Red Hat Integration 2021.Q3

Integrating Applications with Kamelets

Configuring connectors to simplify application integration

Last Updated: 2021-10-14

Red Hat Integration 2021.Q3 Integrating Applications with Kamelets

Configuring connectors to simplify application integration

Legal Notice

Abstract

Kamelets offer an alternative approach to application integration. Instead of using Camel components directly, you can configure kamelets (opinionated route templates) to create connections.

Table of Contents

PREFACE

Kamelets are reusable route components that hide the complexity of creating data pipelines that connect to external systems.

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. OVERVIEW OF KAMELETS

Kamelets are high-level connectors that can serve as building blocks in an event driven architecture solution. They are custom resources that you can install on an OpenShift cluster and use in Camel K integrations. Kamelets accelerate your development efforts. They simplify how you connect data sources (that emit events) and data sinks (that consume events). Because you configure kamelet parameters rather than writing code, you do not need to be familiar with the Camel DSL to use kamelets.

You can use kamelets to connect applications and services directly to each other or to:

- Kafka topics, as described in Connecting to Kafka with Kamelets .
- Knative destinations (channels or brokers), as described in Connecting to Knative with Kamelets .
- Specific Camel URIs, as described in Connecting to an explicit Camel URI .

1.1. ABOUT KAMELETS

Kamelets are route components (encapsulated code) that work as connectors in a Camel integration. You can think of kamelets as templates that define where to consume data from (a source) and where to send data to (a sink) - allowing you to assemble data pipelines. Kamelets can also filter, mask, and perform simple calculation logic on data.

There are three different types of kamelets:

- **source** - A route that produces data. You use a source kamelet to retrieve data from a component.
- **sink** - A route that consumes data. You use a sink kamelet to send data to a component.
- **action** - A route that performs an action on data. You can use an action kamelet to manipulate data when it passes from a source kamelet to a sink kamelet.

1.1.1. Why use kamelets?

In a microservices and event-driven architecture solution, kamelets can serve as building blocks for sources that emit events and sinks which consume events.

Kamelets provide abstraction (they hide the complexity of connecting to external systems) and reusability (they are a simple way to reuse code and apply it to different use cases).

Here are some example use cases:

- You want your application to consume events from Telegram, you can use kamelets to bind the Telegram source to a channel of events. Later, you can connect your application to that channel so that it reacts to those events.
- You want your application to connect Salesforce directly to Slack.

Kamelets allow you, and your integration development team, to be more efficient. You can reuse kamelets and share them with your team members who can configure instances for their specific needs. The underlying Camel K operator does the hard work: it compiles, builds, packages and deploys the integration defined by the kamelet.

1.1.2. Who uses kamelets?

Because kamelets allow you to reduce the amount of coding you need to do in your Camel integration, they are ideal for developers who are not familiar with the Camel DSL. Kamelets can help smooth the learning curve for a non-Camel developer. There is no need for you to learn another framework or language to get Camel running.

Kamelets are also useful for experienced Camel developers who want to encapsulate complex Camel integration logic into a reusable kamelet, and then share it with other users.

1.1.3. What are the prerequisites for using kamelets?

To use Kamelets, you need the following environment setup:

- 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 Camel K CLI tools on your local system.
- You installed the Camel K operator in your namespace or cluster-wide as described in Installing Camel K
- You installed the OpenShift command line (**oc**) interface tool.
- Optionally, you installed VS code or another development tool with the Camel K plugin. The Camel-based tooling extensions include features such as automatic completion of Camel URIs based on the embedded kamelet catalog. For more information, see the Camel K development tooling section in *Getting Started with Camel K* .
  Note: Visual Studio (VS) Code Tooling extensions are community only.

1.1.4. How do you use kamelets?

Using a kamelet typically involves two components: the kamelet itself, which defines a reusable route snippet, and a kamelet binding, in which you reference and bind together one or more kamelets. A kamelet binding is an OpenShift resource (**KameletBinding**).

Within the kamelet binding resource, you can:

- Connect a sink or a source kamelet to a channel of events: a Kafka topic or a Knative destination (channel or broker).
- Connect a sink kamelet directly to a Camel Uniform Resource Identifier (URI). You can also connect a source kamelet to a Camel URI, although connecting a URI and a sink kamelet is the most common use case.
- Connect a sink and a source kamelet directly to each other, without using a channel of events as a middle-layer.
- Reference the same kamelet multiple times in the same kamelet binding.
- Add action kamelets to manipulate data when it passes from a source kamelet to a sink kamelet.
- Define an error handling strategy to specify what Camel K should do if there is a failure when sending or receiving event data.

At runtime, the Camel K operator uses the kamelet binding to generate and run a Camel K integration.

Note: While Camel DSL developers can use kamelets directly in Camel K integrations, the simpler way to implement kamelets is by specifying a kamelet binding resource to build a high-level event flow.

1.2. CONNECTING SOURCES AND SINKS

Use kamelets when you want to connect two or more components (external applications or services). Each kamelet is basically a route template with configuration properties. You need to know which component you want to get data from (a source) and which component you want to send data to (a sink). You connect the source and sink components by adding kamelets in a kamelet binding as illustrated in Figure 1.1.

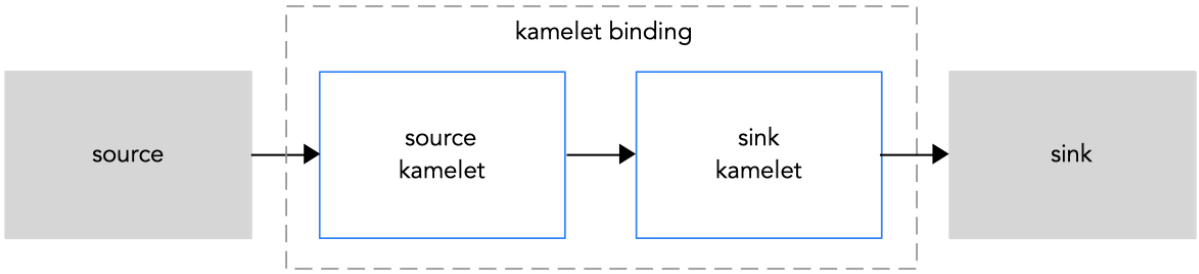


Figure 1.1: Kamelet binding source to sink

Here is an overview of the steps for using kamelets in a kamelet binding:

1. Install the Camel K operator. It includes a catalog of kamelets as resources in your OpenShift project.
2. Create a kamelet binding. Determine which services or applications you want to connect within the kamelet binding.
3. View the Kamelet Catalog to find the kamelets for the source and sink components that you want to use.
4. For each kamelet that you want to include in the kamelet binding, determine the configuration properties that you need to set.
5. In the kamelet binding code, add a reference to each kamelet and configure the required properties.
6. Apply the kamelet binding as a resource in your OpenShift project.

The Camel K operator uses the kamelet binding to generate and run an integration.

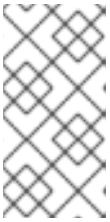
1.2.1. Installing Camel K

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

After you install the Camel K Operator, you can install the Camel K CLI tool for command line access to all Camel K features.

Prerequisites

- You have access to 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 CLI tools on your local system.



NOTE

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

- You installed the OpenShift CLI tool (**oc**) so that you can interact with the OpenShift cluster at the command line. For details on how to install the OpenShift CLI, see Installing the OpenShift CLI.

Procedure

1. In the OpenShift Container Platform web console, log in by using an account with cluster administrator privileges.
2. Create a new OpenShift project:
   a. In the left navigation menu, click **Home** > **Project** > **Create Project**.
   b. Enter a project name, for example, **my-camel-k-project**, and then click **Create**.
3. In the left navigation menu, click **Operators** > **OperatorHub**.
4. In the **Filter by keyword** text box, type **Camel K** and then click the **Red Hat Integration - Camel K Operator** card.
5. Read the information about the operator and then click **Install**. The Operator installation page opens.
6. Select the following subscription settings:
   - **Update Channel** > **1.4.x**
   - **Installation Mode** > **A specific namespace on the cluster**> **my-camel-k-project**
   - **Approval Strategy** > **Automatic**

   NOTE

   The **Installation mode** > **All namespaces on the cluster**and **Approval Strategy** > **Manual** settings are also available if required by your environment.
7. Click **Install**, and then wait a few moments until the Camel K Operator is ready for use.
8. Download and install the Camel K CLI tool:
   a. From the Help menu (?) at the top of the OpenShift web console, select **Command line tools**.
   b. Scroll down to the **kamel - Red Hat Integration - Camel K - Command Line Interface** section.
   c. Click the link to download the binary for your local operating system (Linux, Mac, Windows).

d. Unzip and install the CLI in your system path.

e. To verify that you can access the Kamel K CLI, open a command window and then type the following:
kamel --help

This command shows information about Camel K CLI commands.

Next step

(optional) Specifying Camel K resource limits

1.2.2. Viewing the Kamelet Catalog

When you install the Camel K operator, it includes a catalog of kamelets that you can use in your Camel K integrations.

Prerequisite

You installed the Camel K operator in your working namespace or cluster-wide as described in Installing Camel K.

Procedure

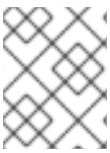
To view a list of kamelets installed with the Camel K operator:

1. In a Terminal window, login to your OpenShift cluster.
2. Viewing the list of available kamelets depends on how the Camel K operator was installed (in a specific namespace or cluster-mode):
   - If the Camel K operator is installed in cluster-mode, use this command to view the available kamelets:
     oc get kamelet -n openshift-operators
   - If the Camel K operator is installed in a specific namespace:
     a. Open a project in which the Camel K operator is installed.
     oc project <camelk-project>

     For example, if the Camel K operator is installed in the **my-camel-k-project** project:

     oc project my-camel-k-project

     b. Run the following command:
     oc get kamelets



NOTE

For a list of the kamelets that are supported by Red Hat, see the Red Hat Integration Release Notes .

