Red Hat build of Apache Camel K 1.10.5

Developing and Managing Integrations Using Camel K

A developer's guide to Camel K

Last Updated: 2024-02-06

Red Hat build of Apache Camel K 1.10.5 Developing and Managing Integrations Using Camel K

A developer's guide to Camel K

Legal Notice

Abstract

The essentials of developing, configuring, and managing Red Hat build of Apache Camel K applications.

Table of Contents

PREFACE

MAKING OPEN SOURCE MORE INCLUSIVE

Red Hat is committed to replacing problematic language in our code, documentation, and web properties. We are beginning with these four terms: master, slave, blacklist, and whitelist. Because of the enormity of this endeavor, these changes will be implemented gradually over several upcoming releases. For more details, see our CTO Chris Wright's message .

CHAPTER 1. 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.

1.1. 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

- Setting up your Camel K development environment
- You must already have a Camel integration written in Java 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 to print detailed information about the integration:

```
kamel describe integration hello
```

```
Name:               hello
Namespace:          myproject
Creation Timestamp:  Fri, 13 Aug 2021 16:23:21 +0200
Phase:              Building Kit
Runtime Version:    1.7.1.fuse-800025-redhat-00001
Kit:                myproject/kit-c4ci6mbe9hl5ph5c9sjg
Image:
Version:            1.6.6
Dependencies:
  camel:core
  camel:log
  camel:timer
  mvn:org.apache.camel.k:camel-k-runtime
  mvn:org.apache.camel.quarkus:camel-quarkus-yaml-dsl
Sources:
  Name                    Language  Compression  Ref  Ref Key
  camel-k-embedded-flow.yaml  yaml      false
Conditions:
  Type                    Status  Reason                    Message
  IntegrationPlatformAvailable  True    IntegrationPlatformAvailable  myproject/camel-k
  IntegrationKitAvailable       True    IntegrationKitAvailable       kit-c4ci6mbe9hl5ph5c9sjg
  CronJobAvailable              False   CronJobNotAvailableReason     different controller
strategy used (deployment)
  DeploymentAvailable           True    DeploymentAvailable           deployment name is hello
  KnativeServiceAvailable       False   KnativeServiceNotAvailable    different controller
strategy used (deployment)
  Ready                   True    ReplicaSetReady
```

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

```
kamel log hello
```

```
...
[1] 2021-08-13 14:37:15,860 INFO [info] (Camel (camel-1) thread #0 - timer://yaml)
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 Managing Camel K integration logging levels

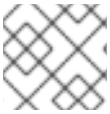
- For faster deployment turnaround times, see Running Camel K integrations in development mode
- For details of development tools to manage integrations, see VS Code Tooling for Apache Camel K by Red Hat

1.2. MANAGING CAMEL K INTEGRATION LOGGING LEVELS

Camel K uses Quarkus Logging mechanism as the logging framework for integrations. You can configure the logging levels of various loggers on the command line at runtime by specifying the **quarkus.log.category** prefix as an integration property. For example:

Example

```
--property 'quarkus.log.category."org".level'=DEBUG
```



NOTE

It is important to escape the property with single quotes.

Prerequisites

- 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 --dev --property 'quarkus.log.category."org.apache.camel.support".level'=DEBUG
Basic.java

...
integration "basic" created
    Progress: integration "basic" in phase Initialization
    Progress: integration "basic" in phase Building Kit
    Progress: integration "basic" in phase Deploying
    Condition "IntegrationPlatformAvailable" is "True" for Integration basic: myproject/camel-k
    Integration basic in phase "Initialization"
    Integration basic in phase "Building Kit"
    Integration basic in phase "Deploying"
    Condition "IntegrationKitAvailable" is "True" for Integration basic: kit-
c4dn5l62v9g3aopkocag
    Condition "DeploymentAvailable" is "True" for Integration basic: deployment name is basic
    Condition "CronJobAvailable" is "False" for Integration basic: different controller strategy
used (deployment)
    Progress: integration "basic" in phase Running
    Condition "KnativeServiceAvailable" is "False" for Integration basic: different controller
strategy used (deployment)
    Integration basic in phase "Running"
    Condition "Ready" is "False" for Integration basic
    Condition "Ready" is "True" for Integration basic
    [1] Monitoring pod basic-575b97f64b-7l5rl
    [1] 2021-08-17 08:35:22,906 DEBUG [org.apa.cam.sup.LRUCacheFactory] (main)
```

```
Creating DefaultLRUCacheFactory
    [1] 2021-08-17 08:35:23,132 INFO  [org.apa.cam.k.Runtime] (main) Apache Camel K
Runtime 1.7.1.fuse-800025-redhat-00001
    [1] 2021-08-17 08:35:23,134 INFO  [org.apa.cam.qua.cor.CamelBootstrapRecorder] (main)
bootstrap runtime: org.apache.camel.quarkus.main.CamelMainRuntime
    [1] 2021-08-17 08:35:23,224 INFO  [org.apa.cam.k.lis.SourcesConfigurer] (main) Loading
routes from: SourceDefinition{name='Basic', language='java',
location='file:/etc/camel/sources/Basic.java', }
    [1] 2021-08-17 08:35:23,232 DEBUG [org.apa.cam.sup.ResolverHelper] (main) Found
RoutesBuilderLoader: org.apache.camel.dsl.java.joor.JavaRoutesBuilderLoader via: META-
INF/services/org/apache/camel/java
    [1] 2021-08-17 08:35:23,232 DEBUG [org.apa.cam.sup.ResolverHelper] (main) Detected
and using RoutesBuilderLoader:
org.apache.camel.dsl.java.joor.JavaRoutesBuilderLoader@68dc098b
    [1] 2021-08-17 08:35:23,236 DEBUG [org.apa.cam.sup.ResolverHelper] (main) Found
ResourceResolver: org.apache.camel.impl.engine.DefaultResourceResolvers$FileResolver
via: META-INF/services/org/apache/camel/file
    [1] 2021-08-17 08:35:23,237 DEBUG [org.apa.cam.sup.ResolverHelper] (main) Detected
and using ResourceResolver:
org.apache.camel.impl.engine.DefaultResourceResolvers$FileResolver@5b67bb7e
    [1] 2021-08-17 08:35:24,320 DEBUG [org.apa.cam.sup.ResolverHelper] (main) Lookup
Language with name simple in registry. Found:
org.apache.camel.language.simple.SimpleLanguage@74d7184a
    [1] 2021-08-17 08:35:24,328 DEBUG [org.apa.cam.sup.EventHelper] (main) Ignoring
notifying event Initializing CamelContext: camel-1. The EventNotifier has not been started
yet: org.apache.camel.quarkus.core.CamelManagementEventBridge@3301500b
    [1] 2021-08-17 08:35:24,336 DEBUG [org.apa.cam.sup.ResolverHelper] (main) Lookup
Component with name timer in registry. Found:
org.apache.camel.component.timer.TimerComponent@3ef41c66
    [1] 2021-08-17 08:35:24,342 DEBUG [org.apa.cam.sup.DefaultComponent] (main)
Creating endpoint uri=[timer://java?period=1000], path=[java]
    [1] 2021-08-17 08:35:24,350 DEBUG [org.apa.cam.sup.ResolverHelper] (main) Found
ProcessorFactory: org.apache.camel.processor.DefaultProcessorFactory via: META-
INF/services/org/apache/camel/processor-factory
    [1] 2021-08-17 08:35:24,351 DEBUG [org.apa.cam.sup.ResolverHelper] (main) Detected
and using ProcessorFactory:
org.apache.camel.processor.DefaultProcessorFactory@704b2127
    [1] 2021-08-17 08:35:24,369 DEBUG [org.apa.cam.sup.ResolverHelper] (main) Found
InternalProcessorFactory: org.apache.camel.processor.DefaultInternalProcessorFactory via:
META-INF/services/org/apache/camel/internal-processor-factory
    [1] 2021-08-17 08:35:24,369 DEBUG [org.apa.cam.sup.ResolverHelper] (main) Detected
and using InternalProcessorFactory:
org.apache.camel.processor.DefaultInternalProcessorFactory@4f8caaf3
    [1] 2021-08-17 08:35:24,442 DEBUG [org.apa.cam.sup.ResolverHelper] (main) Lookup
Component with name log in registry. Found:
org.apache.camel.component.log.LogComponent@46b695ec
    [1] 2021-08-17 08:35:24,444 DEBUG [org.apa.cam.sup.DefaultComponent] (main)
Creating endpoint uri=[log://info], path=[info]
    [1] 2021-08-17 08:35:24,461 DEBUG [org.apa.cam.sup.EventHelper] (main) Ignoring
notifying event Initialized CamelContext: camel-1. The EventNotifier has not been started yet:
org.apache.camel.quarkus.core.CamelManagementEventBridge@3301500b
    [1] 2021-08-17 08:35:24,467 DEBUG [org.apa.cam.sup.DefaultProducer] (main) Starting
producer: Producer[log://info]
    [1] 2021-08-17 08:35:24,469 DEBUG [org.apa.cam.sup.DefaultConsumer] (main) Build
consumer: Consumer[timer://java?period=1000]
    [1] 2021-08-17 08:35:24,475 DEBUG [org.apa.cam.sup.DefaultConsumer] (main) Starting
```

```
consumer: Consumer[timer://java?period=1000]
    [1] 2021-08-17 08:35:24,481 INFO  [org.apa.cam.imp.eng.AbstractCamelContext] (main)
Routes startup summary (total:1 started:1)
    [1] 2021-08-17 08:35:24,481 INFO  [org.apa.cam.imp.eng.AbstractCamelContext] (main)
Started java (timer://java)
    [1] 2021-08-17 08:35:24,482 INFO  [org.apa.cam.imp.eng.AbstractCamelContext] (main)
Apache Camel 3.10.0.fuse-800010-redhat-00001 (camel-1) started in 170ms (build:0ms
init:150ms start:20ms)
    [1] 2021-08-17 08:35:24,487 INFO  [io.quarkus] (main) camel-k-integration 1.6.6 on JVM
(powered by Quarkus 1.11.7.Final-redhat-00009) started in 2.192s.
    [1] 2021-08-17 08:35:24,488 INFO  [io.quarkus] (main) Profile prod activated.
    [1] 2021-08-17 08:35:24,488 INFO  [io.quarkus] (main) Installed features: [camel-bean,
camel-core, camel-java-joor-dsl, camel-k-core, camel-k-runtime, camel-log, camel-support-
common, camel-timer, cdi]
    [1] 2021-08-17 08:35:25,493 INFO  [info] (Camel (camel-1) thread #0 - timer://java)
Exchange[ExchangePattern: InOnly, BodyType: String, Body: Hello Camel K from java]
    [1] 2021-08-17 08:35:26,479 INFO  [info] (Camel (camel-1) thread #0 - timer://java)
Exchange[ExchangePattern: InOnly, BodyType: String, Body: Hello Camel K from java]
...
```

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

Additional resources

- For more details on the logging framework, see the Configuring logging format
- For details of development tools to view logging, see VS Code Tooling for Apache Camel K by Red Hat

1.3. SCALING CAMEL K INTEGRATIONS

You can scale your integrations using the **oc scale** command.

