Red Hat Integration 2020-Q4

Deploying Camel K integrations on OpenShift

TECHNOLOGY PREVIEW - Getting started with Red Hat Integration - Camel K

Last Updated: 2021-02-16

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

TECHNOLOGY PREVIEW - Getting started with Red Hat Integration - Camel K

Legal Notice

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

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

- Apache Camel K website
- Getting started with OpenShift Serverless

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

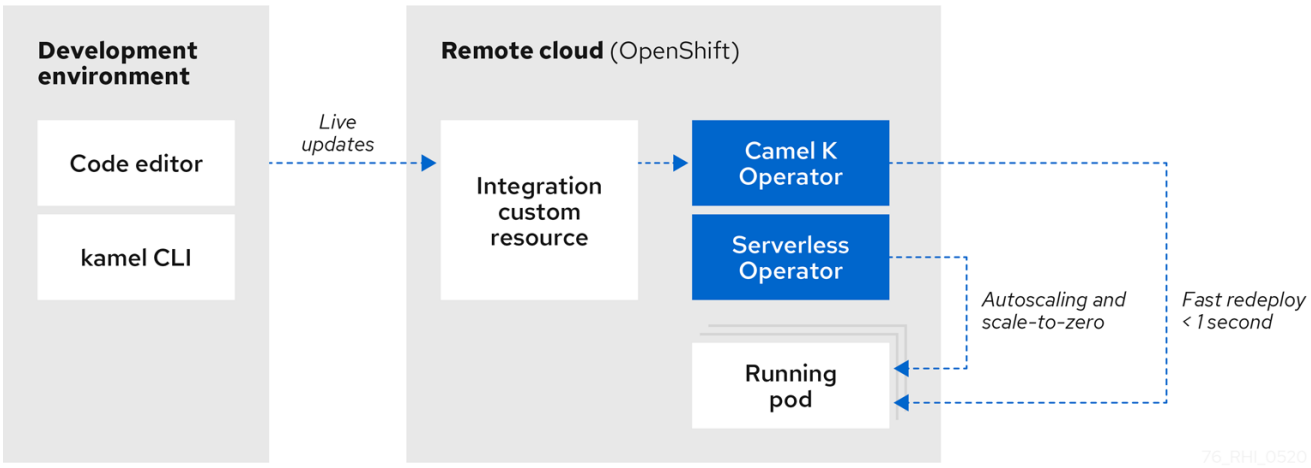
- Knative Serving for autoscaling and scale-to-zero
- Knative Eventing for event-driven architectures
- Performance optimizations using Quarkus runtime by default
- Camel integrations written in Java, XML, or YAML DSL
- Development tooling with Visual Studio Code
- Monitoring of integrations using Prometheus 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

Distribution	Description	Location
Operator image	Container image for the Red Hat Integration - Camel K Operator: **integration-tech-preview/camel-k-rhel8-operator**	• OpenShift web console under **Operators → OperatorHub** • registry.redhat.io
Maven repository	Maven artifacts for Red Hat Integration - Camel K	Software Downloads for Red Hat Integration
Source code	Source code for Red Hat Integration - Camel K	Software Downloads for Red Hat Integration

Distribution	Description	Location
Quickstarts	Quick start tutorials: • Basic Java integration • Event streaming integration • Kamelet catalog • Knative integration • SaaS integration • Serverless API integration • Transformations integration	https://github.com/openshift-integration

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.

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

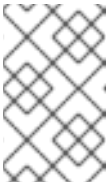
- You must have cluster administrator access to an OpenShift 4.6 cluster

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 OPENSHIFT 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.

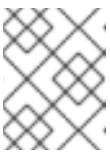
The OpenShift Serverless Operator supports both Knative Serving and Knative Eventing features. For more details, see Getting started with OpenShift Serverless.

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 OPENSHIFT 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:

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

8. Enter the following command to verify the installation of the **kamel** client tool:

```
$ 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:

```
$ 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=
```

```
\ngpgkey=https://packages.microsoft.com/keys/microsoft.asc" >
/etc/yum.repos.d/vscode.repo'
```

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.

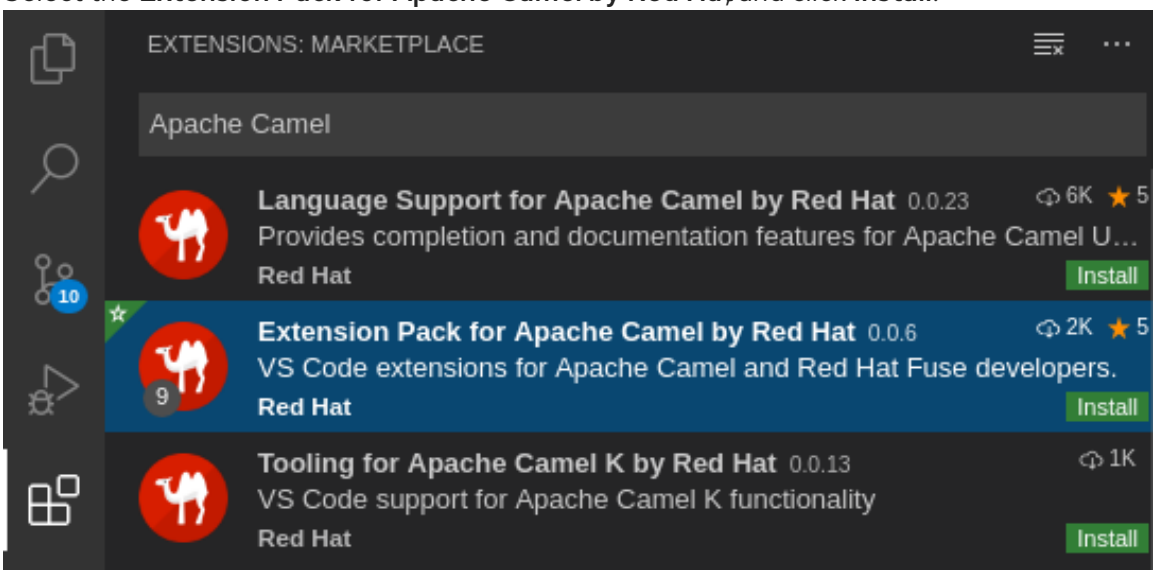
2. Enter the **code** command to launch the VS Code editor. For more details, see the VS Code command line documentation.

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**.



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

- Apache Camel K and VS Code tooling example

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:

```
$ 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:

```
// 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:

```
$ 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:

```
# 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:

```
$ 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 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
   [1] 2020-04-11 14:26:43.449 INFO  [main] ApplicationRuntime - Listener
   org.apache.camel.k.listener.PropertiesFunctionsConfigurer@5223e5ee executed in phase
   Starting
   [1] 2020-04-11 14:26:43.457 INFO  [main] RuntimeSupport - Looking up loader for language:
   yaml
   [1] 2020-04-11 14:26:43.655 INFO  [main] RuntimeSupport - Found loader
   org.apache.camel.k.loader.yaml.YamlSourceLoader@1224144a for language yaml from
   service definition
   [1] 2020-04-11 14:26:43.658 INFO  [main] RoutesConfigurer - Loading routes from:
   file:/etc/camel/sources/i-source-000/hello.camelk.yaml?language=yaml
   [1] 2020-04-11 14:26:43.658 INFO  [main] ApplicationRuntime - Listener
   org.apache.camel.k.listener.RoutesConfigurer@36c88a32 executed in phase
   ConfigureRoutes
   [1] 2020-04-11 14:26:43.661 INFO  [main] BaseMainSupport - Using properties from:
   file:/etc/camel/conf/application.properties
   [1] 2020-04-11 14:26:43.878 INFO  [main] ApplicationRuntime - Listener
   org.apache.camel.k.listener.ContextConfigurer@65466a6a executed in phase
   ConfigureContext
   ```