See also

Adding a custom kamelet to your Kamelet Catalog

1.2.2.1. Adding a custom kamelet to your Kamelet Catalog

If you don't see a kamelet in the catalog that suits your requirements, a Camel DSL developer can create a custom kamelet as described in the *Apache Camel Kamelets Developers Guide* (community documentation). A kamelet is coded in **YAML** format and, by convention, has a **.kamelet.yaml** file extension.

Prerequisites

- A Camel DSL developer has provided you with a custom kamelet file.
- The kamelet name must be unique to the OpenShift namespace in which the Camel K operator is installed.

Procedure

To make a custom kamelet available as a resource in your OpenShift namespace:

1. Download the kamelet **YAML** file (for example, **custom-sink.kamelet.yaml**) to a local folder.
2. Login to your OpenShift cluster.
3. In a Terminal window, open the project in which the Camel K operator is installed, for example **my-camel-k-project**:
   oc project my-camel-k-project
4. Run the **oc apply** command to add the custom kamelet as a resource to the namespace:
   oc apply -f <custom-kamelet-filename>

   For example, use the following command to add the **custom-sink.kamelet.yaml** file that is located in the current directory:

   oc apply -f custom-sink.kamelet.yaml
5. To verify that the kamelet is available as a resource, use the following command to view an alphabetical list of all kamelets in the current namespace and then look for your custom kamelet:
   oc get kamelets

1.2.2.2. Determining a kamelet's configuration parameters

In a kamelet binding, when you add a reference to a kamelet, you specify the name of the kamelet and you configure the kamelet's parameters.

Prerequisite

- You installed the Camel K operator in your working namespace or cluster-wide.

Procedure

To determine a kamelet's name and parameters:

1. In a terminal window, login to your OpenShift cluster.
2. Open the kamelet's YAML file::
   oc describe kamelets/<kamelet-name>

   For example, to view the **ftp-source** kamelet's code, if the Camel K operator is installed in the current namespace, use this command:

oc describe kamelets/ftp-source

If the Camel K operator is installed in cluster-mode, use this command:

oc describe -n openshift-operators kamelets/ftp-source

3. In the YAML file, scroll down to the **spec.definition** section (which is written in JSON-schema format) to see the list of the kamelet's properties. At the end of the section, the required field lists the properties that you must configure when you reference the kamelet.
For example, the following code is an excerpt from the **spec.definition** section of the **ftp-source** kamelet. This section provides details for all of the kamelet's configuration properties. The required properties for this kamelet are **connectionHost**, **connectionPort**, **username**, **password**, and **directoryName**:

```
spec:
  definition:
    title: "FTP Source"
    description: |-
      Receive data from an FTP Server.
    required:
      - connectionHost
      - connectionPort
      - username
      - password
      - directoryName
    type: object
    properties:
      connectionHost:
        title: Connection Host
        description: Hostname of the FTP server
        type: string
      connectionPort:
        title: Connection Port
        description: Port of the FTP server
        type: string
        default: 21
      username:
        title: Username
        description: The username to access the FTP server
        type: string
      password:
        title: Password
        description: The password to access the FTP server
        type: string
        format: password
        x-descriptors:
        - urn:alm:descriptor:com.tectonic.ui:password
      directoryName:
        title: Directory Name
        description: The starting directory
        type: string
      passiveMode:
        title: Passive Mode
        description: Sets passive mode connection
        type: boolean
        default: false
```

```
    x-descriptors:
    - 'urn:alm:descriptor:com.tectonic.ui:checkbox'
  recursive:
    title: Recursive
    description: If a directory, will look for files in all the sub-directories as well.
    type: boolean
    default: false
    x-descriptors:
    - 'urn:alm:descriptor:com.tectonic.ui:checkbox'
  idempotent:
    title: Idempotency
    description: Skip already processed files.
    type: boolean
    default: true
    x-descriptors:
    - 'urn:alm:descriptor:com.tectonic.ui:checkbox'
```

See also

Configuring kamelet instance parameters

1.2.3. Connecting source and sink components in a kamelet binding

Within a kamelet binding, you connect source and sink components.

The example in this procedure uses the following kamelets as shown in Figure 1.2:

- The example source kamelet is named **coffee-source**. This simple kamelet retrieves randomly-generated data about types of coffee from a web site catalog. It has one parameter (**period** - an **integer** value) that determines how frequently (in seconds) to retrieve the coffee data. The parameter is not required since there is a default value (1000 seconds).
- The example sink kamelet is named **log-sink**. It retrieves data and outputs it to a log file.

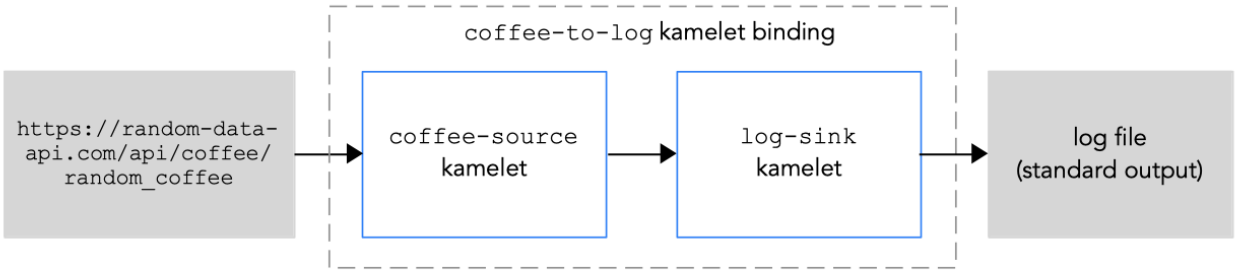


Figure 1.2: Example kamelet binding

Prerequisites

- You know how to create and edit a Camel K integration.
- The **Red Hat Integration - Camel K** operator is installed on your OpenShift namespace or cluster and you have downloaded the Red Hat Integration Camel K CLI tool as described in Installing Camel K .

- You know which kamelets you want to add to your Camel K integration and their required instance parameters.
- The kamelets that you want to use are available in your Kamelet catalog.
  If you want to use the kamelets in this example, copy and save the **coffee-source** code to a local file named **coffee-source.kamelet.yaml** and the **log-sink** code to a file named **log-sink.kamelet.yaml**. Then run the following commands to add them to your Kamelet catalog:

  oc apply -f coffee-source.kamelet.yaml

  oc apply -f log-sink.kamelet.yaml

Procedure

1. Login to your OpenShift cluster.
2. Open your working project where the Camel K operator is installed. If you installed the Camel K operator in cluster-mode, it is available to any project on the cluster.
   For example, to open an existing project named **my-camel-k-project**:

   oc project my-camel-k-project

3. Create a new **KameletBinding** resource:

   a. In an editor of your choice, create a YAML file with the following structure:

   ```
   apiVersion: camel.apache.org/v1alpha1
   kind: KameletBinding
   metadata:
     name:
   spec:
     source:
     sink:
   ```

   b. Add a name for the kamelet binding.
   For this example, the name is **coffee-to-log** because the binding connects the **coffee-source** kamelet to the **log-sink** kamelet.

   ```
   apiVersion: camel.apache.org/v1alpha1
   kind: KameletBinding
   metadata:
     name: coffee-to-log
   spec:
     source:
     sink:
   ```

4. Specify the source kamelet (for example, **coffee-source**) and configure any parameters for the kamelet.
   Note: For this example, the parameter is defined within the kamelet binding's YAML file. Alternatively, you can configure a kamelet's parameters in a property file, ConfigMap, or Secret as described in Configuring kamelet instance parameters.

   ```
   apiVersion: camel.apache.org/v1alpha1
   kind: KameletBinding
   metadata:
   ```

```
  name: coffee-to-log
spec:
  source:
    ref
      kind: Kamelet
      apiVersion: camel.apache.org/v1alpha1
      name: coffee-source
    properties:
      period: 5000
  sink:
```

5. Specify the sink kamelet (for example, **log-sink**) and configure any parameters for the kamelet. The example **log-sink** kamelet does not have any required parameters.

```
apiVersion: camel.apache.org/v1alpha1
kind: KameletBinding
metadata:
  name: coffee-to-log
spec:
  source:
    ref:
      kind: Kamelet
      apiVersion: camel.apache.org/v1alpha1
      name: coffee-source
    properties:
      period: 5000
  sink:
    ref:
      kind: Kamelet
      apiVersion: camel.apache.org/v1alpha1
      name: log-sink
```

6. Save the YAML file (for example, **coffee-to-log.yaml**).

7. Add the **KameletBinding** as a resource to your OpenShift namespace:
oc apply -f <kamelet>.yaml

   For example:

   oc apply -f coffee-to-log.yaml

   The Camel K operator generates and runs a Camel K integration by using the KameletBinding resource.

8. To see the status of the KameletBinding:
oc get kameletbindings

9. To see the status of the corresponding integration: **oc get integrations**

10. To view the output:

    - To view the logs from the command line, open a Terminal window and then type the following command:
      kamel log <integration-name>

      For example, if the integration name is **coffee-to-log**, use this command:

kamel log coffee-to-log

- To view the logs from OpenShift web console:

  a. Select **Workloads** > **Pods**.

  b. Click the name of the Camel K integration's pod, and then click **Logs**.
  You should see a list of coffee events similar to the following example:

  ```
  INFO  [log-sink-E80C5C904418150-0000000000000001] (Camel (camel-1) thread #0 - timer://tick) {"id":7259,"uid":"a4ecb7c2-05b8-4a49-b0d2-d1e8db5bc5e2","blend_name":"Postmodern Symphony","origin":"Huila, Colombia","variety":"Kona","notes":"delicate, chewy, black currant, red apple, star fruit","intensifier":"balanced"}
  ```

11. To stop the integration, delete the kamelet binding:
oc delete kameletbindings/<kameletbinding-name>

    For example:

    oc delete kameletbindings/coffee-to-log

Next steps

Optionally:

- Add action kamelets as intermediary steps, as described in Adding an operation to a kamelet binding.
- Add error handling to the kamelet binding, as described in Adding an error handler policy to a kamelet binding.