Procedure

- To scale the Camel K integrations, run the following command.

```
oc scale it <integration_name> --replicas <number_of_replicas>
```

- You can also edit the Integration resource directly to scale the integration.

```
oc patch it <integration_name> --type merge -p '{"spec":{"replicas":<number_of_replicas>}}'
```

To view the number of replicas for the integration use following command.

```
oc get it <integration_name> -o jsonpath='{.status.replicas}'
```

CHAPTER 2. MONITORING CAMEL K INTEGRATIONS

Red Hat Integration - Camel K monitoring is based on the OpenShift monitoring system. 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 2.1, “Enabling user workload monitoring in OpenShift”
- Section 2.2, “Configuring Camel K integration metrics”
- Section 2.3, “Adding custom Camel K integration metrics”

2.1. 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.

Prerequisites

- You must have cluster administrator access to an OpenShift cluster on which the Camel K Operator is installed. See Installing Camel K .

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

- Enabling monitoring for user-defined projects

2.2. 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 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.

Default Camel Metrics provided by PROMETHEUS TRAIT

Some Camel specific metrics are available out of the box.

Name	Type	Description
application_camel_message_history_processing	timer	Sample of performance of each node in the route when message history is enabled
application_camel_route_count	gauge	Number of routes added
application_camel_route_running_count	gauge	Number of routes runnning

Name	Type	Description
application_camel_[route or context]_exchanges_inflight_count	gauge	Route inflight messages for a CamelContext or a route
application_camel_[route or context]_exchanges_total	counter	Total number of processed exchanges for a CamelContext or a route
application_camel_[route or context]_exchanges_completed_total	counter	Number of successfully completed exchange for a CamelContext or a route
application_camel_[route or context]_exchanges_failed_total	counter	Number of failed exchanges for a CamelContext or a route
application_camel_[route or context]_failuresHandled_total	counter	Number of failures handled for a CamelContext or a route
application_camel_[route or context]_externalRedeliveries_total	counter	Number of external initiated redeliveries (such as from JMS broker) for a CamelContext or a route
application_camel_context_status	gauge	The status of the Camel Context
application_camel_context_uptime_seconds	gauge	The amount of time since the Camel Context was started
application_camel_[route or exchange]*processing*[rate_per_second or one_min_rate_per_second or five_min_rate_per_second or fifteen_min_rate_per_second or min_seconds or max_seconds or mean_second or stddev_seconds]	gauge	Exchange message or route processing with multiple options
application_camel_[route or exchange]_processing_seconds	summary	Exchange message or route processing metric

Additional resources

- Prometheus Trait
- Camel Quarkus MicroProfile Metrics

2.3. 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 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 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 3. MONITORING CAMEL K OPERATOR

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

- Section 3.1, "Camel K Operator metrics"
- Section 3.2, "Enabling Camel K Operator monitoring"
- Section 3.3, "Camel K operator alerts"

3.1. CAMEL K OPERATOR METRICS

The Camel K operator monitoring endpoint exposes the following metrics:

Table 3.1. Camel K operator metrics

Name	Type	Description	Buckets	Labels
camel_k_reconciliation_duration_seconds	**HistogramVec**	Reconciliation request duration	0.25s, 0.5s, 1s, 5s	**namespace**, **group**, **version**, **kind**, **result**: **Reconciled**|**Errored**|**Requeued**, **tag**: **""**|**PlatformError**|**UserError**
camel_k_build_duration_seconds	**HistogramVec**	Build duration	30s, 1m, 1.5m, 2m, 5m, 10m	**result**: **Succeeded**|**Error**
camel_k_build_recovery_attempts	**Histogram**	Build recovery attempts	0, 1, 2, 3, 4, 5	**result**: **Succeeded**|**Error**
camel_k_build_queue_duration_seconds	**Histogram**	Build queue duration	5s, 15s, 30s, 1m, 5m,	N/A
camel_k_integration_first_readiness_seconds	**Histogram**	Time to first integration readiness	5s, 10s, 30s, 1m, 2m	N/A

3.2. ENABLING CAMEL K OPERATOR MONITORING

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.

Prerequisites

- You must have cluster administrator access to an OpenShift cluster on which the Camel K Operator is installed. See Installing Camel K .
- You must have already enabled monitoring of your own services in OpenShift. See Enabling user workload monitoring in OpenShift.

Procedure

1. Create a **PodMonitor** resource targeting the operator metrics endpoint, so that the Prometheus server can scrape the metrics exposed by the operator.

   operator-pod-monitor.yaml

   ```
   apiVersion: monitoring.coreos.com/v1
   kind: PodMonitor
   metadata:
     name: camel-k-operator
     labels:
       app: "camel-k"
       camel.apache.org/component: operator
   spec:
     selector:
       matchLabels:
         app: "camel-k"
         camel.apache.org/component: operator
     podMetricsEndpoints:
       - port: metrics
   ```

2. Create **PodMonitor** resource.

   ```
   oc apply -f operator-pod-monitor.yaml
   ```

Additional Resources

- For more information about the discovery mechanism and the relationship between the operator resources see Prometheus Operator getting started guide.
- In case your operator metrics are not discovered, you can find more information in Troubleshooting ServiceMonitor changes, which also applies to **PodMonitor** resources troubleshooting.

3.3. CAMEL K OPERATOR ALERTS

You can create a **PrometheusRule** resource so that the AlertManager instance from the OpenShift monitoring stack can trigger alerts, based on the metrics exposed by the Camel K operator.

Example

You can create a **PrometheusRule** resource with alerting rules based on the exposed metrics as shown below.

```
apiVersion: monitoring.coreos.com/v1
kind: PrometheusRule
```

```
metadata:
 name: camel-k-operator
spec:
 groups:
   - name: camel-k-operator
     rules:
       - alert: CamelKReconciliationDuration
         expr: |
           (
           1 - sum(rate(camel_k_reconciliation_duration_seconds_bucket{le="0.5"}[5m])) by (job)
           /
           sum(rate(camel_k_reconciliation_duration_seconds_count[5m])) by (job)
           )
           * 100
           > 10
         for: 1m
         labels:
           severity: warning
         annotations:
           message: |
             {{ printf "%0.0f" $value }}% of the reconciliation requests
             for {{ $labels.job }} have their duration above 0.5s.
       - alert: CamelKReconciliationFailure
         expr: |
           sum(rate(camel_k_reconciliation_duration_seconds_count{result="Errored"}[5m])) by (job)
           /
           sum(rate(camel_k_reconciliation_duration_seconds_count[5m])) by (job)
           * 100
           > 1
         for: 10m
         labels:
           severity: warning
         annotations:
           message: |
             {{ printf "%0.0f" $value }}% of the reconciliation requests
             for {{ $labels.job }} have failed.
       - alert: CamelKSuccessBuildDuration2m
         expr: |
           (
           1 - sum(rate(camel_k_build_duration_seconds_bucket{le="120",result="Succeeded"}[5m])) by
(job)
           /
           sum(rate(camel_k_build_duration_seconds_count{result="Succeeded"}[5m])) by (job)
           )
           * 100
           > 10
         for: 1m
         labels:
           severity: warning
         annotations:
           message: |
             {{ printf "%0.0f" $value }}% of the successful builds
             for {{ $labels.job }} have their duration above 2m.
       - alert: CamelKSuccessBuildDuration5m
         expr: |
           (
```

```
        1 - sum(rate(camel_k_build_duration_seconds_bucket{le="300",result="Succeeded"}[5m])) by
(job)
        /
        sum(rate(camel_k_build_duration_seconds_count{result="Succeeded"}[5m])) by (job)
        )
        * 100
        > 1
      for: 1m
      labels:
        severity: critical
      annotations:
        message: |
          {{ printf "%0.0f" $value }}% of the successful builds
          for {{ $labels.job }} have their duration above 5m.
    - alert: CamelKBuildFailure
      expr: |
        sum(rate(camel_k_build_duration_seconds_count{result="Failed"}[5m])) by (job)
        /
        sum(rate(camel_k_build_duration_seconds_count[5m])) by (job)
        * 100
        > 1
      for: 10m
      labels:
        severity: warning
      annotations:
        message: |
          {{ printf "%0.0f" $value }}% of the builds for {{ $labels.job }} have failed.
    - alert: CamelKBuildError
      expr: |
        sum(rate(camel_k_build_duration_seconds_count{result="Error"}[5m])) by (job)
        /
        sum(rate(camel_k_build_duration_seconds_count[5m])) by (job)
        * 100
        > 1
      for: 10m
      labels:
        severity: critical
      annotations:
        message: |
          {{ printf "%0.0f" $value }}% of the builds for {{ $labels.job }} have errored.
    - alert: CamelKBuildQueueDuration1m
      expr: |
        (
        1 - sum(rate(camel_k_build_queue_duration_seconds_bucket{le="60"}[5m])) by (job)
        /
        sum(rate(camel_k_build_queue_duration_seconds_count[5m])) by (job)
        )
        * 100
        > 1
      for: 1m
      labels:
        severity: warning
      annotations:
        message: |
          {{ printf "%0.0f" $value }}% of the builds for {{ $labels.job }}
          have been queued for more than 1m.
```

```
    - alert: CamelKBuildQueueDuration5m
      expr: |
        (
        1 - sum(rate(camel_k_build_queue_duration_seconds_bucket{le="300"}[5m])) by (job)
        /
        sum(rate(camel_k_build_queue_duration_seconds_count[5m])) by (job)
        )
        * 100
        > 1
      for: 1m
      labels:
        severity: critical
      annotations:
        message: |
          {{ printf "%0.0f" $value }}% of the builds for {{ $labels.job }}
          have been queued for more than 5m.
```

Camel K operator alerts

Following table shows the alerting rules that are defined in the **PrometheusRule** resource.

