options specified in the integration.
Additional resources

- Section 9.2, “Camel K modeline options”

- For details of development tools to run modeline integrations, see Introducing IDE support for Apache Camel K Modeline.
CHAPTER 4. CAMEL K QUICK START DEVELOPER TUTORIALS

Red Hat Integration – Camel K provides quick start developer tutorials based on integration use cases available from https://github.com/openshift-integration. This chapter provides details on how to set up and deploy the following tutorials:

- Section 4.1, “Deploying a basic Camel K Java integration”
- Section 4.2, “Deploying a Camel K Serverless integration with Knative”
- Section 4.3, “Deploying a Camel K transformations integration”
- Section 4.4, “Deploying a Camel K Serverless event streaming integration”
- Section 4.5, “Deploying a Camel K Serverless API-based integration”
- Section 4.6, “Deploying a Camel K SaaS integration”

4.1. DEPLOYING A BASIC CAMEL K JAVA INTEGRATION

This tutorial demonstrates how to run a simple Java integration in the cloud on OpenShift, apply configuration and routing to an integration, and run an integration as a Kubernetes CronJob.

Prerequisites

- See the tutorial readme in GitHub: https://github.com/openshift-integration/camel-k-example-basic.
- You must have cluster administrator access to an OpenShift cluster to install Camel K. See Section 2.1, “Installing Camel K from the OpenShift OperatorHub”.
- You must have the kamel command installed. See Section 2.3, “Installing the Camel K and OpenShift command line tools”.
- Visual Studio (VS) Code is optional but recommended for the best developer experience. See Section 3.1, “Setting up your Camel K development environment”.

Procedure

1. Clone the tutorial Git repository:

   ```
   $ git clone git@github.com:openshift-integration/camel-k-example-basic.git
   ```

2. In VS Code, select File → Open Folder → camel-k-example-basic.

3. In the VS Code navigation tree, right-click the readme.didact.md file and select Didact: Start Didact Tutorial from File. For example:
This opens a new Didact tab in VS Code to display the tutorial instructions.

4. Follow the tutorial instructions and click the provided links to run the commands automatically. Alternatively, if you do not have VS Code installed with the Didact extension, you can manually enter the commands from https://github.com/openshift-integration/camel-k-example-basic.

Additional resources

- Section 3.2, “Developing Camel K integrations in Java”

4.2. DEPLOYING A CAMEL K SERVERLESS INTEGRATION WITH KNATIVE

This tutorial demonstrates how to deploy Camel K integrations with OpenShift Serverless in an event-driven architecture. This tutorial uses a Knative Eventing broker to communicate using an event publish-subscribe pattern in a Bitcoin trading demonstration.
This tutorial also shows how to use Camel K integrations to connect to a Knative event mesh with multiple external systems. The Camel K integrations also use Knative Serving to automatically scale up and down to zero as needed.

**Prerequisites**

- See the tutorial readme in GitHub: [https://github.com/openshift-integration/camel-k-example-knative](https://github.com/openshift-integration/camel-k-example-knative).
- You must have cluster administrator access to an OpenShift cluster to install Camel K and OpenShift Serverless:
  - [Section 2.1, “Installing Camel K from the OpenShift OperatorHub”](#)
  - [Section 2.2, “Installing OpenShift Serverless from the OperatorHub”](#)
- You must have the `kamel` command installed. See [Section 2.3, “Installing the Camel K and OpenShift command line tools”](#).
- Visual Studio (VS) Code is optional but recommended for the best developer experience. See [Section 3.1, “Setting up your Camel K development environment”](#).

**Procedure**

1. Clone the tutorial Git repository:
   ```
   $ git clone git@github.com:openshift-integration/camel-k-example-knative.git
   ```
2. In VS Code, select **File → Open Folder → camel-k-example-knative**.
3. In the VS Code navigation tree, right-click the `readme.didact.md` file and select **Didact: Start Didact Tutorial from File**. This opens a new Didact tab in VS Code to display the tutorial instructions.
4. Follow the tutorial instructions and click the provided links to run the commands automatically. Alternatively, if you do not have VS Code installed with the Didact extension, you can manually enter the commands from [https://github.com/openshift-integration/camel-k-example-knative](https://github.com/openshift-integration/camel-k-example-knative).

**Additional resources**

- [How Knative Eventing works](#)
- [How Knative Serving works](#)

### 4.3. DEPLOYING A CAMEL K TRANSFORMATIONS INTEGRATION

This tutorial demonstrates how to run a Camel K Java integration on OpenShift that transforms data such as XML to JSON, and stores it in a database such as PostgreSQL.

The tutorial example uses a CSV file to query an XML API and uses the data collected to build a valid GeoJSON file, which is stored in a PostgreSQL database.

**Prerequisites**
• See the tutorial readme in GitHub: https://github.com/openshift-integration/camel-k-example-transformations.

• You must have cluster administrator access to an OpenShift cluster to install Camel K. See Section 2.1, “Installing Camel K from the OpenShift OperatorHub”.

• You must have the kamel command installed. See Section 2.3, “Installing the Camel K and OpenShift command line tools”.

• You must follow the instructions in the tutorial readme to install the PostgreSQL Operator by Dev4Ddevs.com, which is required on your OpenShift cluster.

• Visual Studio (VS) Code is optional but recommended for the best developer experience. See Section 3.1, “Setting up your Camel K development environment”.

Procedure

1. Clone the tutorial Git repository:

   $ git clone git@github.com:openshift-integration/camel-k-example-transformations.git

2. In VS Code, select File → Open Folder → camel-k-example-transformations.

3. In the VS Code navigation tree, right-click the readme.didact.md file and select Didact: Start Didact Tutorial from File. This opens a new Didact tab in VS Code to display the tutorial instructions.

4. Follow the tutorial instructions and click the provided links to run the commands automatically. Alternatively, if you do not have VS Code installed with the Didact extension, you can manually enter the commands from https://github.com/openshift-integration/camel-k-example-transformations.

Additional resources

• https://operatorhub.io/operator/postgresql-operator-dev4devs-com

• https://geojson.org/

4.4. DEPLOYING A CAMEL K SERVERLESS EVENT STREAMING INTEGRATION

This tutorial demonstrates using Camel K and OpenShift Serverless with Knative Eventing for an event-driven architecture.

The tutorial shows how to install Camel K and Serverless with Knative in an AMQ Streams cluster with an AMQ Broker cluster, and how to deploy an event streaming project to run a global hazard alert demonstration application.

Prerequisites

• See the tutorial readme in GitHub: https://github.com/openshift-integration/camel-k-example-event-streaming.

• You must have cluster administrator access to an OpenShift cluster to install Camel K and OpenShift Serverless:
You must have the `kamel` command installed. See Section 2.3, “Installing the Camel K and OpenShift command line tools”.

- You must follow the instructions in the tutorial readme to install the additional required Operators on your OpenShift cluster:
  - AMQ Streams Operator
  - AMQ Broker Operator
- Visual Studio (VS) Code is optional but recommended for the best developer experience. See Section 3.1, “Setting up your Camel K development environment”.

Procedure

1. Clone the tutorial Git repository:

   ```
   $ git clone git@github.com:openshift-integration/camel-k-example-event-streaming.git
   ```

2. In VS Code, select File → Open Folder → camel-k-example-event-streaming.

3. In the VS Code navigation tree, right-click the `readme.didact.md` file and select Didact: Start Didact Tutorial from File. This opens a new Didact tab in VS Code to display the tutorial instructions.

4. Follow the tutorial instructions and click the provided links to run the commands automatically. Alternatively, if you do not have VS Code installed with the Didact extension, you can manually enter the commands from https://github.com/openshift-integration/camel-k-example-event-streaming.

Additional resources

- Red Hat AMQ documentation
- OpenShift Serverless documentation

4.5. DEPLOYING A CAMEL K SERVERLESS API-BASED INTEGRATION

This tutorial demonstrates using Camel K and OpenShift Serverless with Knative Serving for an API-based integration, and managing an API with 3scale API Management on OpenShift.

The tutorial shows how to configure Amazon S3-based storage, design an OpenAPI definition, and run an integration that calls the demonstration API endpoints.

Prerequisites

- See the tutorial readme in GitHub: https://github.com/openshift-integration/camel-k-example-api.
- You must have cluster administrator access to an OpenShift cluster to install Camel K and OpenShift Serverless:
4.6. DEPLOYING A CAMEL K SAAS INTEGRATION

This tutorial demonstrates how to run a Camel K Java integration on OpenShift that connects two widely-used Software as a Service (SaaS) providers.

The tutorial example shows how to integrate the Salesforce and ServiceNow SaaS providers using REST-based Camel components. In this simple example, each new Salesforce Case is copied to a corresponding ServiceNow Incident that includes the Salesforce Case Number.

Prerequisites

- See the tutorial readme in GitHub: https://github.com/openshift-integration/camel-k-example-saas.
- You must have cluster administrator access to an OpenShift cluster to install Camel K. See Section 2.1, “Installing Camel K from the OpenShift OperatorHub”.
- You must have the kamel command installed. See Section 2.3, “Installing the Camel K and OpenShift command line tools”.