```
[1] 2020-04-11 14:26:43.879 INFO  [main] DefaultCamelContext - Apache Camel 3.0.1
(CamelContext: camel-k) is starting
[1] 2020-04-11 14:26:43.880 INFO  [main] DefaultManagementStrategy - JMX is disabled
[1] 2020-04-11 14:26:44.147 INFO  [main] DefaultCamelContext - StreamCaching is not in
use. If using streams then its recommended to enable stream caching. See more details at
http://camel.apache.org/stream-caching.html
[1] 2020-04-11 14:26:44.157 INFO  [main] DefaultCamelContext - Route: route1 started and
consuming from: timer://yaml?period=1s
[1] 2020-04-11 14:26:44.161 INFO  [main] DefaultCamelContext - Total 1 routes, of which 1
are started
[1] 2020-04-11 14:26:44.162 INFO  [main] DefaultCamelContext - Apache Camel 3.0.1
(CamelContext: camel-k) started in 0.283 seconds
[1] 2020-04-11 14:26:44.163 INFO  [main] ApplicationRuntime - Listener
org.apache.camel.k.listener.RoutesDumper@1c93084c executed in phase Started
[1] 2020-04-11 14:26:45.183 INFO  [Camel (camel-k) thread #1 - timer://yaml] info -
Exchange[ExchangePattern: InOnly, BodyType: String, Body: Hello Camel K from yaml]
...
```

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
[2] Monitoring pod hello-camel-k-866ccb5976-sjh8x[1] Monitoring pod hello-camel-k-
866ccb5976-l288p[2] 2020-04-11 14:44:53.691 INFO  [main] ApplicationRuntime - Add
listener: org.apache.camel.k.listener.ContextConfigurer@159f197
[2] 2020-04-11 14:44:53.694 INFO  [main] ApplicationRuntime - Add listener:
org.apache.camel.k.listener.RoutesConfigurer@1f0f1111
[2] 2020-04-11 14:44:53.695 INFO  [main] ApplicationRuntime - Add listener:
org.apache.camel.k.listener.RoutesDumper@6e0dec4a
[2] 2020-04-11 14:44:53.695 INFO  [main] ApplicationRuntime - Add listener:
org.apache.camel.k.listener.PropertiesFunctionsConfigurer@794cb805
[2] 2020-04-11 14:44:53.712 INFO  [main] ApplicationRuntime - Listener
org.apache.camel.k.listener.PropertiesFunctionsConfigurer@794cb805 executed in phase
Starting
[2] 2020-04-11 14:44:53.721 INFO  [main] RuntimeSupport - Looking up loader for language:
java
[2] 2020-04-11 14:44:53.723 INFO  [main] RuntimeSupport - Found loader
org.apache.camel.k.loader.java.JavaSourceLoader@3911c2a7 for language java from
service definition
[2] 2020-04-11 14:44:54.220 INFO  [main] RoutesConfigurer - Loading routes from:
file:/etc/camel/sources/i-source-000/HelloCamelK.java?language=java
[2] 2020-04-11 14:44:54.220 INFO  [main] ApplicationRuntime - Listener
org.apache.camel.k.listener.RoutesConfigurer@1f0f1111 executed in phase
ConfigureRoutes
```

```
[2] 2020-04-11 14:44:54.224 INFO  [main] BaseMainSupport - Using properties from:
file:/etc/camel/conf/application.properties
[2] 2020-04-11 14:44:54.385 INFO  [main] ApplicationRuntime - Listener
org.apache.camel.k.listener.ContextConfigurer@159f197 executed in phase
ConfigureContext
[2] 2020-04-11 14:44:54.386 INFO  [main] DefaultCamelContext - Apache Camel 3.0.1
(CamelContext: camel-k) is starting
[2] 2020-04-11 14:44:54.387 INFO  [main] DefaultManagementStrategy - JMX is disabled
[2] 2020-04-11 14:44:54.630 INFO  [main] DefaultCamelContext - StreamCaching is not in
use. If using streams then its recommended to enable stream caching. See more details at
http://camel.apache.org/stream-caching.html
[2] 2020-04-11 14:44:54.639 INFO  [main] DefaultCamelContext - Route: java started and
consuming from: timer://java?period=1s
[2] 2020-04-11 14:44:54.643 INFO  [main] DefaultCamelContext - Total 1 routes, of which 1
are started
[2] 2020-04-11 14:44:54.643 INFO  [main] DefaultCamelContext - Apache Camel 3.0.1
(CamelContext: camel-k) started in 0.258 seconds
[2] 2020-04-11 14:44:54.644 INFO  [main] ApplicationRuntime - Listener
org.apache.camel.k.listener.RoutesDumper@6e0dec4a executed in phase Started
[2] 2020-04-11 14:44:55.671 INFO  [Camel (camel-k) thread #1 - timer://java] info -
Exchange[ExchangePattern: InOnly, BodyType: String, Body: Hello Camel K from java]
...
```

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
...
[3] 2020-04-11 14:45:06.792 INFO  [main] DefaultCamelContext - Route: java started and
consuming from: timer://java?period=1s
[3] 2020-04-11 14:45:06.795 INFO  [main] DefaultCamelContext - Total 1 routes, of which 1
are started
[3] 2020-04-11 14:45:06.796 INFO  [main] DefaultCamelContext - Apache Camel 3.0.1
(CamelContext: camel-k) started in 0.323 seconds
[3] 2020-04-11 14:45:06.796 INFO  [main] ApplicationRuntime - Listener
org.apache.camel.k.listener.RoutesDumper@6e0dec4a executed in phase Started
[3] 2020-04-11 14:45:07.826 INFO  [Camel (camel-k) thread #1 - timer://java] info -
Exchange[ExchangePattern: InOnly, BodyType: String, Body: Ciao Camel K from java]
...
```

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/app: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:

   Hello.java

```
// camel-k: language=java trait=prometheus.enabled=true trait=3scale.enabled=true
dependency=mvn:org.my/application:1.0 (1)