1.2.4. Configuring kamelet instance parameters

When you reference a kamelet, you have the following options for defining the kamelet's instance parameters:

- Directly in a kamelet binding where you specify the kamelet URI. In the following example, the bot authorization token provided by the Telegram BotFather. is **123456**:
from("kamelet:telegram-source?authorizationToken=123456")
- Globally configure a kamelet property (so that you don't have to provide the value in the URI) by using the following format:
"camel.kamelet.<kamelet-name>.<property-name>=<value>"

  As described in the Configuring Camel K integrations chapter in *Developing and Managing Integrations Using Camel K*, you can configure kamelet parameters by:

  - Defining them as properties
  - Defining them in a property file
  - Defining them in an OpenShift ConfigMap or Secret

See also

Determining a kamelet's configuration parameters

1.2.5. Connecting to a channel of events

The most common use case for kamelets is to use a kamelet binding to connect them to a channel of events: a Kafka topic or a Knative destination (channel or broker). The advantage of doing so is that the data source and sink are independent and "unaware" of each other. This decoupling allows the components in your business scenario to be developed and managed separately. If you have multiple data sinks and sources as part of your business scenario, it becomes even more important to decouple the various components. For example, if an event sink needs to be shut down, the event source is not impacted. And, if other sinks use the same source, they are not impacted.

Figure 1.3 illustrates the flow of connecting source and sink kamelets to a channel of events.

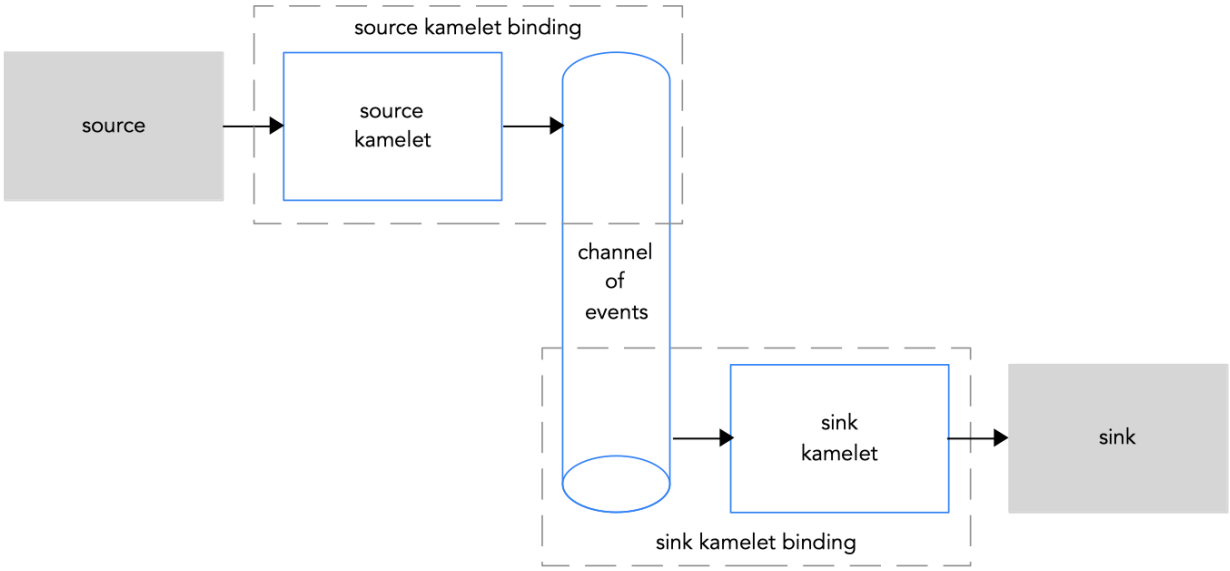


Figure 1.3: Connecting source and sink kamelets to a channel of events

If you use the Apache Kafka stream-processing framework, for details on how to connect to a Kafka topic, see Connecting to Kafka with Kamelets .

If you use the Knative serverless framework, for details on how to connect to a Knative destination (channel or broker), see Connecting to Knative with Kamelets .

1.2.6. Connecting to an explicit Camel URI

You can create a kamelet binding in which a kamelet sends events to—or receives events from—an explicit Camel URI. Typically, you bind a source kamelet to a URI that can receive events (that is, you specify the URI as the sink in a kamelet binding). Examples of Camel URIs that receive events are HTTP or HTTPS endpoints.

It is also possible, but not as common, to specify a URI as the source in a kamelet binding. Examples of Camel URIs that send events are timer, mail, or FTP endpoints.

To connect a kamelet to a Camel URI, follow the steps in Connecting source and sink components in a kamelet binding and for the **sink.uri** field, instead of a kamelet, specify an explicit Camel URI.

In the following example, the URI for the sink is a fictional URI (https://mycompany.com/event-service):

```
apiVersion: camel.apache.org/v1alpha1
kind: KameletBinding
metadata:
  name: coffee-to-event-service
spec:
  source:
    ref:
      kind: Kamelet
      apiVersion: camel.apache.org/v1alpha1
      name: coffee-source
    properties:
      period: 5000
  sink:
    uri: https://mycompany.com/event-service
```

1.3. APPLYING OPERATIONS TO DATA WITHIN A CONNECTION

If you want to perform an operation on the data that passes between a kamelet and an event channel, use action kamelets as intermediary steps within a kamelet binding. For example, you can use an action kamelet to serialize or deserialize data, filter the data, or insert a field or a message header.

Manipulation operations, such as filtering or adding fields, work only with JSON data (that is, when the **Content-Type** header is set to **application/json**). If the event data uses a format other than JSON (for example, Avro or Protocol Buffers), you must convert the format of the data by adding a deserialize step (for example, that references the **protobuf-deserialize-action** or **avro-deserialize-action** kamelet) before the manipulating action and a serialize step (for example, that references the **protobuf-serialize-action** or **avro-serialize-action** kamelet) after it. For more information about converting the format of data in a connection, see Data conversion kamelets.

Action kamelets include:

- Data filtering kamelets
- Data conversion kamelets
- Data transformation kamelets

1.3.1. Adding an operation to a kamelet binding

To implement an action kamelet, in the kamelet binding file's **spec** section, add a **steps** section in between the source and sink sections.

Prerequisites

- You have created a kamelet binding as described in Connecting source and sink components in a kamelet binding.
- You know which action kamelet you want to add to the kamelet binding and the action kamelet's required parameters.
  For the example in this procedure, the parameter for the **predicate-filter-action** kamelet is a **string** type, expression, that provides the JSON Path Expression that filters coffee data to only log coffees that have a "deep" taste intensity. Note that the **predicate-filter-action** kamelet requires that you set a Builder trait configuration property in the kamelet binding.

The example also includes deserialize and serialize actions which are optional in this case because the event data format is JSON.

Procedure

1. Open a **KameletBinding** file in an editor.
   For example, here are the contents of the **coffee-to-log.yaml** file:

```
apiVersion: camel.apache.org/v1alpha1
kind: KameletBinding
metadata:
  name: coffee-to-log
spec:
  source:
    ref:
      kind: Kamelet
      apiVersion: camel.apache.org/v1alpha1
      name: coffee-source
    properties:
      period: 5000
  sink:
    ref:
      kind: Kamelet
      apiVersion: camel.apache.org/v1alpha1
      name: log-sink
```

2. Add an **integration** section above the **source** section and provide the following Builder trait configuration property (as required by the **predicate-filter-action** kamelet):

```
apiVersion: camel.apache.org/v1alpha1
kind: KameletBinding
metadata:
  name: coffee-to-log
spec:
  integration:
        traits:
          builder:
            configuration:
              properties:
                - "quarkus.arc.unremovable-
types=com.fasterxml.jackson.databind.ObjectMapper"
  source:
    ref:
      kind: Kamelet
      apiVersion: camel.apache.org/v1alpha1
      name: coffee-source
    properties:
      period: 5000
  sink:
    ref:
      kind: Kamelet
      apiVersion: camel.apache.org/v1alpha1
      name: log-sink
```

3. Add a **steps** section, between the **source** and **sink** sections and define the action kamelet. For example:

```
apiVersion: camel.apache.org/v1alpha1
kind: KameletBinding
metadata:
  name: coffee-to-log
  spec:
    integration:
          traits:
            builder:
              configuration:
                properties:
                  - "quarkus.arc.unremovable-
types=com.fasterxml.jackson.databind.ObjectMapper"
  source:
    ref:
      kind: Kamelet
      apiVersion: camel.apache.org/v1alpha1
      name: coffee-source
    properties:
      period: 5000
  steps:
  - ref:
      kind: Kamelet
      apiVersion: camel.apache.org/v1alpha1
      name: json-deserialize-action
  - ref:
      kind: Kamelet
      apiVersion: camel.apache.org/v1alpha1
      name: predicate-filter-action
    properties:
      expression: "@.intensifier =~ /.*deep/"
  - ref:
      kind: Kamelet
      apiVersion: camel.apache.org/v1alpha1
      name: json-serialize-action
  sink:
    ref:
      kind: Kamelet
      apiVersion: camel.apache.org/v1alpha1
      name: log-sink
```

4. Save your changes.

5. Update the **KameletBinding** resource:
   oc apply -f coffee-to-log.yaml

   The Camel K operator re-generates and runs the CamelK integration that it generates based upon the updated **KameletBinding** resource.

6. To see the status of the kamelet binding:
   oc get kameletbindings

7. To see the status of its corresponding integration:
   oc get integrations

8. To view the log file output:

   a. Get the pod name for the integration:
   oc get pods

   b. View the log for the integration's pod:
   oc logs <podname of the integration>

   For example, if the pod name is **example-7885bdb9-84ht8**:

   oc logs example-7885bdb9-84ht8

9. To stop the integration, delete the kamelet binding:
   oc delete kameletbindings/<kameletbinding-name>

   For example:

   oc delete kameletbindings/coffee-to-log

1.3.2. Action kamelets

- Section 1.3.2.1, "Data filtering kamelets"
- Section 1.3.2.2, "Data conversion kamelets"
- Section 1.3.2.3, "Data transformation kamelets"

1.3.2.1. Data filtering kamelets

You can filter the data that passes between source and sink components, for example, to prevent leaking sensitive data or to avoid generating unnecessary networking charges.