Procedure

1. Clone the tutorial Git repository:

   $ git clone git@github.com:openshift-integration/camel-k-example-api.git

2. In VS Code, select File → Open Folder → camel-k-example-api.

3. In the VS Code navigation tree, right-click the readme.didact.md file and select Didact: Start Didact Tutorial from File. This opens a new Didact tab in VS Code to display the tutorial instructions.

4. Follow the tutorial instructions and click the provided links to run the commands automatically. Alternatively, if you do not have VS Code installed with the Didact extension, you can manually enter the commands from https://github.com/openshift-integration/camel-k-example-api.

Additional resources

- Red Hat 3scale API Management documentation
- OpenShift Serverless documentation
You must have Salesforce login credentials and ServiceNow login credentials.

Visual Studio (VS) Code is optional but recommended for the best developer experience. See Section 3.1, “Setting up your Camel K development environment”.

Procedure

1. Clone the tutorial Git repository:

   $ git clone git@github.com:openshift-integration/camel-k-example-saas.git

2. In VS Code, select File → Open Folder → camel-k-example-saas.

3. In the VS Code navigation tree, right-click the readme.didact.md file and select Didact: Start Didact Tutorial from File. This opens a new Didact tab in VS Code to display the tutorial instructions.

4. Follow the tutorial instructions and click the provided links to run the commands automatically. Alternatively, if you do not have VS Code installed with the Didact extension, you can manually enter the commands from https://github.com/openshift-integration/camel-k-example-saas.

Additional resources

- https://www.salesforce.com/
- https://www.servicenow.com/
CHAPTER 5. MANAGING CAMEL K INTEGRATIONS

You can manage Red Hat Integration - Camel K integrations using the Camel K command line or using development tools. This chapter explains how to manage Camel K integrations on the command line and provides links to additional resources that explain how to use the VS Code development tools.

- Section 5.1, “Managing Camel K integrations”
- Section 5.2, “Managing Camel K integration logging levels”

5.1. MANAGING CAMEL K INTEGRATIONS

Camel K provides different options for managing Camel K integrations on your OpenShift cluster on the command line. This section shows simple examples of using the following commands:

- `kamel get`
- `kamel describe`
- `kamel log`
- `kamel delete`

Prerequisites

- Section 3.1, “Setting up your Camel K development environment”
- You must already have a Camel integration written in Java, XML, or YAML DSL

Procedure

1. Ensure that the Camel K Operator is running on your OpenShift cluster, for example:

   ```
   $ oc get pod
   NAME                    READY   STATUS    RESTARTS   AGE
   camel-k-operator-86b8d94b4-pk7d6   1/1     Running   0          6m28s
   ```

2. Enter the `kamel run` command to run your integration in the cloud on OpenShift. For example:

   ```
   $ kamel run hello.camelk.yaml
   integration "hello" created
   ```

3. Enter the `kamel get` command to check the status of the integration:

   ```
   $ kamel get
   NAME PHASE        KIT
   hello Building Kit kit-bqatqib5t4kse5vukt40
   ```

4. Enter the `kamel describe` command print detailed information about the integration:

   ```
   $ kamel describe integration hello
   kamel describe integration hello
   Name: hello
   Namespace: camel-k-test
   ```
5. Enter the `kamel log` command to print the log to `stdout`:

```
$ kamel log hello
...
...
```

6. Press `Ctrl-C` to terminate logging in the terminal.

7. Enter the `kamel delete` to delete the integration deployed on OpenShift:

```
$ kamel delete hello
Integration hello deleted
```

**Additional resources**

- For more details on logging, see Section 5.2, “Managing Camel K integration logging levels”
- For faster deployment turnaround times, see Section 3.6, “Running Camel K integrations in development mode”
- For details of development tools to manage integrations, see VS Code Tooling for Apache Camel K by Red Hat
5.2. MANAGING CAMEL K INTEGRATION LOGGING LEVELS

Camel K uses Apache Log4j 2 as the logging framework for integrations. You can configure the logging levels of various loggers on the command line at runtime by specifying the `logging.level` prefix as an integration property. For example:

```
--property logging.level.org.apache.camel=DEBUG
```

Prerequisites

- Section 3.1, "Setting up your Camel K development environment"

Procedure

1. Enter the `kamel run` command and specify the logging level using the `--property` option. For example:

   ```
   $ kamel run --property logging.level.org.apache.camel=DEBUG HelloCamelK.java --dev
   ...
   [1] 2020-04-13 17:02:18.563 DEBUG [main] BaseMainSupport - Properties from Camel properties component:
   ...
   
   2. Press Ctrl-C to terminate logging in the terminal.

Additional resources

- For more details on the logging framework, see the [Apache Log4j 2 documentation](https://logging.apache.org/log4j/2.x/)
• For details of development tools to view logging, see VS Code Tooling for Apache Camel K by Red Hat
CHAPTER 6. MONITORING CAMEL K INTEGRATIONS

Red Hat Integration - Camel K monitoring is based on the Prometheus monitoring system: https://prometheus.io/. This chapter explains how to use the available options for monitoring Red Hat Integration - Camel K integrations at runtime. You can use the Prometheus Operator that is already deployed as part of OpenShift Monitoring to monitor your own applications.

- Section 6.1, “Enabling user workload monitoring in OpenShift”
- Section 6.2, “Configuring Camel K integration metrics”
- Section 6.3, “Adding custom Camel K integration metrics”

6.1. ENABLING USER WORKLOAD MONITORING IN OPENSSHIFT

OpenShift 4.3 or higher includes an embedded Prometheus Operator already deployed as part of OpenShift Monitoring. This section explains how to enable monitoring of your own application services in OpenShift Monitoring. This option avoids the additional overhead of installing and managing a separate Prometheus instance.

IMPORTANT
Monitoring of Camel K integrations using a separate Prometheus Operator is not included in the Technology Preview.

Prerequisites
- You must have cluster administrator access to an OpenShift cluster on which the Camel K Operator is installed. See Section 2.1, “Installing Camel K from the OpenShift OperatorHub”.

Procedure

1. Enter the following command to check if the `cluster-monitoring-config` ConfigMap object exists in the `openshift-monitoring` project:

   ```shell
   $ oc -n openshift-monitoring get configmap cluster-monitoring-config
   ```

2. Create the `cluster-monitoring-config` ConfigMap if this does not already exist:

   ```shell
   $ oc -n openshift-monitoring create configmap cluster-monitoring-config
   ```

3. Edit the `cluster-monitoring-config` ConfigMap:

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

4. Under `data:config.yaml`, set `enableUserWorkload` to `true`:

   ```yaml
   apiVersion: v1
   kind: ConfigMap
   metadata:
     name: cluster-monitoring-config
   namespace: openshift-monitoring
   ```
6.2. CONFIGURING CAMEL K INTEGRATION METRICS

You can configure monitoring of Camel K integrations automatically using the Camel K Prometheus trait at runtime. This automates the configuration of dependencies and integration Pods to expose a metrics endpoint, which is then discovered and displayed by Prometheus. The Camel Quarkus MicroProfile Metrics extension automatically collects and exposes the default Camel K metrics in the OpenMetrics format.

Prerequisites

- You must have already enabled monitoring of your own services in OpenShift. See Section 6.1, “Enabling user workload monitoring in OpenShift”.

Procedure

1. Enter the following command to run your Camel K integration with the Prometheus trait enabled:

   ```
   $ kamel run myIntegration.java -t prometheus.enabled=true
   ```

   Alternatively, you can enable the Prometheus trait globally once, by updating the integration platform as follows:

   ```
   $ oc patch ip camel-k --type=merge -p '{"spec":{"traits":{"prometheus":{"configuration":{"enabled":"true"}}}}}'
   ```

2. View monitoring of Camel K integration metrics in Prometheus. For example, for embedded Prometheus, select Monitoring > Metrics in the OpenShift administrator or developer web console.

3. Enter the Camel K metric that you want to view. For example, in the Administrator console, under Insert Metric at Cursor, enter `application_camel_context_uptime_seconds`, and click Run Queries.

4. Click Add Query to view additional metrics.

Additional resources

- Section 8.2.10, “Prometheus Trait”
- Camel Quarkus MicroProfile Metrics

6.3. ADDING CUSTOM CAMEL K INTEGRATION METRICS

You can add custom metrics to your Camel K integrations by using Camel MicroProfile Metrics:

```
config.yaml:
  enableUserWorkload: true
```

```bash
$ kamel run myIntegration.java -t prometheus.enabled=true
$ oc patch ip camel-k --type=merge -p '{"spec":{"traits":{"prometheus":{"configuration":{"enabled":"true"}}}}}'
```
You can add custom metrics to your Camel K integrations by using Camel MicroProfile Metrics component and annotations in your Java code. These custom metrics will then be automatically discovered and displayed by Prometheus.

This section shows examples of adding Camel MicroProfile Metrics annotations to Camel K integration and service implementation code.

Prerequisites

- You must have already enabled monitoring of your own services in OpenShift. See Section 6.1, “Enabling user workload monitoring in OpenShift”.

Procedure

1. Register the custom metrics in your Camel integration code using Camel MicroProfile Metrics component annotations. The following example shows a `Metrics.java` integration:

```
// camel-k: language=java trait=prometheus.enabled=true dependency=mvn:org.my/app:1.0

import org.apache.camel.Exchange;
import org.apache.camel.LoggingLevel;
import org.apache.camel.builder.RouteBuilder;
import org.apache.camel.component.microprofile.metrics.MicroProfileMetricsConstants;
import javax.enterprise.context.ApplicationScoped;