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) (2)
        .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:

```
$ 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

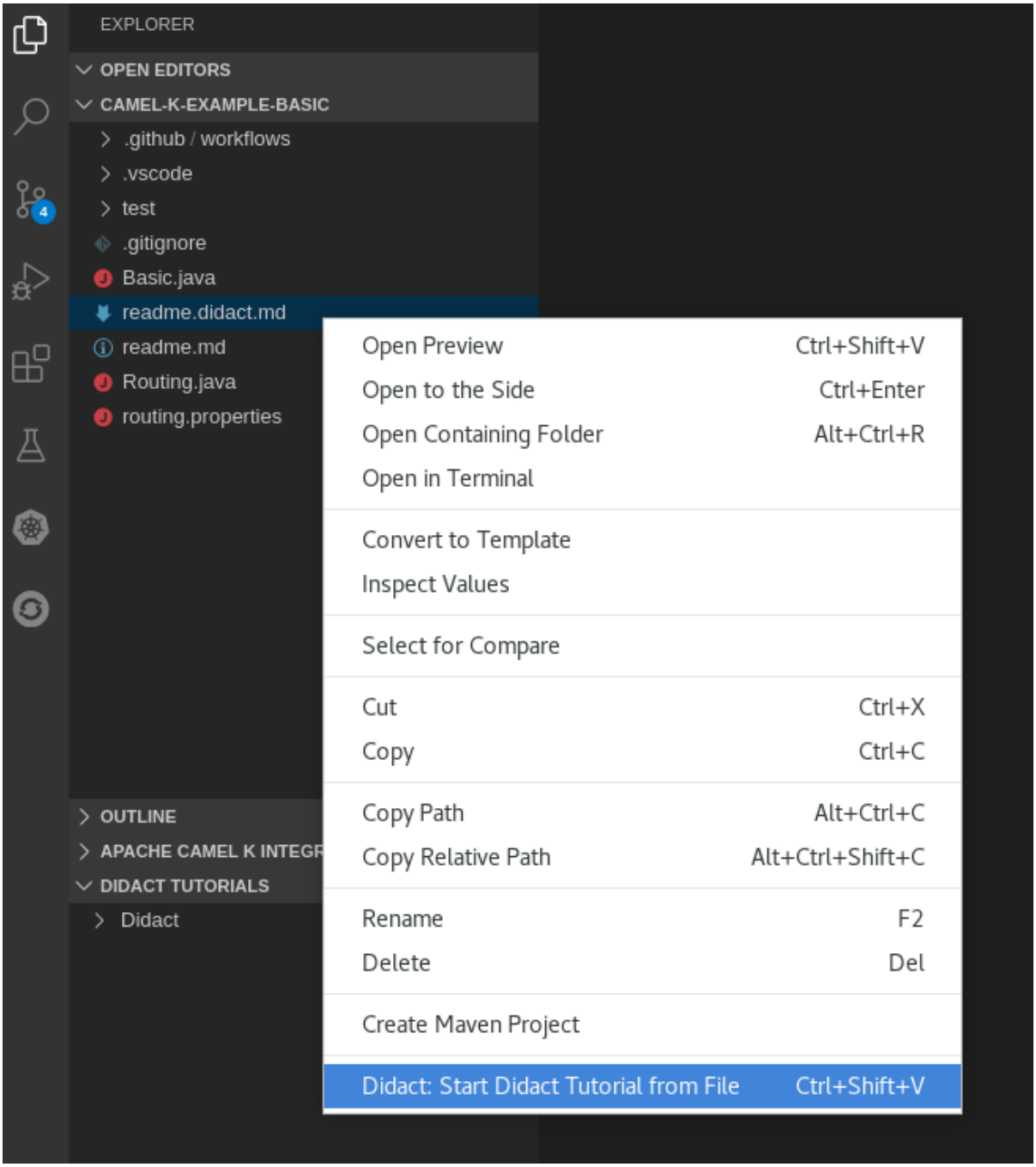
- See the tutorial readme in GitHub: https://github.com/openshift-integration/camel-k-example-basic.
- You must have cluster administrator access to an OpenShift cluster to install Camel K. See Section 2.1, "Installing Camel K from the OpenShift OperatorHub" .
- You must have the **kamel** command installed. 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.
- 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.

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:

- 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”.
- 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:

- 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".
- You can also install the optional Red Hat Integration - 3scale Operator on your OpenShift system to manage the API. See Deploying 3scale using the 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-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

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".

- 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
```

```
Creation Timestamp:  Tue, 14 Apr 2020 16:57:04 +0100
Phase:               Running
Runtime Version:     1.1.0
Kit:                 kit-bqatqib5t4kse5vukt40
Image:               image-registry.openshift-image-registry.svc:5000/camel-k-test/camel-k-kit-
bqatqib5t4kse5vukt40@sha256:3788d571e6534ab27620b6826e6a4f10c23fc871d2f8f60673
b7c20e617d6463
Version:             1.0.0-RC2
Dependencies:
  camel:log
  camel:timer
  mvn:org.apache.camel.k/camel-k-loader-yaml
  mvn:org.apache.camel.k/camel-k-runtime-main
Sources:
  Name              Language  Compression  Ref  Ref Key
  hello.camelk.yaml  yaml      false
Conditions:
  Type                       Status  Reason                      Message
  IntegrationPlatformAvailable  True    IntegrationPlatformAvailable  camel-k
  IntegrationKitAvailable       True    IntegrationKitAvailable       kit-bqatqib5t4kse5vukt40
  CronJobAvailable              False   CronJobNotAvailableReason     different controller
strategy used (deployment)
  DeploymentAvailable           True    DeploymentAvailable           deployment name is hello
  ServiceAvailable              False   ServiceNotAvailable           no http service required
  ExposureAvailable             False   RouteNotAvailable             no target service found
```