Name	Severity	Description
CamelKReconciliationDuration	warning	More than 10% of the reconciliation requests have their duration above 0.5s over at least 1 min.
CamelKReconciliationFailure	warning	More than 1% of the reconciliation requests have failed over at least 10 min.
CamelKSuccessBuildDuration2m	warning	More than 10% of the successful builds have their duration above 2 min over at least 1 min.
CamelKSuccessBuildDuration5m	critical	More than 1% of the successful builds have their duration above 5 min over at least 1 min.
CamelKBuildError	critical	More than 1% of the builds have errored over at least 10 min.
CamelKBuildQueueDuration1m	warning	More than 1% of the builds have been queued for more than 1 min over at least 1 min.
CamelKBuildQueueDuration5m	critical	More than 1% of the builds have been queued for more than 5 min over at least 1 min.

You can find more information about alerts in Creating alerting rules from the OpenShift documentation.

CHAPTER 4. CONFIGURING CAMEL K INTEGRATIONS

There are two configuration phases in a Camel K integration life cycle:

- **Build time** - When Camel Quarkus builds a Camel K integration, it consumes build-time properties.
- **Runtime** - When a Camel K integration is running, the integration uses runtime properties or configuration information from local files, OpenShift ConfigMaps, or Secrets.

You provide configure information by using the following options with the **kamel run** command:

- For build-time configuration, use the **--build-property** option as described in Specifying build-time configuration properties
- For runtime configuration, use the **--property** , **--config**, or **--resource** options as described in Specifying runtime configuration options

For example, you can use build-time and runtime options to quickly configure a datasource in Camel K as shown in the link: Connect Camel K with databases sample configuration.

4.1. SPECIFYING BUILD-TIME CONFIGURATION PROPERTIES

You might need to provide property values to the Camel Quarkus runtime so that it can build a Camel K integration. For more information about Quarkus configurations that take effect during build time, see the Quarkus Build Time configuration documentation . You can specify build-time properties directly at the command line or by referencing a property file. If a property is defined in both places, the value specified directly at the command line takes precedence over the value in the property file.

Prerequisites

- You must have access to an OpenShift cluster on which the Camel K Operator and OpenShift Serverless Operator are installed:
- Installing Camel K
- Installing OpenShift Serverless from the OperatorHub
- You know the Camel Quarkus configuration options that you want to apply to your Camel K integration.

Procedure

- Specify the **--build-property** option with the Camel K **kamel run** command:

```
kamel run --build-property <quarkus-property>=<property-value> <camel-k-integration>
```

For example, the following Camel K integration (named **my-simple-timer.yaml**) uses the **quarkus.application.name** configuration option:

```
- from:
    uri: "timer:tick"
    steps:
      - set-body:
          constant: "{{quarkus.application.name}}"
      - to: "log:info"
```

To override the default application name, specify a value for the **quarkus.application.name** property when you run the integration.

For example, to change the name from **my-simple-timer** to **my-favorite-app**:

```
kamel run --build-property quarkus.application.name=my-favorite-app my-simple-timer.yaml
```

- To provide more than one build-time property, add additional **--build-property** options to the **kamel run** command:

```
kamel run --build-property <quarkus-property1>=<property-value1> -build-property=
<quarkus-property2>=<property-value12> <camel-k-integration>
```

Alternately, if you need to specify multiple properties, you can create a property file and specify the property file with the **--build-property file** option:

```
kamel run --build-property file:<property-filename> <camel-k-integration>
```

For example, the following property file (named **quarkus.properties**) defines two Quarkus properties:

```
quarkus.application.name = my-favorite-app
quarkus.banner.enabled = true
```

The **quarkus.banner.enabled** property specifies to display the Quarkus banner when the integration starts up.

To specify the **quarkus.properties** file with the Camel K kamel run command:

```
kamel run --build-property file:quarkus.properties my-simple-timer.yaml
```

Quarkus parses the property file and uses the property values to configure the Camel K integration.

Additional resources

For information about Camel Quarkus as the runtime for Camel K integrations, see Quarkus Trait.

4.2. SPECIFYING RUNTIME CONFIGURATION OPTIONS

You can specify the following runtime configuration information for a Camel K integration to use when it is running:

- Runtime properties that you provide at the command line or in a .properties file.

- Configuration values that you want the Camel K operator to process and parse as runtime properties when the integration starts. You can provide the configuration values in a local text file, an OpenShift ConfigMap, or an OpenShift secret.

- Resource information that is not parsed as a property file when the integration starts. You can provide resource information in a local text file, a binary file, an OpenShift ConfigMap, or an OpenShift secret.

Use the following **kamel run** options:

- **--property**
  Use the **--property** option to specify runtime properties directly at the command line or by referencing a Java ***.properties** file. The Camel K operator appends the contents of the properties file to the running integration's **user.properties** file.

- **--config**
  Use the **--config** option to provide configuration values that you want the Camel K operator to process and parse as runtime properties when the integration starts.

  You can provide a local text file (1 MiB maximum file size), a ConfigMap (3MB) or a Secret (3MB). The file must be a UTF-8 resource. The materialized file (that is generated at integration startup from the file that you provide) is made available at the classpath level so that you can reference it in your integration code without having to provide an exact location.

  Note: If you need to provide a non-UTF-8 resource (for example, a binary file), use the **--resource** option.

- **--resource**
  Use the **--resource** option to provide a resource for the integration to access when it is running. You can provide a local text or a binary file (1 MiB maximum file size), a ConfigMap (3MB maximum), or a Secret (3MB maximum). Optionally, you can specify the destination of the file that is materialized for the resource. For example, if you want to set an HTTPS connection, use the **--resource** option to provide an SSL certificate (a binary file) that is expected in a specified location.

  The Camel K operator does not parse the resource for properties and does not add the resource to the classpath. (If you want to add the resource to the classpath, you can use the JVM trait in your integration).

4.2.1. Providing runtime properties

You can specify runtime properties directly at the command line or by referencing a Java ***.properties** file by using the **kamel run** command's **--property** option.

When you run an integration with the **--property** option, the Camel K operator appends the properties to the running integration's **user.properties** file.

4.2.1.1. Providing runtime properties at the command line

You can configure properties for Camel K integrations on the command line at runtime. When you define a property in an integration by 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

- Setting up your Camel K development environment

Procedure

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

```
...
- from:
    uri: "timer:tick"
    steps:
      - set-body:
          constant: "{{my.message}}"
      - to: "log:info"
...
```

2. Run the integration by using the following syntax to set the property value at runtime.

```
kamel run --property <property>=<value> <integration>
```

Alternately, you can use the **--p** shorthand notation (in place of **--property**):

```
kamel run --property <property>=<value> <integration>
```

For example:

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

or

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

Here is the example result:

```
...
[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]
...
```

See also

- Providing runtime properties in a property file

4.2.1.2. Providing runtime properties in a property file

You can configure multiple properties for Camel K integrations by specifying a property file (***.properties**) 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 by using a properties file, for example **--p file my-integration.properties**.

Prerequisite

- Setting up your Camel K development environment

Procedure

1. Create an integration properties file. The following example is from a file named **my.properties**:

```
my.key.1=hello
my.key.2=world
```

2. Develop a Camel integration that uses properties that are defined in the properties file. The following example **Routing.java** integration uses the **{{my.key.1}}** and **{{my.key.2=world}}** property placeholders:

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

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

    from("timer:property-file")
      .routeId("property-file")
      .log("property file content is: {{my.key.1}} {{my.key.2}}");

  }
}
```

3. Run the integration by using the following syntax to reference the property file:

```
kamel run --property file:<my-file.properties> <integration>
```

Alternately, you can use the **--p** shorthand notation (in place of **--property**):

```
kamel run --p file:<my-file.properties> <integration>
```

For example:

```
kamel run Routing.java --property:file=my.properties --dev
```

Additional resources

- Deploying a basic Camel K Java integration
- Providing runtime properties at the command line

4.2.2. Providing configuration values

You can provide configuration values that you want the Camel K operator to process and parse as runtime properties by using the **kamel run** command's **--config** option. You can provide the configuration values in a local text (UTF-8) file, an OpenShift ConfigMap, or an OpenShift secret.

When you run the integration, the Camel K operator materializes the provided file and adds it to the classpath so that you can reference the configuration values in your integration code without having to provide an exact location.

4.2.2.1. Specifying a text file

If you have a UTF-8 text file that contains configuration values, you can use the **--config file:/path/to/file** option to make the file available (with the same file name) on the running integration's classpath.

Prerequisites

- Setting up your Camel K development environment
- You have one or more (non-binary) text files that contain configuration values.
  For example, create a file named **resources-data.txt** that contains the following line of text:

```
the file body
```

Procedure

1. Create a Camel K integration that references the text file that contains configuration values.
   For example, the following integration (**ConfigFileRoute.java**) expects the **resources-data.txt** file to be available on the classpath at runtime:

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

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

    from("timer:config-file")
        .setBody()
            .simple("resource:classpath:resources-data.txt")
        .log("resource file content is: ${body}");

  }
}
```

2. Run the integration and use the **--config** option to specify the text file so that it is available to the running integration. For example:

```
kamel run --config file:resources-data.txt ConfigFileRoute.java --dev
```

   Optionally, you can provide more than one file by adding the **--config** option repeatedly, for example:

```
kamel run --config file:resources-data1.txt --config file:resources-data2.txt
ConfigFileRoute.java --dev
```

4.2.2.2. Specifying a ConfigMap

If you have an OpenShift ConfigMap that contains configuration values, and you need to materialize a ConfigMap so that it is available to your Camel K integration, use the --config configmap:<configmap-name> syntax.

Prerequisites

- Setting up your Camel K development environment
- You have one or more ConfigMap files stored on your OpenShift cluster.
  For example, you can create a ConfigMap by using the following command:

```
oc create configmap my-cm --from-literal=my-configmap-key="configmap content"
```

Procedure

1. Create a Camel K integration that references the ConfigMap.
   For example, the following integration (named **ConfigConfigmapRoute.java**) references a configuration value named **my-configmap-key** in a ConfigMap named **my-cm**.