You can filter data based on the following criteria:

- **Kafka topic name** - Filter events for a Kafka topic with a name that matches the given Java regular expression by configuring the Topic Name Matches Filter Action Kamelet (**topic-name-matches-filter-action**).
- **Header key** - Filter events that have a given message header by configuring the Header Filter Action Kamelet (**has-header-filter-action**).
- **Null value** - Filters tombstone events (events with a null payload) by configuring the Tombstone Filter Action Kamelet (**is-tombstone-filter-action**).
- **Predicate** - Filter events based on the given JSON path expression by configuring the Predicate Filter Action Kamelet (**predicate-filter-action**). The **predicate-filter-action** kamelet requires that you set the following Builder trait configuration property in the kamelet binding:

```
spec:
  integration:
    traits:
      builder:
        configuration:
          properties:
           - "quarkus.arc.unremovable-types=com.fasterxml.
                jackson.databind.ObjectMapper"
```

NOTE

Data filtering kamelets work out-of-the-box with JSON data (that is, when the Content-Type header is set to application/json). If the event data uses a format other than JSON, you must convert the format of the data by adding a deserialize step (for example, **protobuf-deserialize-action** or **avro-deserialize-action**) before the manipulating action and a serialize step (for example, **protobuf-serialize-action** or **avro-serialize-action**) after it. For more information about converting the format of data in a connection, see Data conversion kamelets.

1.3.2.2. Data conversion kamelets

With the following data conversion kamelets, you can serialize and deserialize the format of data that passes between source and sink components. The data conversion applies to the payload of event data (not the key or the header).

- **Avro** - An open source project that provides data serialization and data exchange services for Apache Hadoop.
  - Avro Deserialize Action Kamelet (**avro-deserialize-action**)
  - Avro Serialize Action Kamelet (**avro-serialize-action**)
- **Protocol Buffers** - A high-performance, compact binary wire format invented by Google who use it internally so they can communicate with their internal network services.
  - Protobuf Deserialize Action Kamelet (**protobuf-deserialize-action**)
  - Protobuf Serialize Action Kamelet (**protobuf-serialize-action**)
- **JSON** (JavaScript Object Notation) - A data-interchange format that is based on a subset of the JavaScript Programming Language. JSON is a text format that is completely language independent.
  - JSON Deserialize Action Kamelet (**json-deserialize-action**)
  - JSON Serialize Action Kamelet (**json-serialize-action**)

NOTE

You must specify the schema (as a single-line, using JSON format) in the Avro and Protobuf serialize/deserialize kamelets. You do not need to do so for JSON serialize/deserialize kamelets.

1.3.2.3. Data transformation kamelets

With the following data transformation kamelets, you can perform simple manipulations on the data that passes between the source and sink components:

- **ValueToKey** - (for Kafka) Use the **value-to-key-action** kamelet to replace the record key with a new key formed from a subset of fields in the payload. You can set the event key to a value that is based on the event information before the data is written to Kafka. For example, when reading records from a database table, you can partition the records in Kafka based on the customer ID.

- **MaskField** - Use the **mask-field-action** kamelet to replace a field value with a valid null value for the field type (such as 0 or an empty string) or with a given replacement (the replacement must be a non-empty string or a numeric value).
  For example, if you want to capture data from a relational database to send to Kafka and the data includes protected (PCI / PII) information, you must mask the protected information if your Kafka cluster is not certified yet.

- **InsertHeader** - Use the **insert-header-action** kamelet to add a field (header) by using either static data or record metadata.

- **InsertField** - Use the **insert-field-action** kamelet to add a field (value) by using either static data or record metadata.

1.4. HANDLING ERRORS WITHIN A CONNECTION

To specify what the Camel K operator should do if a running integration encounters a failure when sending or receiving event data, you can optionally add one of the following error handling policies to the kamelet binding:

- No error handler - Ignores any failure happening in your integration.

- Log error handler - Sends a log message to standard output.

- Dead letter channel error handler - Redirects a failing event to another component, such as a third-party URI, a queue, or another kamelet which can perform certain logic with the failing event. Also supports attempting to redeliver the message exchange a number of times before sending it to a dead letter endpoint.

- Bean error handler - Specifies to use a custom bean for handling errors.

- Ref error handler - Specifies to use a bean for handling errors. The bean must be available in the Camel registry at runtime.

1.4.1. Adding an error handler policy to a kamelet binding

To handle errors when sending or receiving event data between a source and a sink connection, add an error handler policy to the kamelet binding.

Prerequisites

- You know which type of error handler policy you want to use.

- You have an existing **KameletBinding** YAML file.

Procedure

To implement error handling in a kamelet binding:

1. Open a **KameletBinding** YAML file in an editor.

2. Add an error handler section to the **spec** section, after the **sink** definition:

```
apiVersion: camel.apache.org/v1alpha1
kind: KameletBinding
metadata:
  name: example-kamelet-binding
```

```
spec:
  source:
    ...
  sink:
    ...
  errorHandler: ...
```

For example, in the **coffee-to-log** kamelet binding, specify the maximum number of times an error is sent to the log file by adding a log error handler:

```
apiVersion: camel.apache.org/v1alpha1
kind: KameletBinding
metadata:
  name: coffee-to-log
spec:
  source:
    ref:
      kind: Kamelet
      apiVersion: camel.apache.org/v1alpha1
      name: coffee-source
    properties:
      period: 5000
  sink:
    ref:
      kind: Kamelet
      apiVersion: camel.apache.org/v1alpha1
      name: log-sink
  errorHandler: log: parameters: maximumRedeliveries: 3
```

3. Save your file.

1.4.2. Error handlers

- Section 1.4.2.1, “No error handler”
- Section 1.4.2.2, “Log error handler”
- Section 1.4.2.3, “Dead letter channel error handler”
- Section 1.4.2.4, “Bean error handler”
- Section 1.4.2.5, “Ref error handler”

1.4.2.1. No error handler

If you want to ignore any failure happening in your integration, you can either not include an **errorHandler** section in the kamelet binding or set it to **none** as shown in the following example:

```
apiVersion: camel.apache.org/v1alpha1
kind: KameletBinding
metadata:
  name: my-kamelet-binding
spec:
  source:
...
```

```
  sink:
...
  errorHandler:
    none:
```

1.4.2.2. Log error handler

The default behavior for handling any failure is to send a log message to standard output. Optionally, you can use the log error handler to specify other behaviors, such as a redelivery or delay policy, as shown in the following example:

```
apiVersion: camel.apache.org/v1alpha1
kind: KameletBinding
metadata:
  name: my-kamelet-binding
spec:
  source:
...
  sink:
...
  errorHandler:
    log: parameters: maximumRedeliveries: 3 redeliveryDelay: 2000
```

1.4.2.3. Dead letter channel error handler

The Dead Letter Channel allows you to redirect any failing event to any other component (such as a third party URI, a queue, or another kamelet) that can define how to handle a failing event, as shown in the following example:

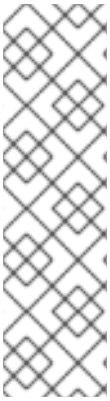
```
apiVersion: camel.apache.org/v1alpha1
kind: KameletBinding
metadata:
  name: my-kamelet-binding
spec:
  source:
  ...
  sink:
  ...
  errorHandler:
    dead-letter-channel:
      endpoint:
        ref: 1
          kind: Kamelet
          apiVersion: camel.apache.org/v1alpha1
          name: error-handler
        properties: 2
          message: "ERROR!"
          ...
      parameters: 3
        maximumRedeliveries: 1
```

1. For the **endpoint**, you can use **ref** or **uri**. The Camel K operator interprets **ref** according to the **kind**, **apiVersion** and **name** values. You can use any kamelet, Kafka Topic channel, or Knative destination.

2. **Properties** that belong to the endpoint (in this example, to a kamelet named **error-handler**).
3. **Parameters** that belong to the dead-letter-channel error handler type.

1.4.2.4. Bean error handler

With the Bean error handler you can extend the functionality of the Error Handler by providing a custom bean that handles errors. For **type**, specify the fully-qualified name of the **ErrorHandlerBuilder**. For **properties**, configure the properties expected by the **ErrorHandlerBuilder** that you specified in **type**.



NOTE

Within a kamelet binding, Camel components in URIs require dependency declarations. The Camel K operator generates an integration from a kamelet binding at runtime. For integrations that are not generated by a kamelet binding, Camel K automatically handles the dependency management and imports all the required libraries from the Camel catalog. However, for this release, you must specify any Camel components that you reference in a URI within a kamelet binding as a dependency. In the following example, because the kamelet binding references the **camel-log** component in the **deadLetterUri**, it includes the **camel-log** as a dependency in the **spec.integration.dependencies** section.

```
apiVersion: camel.apache.org/v1alpha1
kind: KameletBinding
metadata:
  name: my-kamelet-binding
spec:
  integration: dependencies: - camel:log
  source:

  source:
...
  sink:
...
  errorHandler:
    bean: type: "org.apache.camel.builder.DeadLetterChannelBuilder" properties: deadLetterUri:
log:error
```

1.4.2.5. Ref error handler

With the Ref error handler, you can use any bean that you expect to be available in the Camel registry at runtime. In the following example, **my-custom-builder** is the name of the bean to look up at runtime.

```
apiVersion: camel.apache.org/v1alpha1
kind: KameletBinding
metadata:
  name: my-kamelet-binding
spec:
  source:
    ...
  sink:
    ...
  errorHandler:
    ref: my-custom-builder
```

See also:

- Camel K error handling
- Features support by various Error Handlers

CHAPTER 2. CONNECTING TO KAFKA WITH KAMELETS

Apache Kafka is an open-source, distributed, publish-subscribe messaging system for creating fault-tolerant, real-time data feeds. Kafka quickly stores and replicates data for a large number of consumers (external connections).