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

```
$ kamel log hello
...
1] 2020-04-14 16:03:41.205 INFO  [Camel (camel-k) thread #1 - timer://yaml] info -
Exchange[ExchangePattern: InOnly, BodyType: String, Body: Hello Camel K from yaml]
[1] 2020-04-14 16:03:42.205 INFO  [Camel (camel-k) thread #1 - timer://yaml] info -
Exchange[ExchangePattern: InOnly, BodyType: String, Body: Hello Camel K from yaml]
[1] 2020-04-14 16:03:43.204 INFO  [Camel (camel-k) thread #1 - timer://yaml] info -
Exchange[ExchangePattern: InOnly, BodyType: String, Body: Hello Camel K from yaml]
...
```

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 17:02:17.970 DEBUG [main] PropertiesComponentFactoryResolver -
Detected and using PropertiesComponent:
org.apache.camel.component.properties.PropertiesComponent@3e92efc3
[1] 2020-04-13 17:02:17.974 INFO  [main] ApplicationRuntime - Listener
org.apache.camel.k.listener.PropertiesFunctionsConfigurer@4b5a5ed1 executed in phase
Starting
[1] 2020-04-13 17:02:17.984 INFO  [main] RuntimeSupport - Looking up loader for language:
java
[1] 2020-04-13 17:02:17.987 INFO  [main] RuntimeSupport - Found loader
org.apache.camel.k.loader.java.JavaSourceLoader@4facf68f for language java from service
definition
[1] 2020-04-13 17:02:18.553 INFO  [main] RoutesConfigurer - Loading routes from:
file:/etc/camel/sources/i-source-000/HelloCamelK.java?language=java
[1] 2020-04-13 17:02:18.553 INFO  [main] ApplicationRuntime - Listener
org.apache.camel.k.listener.RoutesConfigurer@49c386c8 executed in phase
ConfigureRoutes
[1] 2020-04-13 17:02:18.555 DEBUG [main] PropertiesComponent - Parsed location:
/etc/camel/conf/application.properties
[1] 2020-04-13 17:02:18.557 INFO  [main] BaseMainSupport - Using properties from:
file:/etc/camel/conf/application.properties
[1] 2020-04-13 17:02:18.563 DEBUG [main] BaseMainSupport - Properties from Camel
properties component:
[1] 2020-04-13 17:02:18.598 DEBUG [main] RoutesConfigurer - RoutesCollectorEnabled:
org.apache.camel.k.main.ApplicationRuntime$NoRoutesCollector@2f953efd
[1] 2020-04-13 17:02:18.598 DEBUG [main] RoutesConfigurer - Adding routes into
CamelContext from RoutesBuilder: Routes: []
[1] 2020-04-13 17:02:18.598 DEBUG [main] DefaultCamelContext - Adding routes from
builder: Routes: []
...
```

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

- For details of development tools to view logging, see VS Code Tooling for Apache Camel K by Red Hat

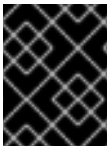
CHAPTER 6. MONITORING CAMEL K INTEGRATIONS

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 OPENSHIFT

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**:

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

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

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

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

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

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

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: cluster-monitoring-config
  namespace: openshift-monitoring
```

```
data:
  config.yaml: |
    enableUserWorkload: true
```

Additional resources

- Monitoring your own services in the OpenShift documentation

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 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
(1)

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;

@ApplicationScoped
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"); (2)

      from("timer:stream?period=1000")
          .routeId("unreliable-service")
          .setBody(header(Exchange.TIMER_COUNTER).prepend("event #"))
          .log("Processing ${body}...")
          // Register the 'generated' meter
          .to("microprofile-metrics:meter:generated") (3)
          // Register the 'attempt' meter via @Metered in Service.java
          .bean("service") (4)
          .filter(header(Exchange.REDELIVERED))
              .log(LoggingLevel.WARN, "Processed ${body} after
${header.CamelRedeliveryCounter} retries")
              .setHeader(MicroProfileMetricsConstants.HEADER_METER_MARK,
```

```
header(Exchange.REDELIVERY_COUNTER))
            // Register the 'redelivery' meter
            .to("microprofile-metrics:meter:redelivery") 5
        .end()
        .log("Successfully processed ${body}")
        // Register the 'success' meter
        .to("microprofile-metrics:meter:success"); 6
    }
}
```

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:

```
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) { 1
    Random rand = new Random();
      if (rand.nextDouble() < 0.5) {
        throw new RuntimeExchangeException("Random failure", exchange); 2
```

```
    }
  }
}
```

1. The **@Metered** MicroProfile Metrics annotation declares the meter and the name is automatically generated based on the metrics method name, in this case, **attempt**.
2. 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:

```
...
  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
...
[1] 2020-04-13 15:39:59.213 INFO  [main] ApplicationRuntime - Listener
org.apache.camel.k.listener.RoutesDumper@6e0dec4a executed in phase Started
[1] 2020-04-13 15:40:00.237 INFO  [Camel (camel-k) thread #1 - timer://java] info -
Exchange[ExchangePattern: InOnly, BodyType: String, Body: Hola Mundo from java]
...
```

Additional resources

- 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”

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:

```
...
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:

```
$ kamel run Routing.java --property-file routing.properties --dev
...
[1] 2020-04-13 15:20:30.424 INFO  [main] ApplicationRuntime - Listener
org.apache.camel.k.listener.RoutesDumper@6e0dec4a executed in phase Started
[1] 2020-04-13 15:20:31.461 INFO  [Camel (camel-k) thread #1 - timer://java] priority -
!!Priority item: engine
[1] 2020-04-13 15:20:37.426 INFO  [Camel (camel-k) thread #1 - timer://java] standard -
Standard item: window
```

```
[1] 2020-04-13 15:20:43.429 INFO  [Camel (camel-k) thread #1 - timer://java] priority -
!!Priority item: door
...
```

Additional resources

- Section 4.1, "Deploying a basic Camel K Java integration"
- Section 7.1, "Configuring Camel K integrations using properties"

7.3. CONFIGURING CAMEL K PROPERTIES USING AN OPENSHIFT 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:

```
...
  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:

```
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:

```
$ 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:

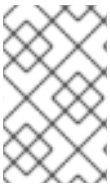
```
$ kamel run --configmap=my-configmap HelloCamelK.java --dev
...
[1] 2020-04-14 14:18:20.654 DEBUG [Camel (camel-k) thread #1 - timer://java]
DefaultReactiveExecutor - Queuing reactive work: CamelInternalProcessor - UnitOfWork -
afterProcess - DefaultErrorHandler[sendTo(log://info)] - ID-hello-camel-k-5df4bcd7dc-zq4vw-
1586873876659-0-25
[1] 2020-04-14 14:18:20.654 DEBUG [Camel (camel-k) thread #1 - timer://java]
SendProcessor - >>>> log://info Exchange[ID-hello-camel-k-5df4bcd7dc-zq4vw-
1586873876659-0-25]
[1] 2020-04-14 14:18:20.655 INFO  [Camel (camel-k) thread #1 - timer://java] info -
Exchange[ExchangePattern: InOnly, BodyType: String, Body: Bonjour le monde from java]
...
```

Additional resources

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

7.4. CONFIGURING CAMEL K PROPERTIES USING AN OPENSHIFT 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")
    .routeId("java")
    .setBody()
      .simple("{{my.message}} from ${routeId}")
    .to("log:info");
...
```

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

```
apiVersion: v1
kind: Secret
metadata:
  name: my-secret
data:
  application.properties: |

bXkubWVzc2FnZT1IZWxsbyBXb3JsZAogICAgbG9nZ2luZy5sZXZlbC5vcmcuYXBhY2hlLmNhbWVs
    PURFQlVHCg==
```

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:

```
$ 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:

```
$ kamel run --secret=my-secret HelloCamelK.java --dev
[1] 2020-04-14 14:30:29.788 DEBUG [Camel (camel-k) thread #1 - timer://java]
DefaultReactiveExecutor - Queuing reactive work: CamelInternalProcessor - UnitOfWork -
afterProcess - DefaultErrorHandler[sendTo(log://info)] - ID-hello-camel-k-68f85d99b9-srd92-
1586874486770-0-144
[1] 2020-04-14 14:30:29.789 DEBUG [Camel (camel-k) thread #1 - timer://java]
SendProcessor - >>>> log://info Exchange[ID-hello-camel-k-68f85d99b9-srd92-
1586874486770-0-144]
[1] 2020-04-14 14:30:29.789 INFO  [Camel (camel-k) thread #1 - timer://java] info -
Exchange[ExchangePattern: InOnly, BodyType: String, Body: Hello World from java]
```

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
```

Prerequisites

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

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:

```
...
  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

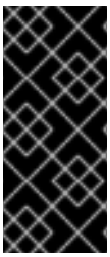
You can disable automatic dependency resolution by disabling the dependencies trait: **-trait dependencies.enabled=false**. However, this is not recommended in most cases.

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 8.3.3, “Camel Trait”
- Section 8.3.2, “Container Trait”
- Section 8.3.4, “Dependencies Trait”

8.1. CAMEL K TRAIT AND PROFILE CONFIGURATION

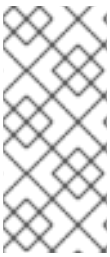
This section explains the important Camel K concepts of *traits* and *profiles*, which are used to configure advanced Camel K features at runtime.

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:

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

Property	Type	Description
3scale.description-path	string	The path where the Open-API specification is published (default **/openapi.json**)

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:

Property	Type	Description
affinity.enabled	bool	Can be used to enable or disable a trait. All traits share this common property.
affinity.pod-affinity	bool	Always co-locates multiple replicas of the integration in the same node (default **false**).
affinity.pod-anti-affinity	bool	Never co-locates multiple replicas of the integration in the same node (default **false**).
affinity.node-affinity-labels	[]string	Defines a set of nodes the integration pod(s) are eligible to be scheduled on, based on labels on the node.
affinity.pod-affinity-labels	[]string	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.
affinity.pod-anti-affinity-labels	[]string	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.

8.2.2.2. Examples

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

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

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