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

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

    from("timer:configmap")
        .setBody()
            .simple("resource:classpath:my-configmap-key")
        .log("configmap content is: ${body}");

  }
}
```

2. Run the integration and use the **--config** option to materialize the ConfigMap file so that it is available to the running integration. For example:

```
kamel run --config configmap:my-cm ConfigConfigmapRoute.java --dev
```

   When the integration starts, the Camel K operator mounts an OpenShift volume with the ConfigMap's content.

Note: If you specify a ConfigMap that is not yet available on the cluster, the Integration waits and starts only after the ConfigMap becomes available.

4.2.2.3. Specifying a Secret

You can use an OpenShift Secret to securely contain configuration information. To materialize a secret so that it is available to your Camel K integration, you can use the **--config secret** syntax.

Prerequisites

- Setting up your Camel K development environment
- You have one or more Secrets stored on your OpenShift cluster.

For example, you can create a Secret by using the following command:

```
oc create secret generic my-sec --from-literal=my-secret-key="very top secret"
```

Procedure

1. Create a Camel K integration that references the ConfigMap.
   For example, the following integration (named **ConfigSecretRoute.java**) references the **my-secret** property that is in a Secret named **my-sec**:

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

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

    from("timer:secret")
        .setBody()
            .simple("resource:classpath:my-secret")
        .log("secret content is: ${body}");

  }
}
```

2. Run the integration and use the **--config** option to materialize the Secret so that it is available to the running integration. For example:

```
kamel run --config secret:my-sec ConfigSecretRoute.java --dev
```

   When the integration starts, the Camel K operator mounts an OpenShift volume with the Secret's content.

4.2.2.4. Referencing properties that are contained in ConfigMaps or Secrets

When you run an integration and you specify a ConfigMap or Secret with the **--config** option, the Camel K operator parses the ConfigMap or Secret as a runtime property file. Within your integration, you can reference the properties as you would reference any other runtime property.

Prerequisite

- Setting up your Camel K development environment

Procedure

1. Create a text file that contains properties.
   For example, create a file named **my.properties** that contains the following properties:

```
my.key.1=hello
my.key.2=world
```

2. Create a ConfigMap or a Secret based on the properties file.
   For example, use the following command to create a secret from the my.properties file:

```
oc create secret generic my-sec --from-file my.properties
```

3. In the integration, refer to the properties defined in the Secret.
   For example, the following integration (named **ConfigSecretPropertyRoute.java**) references the **my.key.1** and **my.key.2** properties:

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

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

    from("timer:secret")
        .routeId("secret")
        .log("{{my.key.1}} {{my.key.2}}");

  }
}
```

4. Run the integration and use the **--config** option to specify the Secret that contains the **my.key.1** and **my.key.2** properties.
   For example:

```
kamel run --config secret:my-sec ConfigSecretPropertyRoute.java --dev
```

4.2.2.5. Filtering configuration values obtained from a ConfigMap or Secret

ConfigMaps and Secrets can hold more than one source. For example, the following command creates a secret (**my-sec-multi**) from two sources:

```
oc create secret generic my-sec-multi --from-literal=my-secret-key="very top secret" --from-literal=my-secret-key-2="even more secret"
```

You can limit the quantity of information that your integration retrieves to just one source by using the **/key** notation after with the **--config configmap** or **--config secret** options.

Prerequisites

- Setting up your Camel K development environment
- You have a ConfigMap or a Secret that holds more than one source.

Procedure

1. Create an integration that uses configuration values from only one of the sources in the ConfigMap or Secret.
   For example, the following integration (**ConfigSecretKeyRoute.java**) uses the property from only one of the sources in the **my-sec-multi** secret.

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

public class ConfigSecretKeyRoute extends RouteBuilder {
  @Override
```

```
    public void configure() throws Exception {

      from("timer:secret")
        .setBody()
            .simple("resource:classpath:my-secret-key-2")
        .log("secret content is: ${body}");
    }
}
```

2. Run the integration by using the **--config secret** option and the **/key** notation.
For example:

```
kamel run --config secret:my-sec-multi/my-secret-key-2 ConfigSecretKeyRoute.java --dev
```

3. Check the integration pod to verify that only the specified source (for example, **my-secret-key-2**) is mounted.
For example, run the following command to list all volumes for a pod:

```
oc set volume pod/<pod-name> --all
```

4.2.3. Providing resources to a running integration

You can provide a resource for the integration to use when it is running by specifying the kamel run command's **--resource** option. You can specify a local text file (1 MiB maximum file size), a ConfigMap (3MB) or a Secret (3MB). You can optionally specify the destination of the file that is materialized for the resource. For example, if you want to set an HTTPS connection, you use the --resource option because you must provide an SSL certificate which is a binary file that is expected in a known location.

When you use the **--resource** option, the Camel K operator does not parse the resource looking for runtime properties and it does not add the resource to the classpath. (If you want to add the resource to the classpath, you can use the JVM trait.

4.2.3.1. Specifying a text or binary file as a resource

If you have a text or binary file that contains configuration values, you can use the **--resource file:/path/to/file** option to materialize the file. By default, the Camel K operator copies the materialized file to the **/etc/camel/resources/** directory. Optionally, you can specify a different destination directory as described in Specifying a destination path for a resource .

Prerequisites

- Setting up your Camel K development environment
- You have one or more text or binary files that contain configuration properties.

Procedure

1. Create a Camel K integration that reads the contents of a file that you provide.
For example, the following integration (**ResourceFileBinaryRoute.java**) unzips and reads the **resources-data.zip** file:

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

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

    from("file:/etc/camel/resources/?fileName=resources-
data.zip&noop=true&idempotent=false")
        .unmarshal().zipFile()
        .log("resource file unzipped content is: ${body}");

  }
}
```

2. Run the integration and use the **--resource** option to copy the file to the default destination directory (**/etc/camel/resources/**). For example:

```
kamel run --resource file:resources-data.zip ResourceFileBinaryRoute.java -d camel-zipfile --
dev
```

Note: If you specify a binary file, a binary representation of the contents of the file is created and decoded transparently in the integration.

Optionally, you can provide more than one resource by adding the **--resource** option repeatedly, for example:

```
kamel run --resource file:resources-data1.txt --resource file:resources-data2.txt
ResourceFileBinaryRoute.java -d camel-zipfile --dev
```

4.2.3.2. Specifying a ConfigMap as a resource

If you have an OpenShift ConfigMap that contains configuration values, and you need to materialize the ConfigMap as a resource for an integration, use the **--resource <configmap-file>** option.

Prerequisites

- Setting up your Camel K development environment
- You have one or more ConfigMap files stored on your OpenShift cluster. For example, you can create a ConfigMap by using the following command:

```
oc create configmap my-cm --from-literal=my-configmap-key="configmap content"
```

Procedure

1. Create a Camel K integration that references a ConfigMap stored on your OpenShift cluster. For example, the following integration (named **ResourceConfigmapRoute.java**) references a ConfigMap named **my-cm** that contains **my-configmap-key**.

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

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

    from("file:/etc/camel/resources/my-cm/?fileName=my-configmap-
```

```
key&noop=true&idempotent=false")
        .log("resource file content is: ${body}");

  }
}
```

2. Run the integration and use the **--resource** option to materialize the ConfigMap file in the default **/etc/camel/resources/** directory so that it is available to the running integration.
For example:

```
kamel run --resource configmap:my-cm ResourceConfigmapRoute.java --dev
```

When the integration starts, the Camel K operator mounts a volume with the ConfigMap's content (for example, **my-configmap-key**).

Note: If you specify a ConfigMap that is not yet available on the cluster, the Integration waits and starts only after the ConfigMap becomes available.

4.2.3.3. Specifying a Secret as a resource

If you have an OpenShift Secret that contains configuration information, and you need to materialize it as a resource that is available to one or more integrations, use the **--resource <secret>** syntax.

Prerequisites

- Setting up your Camel K development environment
- You have one or more Secrets files stored on your OpenShift cluster. For example, you can create a Secret by using the following command:

```
oc create secret generic my-sec --from-literal=my-secret-key="very top secret"
```

Procedure

1. Create a Camel K integration that references a Secret stored on your OpenShift cluster.
For example, the following integration (named **ResourceSecretRoute.java**) references the **my-sec** Secret:

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

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

      from("file:/etc/camel/resources/my-sec/?fileName=my-secret-
key&noop=true&idempotent=false")
          .log("resource file content is: ${body}");

  }
}
```

2. Run the integration and use the **--resource** option to materialize the Secret in the default **/etc/camel/resources/** directory so that it is available to the running integration.
For example:

```
kamel run --resource secret:my-sec ResourceSecretRoute.java --dev
```

When the integration starts, the Camel K operator mounts a volume with the Secret's content (for example, **my-sec**).

Note: If you specify a Secret that is not yet available on the cluster, the Integration waits and starts only after the Secret becomes available.

4.2.3.4. Specifying a destination path for a resource

The **/etc/camel/resources/** directory is the default location for mounting a resource that you specify with the **--resource** option. If you need to specify a different directory on which to mount a resource, use the **--resource @path** syntax.

Prerequisites

- Setting up your Camel K development environment
- You have a file, ConfigMap, or Secret that contains one or more configuration properties.

Procedure

1. Create a Camel K integration that references the file, ConfigMap or Secret that contains configuration properties. For example, the following integration (named **ResourceFileLocationRoute.java**) references the **myprops** file:

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

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

      from("file:/tmp/?fileName=input.txt&noop=true&idempotent=false")
        .log("resource file content is: ${body}");

  }
}
```

2. Run the integration and use the **--resource** option with the **@path** syntax and specify where to mount the resource content (either a file, ConfigMap or Secret):
For example, the following command specifies to use the **/tmp** directory to mount the **input.txt** file:

```
kamel run --resource file:resources-data.txt@/tmp/input.txt ResourceFileLocationRoute.java --dev
```

3. Check the integration's pod to verify that the file (for example, **input.txt**) was mounted in the correct location (for example, in the **tmp** directory). For example, run the following command:

```
oc exec <pod-name> -- cat /tmp/input.txt
```

4.2.3.5. Filtering ConfigMap or Secret data

When you create a ConfigMap or a Secret, you can specify more than one source of information. For example, the following command creates a ConfigMap (named **my-cm-multi**) from two sources:

```
oc create configmap my-cm-multi --from-literal=my-configmap-key="configmap content" --from-literal=my-configmap-key-2="another content"
```

When you run an integration with the **--resource** option, a ConfigMap or Secret that was created with more than one source, by default, both sources are materialized.