Kafka can help you build solutions that process streaming events. A distributed, event-driven architecture requires a "backbone" that captures, communicates and helps process events. Kafka can serve as the communication backbone that connects your data sources and events to applications.

You can use kamelets to configure communication between Kafka and external resources. Kamelets allow you to configure how data moves from one endpoint to another in a Kafka stream-processing framework without writing code. Kamelets are route templates that you configure by specifying parameter values.

For example, Kafka stores data in a binary form. You can use kamelets to serialize and deserialize the data for sending to, and receiving from, external connections. With kamelets, you can validate the schema and make changes to the data, such as adding to it, filtering it, or masking it. Kamelets can also handle and process errors.

2.1. OVERVIEW OF CONNECTING TO KAFKA WITH KAMELETS

If you use an Apache Kafka stream-processing framework, you can use kamelets to connect services and applications to a Kafka topic. The Kamelet Catalog provides the following kamelets specifically for making connections to a Kafka topic:

- **kafka-sink** - Moves events from a data producer to a Kafka topic. In a kamelet binding, specify the **kafka-sink** kamelet as the sink.
- **kafka-source** - Moves events from a Kafka topic to a data consumer. In a kamelet binding, specify the **kafka-source** kamelet as the source.

Figure 2.1 illustrates the flow of connecting source and sink kamelets to a Kafka topic.

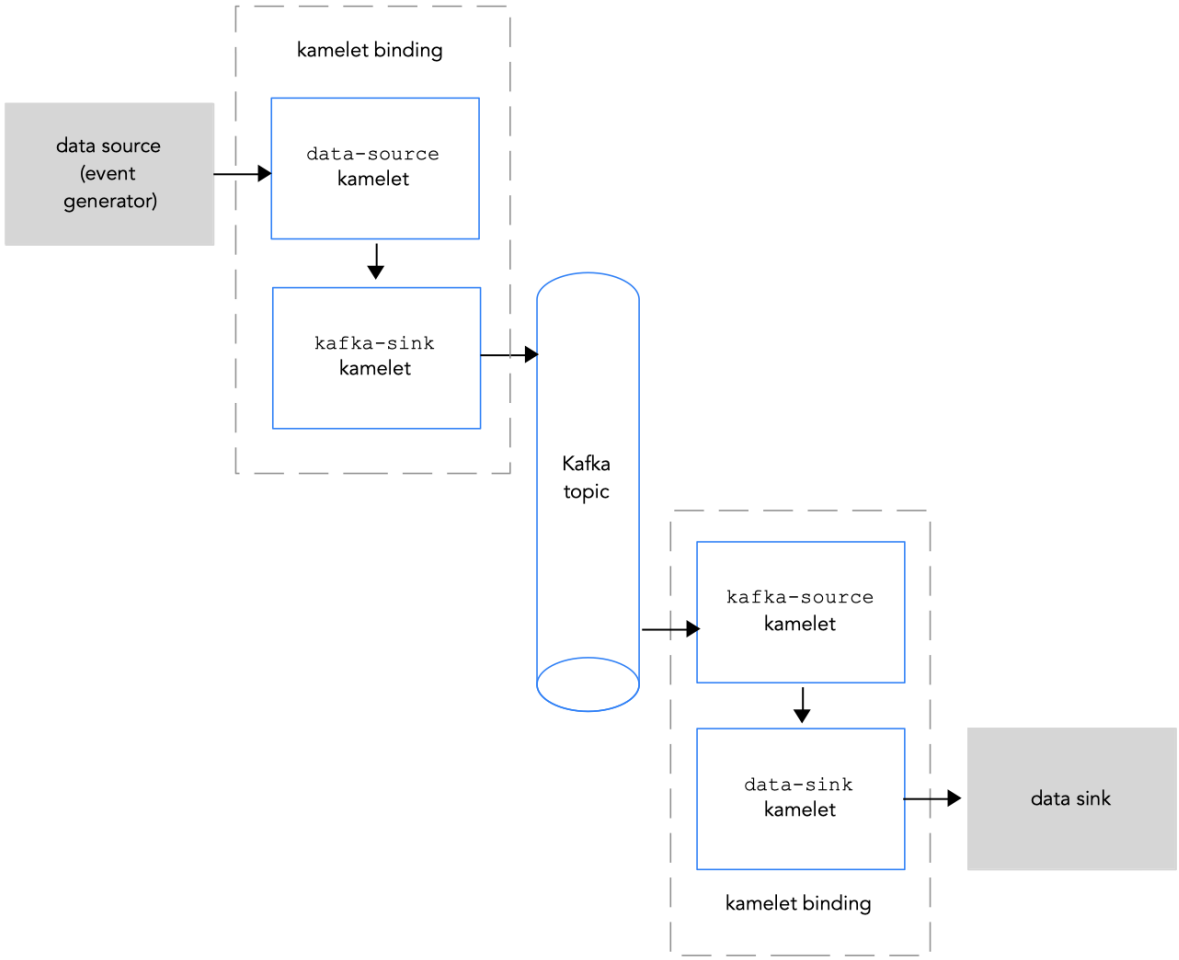


Figure 2.1: Data flow with kamelets and a Kafka topic

Here is an overview of the basic steps for using kamelets and kamelet bindings to connect applications and services to a Kafka topic:

1. Set up Kafka:
    a. Install the needed OpenShift operators.
        - For OpenShift Streams for Apache Kafka, install the RHOAS and Camel K operators.
        - For AMQ streams, install the Camel K and AMQ streams operators.
    b. Create a Kafka instance. A Kafka instance operates as a message broker. A broker contains topics and orchestrates the storage and passing of messages.
    c. Create a Kafka topic. A topic provides a destination for the storage of data.
    d. Obtain Kafka authentication credentials.
2. Determine which services or applications you want to connect to your Kafka topic.
3. View the kamelet catalog to find the kamelets for the source and sink components that you want to add to your integration. Also, determine the required configuration parameters for each kamelet that you want to use.

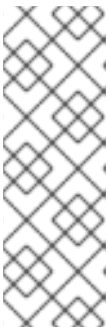
4. Create kamelet bindings:
    - Create a kamelet binding that connects a data source (a component that produces data) to the Kafka topic (by using the **kafka-sink** kamelet).
    - Create a kamelet binding that connects the kafka topic (by using **kafka-source** kamelet) to a data sink (a component that consumes data).
5. Optionally, manipulate the data that passes between the Kafka topic and the data source or sink by adding one or more action kamelets as intermediary steps within a kamelet binding.
6. Optionally, define how to handle errors within a kamelet binding.
7. Apply the kamelet bindings as resources to the project.
   The Camel K operator generates a separate Camel K integration for each kamelet binding.

2.2. SETTING UP KAFKA

Setting up Kafka involves installing the required OpenShift operators, creating a Kafka instance, and creating a Kafka topic.

Use one of these Red Hat products 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.
- **Red Hat OpenShift Streams for Apache Kafka**(Development Preview) - A managed cloud service that simplifies the process of running Apache Kafka. It provides a streamlined developer experience for building, deploying, and scaling new cloud-native applications or modernizing existing systems.



NOTE

Red Hat OpenShift Streams for Apache Kafka is currently available for Development Preview. Development Preview releases provide early access to a limited set of features that might not be fully tested and that might change in the final GA version. Users should not use Development Preview software in production or for business-critical workloads. Limited documentation is available for Development Preview releases and is typically focused on fundamental user goals.

2.2.1. Setting up Kafka by using AMQ streams

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

2.2.1.1. 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.
6. Set up Kafka authentication as described in Managing secure access to Kafka .

Next steps

Setting up a Kafka topic with AMQ Streams

2.2.1.2. 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**.

2.2.2. Setting up Kafka by using OpenShift streams

Red Hat OpenShift Streams for Apache Kafka is a managed cloud service that simplifies the process of running Apache Kafka.

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

NOTE

Red Hat OpenShift Streams for Apache Kafka is currently available for Development Preview. Development Preview releases provide early access to a limited set of features that might not be fully tested and that might change in the final GA version. Users should not use Development Preview software in production or for business-critical workloads. Limited documentation is available for Development Preview releases and is typically focused on fundamental user goals.

See Also

- Product documentation for Red Hat OpenShift Streams for Apache Kafka

2.2.2.1. Preparing your OpenShift cluster for OpenShift Streams

To use the Red Hat OpenShift Streams for Apache Kafka managed cloud service, you must install the following operators and tools:

- **OpenShift Application Services (RHOAS)** operator - Manages the communication between your OpenShift Cluster and the Red Hat OpenShift Streams for Apache Kafka instances.
  Note: This is a community operator.
- **RHOAS CLI** - Allows you to manage your application services from a terminal.
  Note: This is a Developer Preview feature.
- **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. Install the RHOAS operator:
   a. From any project, select **Operators** > **OperatorHub**.
   b. In the **Filter by Keyword** field, type **RHOAS**.
   c. Click the **OpenShift Application Services (RHOAS)** card (it is a community operator) and then click **Install**.
   The **Install Operator** page opens.
   d. Accept the default mode (**All namespaces on the cluster**) or select the namespace for your project, then click **Install**.
4. Download and install the RHOAS CLI as described in Getting started with the rhoas CLI.

5. Install the Camel K operator and Camel K CLI as described in Installing Camel K.

6. To verify that the **Red Hat Integration - Camel K**and **OpenShift Application Services (RHOAS)** operators are installed, click **Operators** > **Installed Operators**.

Next step

Setting up a Kafka topic with RHOAS

2.2.2.2. 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**and **OpenShift Application Services (RHOAS)** operators as described in Preparing your OpenShift cluster.
- You are logged in to the Red Hat Cloud (Beta) site.