@SpringBootApplication
public class Metrics extends RouteBuilder {

    @Override
    public void configure() {
        onException()
            .handled(true)
            .maximumRedeliveries(2)
            .logStackTrace(false)
            .logExhausted(false)
            .log(LoggingLevel.ERROR, "Failed processing ${body}")
            // Register the 'redelivery' meter
            .to("microprofile-metrics:meter:redelivery?mark=2")
            // Register the 'error' meter
            .to("microprofile-metrics:meter:error");

        from("timer:stream?period=1000")
            .routeId("unreliable-service")
            .setBody(header(Exchange.TIMER_COUNTER).prepend("event ")
            .log(LoggingLevel.ERROR, "Failed processing ${body}"))
            // Register the 'generated' meter
            .to("microprofile-metrics:meter:generated")
            // Register the 'attempt' meter via @Metered in Service.java
            .bean("service")
            .filter(header(Exchange.REDELIVERED))
            .log(LoggingLevel.WARN, "Processed ${body} after
            ${header.CamelRedeliveryCounter} retries")
            .setHeader(MicroProfileMetricsConstants.HEADER_METER_MARK,
```

---

Red Hat Integration 2020-Q4 Deploying Camel K integrations on OpenShift

38
1. Uses the Camel K modeline to automatically configure the Prometheus trait and Maven dependencies

2. Error: Metric for the number of errors corresponding to the number of events that have not been processed

3. Generated: Metric for the number of events to be processed

4. Attempt: Metric for the number of calls made to the service bean to process incoming events

5. Redelivery: Metric for the number of retries made to process the event

6. Success: Metric for the number of events successfully processed

2. Add Camel MicroProfile Metrics annotations to any implementation files as needed. The following example shows the service bean called by the Camel K integration, which generates random failures:

```java
package com.redhat.integration;

import java.util.Random;
import org.apache.camel.Exchange;
import org.apache.camel.RuntimeExchangeException;
import org.eclipse.microprofile.metrics.Meter;
import org.eclipse.microprofile.metrics.annotation.Metered;
import org.eclipse.microprofile.metrics.annotation.Metric;
import javax.inject.Named;
import javax.enterprise.context.ApplicationScoped;

@Named("service")
@ApplicationScoped
@io.quarkus.arc.Unremovable
public class Service {

    // Register the attempt meter
    @Metered(absolute = true)
    public void attempt(Exchange exchange) {
        Random rand = new Random();
        if (rand.nextDouble() < 0.5) {
            throw new RuntimeExchangeException("Random failure", exchange);
        }
    }

    // Register the 'redelivery' meter
    .to("microprofile-metrics: meter: redelivery")
    // Register the 'success' meter
    .to("microprofile-metrics: meter: success");
}
```

CHAPTER 6. MONITORING CAMEL K INTEGRATIONS

39
The @Metered MicroProfile Metrics annotation declares the meter and the name is automatically generated based on the metrics method name, in this case, attempt.

This example fails randomly to help generate errors for metrics.

3. Follow the steps in Section 6.2, “Configuring Camel K integration metrics” to run the integration and view the custom Camel K metrics in Prometheus. In this case, the example already uses the Camel K modeline in Metrics.java to automatically configure Prometheus and the required Maven dependencies for Service.java.

Additional resources

- Camel MicroProfile Metrics component
- Camel Quarkus MicroProfile Metrics Extension
CHAPTER 7. CONFIGURING CAMEL K INTEGRATIONS

This chapter explains available options for configuring Red Hat Integration - Camel K integrations using properties:

- Section 7.1, “Configuring Camel K integrations using properties”
- Section 7.2, “Configuring Camel K integrations using property files”
- Section 7.3, “Configuring Camel K properties using an OpenShift ConfigMap”
- Section 7.4, “Configuring Camel K properties using an OpenShift Secret”
- Section 7.5, “Configuring Camel integration components”
- Section 7.6, “Configuring Camel K integration dependencies”

7.1. CONFIGURING CAMEL K INTEGRATIONS USING PROPERTIES

You can configure properties for Camel K integrations on the command line at runtime. When you define a property in an integration using a property placeholder, for example, {{my.message}}, you can specify the property value on the command line, for example --property my.message=Hello. You can specify multiple properties in a single command.

Prerequisites

- Section 3.1, “Setting up your Camel K development environment”

Procedure

1. Develop a Camel integration that uses a property. The following simple route includes a {{my.message}} property placeholder:

```java
... from("timer:java?period=1s") .routeId("java") .setBody() .simple("{{my.message}} from ${routeId}") .to("log:info"); ...
```

2. Enter the kamel run command using the --property option to set the property value at runtime. For example:

```
$ kamel run --property my.message="Hola Mundo" HelloCamelK.java --dev
```

Additional resources
7.2. CONFIGURING CAMEL K INTEGRATIONS USING PROPERTY FILES

You can configure multiple properties for Camel K integrations by specifying a property file on the command line at runtime. When you define properties in an integration using property placeholders, for example, `{{my.items}}`, you can specify the property values on the command line using a properties file, for example `--property-file my-integration.properties`.

**Prerequisites**

- Section 3.1, “Setting up your Camel K development environment”

**Procedure**

1. Define your integration properties file. The following shows a simple example from a `routing.properties` file:

   ```
   # List of items for random generation
   items=*radiator *engine *door window
   
   # Marker to identify priority items
   priority-marker=*
   ```

2. Develop a Camel integration that uses properties defined in the properties file. The following example from the `Routing.java` integration uses the `{{items}}` and `{{priority-marker}}` property placeholders:

   ```java
   from("timer:java?period=6000")
     .id("generator")
     .bean(this, "generateRandomItem({{items}})")
     .choice()
       .when().simple("${body.startsWith('{{priority-marker}}')}")
       .transform().body(String.class, item -> item.substring(priorityMarker.length()))
       .to("direct:priorityQueue")
     .otherwise()
       .to("direct:standardQueue");
   ```

3. Enter the `kamel run` command with the `--property-file` option. For example:

   ```bash
   $ kamel run Routing.java --property-file routing.properties --dev
   ...
   ```
7.3. CONFIGURING CAMEL K PROPERTIES USING AN OPENSIFT CONFIGMAP

You can configure multiple properties for Camel K integrations using an OpenShift ConfigMap. When you define properties in an integration using property placeholders, for example, `{{my.message}}`, you can specify the property values at runtime using a ConfigMap. You can also specify additional properties such as logging levels in the `application.properties` section of the ConfigMap.

**Prerequisites**

- Section 3.1, “Setting up your Camel K development environment”

**Procedure**

1. Develop a Camel integration that uses properties. The following simple route includes the `{{my.message}}` property placeholder:

   ```java
   from("timer:java?period=1s")
   .routeId("java")
   .setBody()
     .simple("{{my.message}} from ${routeId}")
   .to("log:info");
   ...
   ```

2. Define a ConfigMap that contains your configuration properties. For example:

   ```yaml
   apiVersion: v1
   kind: ConfigMap
   metadata:
     name: my-configmap
   data:
     application.properties:
       my.message=Bonjour le monde
       logging.level.org.apache.camel=DEBUG
   ```

   This example sets the value of the `my.message` property and sets the logging level for the `org.apache.camel` package in the `application.properties`.

3. Create the ConfigMap in the same OpenShift namespace as your integration:

   ```bash
   $ oc apply -f my-configmap.yaml
   configmap/my-configmap created
   ```
4. Run the integration with the `--configmap` option to specify the configuration properties in the ConfigMap:

```
$ kamel run --configmap=my-configmap HelloCamelK.java --dev
```

Additional resources

- **Section 7.4, “Configuring Camel K properties using an OpenShift Secret”**

### 7.4. CONFIGURING CAMEL K PROPERTIES USING AN OPENSSHIFT SECRET

You can configure multiple properties for Camel K integrations using an OpenShift Secret. When you define properties in an integration using property placeholders, for example, `{my.message}`, you can specify the property values at runtime using a Secret. You can also specify additional properties such as logging levels in the `application.properties` section of the Secret.

**NOTE**

Configuring integration properties using a Secret is similar to configuring using a ConfigMap. The main difference is that you may need to base64-encode the content of the `application.properties` in the Secret.

**Prerequisites**

- **Section 3.1, “Setting up your Camel K development environment”**

**Procedure**

1. Develop a Camel integration that uses properties. The following simple route includes the `{my.message}` property placeholder:

```
from("timer:java?period=1s")
.setBody()
   .simple("{{my.message}} from ${routeId}")
.to("log:info");
```

2. Define a Secret that contains your configuration properties. For example:
This example sets the value of the `my.message` property to `Hello World` and sets the logging level for the `org.apache.camel` package to `DEBUG`. These settings are specified in base64-encoded format in the `application.properties`.

3. Create the Secret in the same OpenShift namespace as your integration:

```bash
$ oc apply -f my-secret.yaml
secret/my-secret created
```

4. Run the integration with the `--secret` option to specify the configuration properties in the Secret:

```bash
$ kamel run --secret=my-secret HelloCamelK.java --dev
```

Additional resources

- Section 7.3, “Configuring Camel K properties using an OpenShift ConfigMap”

### 7.5. CONFIGURING CAMEL INTEGRATION COMPONENTS

You can configure Camel components programmatically in your integration code or by using configuration properties on the command line at runtime. You can configure Camel components using the following syntax:

```
camel.component.$(scheme).$(property)=$value
```

For example, to change the queue size of the Camel `seda` component for staged event-driven architecture, you can configure the following property on the command line:

```
camel.component.seda.queueSize=10
```
Procedure

- Enter the `kamel run` command and specify the Camel component configuration using the `--property` option. For example:

  ```
  $ kamel run --property camel.component.seda.queueSize=10 examples/Integration.java
  ```

Additional resources

- Section 7.1, “Configuring Camel K integrations using properties”
- Apache Camel SEDA component

### 7.6. CONFIGURING CAMEL K INTEGRATION DEPENDENCIES

Camel K automatically resolves a wide range of dependencies that are required to run your integration code. However, you can explicitly add dependencies on the command line at runtime using the `kamel run --dependency` option.

The following example integration uses Camel K automatic dependency resolution:

```java
... from("imap://admin@myserver.com") .to("seda:output") ...
```

Because this integration has an endpoint starting with the `imap:` prefix, Camel K can automatically add the `camel-mail` component to the list of required dependencies. The `seda:` endpoint belongs to `camel-core`, which is automatically added to all integrations, so Camel K does not add additional dependencies for this component.

Camel K automatic dependency resolution is transparent to the user at runtime. This is very useful in development mode because you can quickly add all the components that you need without exiting the development loop.

You can explicitly add a dependency using the `kamel run --dependency` or `-d` option. You might need to use this to specify dependencies that are not included in the Camel catalog. You can specify multiple dependencies on the command line.

Prerequisites

- Section 3.1, “Setting up your Camel K development environment”

Procedure

- Enter the `kamel run` command and specify dependencies using the `-d` option. For example:

  ```
  $ kamel run -d mvn:com.google.guava:guava:26.0-jre -d camel-mina2 Integration.java
  ```
NOTE
You can disable automatic dependency resolution by disabling the dependencies trait: `-trait dependencies.enabled=false`. However, this is not recommended in most cases.

Additional resources

- Section 3.6, “Running Camel K integrations in development mode”
- Section 8.1, “Camel K trait and profile configuration”
- Apache Camel Mail component
- Apache Camel SEDA component
CHAPTER 8. CAMEL K TRAIT CONFIGURATION REFERENCE

This chapter provides reference information about advanced features and core capabilities that you can configure on the command line at runtime using traits. Camel K provides feature traits to configure specific features and technologies. Camel K provides platform traits to configure internal Camel K core capabilities.

IMPORTANT

The Red Hat Integration - Camel K Technology Preview includes the OpenShift and Knative profiles. The Kubernetes profile has community-only support.

This Technology Preview includes Java, XML, and YAML DSL for integrations. Other languages such as Groovy, JavaScript, and Kotlin have community-only support.

This chapter includes the following sections:

- Section 8.1, “Camel K trait and profile configuration”

Camel K feature traits

- Section 8.2.1, “3scale Trait”
- Section 8.2.2, “Affinity Trait”
- Section 8.2.3, “Cron Trait”
- Section 8.2.4, “Gc Trait”
- Section 8.2.5, “Istio Trait”
- Section 8.2.6, “Jolokia Trait”
- Section 8.2.7, “Knative Trait”
- Section 8.2.8, “Knative Service Trait”
- Section 8.2.9, “Master Trait”
- Section 8.2.10, “Prometheus Trait”
- Section 8.2.11, “Quarkus Trait”
- Section 8.2.12, “Route Trait”
- Section 8.2.13, “Service Trait”

Camel K core platform traits

- Section 8.3.1, “Builder Trait”
- Section 8.3.2, “Camel Trait”
- Section 8.3.3, “Container Trait”
- Section 8.3.4, “Dependencies Trait”
8.1. CAMEL K TRAIT AND PROFILE CONFIGURATION

This section explains the important Camel K concepts of traits and profiles, which are used to configure advanced Camel K features at runtime.

Camel K traits

Camel K traits are advanced features and core capabilities that you can configure on the command line to customize Camel K integrations. For example, this includes feature traits that configure interactions with technologies such as 3scale API Management, Quarkus, Knative, and Prometheus. Camel K also provides internal platform traits that configure important core platform capabilities such as Camel support, containers, dependency resolution, and JVM support.

Camel K profiles

Camel K profiles define the target cloud platforms on which Camel K integrations run. The Camel K Technology Preview supports the OpenShift and Knative profiles.

NOTE

When you run an integration on OpenShift, Camel K uses the Knative profile when OpenShift Serverless is installed on the cluster. Camel K uses the OpenShift profile when OpenShift Serverless is not installed.

You can also specify the profile at runtime using the kamel run --profile option.

Camel K provides useful defaults for all traits, taking into account the target profile on which the integration runs. However, advanced users can configure Camel K traits for custom behavior. Some traits only apply to specific profiles such as OpenShift or Knative. For more details, see the available profiles in each trait description.

Camel K trait configuration

Each Camel trait has a unique ID that you can use to configure the trait on the command line. For example, the following command disables creating an OpenShift Service for an integration:

$ kamel run --trait service.enabled=false my-integration.yaml

You can also use the -t option to specify traits.

Camel K trait properties
You can use the `enabled` property to enable or disable each trait. All traits have their own internal logic to determine if they need to be enabled when the user does not activate them explicitly.

**WARNING**
Disabling a platform trait may compromise the platform functionality.

Some traits have an `auto` property, which you can use to enable or disable automatic configuration of the trait based on the environment. For example, this includes traits such as 3scale, Cron, and Knative. This automatic configuration can enable or disable the trait when the `enabled` property is not explicitly set, and can change the trait configuration.

Most traits have additional properties that you can configure on the command line. For more details, see the descriptions for each trait in the sections that follow.

### 8.2. CAMEL K FEATURE TRAITS

#### 8.2.1. 3scale Trait

The 3scale trait can be used to automatically create annotations that allow 3scale to discover the generated service and make it available for API management.

The 3scale trait is disabled by default.

This trait is available in the following profiles: Kubernetes, Knative, OpenShift.

**8.2.1.1. Configuration**

Trait properties can be specified when running any integration with the CLI:

```
kamel run --trait 3scale.[key]=[value] --trait 3scale.[key2]=[value2] Integration.java
```

The following configuration options are available:

<table>
<thead>
<tr>
<th>Property</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>3scale.enabled</td>
<td>bool</td>
<td>Can be used to enable or disable a trait. All traits share this common property.</td>
</tr>
<tr>
<td>3scale.auto</td>
<td>bool</td>
<td>Enables automatic configuration of the trait.</td>
</tr>
<tr>
<td>3scale.scheme</td>
<td>string</td>
<td>The scheme to use to contact the service (default http)</td>
</tr>
<tr>
<td>3scale.path</td>
<td>string</td>
<td>The path where the API is published (default /)</td>
</tr>
<tr>
<td>3scale.port</td>
<td>int</td>
<td>The port where the service is exposed (default 80)</td>
</tr>
</tbody>
</table>
8.2.2. Affinity Trait

Allows constraining which nodes the integration pod(s) are eligible to be scheduled on, based on labels on the node, or with inter-pod affinity and anti-affinity, based on labels on pods that are already running on the nodes.

It’s disabled by default.

This trait is available in the following profiles: Kubernetes, Knative, OpenShift.

8.2.2.1. Configuration

Trait properties can be specified when running any integration with the CLI:

```
kamel run --trait affinity.[key]=[value] --trait affinity.[key2]=[value2] Integration.java
```

The following configuration options are available:

<table>
<thead>
<tr>
<th>Property</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>affinity.enabled</td>
<td>bool</td>
<td>Can be used to enable or disable a trait. All traits share this common property.</td>
</tr>
<tr>
<td>affinity.pod-affinity</td>
<td>bool</td>
<td>Always co-locates multiple replicas of the integration in the same node (default false).</td>
</tr>
<tr>
<td>affinity.pod-anti-affinity</td>
<td>bool</td>
<td>Never co-locates multiple replicas of the integration in the same node (default false).</td>
</tr>
<tr>
<td>affinity.node-affinity-labels</td>
<td>[]string</td>
<td>Defines a set of nodes the integration pod(s) are eligible to be scheduled on, based on labels on the node.</td>
</tr>
<tr>
<td>affinity.pod-affinity-labels</td>
<td>[]string</td>
<td>Defines a set of pods (namely those matching the label selector, relative to the given namespace) that the integration pod(s) should be co-located with.</td>
</tr>
<tr>
<td>affinity.pod-anti-affinity-labels</td>
<td>[]string</td>
<td>Defines a set of pods (namely those matching the label selector, relative to the given namespace) that the integration pod(s) should not be co-located with.</td>
</tr>
</tbody>
</table>

8.2.2.2. Examples
To schedule the integration pod(s) on a specific node using the built-in node label 
kubernetes.io/hostname:

$$\text{kamel run -t affinity.node-affinity-labels="}\text{kubernetes.io/hostname in(node-66-}
\text{50.hosted.k8s.tld)" ...}$$

- To schedule a single integration pod per node (using the **Exists** operator):

  $$\text{kamel run -t affinity.pod-anti-affinity-labels="}\text{camel.apache.org/integration" ...}$$

- To co-locate the integration pod(s) with other integration pod(s):

  $$\text{kamel run -t affinity.pod-affinity-labels="}\text{camel.apache.org/integration in(it1, it2)" ...}$$

The *-labels options follow the requirements from Label selectors. They can be multi-valuated, then the requirements list is ANDed, e.g., to schedule a single integration pod per node AND not co-located with the Camel K operator pod(s):

$$\text{kamel run -t affinity.pod-anti-affinity-labels="}\text{camel.apache.org/integration" -t affinity.pod-anti-}
\text{affinity-labels="}\text{camel.apache.org/component=operator" ...}$$

More information can be found in the official Kubernetes documentation about Assigning Pods to Nodes.

### 8.2.3. Cron Trait

The Cron trait can be used to customize the behaviour of periodic timer/cron based integrations.

While normally an integration requires a pod to be always up and running, some periodic tasks, such as batch jobs, require to be activated at specific hours of the day or with a periodic delay of minutes. For such tasks, the cron trait can materialize the integration as a Kubernetes CronJob instead of a standard deployment, in order to save resources when the integration does not need to be executed.

Integrations that start from the following components are evaluated by the cron trait: **timer**, **cron**, **quartz**.

The rules for using a Kubernetes CronJob are the following: - **timer**: when periods can be written as cron expressions. E.g. `timer:tick?period=60000`. - **cron, quartz**: when the cron expression does not contain seconds (or the "seconds" part is set to 0). E.g. `cron:tab?schedule=0/2$+*+*+*+*` or `quartz:trigger?cron=0+0/2+*+*+*+?`. This trait is available in the following profiles: **Kubernetes, Knative, OpenShift**.

### 8.2.3.1. Configuration

Trait properties can be specified when running any integration with the CLI:

```
$ kamel run --trait cron.[key]=[value] --trait cron.[key2]=[value2] Integration.java
```

The following configuration options are available:
<table>
<thead>
<tr>
<th>Property</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cron.enabled</td>
<td>bool</td>
<td>Can be used to enable or disable a trait. All traits share this common property.</td>
</tr>
<tr>
<td>cron.schedule</td>
<td>string</td>
<td>The CronJob schedule for the whole integration. If multiple routes are declared, they must have the same schedule for this mechanism to work correctly.</td>
</tr>
<tr>
<td>cron.components</td>
<td>string</td>
<td>A comma separated list of the Camel components that need to be customized in order for them to work when the schedule is triggered externally by Kubernetes. A specific customizer is activated for each specified component. E.g. for the timer component, the cron-timer customizer is activated (it's present in the org.apache.camel.k:camel-k-runtime-cron library). Supported components are currently: cron, timer and quartz.</td>
</tr>
<tr>
<td>cron.fallback</td>
<td>bool</td>
<td>Use the default Camel implementation of the cron endpoint (quartz) instead of trying to materialize the integration as Kubernetes CronJob.</td>
</tr>
<tr>
<td>cron.concurrency-policy</td>
<td>string</td>
<td>Specifies how to treat concurrent executions of a Job. Valid values are: - &quot;Allow&quot;: allows CronJobs to run concurrently; - &quot;Forbid&quot; (default): forbids concurrent runs, skipping next run if previous run hasn’t finished yet; - &quot;Replace&quot;: cancels currently running job and replaces it with a new one</td>
</tr>
<tr>
<td>cron.auto</td>
<td>bool</td>
<td>Automatically deploy the integration as CronJob when all routes are either starting from a periodic consumer (only cron, timer and quartz are supported) or a passive consumer (e.g. direct is a passive consumer). It’s required that all periodic consumers have the same period and it can be expressed as cron schedule (e.g. 1m can be expressed as 0/1 * * * *, while 35m or 50s cannot).</td>
</tr>
</tbody>
</table>

### 8.2.4. Gc Trait

The GC Trait garbage-collects all resources that are no longer necessary upon integration updates.

This trait is available in the following profiles: Kubernetes, Knative, OpenShift.

#### 8.2.4.1. Configuration

Trait properties can be specified when running any integration with the CLI:

```
kamel run --trait gc.[key]=[value] --trait gc.[key2]=[value2] Integration.java
```

The following configuration options are available:
### 8.2.5. Istio Trait

The Istio trait allows to configure properties related to the Istio service mesh, such as sidecar injection and outbound IP ranges.

This trait is available in the following profiles: **Kubernetes, Knative, OpenShift**.

#### 8.2.5.1. Configuration

Trait properties can be specified when running any integration with the CLI:

```bash
kamel run --trait istio.[key]=[value] --trait istio.[key2]=[value2] Integration.java
```

The following configuration options are available:

<table>
<thead>
<tr>
<th>Property</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>istio.enabled</td>
<td>bool</td>
<td>Can be used to enable or disable a trait. All traits share this common property.</td>
</tr>
<tr>
<td>istio.allow</td>
<td>string</td>
<td>Configures a (comma-separated) list of CIDR subnets that should not be intercepted by the Istio proxy (10.0.0/8, 172.16.0.0/12, 192.168.0.0/16 by default).</td>
</tr>
<tr>
<td>istio.inject</td>
<td>bool</td>
<td>Forces the value for labels <code>sidecar.istio.io/inject</code>. By default the label is set to <code>true</code> on deployment and not set on Knative Service.</td>
</tr>
</tbody>
</table>

### 8.2.6. Jolokia Trait

The Jolokia trait activates and configures the Jolokia Java agent.


This trait is available in the following profiles: **Kubernetes, Knative, OpenShift**.

#### 8.2.6.1. Configuration

Trait properties can be specified when running any integration with the CLI:

```bash
kamel run --trait jolokia.[key]=[value] --trait jolokia.[key2]=[value2] Integration.java
```

---

<table>
<thead>
<tr>
<th>Property</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>gc.enabled</td>
<td>bool</td>
<td>Can be used to enable or disable a trait. All traits share this common property.</td>
</tr>
<tr>
<td>gc.discovery-cache</td>
<td>pkg/trait. discovery CacheType</td>
<td>Discovery client cache to be used, either disabled, disk or memory (default memory)</td>
</tr>
</tbody>
</table>
The following configuration options are available:

<table>
<thead>
<tr>
<th>Property</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>jolokia.enabled</td>
<td>bool</td>
<td>Can be used to enable or disable a trait. All traits share this common property.</td>
</tr>
<tr>
<td>jolokia.ca-cert</td>
<td>string</td>
<td>The PEM encoded CA certification file path, used to verify client certificates, applicable when protocol is https and use-ssl-client-authentication is true (default /var/run/secrets/kubernetes.io/serviceaccount/service-ca.crt for OpenShift).</td>
</tr>
<tr>
<td>jolokia.client-principal</td>
<td>[]string</td>
<td>The principal(s) which must be given in a client certificate to allow access to the Jolokia endpoint, applicable when protocol is https and use-ssl-client-authentication is true (default clientPrincipal=cn=system:master-proxy, cn=hawtio-online.hawtio.svc and cn=fuse-console.fuse.svc for OpenShift).</td>
</tr>
<tr>
<td>jolokia.discovery-enabled</td>
<td>bool</td>
<td>Listen for multicast requests (default false)</td>
</tr>
<tr>
<td>jolokia.extended-client-check</td>
<td>bool</td>
<td>Mandate the client certificate contains a client flag in the extended key usage section, applicable when protocol is https and use-ssl-client-authentication is true (default true for OpenShift).</td>
</tr>
</tbody>
</table>
| jolokia.host                      | string   | The Host address to which the Jolokia agent should bind to. If "*" or "0.0.0.0" is given, the servers binds to every network interface (default ")
| jolokia.password                  | string   | The password used for authentication, applicable when the user option is set. |
| jolokia.port                      | int      | The Jolokia endpoint port (default 8778).                                    |
| jolokia.protocol                  | string   | The protocol to use, either http or https (default https for OpenShift)     |
| jolokia.user                      | string   | The user to be used for authentication                                         |
| jolokia.use-ssl-client-authentication | bool   | Whether client certificates should be used for authentication (default true for OpenShift). |
| jolokia.options                   | []string | A list of additional Jolokia options as defined in JVM agent configuration options |

8.2.7. Knative Trait
The Knative trait automatically discovers addresses of Knative resources and injects them into the running integration.

The full Knative configuration is injected in the CAMEL_KNATIVE_CONFIGURATION in JSON format. The Camel Knative component will then use the full configuration to configure the routes.

The trait is enabled by default when the Knative profile is active.

This trait is available in the following profiles: Knative.

### 8.2.7.1. Configuration

Trait properties can be specified when running any integration with the CLI:

```
  kamel run --trait knative.[key]=[value] --trait knative.[key2]=[value2] Integration.java
```

The following configuration options are available:

<table>
<thead>
<tr>
<th>Property</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>knative.enabled</td>
<td>bool</td>
<td>Can be used to enable or disable a trait. All traits share this common property.</td>
</tr>
<tr>
<td>knative.configuration</td>
<td>string</td>
<td>Can be used to inject a Knative complete configuration in JSON format.</td>
</tr>
<tr>
<td>knative.channel-sources</td>
<td>[]string</td>
<td>List of channels used as source of integration routes. Can contain simple channel names or full Camel URIs.</td>
</tr>
<tr>
<td>knative.channel-sinks</td>
<td>[]string</td>
<td>List of channels used as destination of integration routes. Can contain simple channel names or full Camel URIs.</td>
</tr>
<tr>
<td>knative.endpoint-sources</td>
<td>[]string</td>
<td>List of channels used as source of integration routes.</td>
</tr>
<tr>
<td>knative.endpoint-sinks</td>
<td>[]string</td>
<td>List of endpoints used as destination of integration routes. Can contain simple endpoint names or full Camel URIs.</td>
</tr>
<tr>
<td>knative.event-sources</td>
<td>[]string</td>
<td>List of event types that the integration will be subscribed to. Can contain simple event types or full Camel URIs (to use a specific broker different from &quot;default&quot;).</td>
</tr>
<tr>
<td>knative.event-sinks</td>
<td>[]string</td>
<td>List of event types that the integration will produce. Can contain simple event types or full Camel URIs (to use a specific broker).</td>
</tr>
<tr>
<td>knative.filter-source-channels</td>
<td>bool</td>
<td>Enables filtering on events based on the header &quot;ce-knativehistory&quot;. Since this is an experimental header that can be removed in a future version of Knative, filtering is enabled only when the integration is listening from more than 1 channel.</td>
</tr>
</tbody>
</table>
8.2.8. Knative Service Trait

The Knative Service trait allows to configure options when running the integration as Knative service instead of a standard Kubernetes Deployment.

Running integrations as Knative Services adds auto-scaling (and scaling-to-zero) features, but those features are only meaningful when the routes use a HTTP endpoint consumer.

This trait is available in the following profiles: Knative.

8.2.8.1. Configuration

Trait properties can be specified when running any integration with the CLI:

```
kamel run --trait knative-service.[key]=[value] --trait knative-service.[key2]=[value2] Integration.java
```

The following configuration options are available:

<table>
<thead>
<tr>
<th>Property</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>knative-service.enabled</td>
<td>bool</td>
<td>Can be used to enable or disable a trait. All traits share this common property.</td>
</tr>
<tr>
<td>knative-service.autoscaling-class</td>
<td>string</td>
<td>Configures the Knative autoscaling class property (e.g. to set \texttt{hpa.autoscaling.knative.dev} or \texttt{kpa.autoscaling.knative.dev} autoscaling).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Refer to the Knative documentation for more information.</td>
</tr>
<tr>
<td>knative-service.autoscaling-metric</td>
<td>string</td>
<td>Configures the Knative autoscaling metric property (e.g. to set \texttt{concurrency} based or \texttt{cpu} based autoscaling).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Refer to the Knative documentation for more information.</td>
</tr>
<tr>
<td>knative-service.autoscaling-target</td>
<td>int</td>
<td>Sets the allowed concurrency level or CPU percentage (depending on the autoscaling metric) for each Pod.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Refer to the Knative documentation for more information.</td>
</tr>
</tbody>
</table>
### knative-service.min-scale

**Type**: int  
**Description**: The minimum number of Pods that should be running at any time for the integration. It’s **zero** by default, meaning that the integration is scaled down to zero when not used for a configured amount of time.  
Refer to the Knative documentation for more information.

### knative-service.max-scale

**Type**: int  
**Description**: An upper bound for the number of Pods that can be running in parallel for the integration. Knative has its own cap value that depends on the installation.  
Refer to the Knative documentation for more information.

### knative-service.auto

**Type**: bool  
**Description**: Automatically deploy the integration as Knative service when all conditions hold:  
- Integration is using the Knative profile  
- All routes are either starting from a HTTP based consumer or a passive consumer (e.g. direct is a passive consumer)

### 8.2.9. Master Trait

The Master trait allows to configure the integration to automatically leverage Kubernetes resources for doing leader election and starting **master** routes only on certain instances.

It’s activated automatically when using the master endpoint in a route, e.g. `from("master:lockname:telegram:bots")`...

**NOTE**  
this trait adds special permissions to the integration service account in order to read/write configmaps and read pods. It’s recommended to use a different service account than "default" when running the integration.

This trait is available in the following profiles: **Kubernetes, Knative, OpenShift**.

### 8.2.9.1. Configuration

Trait properties can be specified when running any integration with the CLI:

```
  kamel run --trait master.[key]=[value] --trait master.[key2]=[value2] Integration.java
```

The following configuration options are available:

<table>
<thead>
<tr>
<th>Property</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Property</td>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------------------------</td>
<td>--------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>master.enabled</td>
<td>bool</td>
<td>Can be used to enable or disable a trait. All traits share this common property.</td>
</tr>
<tr>
<td>master.auto</td>
<td>bool</td>
<td>Enables automatic configuration of the trait.</td>
</tr>
<tr>
<td>master.include-delegate-dependencies</td>
<td>bool</td>
<td>When this flag is active, the operator analyzes the source code to add dependencies required by delegate endpoints. E.g. when using master:lockname:timer, then camel:timer is automatically added to the set of dependencies. It’s enabled by default.</td>
</tr>
<tr>
<td>master.configmap</td>
<td>string</td>
<td>Name of the configmap that will be used to store the lock. Defaults to &quot;&lt;integration-name&gt;-lock&quot;.</td>
</tr>
<tr>
<td>master.label-key</td>
<td>string</td>
<td>Label that will be used to identify all pods contending the lock. Defaults to &quot;camel.apache.org/integration&quot;.</td>
</tr>
<tr>
<td>master.label-value</td>
<td>string</td>
<td>Label value that will be used to identify all pods contending the lock. Defaults to the integration name.</td>
</tr>
</tbody>
</table>

### 8.2.10. Prometheus Trait

The Prometheus trait configures a Prometheus-compatible endpoint. This trait also exposes the integration with Service and ServiceMonitor resources, so that the endpoint can be scraped automatically, when using the Prometheus Operator.

The metrics exposed vary depending on the configured runtime. With the default Quarkus runtime, metrics are exposed using MicroProfile Metrics. While with the Java main runtime, metrics are exposed using the Prometheus JMX exporter.

**WARNING**

The creation of the ServiceMonitor resource requires the Prometheus Operator custom resource definition to be installed. You can set service-monitor to false for the Prometheus trait to work without the Prometheus Operator.

The Prometheus trait is disabled by default.

This trait is available in the following profiles: Kubernetes, Knative, OpenShift.

### 8.2.10.1. Configuration

Trait properties can be specified when running any integration with the CLI:

```
kamel run --trait prometheus.[key]=[value] --trait prometheus.[key2]=[value2] Integration.java
```
The following configuration options are available:

<table>
<thead>
<tr>
<th>Property</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>prometheus.enabled</td>
<td>bool</td>
<td>Can be used to enable or disable a trait. All traits share this common property.</td>
</tr>
<tr>
<td>prometheus.port</td>
<td>int</td>
<td>The Prometheus endpoint port (default 9779, or 8080 with Quarkus).</td>
</tr>
<tr>
<td>prometheus.service-monitor</td>
<td>bool</td>
<td>Whether a ServiceMonitor resource is created (default true).</td>
</tr>
<tr>
<td>prometheus.service-monitor-labels</td>
<td>[]string</td>
<td>The ServiceMonitor resource labels, applicable when service-monitor is true.</td>
</tr>
<tr>
<td>prometheus.configmap</td>
<td>string</td>
<td>To use a custom ConfigMap containing the Prometheus JMX exporter configuration (under the content ConfigMap key). When this property is left empty (default), Camel K generates a standard Prometheus configuration for the integration. It is not applicable when using Quarkus.</td>
</tr>
</tbody>
</table>

8.2.11. Quarkus Trait

The Quarkus trait activates the Quarkus runtime.

It’s enabled by default.

This trait is available in the following profiles: Kubernetes, Knative, OpenShift.

8.2.11.1. Configuration

Trait properties can be specified when running any integration with the CLI:

```
   kamel run --trait quarkus.[key]=[value] --trait quarkus.[key2]=[value2] Integration.java
```

The following configuration options are available:

<table>
<thead>
<tr>
<th>Property</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>quarkus.enabled</td>
<td>bool</td>
<td>Can be used to enable or disable a trait. All traits share this common property.</td>
</tr>
<tr>
<td>quarkus.native</td>
<td>bool</td>
<td>The Quarkus runtime type (reserved for future use)</td>
</tr>
</tbody>
</table>

8.2.11.2. Supported Camel Components

When running with Quarkus enabled, then Camel K only supports out of the box, those Camel components that are available as Camel Quarkus Extensions.
You can see the list of extensions from the Camel Quarkus documentation.

8.2.12. Route Trait

The Route trait can be used to configure the creation of OpenShift routes for the integration.

This trait is available in the following profiles: OpenShift.

8.2.12.1. Configuration

Trait properties can be specified when running any integration with the CLI:

```
kamel run --trait route.[key]=[value] --trait route.[key2]=[value2] Integration.java
```

The following configuration options are available:

<table>
<thead>
<tr>
<th>Property</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>route.enabled</td>
<td>bool</td>
<td>Can be used to enable or disable a trait. All traits share this common property.</td>
</tr>
<tr>
<td>route.host</td>
<td>string</td>
<td>To configure the host exposed by the route.</td>
</tr>
<tr>
<td>route.tls-termination</td>
<td>string</td>
<td>The TLS termination type, like edge, passthrough or reencrypt. Refer to the OpenShift documentation for additional information.</td>
</tr>
<tr>
<td>route.tls-certificate</td>
<td>string</td>
<td>The TLS certificate contents. Refer to the OpenShift documentation for additional information.</td>
</tr>
<tr>
<td>route.tls-key</td>
<td>string</td>
<td>The TLS certificate key contents. Refer to the OpenShift documentation for additional information.</td>
</tr>
<tr>
<td>route.tls-ca-certificate</td>
<td>string</td>
<td>The TLS cert authority certificate contents. Refer to the OpenShift documentation for additional information.</td>
</tr>
<tr>
<td>route.tls-destination-ca-certificate</td>
<td>string</td>
<td>The destination CA certificate provides the contents of the CA certificate of the final destination. When using reencrypt termination this file should be provided in order to have routers use it for health checks on the secure connection. If this field is not specified, the router may provide its own destination CA and perform hostname validation using the short service name (service.namespace.svc), which allows infrastructure generated certificates to automatically verify. Refer to the OpenShift documentation for additional information.</td>
</tr>
</tbody>
</table>
8.2.13. Service Trait

The Service trait exposes the integration with a Service resource so that it can be accessed by other applications (or integrations) in the same namespace.

It’s enabled by default if the integration depends on a Camel component that can expose a HTTP endpoint.

This trait is available in the following profiles: Kubernetes, OpenShift.

8.2.13.1. Configuration

Trait properties can be specified when running any integration with the CLI:

```
kamel run --trait.service.[key]=[value] --trait.service.[key2]=[value2] Integration.java
```

The following configuration options are available:

<table>
<thead>
<tr>
<th>Property</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>service.enabled</td>
<td>bool</td>
<td>Can be used to enable or disable a trait. All traits share this common property.</td>
</tr>
<tr>
<td>service.auto</td>
<td>bool</td>
<td>To automatically detect from the code if a Service needs to be created.</td>
</tr>
<tr>
<td>service.node-port</td>
<td>bool</td>
<td>Enable Service to be exposed as NodePort</td>
</tr>
</tbody>
</table>

8.3. CAMEL K PLATFORM TRAITS

8.3.1. Builder Trait

The builder trait is internally used to determine the best strategy to build and configure IntegrationKits.

This trait is available in the following profiles: Kubernetes, Knative, OpenShift.
WARNING
The builder trait is a platform trait: disabling it may compromise the platform functionality.

8.3.1.1. Configuration
Trait properties can be specified when running any integration with the CLI:

```
kamel run --trait builder.[key]=[value] --trait builder.[key2]=[value2] Integration.java
```

The following configuration options are available:

<table>
<thead>
<tr>
<th>Property</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>builder.enabled</td>
<td>bool</td>
<td>Can be used to enable or disable a trait. All traits share this common property.</td>
</tr>
<tr>
<td>builder.verbose</td>
<td>bool</td>
<td>Enable verbose logging on build components that support it (e.g., OpenShift build pod). Kaniko and Buildah are not supported.</td>
</tr>
</tbody>
</table>

8.3.2. Container Trait
The Container trait can be used to configure properties of the container where the integration will run.

It also provides configuration for Services associated to the container.

This trait is available in the following profiles: Kubernetes, Knative, OpenShift.

WARNING
The container trait is a platform trait: disabling it may compromise the platform functionality.

8.3.2.1. Configuration
Trait properties can be specified when running any integration with the CLI:

```
kamel run --trait container.[key]=[value] --trait container.[key2]=[value2] Integration.java
```

The following configuration options are available:
<table>
<thead>
<tr>
<th>Property</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>container.enabled</td>
<td>bool</td>
<td>Can be used to enable or disable a trait. All traits share this common property.</td>
</tr>
<tr>
<td>container.auto</td>
<td>bool</td>
<td></td>
</tr>
<tr>
<td>container.request-cpu</td>
<td>string</td>
<td>The minimum amount of CPU required.</td>
</tr>
<tr>
<td>container.request-memory</td>
<td>string</td>
<td>The minimum amount of memory required.</td>
</tr>
<tr>
<td>container.limit-cpu</td>
<td>string</td>
<td>The maximum amount of CPU required.</td>
</tr>
<tr>
<td>container.limit-memory</td>
<td>string</td>
<td>The maximum amount of memory required.</td>
</tr>
<tr>
<td>container.expose</td>
<td>bool</td>
<td>Can be used to enable/disable exposure via kubernetes Service.</td>
</tr>
<tr>
<td>container.port</td>
<td>int</td>
<td>To configure a different port exposed by the container (default 8080).</td>
</tr>
<tr>
<td>container.port-name</td>
<td>string</td>
<td>To configure a different port name for the port exposed by the container (default http).</td>
</tr>
<tr>
<td>container.service-port</td>
<td>int</td>
<td>To configure under which service port the container port is to be exposed (default 80).</td>
</tr>
<tr>
<td>container.service-port-name</td>
<td>string</td>
<td>To configure under which service port name the container port is to be exposed (default http).</td>
</tr>
<tr>
<td>container.name</td>
<td>string</td>
<td>The main container name. It's named integration by default.</td>
</tr>
<tr>
<td>container.probes-enabled</td>
<td>bool</td>
<td>ProbesEnabled enable/disable probes on the container (default false)</td>
</tr>
<tr>
<td>container.probe-path</td>
<td>string</td>
<td>Path to access on the probe ( default /health). Note that this property is not supported on quarkus runtime and setting it will result in the integration failing to start.</td>
</tr>
<tr>
<td>container.liveness-initial-delay</td>
<td>int32</td>
<td>Number of seconds after the container has started before liveness probes are initiated.</td>
</tr>
<tr>
<td>container.liveness-timeout</td>
<td>int32</td>
<td>Number of seconds after which the probe times out. Applies to the liveness probe.</td>
</tr>
<tr>
<td>container.liveness-period</td>
<td>int32</td>
<td>How often to perform the probe. Applies to the liveness probe.</td>
</tr>
</tbody>
</table>
### Property Types and Descriptions

<table>
<thead>
<tr>
<th>Property</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>container.liveness-success-threshold</td>
<td>int32</td>
<td>Minimum consecutive successes for the probe to be considered successful after having failed. Applies to the liveness probe.</td>
</tr>
<tr>
<td>container.liveness-failure-threshold</td>
<td>int32</td>
<td>Minimum consecutive failures for the probe to be considered failed after having succeeded. Applies to the liveness probe.</td>
</tr>
<tr>
<td>container.readiness-initial-delay</td>
<td>int32</td>
<td>Number of seconds after the container has started before readiness probes are initiated.</td>
</tr>
<tr>
<td>container.readiness-timeout</td>
<td>int32</td>
<td>Number of seconds after which the probe times out. Applies to the readiness probe.</td>
</tr>
<tr>
<td>container.readiness-period</td>
<td>int32</td>
<td>How often to perform the probe. Applies to the readiness probe.</td>
</tr>
<tr>
<td>container.readiness-success-threshold</td>
<td>int32</td>
<td>Minimum consecutive successes for the probe to be considered successful after having failed. Applies to the readiness probe.</td>
</tr>
<tr>
<td>container.readiness-failure-threshold</td>
<td>int32</td>
<td>Minimum consecutive failures for the probe to be considered failed after having succeeded. Applies to the readiness probe.</td>
</tr>
</tbody>
</table>

#### 8.3.3. Camel Trait

The Camel trait can be used to configure versions of Apache Camel K runtime and related libraries, it cannot be disabled.

This trait is available in the following profiles: **Kubernetes, Knative, OpenShift**.

**WARNING**

The camel trait is a **platform trait**: disabling it may compromise the platform functionality.

#### 8.3.3.1. Configuration

Trait properties can be specified when running any integration with the CLI:

```
kamel run --trait camel.[key]=[value] --trait camel.[key2]=[value2] Integration.java
```

The following configuration options are available:
### 8.3.4. Dependencies Trait

The Dependencies trait is internally used to automatically add runtime dependencies based on the integration that the user wants to run.

This trait is available in the following profiles: **Kubernetes, Knative, OpenShift**.

#### WARNING

The dependencies trait is a **platform trait**: disabling it may compromise the platform functionality.

### 8.3.4.1. Configuration

Trait properties can be specified when running any integration with the CLI:

```bash
kamel run --trait dependencies.[key]=[value] Integration.java
```

The following configuration options are available:

<table>
<thead>
<tr>
<th>Property</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dependencies.enabled</td>
<td>bool</td>
<td>Can be used to enable or disable a trait. All traits share this common property.</td>
</tr>
</tbody>
</table>

### 8.3.5. Deployer Trait

The deployer trait can be used to explicitly select the kind of high level resource that will deploy the integration.

This trait is available in the following profiles: **Kubernetes, Knative, OpenShift**.
8.3.5.1. Configuration

Trait properties can be specified when running any integration with the CLI:

```
kamel run --trait deployer.[key]=[value] --trait deployer.[key2]=[value2] Integration.java
```

The following configuration options are available:

<table>
<thead>
<tr>
<th>Property</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>deployer.enabled</td>
<td>bool</td>
<td>Can be used to enable or disable a trait. All traits share this common property.</td>
</tr>
<tr>
<td>deployer.kind</td>
<td>string</td>
<td>Allows to explicitly select the desired deployment kind between <strong>deployment</strong>, <strong>cron-job</strong> or <strong>knative-service</strong> when creating the resources for running the integration.</td>
</tr>
</tbody>
</table>

8.3.6. Deployment Trait

The Deployment trait is responsible for generating the Kubernetes deployment that will make sure the integration will run in the cluster.

This trait is available in the following profiles: **Kubernetes**, **Knative**, **OpenShift**.

8.3.6.1. Configuration

Trait properties can be specified when running any integration with the CLI:

```
kamel run --trait deployment.[key]=[value] Integration.java
```

The following configuration options are available:
### 8.3.7. Environment Trait

The environment trait is used internally to inject standard environment variables in the integration container, such as `NAMESPACE`, `POD_NAME` and others.

This trait is available in the following profiles: Kubernetes, Knative, OpenShift.

**WARNING**

The environment trait is a platform trait: disabling it may compromise the platform functionality.

#### 8.3.7.1. Configuration

Trait properties can be specified when running any integration with the CLI:

```
kamel run --trait environment.[key]=[value] --trait environment.[key2]=[value2] Integration.java
```

The following configuration options are available:

<table>
<thead>
<tr>
<th>Property</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>environment.enabled</td>
<td>bool</td>
<td>Can be used to enable or disable a trait. All traits share this common property.</td>
</tr>
<tr>
<td>environment.container-meta</td>
<td>bool</td>
<td></td>
</tr>
</tbody>
</table>

### 8.3.8. Jvm Trait

The JVM trait is used to configure the JVM that runs the integration.

This trait is available in the following profiles: Kubernetes, Knative, OpenShift.
8.3.8.1. Configuration

Trait properties can be specified when running any integration with the CLI:

```
kamel run --trait jvm.[key]=[value] --trait jvm.[key2]=[value2] Integration.java
```

The following configuration options are available:

<table>
<thead>
<tr>
<th>Property</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>jvm.enabled</td>
<td>bool</td>
<td>Can be used to enable or disable a trait. All traits share this common property.</td>
</tr>
<tr>
<td>jvm.debug</td>
<td>bool</td>
<td>Activates remote debugging, so that a debugger can be attached to the JVM, e.g., using port-forwarding</td>
</tr>
<tr>
<td>jvm.debug-suspend</td>
<td>bool</td>
<td>Suspends the target JVM immediately before the main class is loaded</td>
</tr>
<tr>
<td>jvm.print-command</td>
<td>bool</td>
<td>Prints the command used the start the JVM in the container logs (default <code>true</code>)</td>
</tr>
<tr>
<td>jvm.debug-address</td>
<td>string</td>
<td>Transport address at which to listen for the newly launched JVM (default <code>*:5005</code>)</td>
</tr>
<tr>
<td>jvm.options</td>
<td>[]string</td>
<td>A list of JVM options</td>
</tr>
</tbody>
</table>

8.3.9. Openapi Trait

The OpenAPI DSL trait is internally used to allow creating integrations from a OpenAPI specs.

This trait is available in the following profiles: **Kubernetes, Knative, OpenShift**.

```
WARNING
The openapi trait is a platform trait: disabling it may compromise the platform functionality.
```
8.3.9.1. Configuration

Trait properties can be specified when running any integration with the CLI:
```
kamel run --trait openapi.[key]=[value] Integration.java
```

The following configuration options are available:

<table>
<thead>
<tr>
<th>Property</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>openapi.enabled</td>
<td>bool</td>
<td>Can be used to enable or disable a trait. All traits share this common property.</td>
</tr>
</tbody>
</table>

8.3.10. Owner Trait

The Owner trait ensures that all created resources belong to the integration being created and transfers annotations and labels on the integration onto these owned resources.

This trait is available in the following profiles: Kubernetes, Knative, OpenShift.

**WARNING**
The owner trait is a platform trait: disabling it may compromise the platform functionality.

8.3.10.1. Configuration

Trait properties can be specified when running any integration with the CLI:
```
kamel run --trait owner.[key]=[value] --trait owner.[key2]=[value2] Integration.java
```

The following configuration options are available:

<table>
<thead>
<tr>
<th>Property</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>owner.enabled</td>
<td>bool</td>
<td>Can be used to enable or disable a trait. All traits share this common property.</td>
</tr>
<tr>
<td>owner.target-annotations</td>
<td>[]string</td>
<td>The set of annotations to be transferred</td>
</tr>
<tr>
<td>owner.target-labels</td>
<td>[]string</td>
<td>The set of labels to be transferred</td>
</tr>
</tbody>
</table>

8.3.11. Platform Trait
The platform trait is a base trait that is used to assign an integration platform to an integration.

In case the platform is missing, the trait is allowed to create a default platform. This feature is especially useful in contexts where there’s no need to provide a custom configuration for the platform (e.g. on OpenShift the default settings work, since there’s an embedded container image registry).

This trait is available in the following profiles: **Kubernetes, Knative, OpenShift**.

---

**WARNING**

The platform trait is a **platform trait**: disabling it may compromise the platform functionality.

---

### 8.3.11.1. Configuration

Trait properties can be specified when running any integration with the CLI:

```
kamel run --trait platform.[key]=[value] --trait platform.[key2]=[value2] Integration.java
```

The following configuration options are available:

<table>
<thead>
<tr>
<th>Property</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>platform.enabled</td>
<td>bool</td>
<td>Can be used to enable or disable a trait. All traits share this common property.</td>
</tr>
<tr>
<td>platform.create-default</td>
<td>bool</td>
<td>To create a default (empty) platform when the platform is missing.</td>
</tr>
<tr>
<td>platform.auto</td>
<td>bool</td>
<td>To automatically detect from the environment if a default platform can be created (it will be created on OpenShift only).</td>
</tr>
</tbody>
</table>
CHAPTER 9. CAMEL K COMMAND REFERENCE

This chapter provides reference details on the Camel K command line interface (CLI), and provides examples of using the `kamel` command. This chapter also provides reference details on Camel K modeline options that you can specify in a Camel K integration source file, which are executed at runtime.

This chapter includes the following sections:

- Section 9.1, “Camel K command line”
- Section 9.2, “Camel K modeline options”

### 9.1. CAMEL K COMMAND LINE

The Camel K CLI provides the `kamel` command as the main entry point for running Camel K integrations on OpenShift. This section provides details on the most commonly used `kamel` commands.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
</table>
| `help` | Get the full list of available commands. You can enter `--help` as a parameter to each command for more details. | `kamel help`  
 `kamel run --help` |
| `init` | Initialize an empty Camel K file implemented in Java, XML, or YAML. | `kamel init MyIntegration.java` |
| `run` | Run an integration on OpenShift. | `kamel run MyIntegration.java` |
| `debug` | Debug a remote integration using a local debugger. | `kamel debug my-integration` |
| `get` | Get integrations deployed on OpenShift. | `kamel get` |
| `describe` | Get detailed information on a Camel K resource. This includes an integration, kit, or platform. | `kamel describe integration my-integration` |
| `log` | Print the logs of a running integration. | `kamel log my-integration` |
| `delete` | Delete an integration deployed on OpenShift. | `kamel delete my-integration` |

### Additional resources

Red Hat Integration 2020-Q4 Deploying Camel K integrations on OpenShift
9.2. CAMEL K MODELINE OPTIONS

You can use the Camel K modeline to enter configuration options in a Camel K integration source file, which are executed at runtime, for example, using `kamel run MyIntegration.java`. For more details, see Section 3.7, “Running Camel K integrations using modeline”.

This section provides reference details about the most commonly used modeline options.

Table 9.2. Camel K modeline options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dependency</td>
<td>Add an external library to be included in the integration. For example, for Maven, use <code>dependency=mvn:org.my/app:1.0</code>, or for GitHub, use <code>dependency=github:my-account:camel-k-example-project:master</code>.</td>
</tr>
<tr>
<td>env</td>
<td>Set an environment variable in the integration container. For example, <code>env=MY_ENV_VAR=my-value</code>.</td>
</tr>
<tr>
<td>label</td>
<td>Add a label for the integration. For example, <code>label=my.company=hello</code>.</td>
</tr>
<tr>
<td>name</td>
<td>Add an integration name. For example, <code>name=my-integration</code>.</td>
</tr>
<tr>
<td>open-api</td>
<td>Add an OpenAPI v2 specification. For example, <code>open-api=path/to/my-hello-api.json</code>.</td>
</tr>
<tr>
<td>profile</td>
<td>Set the Camel K trait profile used for deployment. For example, <code>profile=openshift</code>.</td>
</tr>
<tr>
<td>property</td>
<td>Add a integration property. For example, <code>property=my.message=&quot;Hola Mundo&quot;</code>.</td>
</tr>
<tr>
<td>property-file</td>
<td>Bind a property file to the integration. For example, <code>property-file=my-integration.properties</code>.</td>
</tr>
<tr>
<td>resource</td>
<td>Add an external resource. For example, <code>resource=path/to/my-hello.txt</code>.</td>
</tr>
<tr>
<td>trait</td>
<td>Configure a Camel K feature or core capability in a trait. For example, <code>trait=crontab.enabled=true</code>.</td>
</tr>
</tbody>
</table>