If you want to limit the quantity of information to recover from a ConfigMap or Secret, you can specify the **--resource** option's /key notation after the ConfigMap or Secret name. For example, **--resource configmap:my-cm/my-key** or **--resource secret:my-secret/my-key**.

You can limit the quantity of information that your integration retrieves to just one resource by using the /key notation after with the **--resource configmap** or **--resource secret** options.

Prerequisites

- Setting up your Camel K development environment
- You have a ConfigMap or a Secret that holds values from more than one source.

Procedure

1. Create an integration that uses configuration values from only one of the resources in the ConfigMap or Secret. For example, the following integration (named **ResourceConfigmapKeyLocationRoute.java**) references the **my-cm-multi** ConfigMap:

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

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

    from("file:/tmp/app/data/?fileName=my-configmap-key-2&noop=true&idempotent=false")
      .log("resource file content is: ${body} consumed from
       ${header.CamelFileName}");

  }
}
```

2. Run the integration and use the **--resource** option with the **@path** syntax and specify where to mount the source content (either a file, ConfigMap or Secret):
   For example, the following command specifies to use only one of the sources (**my-configmap-key-2@**) contained within the ConfigMap and to use the **/tmp/app/data** directory to mount it:

```
kamel run --resource configmap:my-cm-multi/my-configmap-key-2@/tmp/app/data
ResourceConfigmapKeyLocationRoute.java --dev
```

3. Check the integration's pod to verify that only one file (for example, **my-configmap-key-2**) was mounted in the correct location (for example, in the **/tmp/app/data** directory). For example, run the following command:

```
oc exec <pod-name> -- cat /tmp/app/data/my-configmap-key-2
```

4.3. 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

- 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

- Providing runtime properties at the command line
- Apache Camel SEDA component

4.4. 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

- 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.

Types of Dependencies

The **-d** flag of the **kamel run** command is flexible and support multiple kind of dependencies.

Camel dependencies can be added directly using the **-d** flag like this:

```
kamel run -d camel:http Integration.java
```

In this case, the dependency will be added with the correct version. Note that the standard notation for specifying a Camel dependency is **camel:xxx**, while **kamel** also accepts **camel-xxx** for usability.

You can add **External dependencies** using the **-d** flag, the **mvn** prefix, and the maven coordinates:

```
kamel run -d mvn:com.google.guava:guava:26.0-jre Integration.java
```

Note that if your dependencies belong to a private repository, this repository must be defined. See Configure maven.

You can add **Local dependencies** using the **-d** flag and the **file://** prefix.

```
kamel run -d file://path/to/integration-dep.jar Integration.java
```

The content of **integration-dep.jar** will then be accessible in your integration for you to use.

You can also specify data files to be mounted in the running container:

```
kamel run -d file://path/to/data.csv:path/in/container/data.csv Integration.java
```

Specifying a directory will work recursively.

Note that this feature relies on the Image Registry being setup accurately.

Jitpack Dependencies

If your dependency is not published in a **maven** repository, you will find **Jitpack** as a way to provide any

custom dependency to your runtime Integration environment. In certain occasion, you will find it useful to include not only your route definition, but also some helper class or any other class which has to be used while defining the Integration behavior. With **Jitpack** you will be able to compile on the fly a java project hosted in a remote repository and use the produced package as a dependency of your Integration.

The usage is the same as defined above for any maven dependency. It can be added using the **-d** flag, but, this time, you need to define the prefix as expected for the project repository you are using (that is, **github**). It has to be provided in the form **repository-kind:user/repo/version.** As an example, you can provide the Apache Commons CSV dependency by executing:

```
kamel run -d github:apache/commons-csv/1.1 Integration.java
```

We support the most important public code repositories:

```
github:user/repo/version
gitlab:user/repo/version
bitbucket:user/repo/version
gitee:user/repo/version
azure:user/repo/version
```

The **version** can be omitted when you are willing to use the **main** branch. Else, it will represent the branch or tag used in the project repo.

Dynamic URIs

Camel K does not always discover all of your dependencies. When you are creating an URI dynamically, you must instruct Camel K which component to load (using the **-d** parameter). The following code snippet illustrates this.

DynamicURI.java

```
String myTopic = "purchases"
from("kafka:" + myTopic + "? ... ")
    .to(...)
...
```

Here the **from** URI is dynamically created by some variables that are resolved at runtime. In cases like this, you must specify the component and the related dependency to load into the **Integration.**

Additional resources

- Running Camel K integrations in development mode
- Camel K trait and profile configuration
- Apache Camel Mail component
- Apache Camel SEDA component

CHAPTER 5. AUTHENTICATING CAMEL K AGAINST KAFKA

You can authenticate Camel K against Apache Kafka.

The following example demonstrates how to set up a Kafka Topic and use it in a simple Producer/Consumer pattern Integration.

5.1. SETTING UP KAFKA

To set up Kafka, you must:

1. Install the required OpenShift operators
2. Create a Kafka instance
3. Create a Kafka topic

Use the Red Hat product mentioned below to set up Kafka:

- **Red Hat Advanced Message Queuing (AMQ) streams**- A self-managed Apache Kafka offering. AMQ Streams is based on open source Strimzi and is included as part of Red Hat Integration. AMQ Streams is a distributed and scalable streaming platform based on Apache Kafka that includes a publish/subscribe messaging broker. Kafka Connect provides a framework to integrate Kafka-based systems with external systems. Using Kafka Connect, you can configure source and sink connectors to stream data from external systems into and out of a Kafka broker.

5.1.1. Setting up Kafka by using AMQ streams

AMQ Streams simplifies the process of running Apache Kafka in an OpenShift cluster.

5.1.1.1. Preparing your OpenShift cluster for AMQ Streams

To use Camel K or Kamelets and Red Hat AMQ Streams, you must install the following operators and tools:

- **Red Hat Integration - AMQ Streams**operator - Manages the communication between your Openshift Cluster and AMQ Streams for Apache Kafka instances.
- **Red Hat Integration - Camel K**operator - Installs and manages Camel K - a lightweight integration framework that runs natively in the cloud on OpenShift.
- **Camel K CLI** tool - Allows you to access all Camel K features.

Prerequisites

- You are familiar with Apache Kafka concepts.
- You can access an OpenShift 4.6 (or later) cluster with the correct access level, the ability to create projects and install operators, and the ability to install the OpenShift and the Camel K CLI on your local system.
- You installed the OpenShift CLI tool (**oc**) so that you can interact with the OpenShift cluster at the command line.

Procedure

To set up Kafka by using AMQ Streams:

1. Log in to your OpenShift cluster's web console.
2. Create or open a project in which you plan to create your integration, for example **my-camel-k-kafka**.
3. Install the Camel K operator and Camel K CLI as described in Installing Camel K .
4. Install the AMQ streams operator:
    a. From any project, select **Operators** > **OperatorHub**.
    b. In the **Filter by Keyword** field, type **AMQ Streams**.
    c. Click the **Red Hat Integration - AMQ Streams**card and then click **Install**.
    The **Install Operator** page opens.
    d. Accept the defaults and then click **Install**.
5. Select **Operators** > **Installed Operators** to verify that the Camel K and AMQ Streams operators are installed.

Next steps

Setting up a Kafka topic with AMQ Streams

5.1.1.2. Setting up a Kafka topic with AMQ Streams

A Kafka topic provides a destination for the storage of data in a Kafka instance. You must set up a Kafka topic before you can send data to it.

Prerequisites

- You can access an OpenShift cluster.
- You installed the **Red Hat Integration - Camel K**and **Red Hat Integration - AMQ Streams** operators as described in Preparing your OpenShift cluster .
- You installed the OpenShift CLI (**oc**) and the Camel K CLI (**kamel**).

Procedure

To set up a Kafka topic by using AMQ Streams:

1. Log in to your OpenShift cluster's web console.
2. Select **Projects** and then click the project in which you installed the **Red Hat Integration - AMQ Streams** operator. For example, click the **my-camel-k-kafka** project.
3. Select **Operators** > **Installed Operators** and then click **Red Hat Integration - AMQ Streams**
4. Create a Kafka cluster:
    a. Under **Kafka**, click **Create instance**.

b. Type a name for the cluster, for example **kafka-test**.

c. Accept the other defaults and then click **Create**.
The process to create the Kafka instance might take a few minutes to complete.

When the status is ready, continue to the next step.

5. Create a Kafka topic:

a. Select **Operators** > **Installed Operators** and then click **Red Hat Integration - AMQ Streams**.

b. Under **Kafka Topic**, click **Create Kafka Topic**.

c. Type a name for the topic, for example **test-topic**.

d. Accept the other defaults and then click **Create**.

5.1.2. Setting up Kafka by using OpenShift streams

To use OpenShift Streams for Apache Kafka, you must be logged into your Red Hat account.

5.1.2.1. Preparing your OpenShift cluster for OpenShift Streams

To use managed cloud service, you must install the following operators and tools:

- **OpenShift Application Services (RHOAS) CLI** - Allows you to manage your application services from a terminal.
- **Red Hat Integration - Camel K**operator Installs and manages Camel K - a lightweight integration framework that runs natively in the cloud on OpenShift.
- **Camel K CLI tool** - Allows you to access all Camel K features.

Prerequisites

- You are familiar with Apache Kafka concepts.
- You can access an OpenShift 4.6 (or later) cluster with the correct access level, the ability to create projects and install operators, and the ability to install the OpenShift and Apache Camel K CLI on your local system.
- You installed the OpenShift CLI tool (**oc**) so that you can interact with the OpenShift cluster at the command line.

Procedure

1. Log in to your OpenShift web console with a cluster admin account.
2. Create the OpenShift project for your Camel K or Kamelets application.

a. Select **Home** > **Projects**.

b. Click **Create Project**.

c. Type the name of the project, for example **my-camel-k-kafka**, then click **Create**.

3. Download and install the RHOAS CLI as described in Getting started with the rhoas CLI.
4. Install the Camel K operator and Camel K CLI as described in Installing Camel K .
5. To verify that the **Red Hat Integration - Camel K**operator is installed, click **Operators** > **Installed Operators**.

Next step

Setting up a Kafka topic with RHOAS

5.1.2.2. Setting up a Kafka topic with RHOAS

Kafka organizes messages around *topics*. Each topic has a name. Applications send messages to topics and retrieve messages from topics. A Kafka topic provides a destination for the storage of data in a Kafka instance. You must set up a Kafka topic before you can send data to it.

Prerequisites

- You can access an OpenShift cluster with the correct access level, the ability to create projects and install operators, and the ability to install the OpenShift and the Camel K CLI on your local system.
- You installed the OpenShift CLI (**oc**) , the Camel K CLI (**kamel**) , and RHOAS CLI (**rhoas**) tools as described in Preparing your OpenShift cluster .
- You installed the **Red Hat Integration - Camel K**operator as described in Preparing your OpenShift cluster.
- You are logged in to the Red Hat Cloud site .

Procedure

To set up a Kafka topic:

1. From the command line, log in to your OpenShift cluster.
2. Open your project, for example:
   oc project my-camel-k-kafka
3. Verify that the Camel K operator is installed in your project:
   oc get csv

   The result lists the Red Hat Camel K operator and indicates that it is in the **Succeeded** phase.
4. Prepare and connect a Kafka instance to RHOAS:
   a. Login to the RHOAS CLI by using this command:
      rhoas login
   b. Create a kafka instance, for example **kafka-test**:
      rhoas kafka create kafka-test

      The process to create the Kafka instance might take a few minutes to complete.
5. To check the status of your Kafka instance:
   rhoas status

You can also view the status in the web console:

https://cloud.redhat.com/application-services/streams/kafkas/

When the status is **ready**, continue to the next step.

6. Create a new Kafka topic:
rhoas kafka topic create --name test-topic

7. Connect your Kafka instance (cluster) with the Openshift Application Services instance:
rhoas cluster connect

8. Follow the script instructions for obtaining a credential token.
You should see output similar to the following:

```
Token Secret "rh-cloud-services-accesstoken-cli" created successfully
Service Account Secret "rh-cloud-services-service-account" created successfully
KafkaConnection resource "kafka-test" has been created
KafkaConnection successfully installed on your cluster.
```

Next step

- Obtaining Kafka credentials

5.1.2.3. Obtaining Kafka credentials

To connect your applications or services to a Kafka instance, you must first obtain the following Kafka credentials:

- Obtain the bootstrap URL.
- Create a service account with credentials (username and password).

For OpenShift Streams, the authentication protocol is SASL_SSL.

Prerequisite

- You have created a Kafka instance, and it has a ready status.
- You have created a Kafka topic.

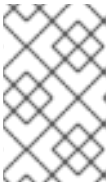
Procedure

1. Obtain the Kafka Broker URL (Bootstrap URL):
rhoas status

This command returns output similar to the following:

```
Kafka
------------------------------------------------------------
ID:                 1ptdfZRHmLKwqW6A3YKM2MawgDh
Name:                 my-kafka
Status:               ready
Bootstrap URL:        my-kafka--ptdfzrhmlkwqw-a-ykm-mawgdh.kafka.devshift.org:443
```

2. To obtain a username and password, create a service account by using the following syntax:
rhoas service-account create --name "<account-name>" --file-format json



NOTE

When creating a service account, you can choose the file format and location to save the credentials. For more information, type **rhoas service-account create --help**

For example:

rhoas service-account create --name "my-service-acct" --file-format json

The service account is created and saved to a JSON file.

3. To verify your service account credentials, view the **credentials.json** file:
cat credentials.json

This command returns output similar to the following:

```
{"clientID":"srvc-acct-eb575691-b94a-41f1-ab97-50ade0cd1094", "password":"facf3df1-3c8d-4253-aa87-8c95ca5e1225"}
```

4. Grant permission for sending and receiving messages to or from the Kakfa topic. Use the following command, where **clientID** is the value provided in the **credentials.json** file (from Step 3).

```
rhoas kafka acl grant-access --producer --consumer --service-account $CLIENT_ID --topic test-topic --group all
```

For example:

```
rhoas kafka acl grant-access --producer --consumer --service-account srvc-acct-eb575691-b94a-41f1-ab97-50ade0cd1094 --topic test-topic --group all
```

5.1.2.4. Creating a secret by using the SASL/Plain authentication method

You can create a secret with the credentials that you obtained (Kafka bootstrap URL, service account ID, and service account secret).

Procedure

1. Edit the **application.properties** file and add the Kafka credentials.

application.properties file

```
camel.component.kafka.brokers = <YOUR-KAFKA-BOOTSTRAP-URL-HERE>
camel.component.kafka.security-protocol = SASL_SSL
camel.component.kafka.sasl-mechanism = PLAIN
camel.component.kafka.sasl-jaas-config=org.apache.kafka.common.security.plain.PlainLoginModule required username='<YOUR-SERVICE-ACCOUNT-ID-HERE>' password='<YOUR-SERVICE-
```

```
ACCOUNT-SECRET-HERE>';
consumer.topic=<TOPIC-NAME>
producer.topic=<TOPIC-NAME>
```

2. Run the following command to create a secret that contains the sensitive properties in the **application.properties** file:

```
oc create secret generic kafka-props --from-file application.properties
```

You use this secret when you run a Camel K integration.

See Also

The Camel K Kafka Basic Quickstart

5.1.2.5. Creating a secret by using the SASL/OAUTHBearer authentication method

You can create a secret with the credentials that you obtained (Kafka bootstrap URL, service account ID, and service account secret).

Procedure

1. Edit the **application-oauth.properties** file and add the Kafka credentials.

application-oauth.properties file

```
camel.component.kafka.brokers = <YOUR-KAFKA-BOOTSTRAP-URL-HERE>
camel.component.kafka.security-protocol = SASL_SSL
camel.component.kafka.sasl-mechanism = OAUTHBEARER
camel.component.kafka.sasl-jaas-config =
org.apache.kafka.common.security.oauthbearer.OAuthBearerLoginModule required \
oauth.client.id='<YOUR-SERVICE-ACCOUNT-ID-HERE>' \
oauth.client.secret='<YOUR-SERVICE-ACCOUNT-SECRET-HERE>' \
oauth.token.endpoint.uri="https://identity.api.openshift.com/auth/realms/rhoas/protocol/openid-connect/token" ;
camel.component.kafka.additional-
properties[sasl.login.callback.handler.class]=io.strimzi.kafka.oauth.client.JaasClientOauthLoginC
allbackHandler

consumer.topic=<TOPIC-NAME>
producer.topic=<TOPIC-NAME>
```

2. Run the following command to create a secret that contains the sensitive properties in the **application.properties** file:

```
oc create secret generic kafka-props --from-file application-oauth.properties
```

You use this secret when you run a Camel K integration.

See Also

The Camel K Kafka Basic Quickstart

5.2. RUNNING A KAFKA INTEGRATION

Running a producer integration

1. Create a sample producer integration. This fills the topic with a message, every 10 seconds.

   Sample SaslSSLKafkaProducer.java

```
// kamel run --secret kafka-props SaslSSLKafkaProducer.java --dev
// camel-k: language=java dependency=mvn:org.apache.camel.quarkus:camel-quarkus-kafka dependency=mvn:io.strimzi:kafka-oauth-client:0.7.1.redhat-00003

import org.apache.camel.builder.RouteBuilder;
import org.apache.camel.component.kafka.KafkaConstants;

public class SaslSSLKafkaProducer extends RouteBuilder {
  @Override
  public void configure() throws Exception {
  log.info("About to start route: Timer -> Kafka ");
  from("timer:foo")
    .routeId("FromTimer2Kafka")
    .setBody()
      .simple("Message #${exchangeProperty.CamelTimerCounter}")
    .to("kafka:{{producer.topic}}")
    .log("Message correctly sent to the topic!");
  }
}
```

2. Then run the procedure integration.

```
kamel run --secret kafka-props SaslSSLKafkaProducer.java --dev
```

   The producer will create a new message and push into the topic and log some information.

```
[2] 2021-05-06 08:48:11,854 INFO  [FromTimer2Kafka] (Camel (camel-1) thread #1 - KafkaProducer[test]) Message correctly sent to the topic!
[2] 2021-05-06 08:48:11,854 INFO  [FromTimer2Kafka] (Camel (camel-1) thread #3 - KafkaProducer[test]) Message correctly sent to the topic!
[2] 2021-05-06 08:48:11,973 INFO  [FromTimer2Kafka] (Camel (camel-1) thread #5 - KafkaProducer[test]) Message correctly sent to the topic!
[2] 2021-05-06 08:48:12,970 INFO  [FromTimer2Kafka] (Camel (camel-1) thread #7 - KafkaProducer[test]) Message correctly sent to the topic!
[2] 2021-05-06 08:48:13,970 INFO  [FromTimer2Kafka] (Camel (camel-1) thread #9 - KafkaProducer[test]) Message correctly sent to the topic!
```

Running a consumer integration

1. Create a consumer integration.

   Sample SaslSSLKafkaProducer.java

```
// kamel run --secret kafka-props SaslSSLKafkaConsumer.java --dev
// camel-k: language=java dependency=mvn:org.apache.camel.quarkus:camel-quarkus-kafka dependency=mvn:io.strimzi:kafka-oauth-client:0.7.1.redhat-00003

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

```
public class SaslSSLKafkaConsumer extends RouteBuilder {
  @Override
  public void configure() throws Exception {
 log.info("About to start route: Kafka -> Log ");
 from("kafka:{{consumer.topic}}")
    .routeId("FromKafka2Log")
    .log("${body}");
  }
}
```

2. Open another shell and run the consumer integration using the command:

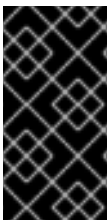
```
kamel run --secret kafka-props SaslSSLKafkaConsumer.java --dev
```

A consumer will start logging the events found in the Topic:

```
[1] 2021-05-06 08:51:08,991 INFO  [FromKafka2Log] (Camel (camel-1) thread #0 -
KafkaConsumer[test]) Message #8
[1] 2021-05-06 08:51:10,065 INFO  [FromKafka2Log] (Camel (camel-1) thread #0 -
KafkaConsumer[test]) Message #9
[1] 2021-05-06 08:51:10,991 INFO  [FromKafka2Log] (Camel (camel-1) thread #0 -
KafkaConsumer[test]) Message #10
[1] 2021-05-06 08:51:11,991 INFO  [FromKafka2Log] (Camel (camel-1) thread #0 -
KafkaConsumer[test]) Message #11
```

CHAPTER 6. 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 1.6 includes the **OpenShift** and **Knative** profiles. The **Kubernetes** profile has community-only support. It also includes Java, and YAML DSL support for integrations. Other languages such as XML, Groovy, JavaScript, and Kotlin have community-only support.