```
$ kamel run -t affinity.pod-anti-affinity-labels="camel.apache.org/integration" ...
```

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

```
$ kamel run -t affinity.pod-affinity-labels="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):

```
$ kamel run -t affinity.pod-anti-affinity-labels="camel.apache.org/integration" -t affinity.pod-anti-affinity-labels="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:

Property	Type	Description
cron.enabled	bool	Can be used to enable or disable a trait. All traits share this common property.
cron.schedule	string	The CronJob schedule for the whole integration. If multiple routes are declared, they must have the same schedule for this mechanism to work correctly.
cron.components	string	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**.
cron.fallback	bool	Use the default Camel implementation of the **cron** endpoint (**quartz**) instead of trying to materialize the integration as Kubernetes CronJob.
cron.concurrency-policy	string	Specifies how to treat concurrent executions of a Job. Valid values are: - "Allow": allows CronJobs to run concurrently; - "Forbid" (default): forbids concurrent runs, skipping next run if previous run hasn't finished yet; - "Replace": cancels currently running job and replaces it with a new one
cron.auto	bool	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).

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:

Property	Type	Description
gc.enabled	bool	Can be used to enable or disable a trait. All traits share this common property.
gc.discovery-cache	./pkg/trait.discoveryCacheType	Discovery client cache to be used, either **disabled**, **disk** or **memory** (default **memory**)

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:

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

The following configuration options are available:

Property	Type	Description
istio.enabled	bool	Can be used to enable or disable a trait. All traits share this common property.
istio.allow	string	Configures a (comma-separated) list of CIDR subnets that should not be intercepted by the Istio proxy (**10.0.0.0/8,172.16.0.0/12,192.168.0.0/16** by default).
istio.inject	bool	Forces the value for labels **sidecar.istio.io/inject**. By default the label is set to **true** on deployment and not set on Knative Service.

8.2.6. Jolokia Trait

The Jolokia trait activates and configures the Jolokia Java agent.

See https://jolokia.org/reference/html/agents.html

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:

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

The following configuration options are available:

Property	Type	Description
jolokia.enabled	bool	Can be used to enable or disable a trait. All traits share this common property.
jolokia.ca-cert	string	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).
jolokia.client-principal	[]string	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).
jolokia.discovery-enabled	bool	Listen for multicast requests (default **false**)
jolokia.extended-client-check	bool	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).
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 inject 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:

Property	Type	Description
knative.enabled	bool	Can be used to enable or disable a trait. All traits share this common property.
knative.configuration	string	Can be used to inject a Knative complete configuration in JSON format.
knative.channel-sources	[]string	List of channels used as source of integration routes. Can contain simple channel names or full Camel URIs.
knative.channel-sinks	[]string	List of channels used as destination of integration routes. Can contain simple channel names or full Camel URIs.
knative.endpoint-sources	[]string	List of channels used as source of integration routes.
knative.endpoint-sinks	[]string	List of endpoints used as destination of integration routes. Can contain simple endpoint names or full Camel URIs.
knative.event-sources	[]string	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 "default").
knative.event-sinks	[]string	List of event types that the integration will produce. Can contain simple event types or full Camel URIs (to use a specific broker).
knative.filter-source-channels	bool	Enables filtering on events based on the header "ce-knativehistory". 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.

Property	Type	Description
knative.camel-source-compat	bool	Enables Knative CamelSource pre 0.15 compatibility fixes (will be removed in future versions).
knative.sink-binding	bool	Allows binding the integration to a sink via a Knative SinkBinding resource. This can be used when the integration targets a single sink. It's disabled by default.
knative.auto	bool	Enable automatic discovery of all trait properties.

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:

Property	Type	Description
knative-service.enabled	bool	Can be used to enable or disable a trait. All traits share this common property.
knative-service.autoscaling-class	string	Configures the Knative autoscaling class property (e.g. to set **hpa.autoscaling.knative.dev** or **kpa.autoscaling.knative.dev** autoscaling). Refer to the Knative documentation for more information.
knative-service.autoscaling-metric	string	Configures the Knative autoscaling metric property (e.g. to set **concurrency** based or **cpu** based autoscaling). Refer to the Knative documentation for more information.
knative-service.autoscaling-target	int	Sets the allowed concurrency level or CPU percentage (depending on the autoscaling metric) for each Pod. Refer to the Knative documentation for more information.

Property	Type	Description
knative-service.min-scale	int	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	int	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	bool	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:

Property	Type	Description

Property	Type	Description
master.enabled	bool	Can be used to enable or disable a trait. All traits share this common property.
master.auto	bool	Enables automatic configuration of the trait.
master.include-delegate-dependencies	bool	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.
master.configmap	string	Name of the configmap that will be used to store the lock. Defaults to "<integration-name>-lock".
master.label-key	string	Label that will be used to identify all pods contending the lock. Defaults to "camel.apache.org/integration".
master.label-value	string	Label value that will be used to identify all pods contending the lock. Defaults to the integration name.

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:

Property	Type	Description
prometheus.enabled	bool	Can be used to enable or disable a trait. All traits share this common property.
prometheus.port	int	The Prometheus endpoint port (default **9779**, or **8080** with Quarkus).