Procedure

To set up a Kafka topic by using Red Hat OpenShift Streams for Apache Kafka:

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 necessary operators are installed in your project:
   oc get csv

   The result lists the Red Hat Camel K and RHOAS operators and indicates that they are 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/beta/application-services/streams/kafkas/

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

6. Create a new Kafka topic:
rhoas kafka topic create 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.
```

The RHOAS operator sets up the KafkaConnection custom resource named **kafka-test**.

Next step

- Obtaining Kafka credentials

2.2.2.3. 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 method 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 kafka

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:

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

2.2.2.4. 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.

2.2.2.5. 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" ;
   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.

2.3. CONNECTING A DATA SOURCE TO A KAFKA TOPIC IN A KAMELET BINDING

To connect a data source to a Kafka topic, you create a kamelet binding as illustrated in *Figure 2.2*.

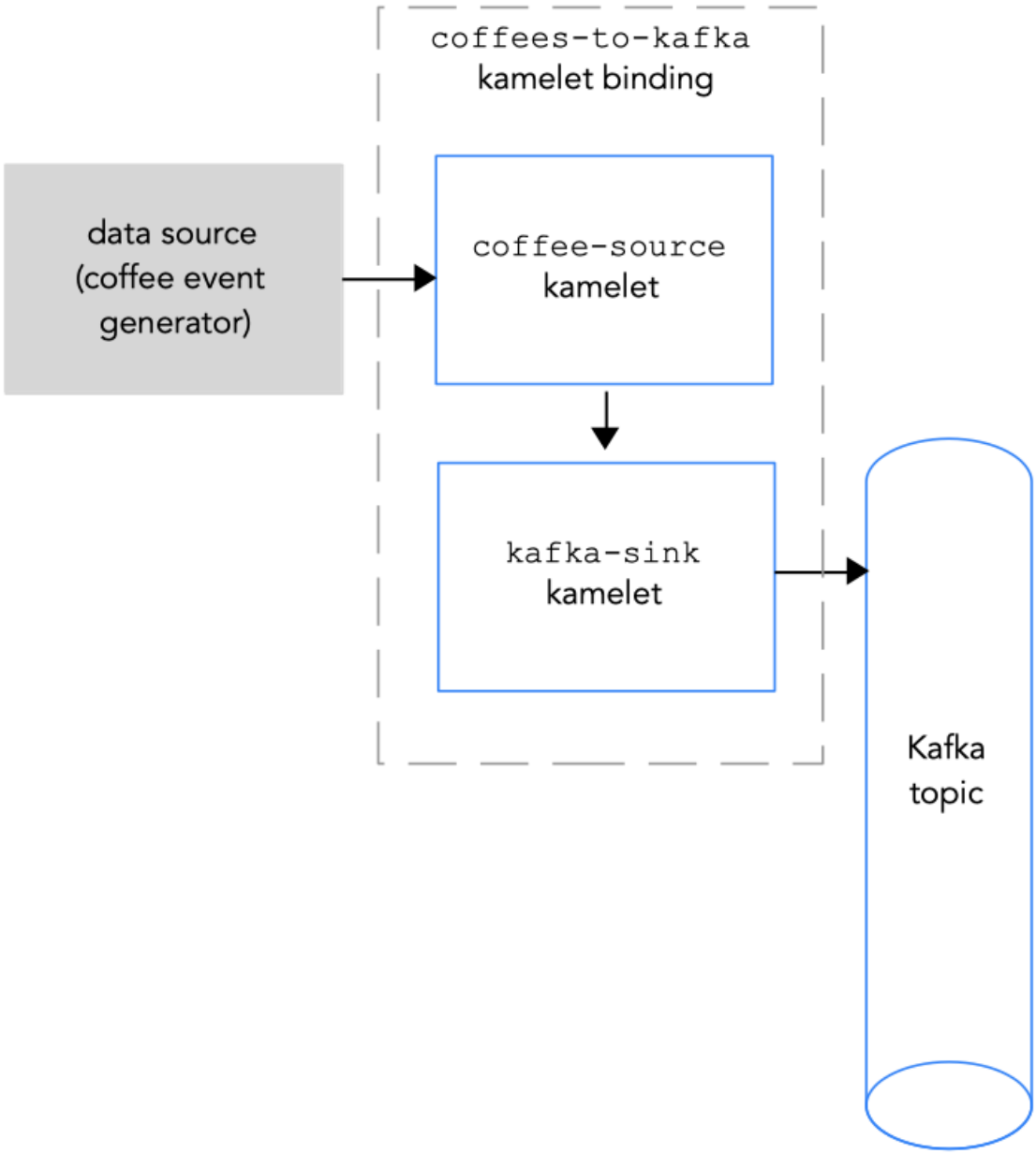


Figure 2.2 Connecting a data source to a Kafka topic

Prerequisites

- You know the name of the Kafka topic to which you want to send events.
  The example in this procedure uses **test-topic** for receiving events.
- You know the values of the following parameters for your Kafka instance:
  - **bootstrapServers** - A comma separated list of Kafka Broker URLs.
  - **password** - The password to authenticate to Kafka.
  - **user** - The user name to authenticate to Kafka.
    For information on how to obtain these values when you use OpenShift Streams, see Obtaining Kafka credentials.

For information about Kafka authentication on AMQ streams, see Managing secure access to Kafka.

- You know the security protocol for communicating with the Kafka brokers. For a Kafka cluster on OpenShift Streams, it is **SASL_SSL** (the default). For a Kafka cluster on AMQ streams, it is **SASL/Plain**.

- You know which kamelets you want to add to your Camel K integration and the required instance parameters.
  The example kamelets for this procedure are: ** The **coffee-source** kamelet - It has an optional parameter, **period**, that specifies how often to send each event. You can copy the code from Example source kamelet to a file named **coffee-source.kamelet.yaml** file and then run the following command to add it as a resource to your namespace:

  + **oc apply -f coffee-source.kamelet.yaml**

  - The **kafka-sink** kamelet provided in the Kamelet Catalog. You use the **kafka-sink** kamelet because the Kafka topic is receiving data (it is the data consumer) in this binding. The example values for the required parameters are:

    - **bootstrapServers** - **"broker.url:9092"**
    - **password** - **"testpassword"**
    - **user** - **"testuser"**
    - **topic** - **"test-topic"**
    - **securityProtocol** - For a Kafka cluster on OpenShift Streams, you do not need to set this parameter because **SASL_SSL** is the default value. For a Kafka cluster on AMQ streams, this parameter value is **"PLAINTEXT"**.

Procedure

To connect a data source to a Kafka topic, create a kamelet binding:

1. In an editor of your choice, create a YAML file with the following basic structure:

```
apiVersion: camel.apache.org/v1alpha1
kind: KameletBinding
metadata:
  name:
spec:
  source:
  sink:
```

2. Add a name for the kamelet binding. For this example, the name is **coffees-to-kafka** because the binding connects the **coffee-source** kamelet to the **kafka-sink** kamelet.

```
apiVersion: camel.apache.org/v1alpha1
kind: KameletBinding
metadata:
  name: coffees-to-kafka
spec:
  source:
  sink:
```

3. For the kamelet binding's source, specify a data source kamelet (for example, the **coffee-source** kamelet produces events that contain data about coffee) and configure any parameters for the kamelet.

```
apiVersion: camel.apache.org/v1alpha1
kind: KameletBinding
metadata:
  name: coffees-to-kafka
spec:
  source:
    ref:
      kind: Kamelet
      apiVersion: camel.apache.org/v1alpha1
      name: coffee-source
    properties:
      period: 5000
  sink:
```

4. For the kamelet binding's sink, specify the **kafka-sink** kamelet and its required parameters. For example, when the Kafka cluster is on OpenShift Streams (you do not need to set the **securityProtocol** parameter):

```
apiVersion: camel.apache.org/v1alpha1
kind: KameletBinding
metadata:
  name: coffees-to-kafka
spec:
  source:
    ref:
      kind: Kamelet
      apiVersion: camel.apache.org/v1alpha1
      name: coffee-source
    properties:
      period: 5000
  sink:
    ref:
      kind: Kamelet
      apiVersion: camel.apache.org/v1alpha1
      name: kafka-sink
    properties:
      bootstrapServers: "broker.url:9092"
      password: "testpassword"
      topic: "test-topic"
      user: "testuser"
```

For example, when the Kafka cluster is on AMQ Streams you must set the **securityProtocol** parameter to **"PLAINTEXT"**:

```
apiVersion: camel.apache.org/v1alpha1
kind: KameletBinding
metadata:
  name: coffees-to-kafka
spec:
  source:
    ref:
```

```
      kind: Kamelet
      apiVersion: camel.apache.org/v1alpha1
      name: coffee-source
    properties:
      period: 5000
  sink:
    ref:
      kind: Kamelet
      apiVersion: camel.apache.org/v1alpha1
      name: kafka-sink
    properties:
      bootstrapServers: "broker.url:9092"
      password: "testpassword"
      topic: "test-topic"
      user: "testuser"
      securityProtocol: "PLAINTEXT"
```

5. Save the YAML file (for example, **coffees-to-kafka.yaml**).
6. Log into your OpenShift project.
7. Add the kamelet binding as a resource to your OpenShift namespace:
   oc apply -f <kamelet binding filename>

   For example:

   oc apply -f coffees-to-kafka.yaml

   The Camel K operator generates and runs a Camel K integration by using the **KameletBinding** resource. It might take a few minutes to build.
8. To see the status of the **KameletBinding** resource:
   oc get kameletbindings
9. To see the status of their integrations:
   oc get integrations
10. To view the integration's log:
    kamel logs <integration> -n <project>

    For example:

    kamel logs coffees-to-kafka -n my-camel-k-kafka

    The result is similar to the following output:

```
...
[1] INFO  [io.quarkus] (main) camel-k-integration 1.4.0 on JVM (powered by Quarkus
1.13.0.Final) started in 2.790s.
```

See Also

- Applying operations to data within a Kafka connection
- Handling errors within a connection

- Connecting a Kafka topic to a data sink in a kamelet binding

2.4. CONNECTING A KAFKA TOPIC TO A DATA SINK IN A KAMELET BINDING

To connect a Kafka topic to a data sink, you create a kamelet binding as illustrated in *Figure 2.3*.

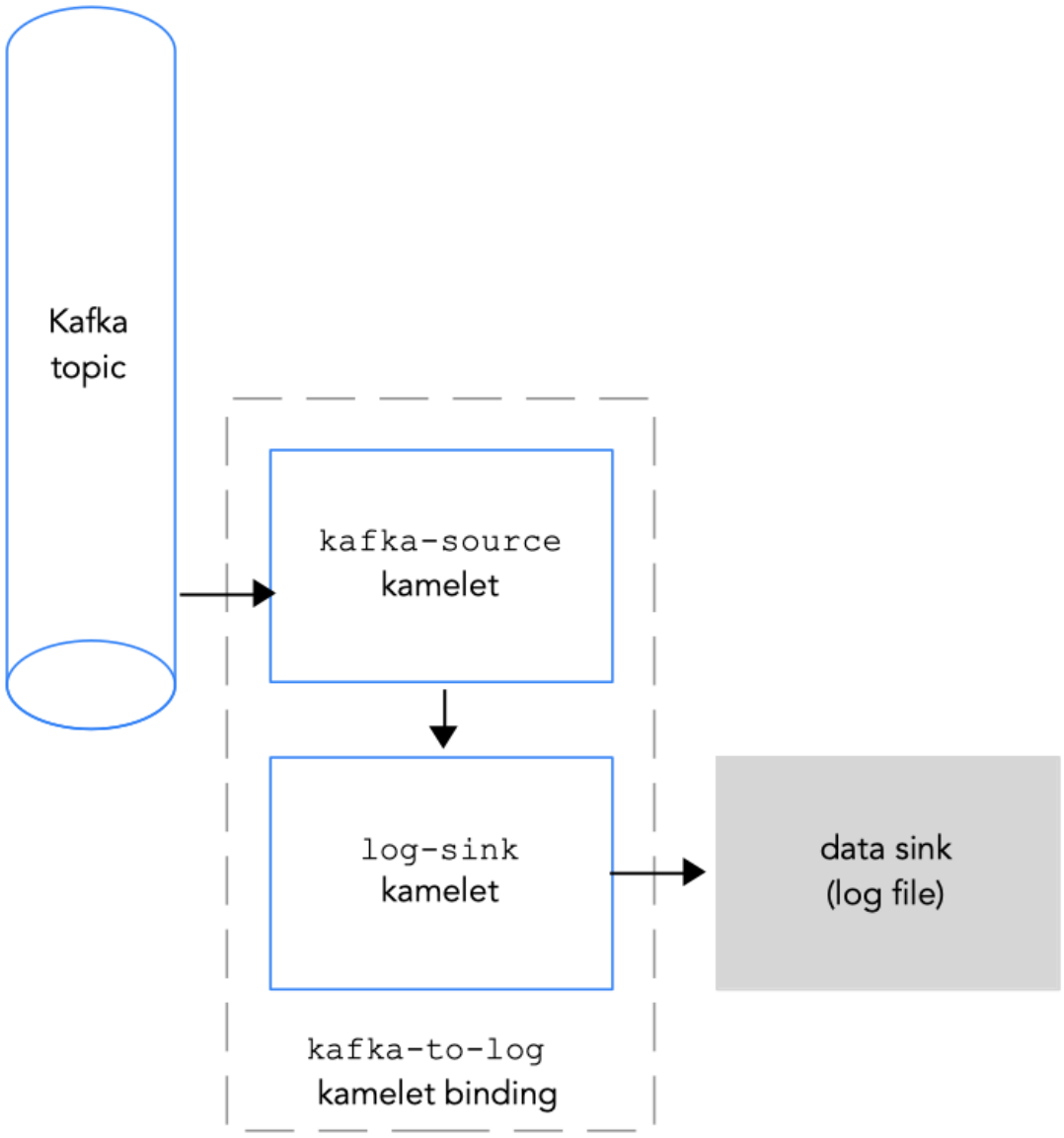


Figure 2.3 Connecting a Kafka topic to a data sink

Prerequisites

- You know the name of the Kafka topic from which you want to send events. The example in this procedure uses **test-topic** for sending events. It is the same topic that you used to receive events from the coffee source in Connecting a data source to a Kafka topic in a kamelet binding.
- You know the values of the following parameters for your Kafka instance:
  - **bootstrapServers** - A comma separated list of Kafka Broker URLs.

- **password** - The password to authenticate to Kafka.
- **user** - The user name to authenticate to Kafka.
  For information on how to obtain these values when you use OpenShift Streams, see Obtaining Kafka credentials.

  For information about Kafka authentication on AMQ streams, see Managing secure access to Kafka.

- You know the security protocol for communicating with the Kafka brokers. For a Kafka cluster on OpenShift Streams, it is **SASL_SSL** (the default). For a Kafka cluster on AMQ streams, it is **SASL/Plain**.
- You know which kamelets you want to add to your Camel K integration and the required instance parameters. The example kamelets for this procedure are:
  - The **kafka-source** kamelet provided in the Kamelet Catalog. You use the **kafka-source** kamelet because the Kafka topic is sending data (it is the data producer) in this binding. The example values for the required parameters are:
    - **bootstrapServers** - **"broker.url:9092"**
    - **password** - **"testpassword"**
    - **user** - **"testuser"**
    - **topic** - **"test-topic"**
    - **securityProtocol** - For a Kafka cluster on OpenShift Streams, you do not need to set this parameter because **SASL_SSL** is the default value. For a Kafka cluster on AMQ streams, this parameter value is **"PLAINTEXT"**.
  - The **log-sink** kamelet - You can copy the code from the Example sink kamelet to a file named **log-sink.kamelet.yaml** file and then run the following command to add it as a resource to your namespace:
    oc apply -f log-sink.kamelet.yaml

Procedure

To connect a Kafka topic to a data sink, create a kamelet binding:

1. In an editor of your choice, create a YAML file with the following basic structure:

```
apiVersion: camel.apache.org/v1alpha1
kind: KameletBinding
metadata:
  name:
spec:
  source:
  sink:
```

2. Add a name for the kamelet binding. For this example, the name is **kafka-to-log** because the binding connects the **kafka-source** kamelet to the **log-sink** kamelet.

```
apiVersion: camel.apache.org/v1alpha1
kind: KameletBinding
metadata:
```

```
  name: kafka-to-log
spec:
  source:
  sink:
```

3. For the kamelet binding's source, specify the **kafka-source** kamelet and configure its parameters.
   For example, when the Kafka cluster is on OpenShift Streams (you do not need to set the **securityProtocol** parameter):

```
apiVersion: camel.apache.org/v1alpha1
kind: KameletBinding
metadata:
  name: kafka-to-log
spec:
  source:
    ref:
      kind: Kamelet
      apiVersion: camel.apache.org/v1alpha1
      name: kafka-source
    properties:
      bootstrapServers: "broker.url:9092"
      password: "testpassword"
      topic: "test-topic"
      user: "testuser"
  sink:
```

   For example, when the Kafka cluster is on AMQ Streams you must set the **securityProtocol** parameter to **"PLAINTEXT"**:

```
apiVersion: camel.apache.org/v1alpha1
kind: KameletBinding
metadata:
  name: kafka-to-log
spec:
  source:
    ref:
      kind: Kamelet
      apiVersion: camel.apache.org/v1alpha1
      name: kafka-source
    properties:
      bootstrapServers: "broker.url:9092"
      password: "testpassword"
      topic: "test-topic"
      user: "testuser"
      securityProtocol: "PLAINTEXT"
  sink:
```

4. For the kamelet binding's sink, specify the data consumer kamelet (for example, the **log-sink** kamelet) and configure any parameters for the kamelet, for example:

```
apiVersion: camel.apache.org/v1alpha1
kind: KameletBinding
metadata:
  name: kafka-to-log
```