This chapter includes the following sections:

Camel K feature traits

- Section 6.2.1, “Knative Trait” - Technology Preview
- Section 6.2.2, “Knative Service Trait”- Technology Preview
- Section 6.2.3, “Prometheus Trait”
- Section 6.2.4, “Pdb Trait”
- Section 6.2.5, “Pull Secret Trait”
- Section 6.2.6, “Route Trait”
- Section 6.2.7, “Service Trait”

Camel K core platform traits

- Section 6.3.1, “Builder Trait”
- Section 6.3.3, “Camel Trait”
- Section 6.3.2, “Container Trait”
- Section 6.3.4, “Dependencies Trait”
- Section 6.3.5, “Deployer Trait”
- Section 6.3.6, “Deployment Trait”
- Section 6.3.7, “Environment Trait”
- Section 6.3.8, “Error Handler Trait”
- Section 6.3.9, “Jvm Trait”
- Section 6.3.10, “Kamelets Trait”
- Section 6.3.11, “NodeAffinity Trait”

- Section 6.3.12, “Openapi Trait”- Technology Preview
- Section 6.3.13, “Owner Trait”
- Section 6.3.14, “Platform Trait”
- Section 6.3.15, “Quarkus Trait”

6.1. 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. Supported profiles are **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.

6.2. CAMEL K FEATURE TRAITS

6.2.1. 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**.

6.2.1.1. 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.

Property	Type	Description
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 header has been removed in newer versions of Knative, filtering is disabled by default.
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 enabled by default when the integration targets a single sink (except when the integration is owned by a Knative source).
knative.auto	**bool**	Enable automatic discovery of all trait properties.

6.2.2. 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**.

6.2.2.1. 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.

Property	Type	Description
knative-service.annotations	**map[string]string**	The annotations are added to route. This can be used to set knative service specific annotations. For more details see, Route Specific Annotations. CLI usage example: -t "knative-service.annotations.'haproxy.router.openshift.io/balance'=roundrobin"
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.
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)

6.2.3. Prometheus Trait

The Prometheus trait configures a Prometheus-compatible endpoint. It also creates a **PodMonitor** resource, so that the endpoint can be scraped automatically, when using the Prometheus operator.

The metrics are exposed using MicroProfile Metrics.

WARNING

The creation of the **PodMonitor** resource requires the Prometheus Operator custom resource definition to be installed. You can set **pod-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**.

6.2.3.1. 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.pod-monitor	**bool**	Whether a **PodMonitor** resource is created (default **true**).
prometheus.pod-monitor-labels	**[]string**	The **PodMonitor** resource labels, applicable when **pod-monitor** is **true**.

6.2.4. Pdb Trait

The PDB trait allows to configure the PodDisruptionBudget resource for the Integration pods.

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

6.2.4.1. Configuration

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

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

The following configuration options are available:

Property	Type	Description
pdb.enabled	**bool**	Can be used to enable or disable a trait. All traits share this common property.
pdb.min-available	**string**	The number of pods for the Integration that must still be available after an eviction. It can be either an absolute number or a percentage. Only one of **min-available** and **max-unavailable** can be specified.
pdb.max-unavailable	**string**	The number of pods for the Integration that can be unavailable after an eviction. It can be either an absolute number or a percentage (default **1** if **min-available** is also not set). Only one of **max-unavailable** and **min-available** can be specified.

6.2.5. Pull Secret Trait

The Pull Secret trait sets a pull secret on the pod, to allow Kubernetes to retrieve the container image from an external registry.

The pull secret can be specified manually or, in case you've configured authentication for an external container registry on the **IntegrationPlatform**, the same secret is used to pull images.

It's enabled by default whenever you configure authentication for an external container registry, so it assumes that external registries are private.

If your registry does not need authentication for pulling images, you can disable this trait.

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

6.2.5.1. Configuration

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

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

The following configuration options are available:

Property	Type	Description
pull-secret.enabled	**bool**	Can be used to enable or disable a trait. All traits share this common property.
pull-secret.secret-name	**string**	The pull secret name to set on the Pod. If left empty this is automatically taken from the **IntegrationPlatform** registry configuration.
pull-secret.image-puller-delegation	**bool**	When using a global operator with a shared platform, this enables delegation of the **system:image-puller** cluster role on the operator namespace to the integration service account.

Property	Type	Description
pull-secret.auto	**bool**	Automatically configures the platform registry secret on the pod if it is of type **kubernetes.io/dockerconfigjson**.

6.2.6. Route Trait

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

The certificate and key contents may be sourced either from the local filesystem or in a Openshift **secret** object. The user may use the parameters ending in **-secret** (example: **tls-certificate-secret**) to reference a certificate stored in a **secret**. Parameters ending in **-secret** have higher priorities and in case the same route parameter is set, for example: **tls-key-secret** and **tls-key**, then **tls-key-secret** is used. The recommended approach to set the key and certificates is to use **secrets** to store their contents and use the following parameters to reference them: **tls-certificate-secret**, **tls-key-secret**, **tls-ca-certificate-secret**, **tls-destination-ca-certificate-secret** See the examples section at the end of this page to see the setup options.

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

6.2.6.1. 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.annotations	**map[string]string**	The annotations are added to route. This can be used to set route specific annotations. For annotations options see Route Specific Annotations. CLI usage example: -t "route.annotations.'haproxy.router.openshift.io/balance'=roundrobin
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 route documentation for additional information.

Property	Type	Description
route.tls-certificate	**string**	The TLS certificate contents. Refer to the OpenShift route documentation for additional information.
route.tls-certificate-secret	**string**	The secret name and key reference to the TLS certificate. The format is "secret-name[/key-name]", the value represents the secret name, if there is only one key in the secret it will be read, otherwise you can set a key name separated with a "/". Refer to the OpenShift route documentation for additional information.
route.tls-key	**string**	The TLS certificate key contents. Refer to the OpenShift route documentation for additional information.
route.tls-key-secret	**string**	The secret name and key reference to the TLS certificate key. The format is "secret-name[/key-name]", the value represents the secret name, if there is only one key in the secret it will be read, otherwise you can set a key name separated with a "/". Refer to the OpenShift route documentation for additional information.
route.tls-ca-certificate	**string**	The TLS CA certificate contents. Refer to the OpenShift route documentation for additional information.
route.tls-ca-certificate-secret	**string**	The secret name and key reference to the TLS CA certificate. The format is "secret-name[/key-name]", the value represents the secret name, if there is only one key in the secret it will be read, otherwise you can set a key name separated with a "/". Refer to the OpenShift route 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 route documentation for additional information.

Property	Type	Description
route.tls-destination-ca-certificate-secret	**string**	The secret name and key reference to the destination CA certificate. The format is "secret-name[/key-name]", the value represents the secret name, if there is only one key in the secret it will be read, otherwise you can set a key name separated with a "/". Refer to the OpenShift route documentation for additional information.
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 route documentation for additional information.

6.2.6.2. Examples

These examples uses **secrets** to store the certificates and keys to be referenced in the integrations. Read Openshift route documentation for detailed information about routes. The PlatformHttpServer.java is the integration example.

As a requirement to run these examples, you should have a **secret** with a key and certificate.

6.2.6.2.1. Generate a self-signed certificate and create a secret

```
openssl genrsa -out tls.key
openssl req -new -key tls.key -out csr.csr -subj "/CN=my-server.com"
openssl x509 -req -in csr.csr -signkey tls.key -out tls.crt
oc create secret tls my-combined-certs --key=tls.key  --cert=tls.crt
```

6.2.6.2.2. Making an HTTP request to the route

For all examples, you can use the following curl command to make an HTTP request. It makes use of inline scripts to retrieve the openshift namespace and cluster base domain, if you are using a shell which doesn't support these inline scripts, you should replace the inline scripts with the values of your actual namespace and base domain.

```
curl -k https://platform-http-server-`oc config view --minify -o 'jsonpath={..namespace}'`.`oc get dnses/cluster -ojsonpath='{.spec.baseDomain}'`/hello?name=Camel-K
```

- To add an **edge** route using secrets, use the parameters ending in **-secret** to set the secret name which contains the certificate. This route example trait references a secret named **my-combined-certs** which contains two keys named **tls.key** and **tls.crt**.

  ```
  kamel run --dev PlatformHttpServer.java -t route.tls-termination=edge -t route.tls-certificate-secret=my-combined-certs/tls.crt -t route.tls-key-secret=my-combined-certs/tls.key
  ```

- To add a **passthrough** route using secrets, the TLS is setup in the integration pod, the keys and certificates should be visible in the running integration pod, to achieve this we are using the **--resource** kamel parameter to mount the secret in the integration pod, then we use some camel

quarkus parameters to reference these certificate files in the running pod, they start with **-p quarkus.http.ssl.certificate**. This route example trait references a secret named **my-combined-certs** which contains two keys named **tls.key** and **tls.crt**.

```
kamel run --dev PlatformHttpServer.java --resource secret:my-combined-certs@/etc/ssl/my-combined-certs -p quarkus.http.ssl.certificate.file=/etc/ssl/my-combined-certs/tls.crt -p quarkus.http.ssl.certificate.key-file=/etc/ssl/my-combined-certs/tls.key -t route.tls-termination=passthrough -t container.port=8443
```

- To add a **reencrypt** route using secrets, the TLS is setup in the integration pod, the keys and certificates should be visible in the running integration pod, to achieve this we are using the **--resource** kamel parameter to mount the secret in the integration pod, then we use some camel quarkus parameters to reference these certificate files in the running pod, they start with **-p quarkus.http.ssl.certificate**. This route example trait references a secret named **my-combined-certs** which contains two keys named **tls.key** and **tls.crt**.

```
kamel run --dev PlatformHttpServer.java --resource secret:my-combined-certs@/etc/ssl/my-combined-certs  -p quarkus.http.ssl.certificate.file=/etc/ssl/my-combined-certs/tls.crt -p quarkus.http.ssl.certificate.key-file=/etc/ssl/my-combined-certs/tls.key -t route.tls-termination=reencrypt -t route.tls-destination-ca-certificate-secret=my-combined-certs/tls.crt -t route.tls-certificate-secret=my-combined-certs/tls.crt -t route.tls-key-secret=my-combined-certs/tls.key -t container.port=8443
```