prometheus.service-monitor	bool	Whether a **ServiceMonitor** resource is created (default **true**).
prometheus.service-monitor-labels	[]string	The **ServiceMonitor** resource labels, applicable when **service-monitor** is **true**.
prometheus.configmap	string	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.

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:

Property	Type	Description
quarkus.enabled	bool	Can be used to enable or disable a trait. All traits share this common property.
quarkus.native	bool	The Quarkus runtime type (reserved for future use)

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:

Property	Type	Description
route.enabled	bool	Can be used to enable or disable a trait. All traits share this common property.
route.host	string	To configure the host exposed by the route.
route.tls-termination	string	The TLS termination type, like **edge**, **passthrough** or **reencrypt**. Refer to the OpenShift documentation for additional information.
route.tls-certificate	string	The TLS certificate contents. Refer to the OpenShift documentation for additional information.
route.tls-key	string	The TLS certificate key contents. Refer to the OpenShift documentation for additional information.
route.tls-ca-certificate	string	The TLS cert authority certificate contents. Refer to the OpenShift documentation for additional information.
route.tls-destination-ca-certificate	string	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.

Property	Type	Description
route.tls-insecure-edge-termination-policy	string	To configure how to deal with insecure traffic, e.g. **Allow**, **Disable** or **Redirect** traffic. Refer to the OpenShift documentation for additional information.

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:

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

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:

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

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:

Property	Type	Description
container.enabled	bool	Can be used to enable or disable a trait. All traits share this common property.
container.auto	bool	
container.request-cpu	string	The minimum amount of CPU required.
container.request-memory	string	The minimum amount of memory required.
container.limit-cpu	string	The maximum amount of CPU required.
container.limit-memory	string	The maximum amount of memory required.
container.expose	bool	Can be used to enable/disable exposure via kubernetes Service.
container.port	int	To configure a different port exposed by the container (default **8080**).
container.port-name	string	To configure a different port name for the port exposed by the container (default **http**).
container.service-port	int	To configure under which service port the container port is to be exposed (default **80**).
container.service-port-name	string	To configure under which service port name the container port is to be exposed (default **http**).
container.name	string	The main container name. It's named **integration** by default.
container.probes-enabled	bool	ProbesEnabled enable/disable probes on the container (default **false**)
container.probe-path	string	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.
container.liveness-initial-delay	int32	Number of seconds after the container has started before liveness probes are initiated.
container.liveness-timeout	int32	Number of seconds after which the probe times out. Applies to the liveness probe.
container.liveness-period	int32	How often to perform the probe. Applies to the liveness probe.

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

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:

Property	Type	Description
camel.enabled	bool	Can be used to enable or disable a trait. All traits share this common property.
camel.runtime-version	string	The camel-k-runtime version to use for the integration. It overrides the default version set in the Integration Platform.

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:

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

The following configuration options are available:

Property	Type	Description
dependencies.enabled	bool	Can be used to enable or disable a trait. All traits share this common property.

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**.

WARNING

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

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:

Property	Type	Description
deployer.enabled	bool	Can be used to enable or disable a trait. All traits share this common property.
deployer.kind	string	Allows to explicitly select the desired deployment kind between **deployment**, **cron-job** or **knative-service** when creating the resources for running the integration.

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**.

WARNING

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

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:

Property	Type	Description
deployment.enabled	bool	Can be used to enable or disable a trait. All traits share this common property.

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:

Property	Type	Description
environment.enabled	bool	Can be used to enable or disable a trait. All traits share this common property.
environment.container-meta	bool	

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**.

WARNING

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

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:

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

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:

Property	Type	Description
openapi.enabled	bool	Can be used to enable or disable a trait. All traits share this common property.

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:

Property	Type	Description
owner.enabled	bool	Can be used to enable or disable a trait. All traits share this common property.
owner.target-annotations	[]string	The set of annotations to be transferred
owner.target-labels	[]string	The set of labels to be transferred

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:

Property	Type	Description
platform.enabled	bool	Can be used to enable or disable a trait. All traits share this common property.
platform.create-default	bool	To create a default (empty) platform when the platform is missing.
platform.auto	bool	To automatically detect from the environment if a default platform can be created (it will be created on OpenShift only).

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 9.1. kamel commands

Name	Description	Example
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

- Section 2.3, “Installing the Camel K and OpenShift command line tools”

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

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