- To add a **reencrypt** route using a specific certificate from a secret for the route and Openshift service serving certificates for the integration endpoint. This way the Openshift service serving certificates is set up only in the integration pod. The keys and certificates should be visible in the running integration pod, to achieve this we are using the **--resource** kamel parameter to mount the secret in the integration pod, then we use some camel quarkus parameters to reference these certificate files in the running pod, they start with **-p quarkus.http.ssl.certificate**. This route example trait references a secret named **my-combined-certs** which contains two keys named **tls.key** and **tls.crt**.

```
kamel run --dev PlatformHttpServer.java --resource secret:cert-from-openshift@/etc/ssl/cert-from-openshift  -p quarkus.http.ssl.certificate.file=/etc/ssl/cert-from-openshift/tls.crt -p quarkus.http.ssl.certificate.key-file=/etc/ssl/cert-from-openshift/tls.key -t route.tls-termination=reencrypt -t route.tls-certificate-secret=my-combined-certs/tls.crt -t route.tls-key-secret=my-combined-certs/tls.key -t container.port=8443
```

Then you should annotate the integration service to inject the Openshift service serving certificates

```
oc annotate service platform-http-server service.beta.openshift.io/serving-cert-secret-name=cert-from-openshift
```

- To add an **edge** route using a certificate and a private key provided from your local filesystem. This example uses inline scripts to read the certificate and private key file contents, then remove all new line characters, (this is required to set the certificate as parameter's values), so the values are in a single line.

```
kamel run PlatformHttpServer.java --dev -t route.tls-termination=edge -t route.tls-certificate="$(cat tls.crt|awk 'NF {sub(/\r/, ""); printf "%s\\n",$0;}')" -t route.tls-key="$(cat tls.key|awk 'NF {sub(/\r/, ""); printf "%s\\n",$0;}')"
```

6.2.7. 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**.

6.2.7.1. 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 (default **false**).

6.3. CAMEL K PLATFORM TRAITS

6.3.1. 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.

6.3.1.1. 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.
builder.properties	**[]string**	A list of properties to be provided to the build task

6.3.2. 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.

6.3.2.1. 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.

Property	Type	Description
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.image	**string**	The main container image
container.probes-enabled	**bool**	ProbesEnabled enable/disable probes on the container (default **false**)
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.
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.

Property	Type	Description
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.

6.3.3. 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.

6.3.3.1. 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.

6.3.4. 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.

6.3.4.1. 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.

6.3.5. 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.

6.3.5.1. 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.

Property	Type	Description
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.

6.3.6. 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.

6.3.6.1. 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.

6.3.7. 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.

6.3.7.1. 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**	Enables injection of **NAMESPACE** and **POD_NAME** environment variables (default **true**)

6.3.8. Error Handler Trait

The error-handler is a platform trait used to inject Error Handler source into the integration runtime.

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

WARNING

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

6.3.8.1. Configuration

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

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

The following configuration options are available:

Property	Type	Description
error-handler.enabled	**bool**	Can be used to enable or disable a trait. All traits share this common property.
error-handler.ref	**string**	The error handler ref name provided or found in application properties

6.3.9. 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.

6.3.9.1. 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
jvm.classpath	**string**	Additional JVM classpath (use **Linux** classpath separator)

6.3.9.2. Examples

- Include an additional classpath to the **Integration**:

```
$ kamel run -t jvm.classpath=/path/to/my-dependency.jar:/path/to/another-dependency.jar ...
```

6.3.10. Kamelets Trait

The kamelets trait is a platform trait used to inject Kamelets into the integration runtime.

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

WARNING

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

6.3.10.1. Configuration

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

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

The following configuration options are available:

Property	Type	Description
kamelets.enabled	**bool**	Can be used to enable or disable a trait. All traits share this common property.
kamelets.auto	**bool**	Automatically inject all referenced Kamelets and their default configuration (enabled by default)
kamelets.list	**string**	Comma separated list of Kamelet names to load into the current integration

6.3.11. NodeAffinity Trait

The NodeAffinity trait enables you to constrain the nodes that the integration pods are eligible to schedule on, through the following paths:

- 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.

This trait is disabled by default.

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

6.3.11.1. 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.

6.3.11.2. 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.

6.3.12. 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.

6.3.12.1. 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.

6.3.13. 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.

6.3.13.1. 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

6.3.14. 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.

6.3.14.1. 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:

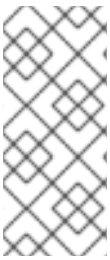
Property	Type	Description
platform.enabled	**bool**	Can be used to enable or disable a trait. All traits share this common property.
platform.create-default	**bool**	To create a default (empty) platform when the platform is missing.
platform.global	**bool**	Indicates if the platform should be created globally in the case of global operator (default true).

Property	Type	Description
platform.auto	**bool**	To automatically detect from the environment if a default platform can be created (it will be created on OpenShift only).

6.3.15. Quarkus Trait

The Quarkus trait activates the Quarkus runtime.

It's enabled by default.



NOTE

Compiling to a native executable, i.e. when using **package-type=native**, is only supported for kamelets, as well as YAML integrations. It also requires at least 4GiB of memory, so the Pod running the native build, that is either the operator Pod, or the build Pod (depending on the build strategy configured for the platform), must have enough memory available.

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

WARNING

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

6.3.15.1. 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.

Property	Type	Description
quarkus.package-type	**[]github.com/apache/camel-k/pkg/trait.quarkusPackageType**	The Quarkus package types, either **fast-jar** or **native** (default **fast-jar**). In case both **fast-jar** and **native** are specified, two **IntegrationKit** resources are created, with the **native** kit having precedence over the **fast-jar** one once ready. The order influences the resolution of the current kit for the integration. The kit corresponding to the first package type will be assigned to the integration in case no existing kit that matches the integration exists.

6.3.15.2. Supported Camel Components

Camel K only supports the Camel components that are available as Camel Quarkus Extensions out-of-the-box.

6.3.15.3. Examples

6.3.15.3.1. Automatic Rollout Deployment to Native Integration

While the compilation to native executables produces integrations that start faster and consume less memory at runtime, the build process is resources intensive, and takes a longer time than the packaging to traditional Java applications.

In order to combine the best of both worlds, it's possible to configure the Quarkus trait to run both traditional and native builds in parallel when running an integration, e.g.:

```
$ kamel run -t quarkus.package-type=fast-jar -t quarkus.package-type=native ...
```

The integration pod will run as soon as the **fast-jar** build completes, and a rollout deployment to the **native** image will be triggered, as soon as the **native** build completes, with no service interruption.

CHAPTER 7. 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 7.1, “Camel K command line”
- Section 7.2, “Camel K modeline options”

7.1. CAMEL K COMMAND LINE

The Camel K CLI provides the **kamel** command as the main entry point for running Camel K integrations on OpenShift.

7.1.1. Supported commands

Note the following key:

Symbol	Description
✔	Supported
▮	Unsupported or not yet supported

Table 7.1. kamel commands

Name	Supported	Description	Example
bind	✔	Bind Kubernetes resources such as Kamelets, in an integration flow, to Knative channels, Kafka topics, or any other endpoint.	**kamel bind telegram-source -p "source.authorizationToken=The Token" channel:mychannel**
completion	▮	Generate completion scripts.	**kamel completion bash**
debug	▮	Debug a remote integration using a local debugger.	**kamel debug my-integration**
delete	✔	Delete an integration deployed on OpenShift.	**kamel delete my-integration**

Name	Supported	Description	Example
describe	✔	Get detailed information on a Camel K resource. This includes an **integration**, **kit**, or **platform**.	**kamel describe integration my-integration**
get	✔	Get the status of integrations deployed on OpenShift.	**kamel get**
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 or YAML.	**kamel init MyIntegration.java**
install	▮	Install Camel K on an OpenShift cluster. **Note:** It is recommended that you use the OpenShift Camel K Operator to install and uninstall Camel K.	**kamel install**
kit	▮	Configure an Integration Kit.	**kamel kit create my-integration --secret**
local	▮	Perform integration actions locally given a set of input integration files.	**kamel local run**
log	✔	Print the logs of a running integration.	**kamel log my-integration**
promote	✔	You can move an integration from one namespace to another.	**kamel promote**

Name	Supported	Description	Example
rebuild	✔	Clear the state of one or more integrations causing a rebuild.	**kamel rebuild my-integration**
reset	✔	Reset the current Camel K installation.	**kamel reset**
run	✔	Run an integration on OpenShift.	**kamel run MyIntegration.java**
uninstall	▮	Uninstall Camel K from an OpenShift cluster. **Note:** It is recommended that you use the OpenShift Camel K Operator to install and uninstall Camel K.	**kamel uninstall**
version	✔	Display Camel-K client version.	**kamel version**

Additional resources

- See Installing Camel K

7.2. 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 Running Camel K integrations using modeline .

All options that are available for the **kamel run** command, you can specify as modeline options.

The following table describes some of the most commonly-used modeline options.

Table 7.2. Camel K modeline options

Option	Description
build-property	Add a build-time property or build-time properties file. Syntax: **[my-key=my-value|file:/path/to/my-conf.properties]**

Option	Description
config	Add a runtime configuration from a Configmap, Secret, or file Syntax: **[configmap|secret|file]:name[/key]** - **name** represents the local file path or the ConfigMap/Secret name. - **key** optionally represents the ConfigMap/Secret key to be filtered.
dependency	Include an external library (for example, a Maven dependency) Example: **dependency=mvn:org.my:app:1.0**
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 runtime property or a runtime properties file. Syntax: **[my-key=my-value|file:/path/to/my-conf.properties])**
resource	Add a run-time resource from a ConfigMap, Secret or file Syntax: **[configmap|secret|file]:name[/key][@path]** - **name** represents the local file path or the ConfigMap/Secret name - **key** (optional) represents the ConfigMap or Secret key to be filtered s - **path** (optional) represents the destination path
trait	Configure a Camel K feature or core capability in a trait. For example, **trait=service.enabled=false**.