```
spec:
  source:
    ref:
      kind: Kamelet
      apiVersion: camel.apache.org/v1alpha1
      name: kafka-source
    properties:
      bootstrapServers: "broker.url:9092"
      password: "testpassword"
      topic: "test-topic"
      user: "testuser"
      securityProtocol: "PLAINTEXT" // only for AMQ streams
  sink:
    ref:
      kind: Kamelet
      apiVersion: camel.apache.org/v1alpha1
      name: log-sink
```

5. Save the YAML file (for example, **kafka-to-log.yaml**).
6. Log into your OpenShift project.
7. Add the kamelet binding as a resource to your OpenShift namespace:
   oc apply -f <kamelet binding filename>

   For example:

   oc apply -f kafka-to-log.yaml

   The Camel K operator generates and runs a Camel K integration by using the **KameletBinding** resource. It might take a few minutes to build.
8. To see the status of the **KameletBinding** resource:
   oc get kameletbindings
9. To see the status of their integrations:
   oc get integrations
10. To view the integration's log:
    kamel logs <integration> -n <project>

    For example:

    kamel logs kafka-to-log -n my-camel-k-kafka

    In the output, you should see coffee events, for example:

    ```
    INFO  [log-sink-E80C5C904418150-0000000000000001] (Camel (camel-1) thread #0 - timer://tick) {"id":7259,"uid":"a4ecb7c2-05b8-4a49-b0d2-d1e8db5bc5e2","blend_name":"Postmodern Symphony","origin":"Huila, Colombia","variety":"Kona","notes":"delicate, chewy, black currant, red apple, star fruit","intensifier":"balanced"}
    ```
11. To stop a running integration, delete the associated kamelet binding resource:
    oc delete kameletbindings/<kameletbinding-name>

For example:

oc delete kameletbindings/kafka-to-log

See also

- Applying operations to data within a Kafka connection
- Adding an error handler policy to a kamelet binding

2.5. APPLYING OPERATIONS TO DATA WITHIN A KAFKA CONNECTION

If you want to perform an operation on the data that passes between a kamelet and a Kafka topic, use action kamelets as intermediary steps within a kamelet binding.

- Applying operations to data within a connection
- Routing event data to different destination topics

2.5.1. Routing event data to different destination topics

When you configure a connection to a Kafka instance, you can optionally transform the topic information from the event data so that the event is routed to a different Kafka topic. Use one of the following transformation action kamelets:

- **Regex Router** - Modify the topic of a message by using a regular expression and a replacement string. For example, if you want to remove a topic prefix, add a prefix, or remove part of a topic name. Configure the Regex Router Action Kamelet (**regex-router-action**).
- **TimeStamp** - Modify the topic of a message based on the original topic and the message's timestamp. For example, when using a sink that needs to write to different tables or indexes based on timestamps. For example, when you want to write events from Kafka to Elasticsearch, but each event needs to go to a different index based on information in the event itself. Configure the Timestamp Router Action Kamelet (**timestamp-router-action**).
- **Message TimeStamp** - Modify the topic of a message based on the original topic value and the timestamp field coming from a message value field. Configure the Message Timestamp Router Action Kamelet (**message-timestamp-router-action**).
- **Predicate** - Filter events based on the given JSON path expression by configuring the Predicate Filter Action Kamelet (**predicate-filter-action**).

Prerequisites

- You have created a kamelet binding in which the sink is a **kafka-sink** kamelet, as described in Connecting a data source to a Kafka topic in a kamelet binding .
- You know which type of transformation you want to add to the kamelet binding.

Procedure

To transform the destination topic, use one of the transformation action kamelets as an intermediary step within the kamelet binding.

For details on how to add an action kamelet to a kamelet binding, see Adding an operation to a kamelet binding.

CHAPTER 3. CONNECTING TO KNATIVE WITH KAMELETS

You can connect kamelets to Knative destinations (channels or brokers). Red Hat OpenShift Serverless is based on the open source Knative project, which provides portability and consistency across hybrid and multi-cloud environments by enabling an enterprise-grade serverless platform. OpenShift Serverless includes support for the Knative Eventing and Knative Serving components.

Red Hat OpenShift Serverless, Knative Eventing, and Knative Serving enable you to use an event-driven architecture with serverless applications, decoupling the relationship between event producers and consumers by using a publish-subscribe or event-streaming model. Knative Eventing uses standard HTTP POST requests to send and receive events between event producers and consumers. These events conform to the CloudEvents specifications, which enables creating, parsing, sending, and receiving events in any programming language.

You can use kamelets to send CloudEvents to Knative and send them from Knative to event consumers. Kamelets can translate messages to CloudEvents and you can use them to apply any pre-processing and post-processing of the data within CloudEvents.

3.1. OVERVIEW OF CONNECTING TO KNATIVE WITH KAMELETS

If you use a Knative stream-processing framework, you can use kamelets to connect services and applications to a Knative destination (channel or broker).

Figure 3.1 illustrates the flow of connecting source and sink kamelets to a Knative destination.

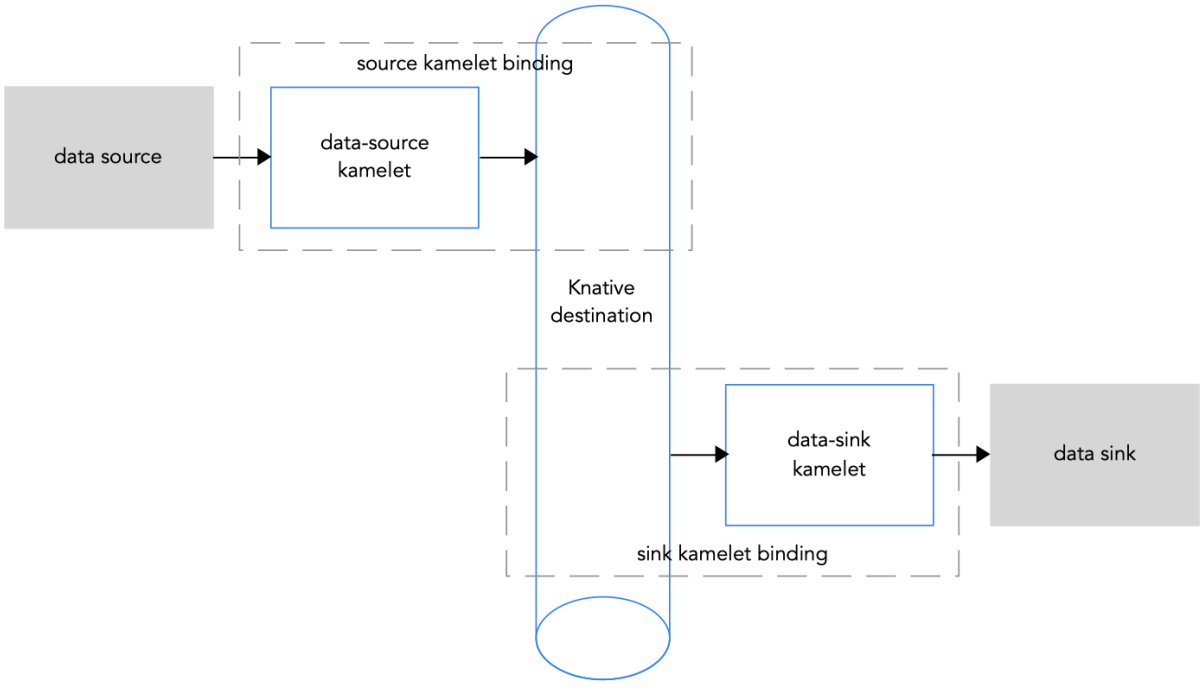


Figure 3.1: Data flow with kamelets and a Knative channel

Here is an overview of the basic steps for using kamelets and kamelet bindings to connect applications and services to a Knative destination:

1. Set up Knative:
    a. Prepare your OpenShift cluster by installing the Camel K and OpenShift Serverless

operators.

b. Install the required Knative Serving and Eventing components.

c. Create a Knative channel or broker.

2. Determine which services or applications you want to connect to your Knative channel or broker.

3. View the Kamelet Catalog to find the kamelets for the source and sink components that you want to add to your integration. Also, determine the required configuration parameters for each kamelet that you want to use.

4. Create kamelet bindings:

   - Create a kamelet binding that connects a source kamelet to a Knative channel (or broker).
   - Create a kamelet binding that connects the Knative channel (or broker) to a sink kamelet.

5. Optionally, manipulate the data that is passing between the Knative channel (or broker) and the data source or sink by adding one or more action kamelets as intermediary steps within a kamelet binding.

6. Optionally, define how to handle errors within a kamelet binding.

7. Apply the kamelet bindings as resources to the project.

The Camel K operator generates a separate Camel integration for each kamelet binding.

When you configure a kamelet binding to use a Knative channel or a broker as the source of events, the Camel K operator materializes the corresponding integration as a Knative Serving service, to leverage the auto-scaling capabilities offered by Knative.

3.2. SETTING UP KNATIVE

Setting up Knative involves installing the required OpenShift operators and creating a Knative channel.

3.2.1. Preparing your OpenShift cluster

To use kamelets and OpenShift Serverless, install the following operators, components, and CLI tools:

- **Red Hat Integration - Camel K** operator and CLI tool - The operator installs and manages Camel K - a lightweight integration framework that runs natively in the cloud on OpenShift. The **kamel** CLI tool allows you to access all Camel K features.
  See the installation instructions in Installing Camel K .

- **OpenShift Serverless** operator - Provides a collection of APIs that enables containers, microservices, and functions to run "serverless". Serverless applications can scale up and down (to zero) on demand and be triggered by a number of event sources. When you install the OpenShift Serverless operator, it automatically creates the **knative-serving** namespace (for installing the Knative Serving component) and the **knative-eventing** namespace (required for installing the Knative Eventing component).

- **Knative Eventing** component

- **Knative Serving** component

- **Knative CLI** tool (**kn**) - Allows you to create Knative resources from the command line or from within Shell scripts.

3.2.1.1. Installing OpenShift Serverless

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

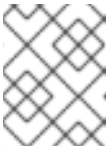
The OpenShift Serverless Operator supports both Knative Serving and Knative Eventing features. For more details, see Getting started with OpenShift Serverless.

Prerequisites

- You have cluster administrator access to an OpenShift project in which the Camel K Operator is installed.
- You installed the OpenShift CLI tool (**oc**) so that you can interact with the OpenShift cluster at the command line. For details on how to install the OpenShift CLI, see Installing the OpenShift CLI.

Procedure

1. In the OpenShift Container Platform web console, log in by using an account with cluster administrator privileges.
2. In the left navigation menu, click **Operators** > **OperatorHub**.
3. In the **Filter by keyword** text box, enter **Serverless** to find the **OpenShift Serverless Operator**.
4. Read the information about the Operator and then click **Install** to display the Operator subscription page.
5. Select the default subscription settings:
   - **Update Channel** > Select the channel that matches your OpenShift version, for example, **4.7**
   - **Installation Mode** > **All namespaces on the cluster**
   - **Approval Strategy** > **Automatic**

   NOTE

   The **Approval Strategy** > **Manual** setting is also available if required by your environment.
6. Click **Install**, and wait a few moments until the Operator is ready for use.
7. Install the required Knative components using the steps in the OpenShift documentation:
   - Installing Knative Serving
   - Installing Knative Eventing

8. (Optional) Download and install the OpenShift Serverless CLI tool:

    a. From the Help menu (?) at the top of the OpenShift web console, select **Command line tools**.

    b. Scroll down to the **kn - OpenShift Serverless - Command Line Interface** section.

    c. Click the link to download the binary for your local operating system (Linux, Mac, Windows)

    d. Unzip and install the CLI in your system path.

    e. To verify that you can access the **kn** CLI, open a command window and then type the following:
    kn --help

    This command shows information about OpenShift Serverless CLI commands.

    For more details, see the OpenShift Serverless CLI documentation.

Additional resources

- Installing OpenShift Serverless in the OpenShift documentation

3.2.2. Creating a Knative channel

A Knative channel is a custom resource that forwards events. After events have been sent to a channel from an event source or producer, these events can be sent to multiple Knative services, or other sinks, by using a subscription.

This example uses an **InMemoryChannel** channel, which you use with OpenShift Serverless for development purposes. Note that **InMemoryChannel** type channels have the following limitations:

- No event persistence is available. If a pod goes down, events on that pod are lost.
- **InMemoryChannel** channels do not implement event ordering, so two events that are received in the channel at the same time can be delivered to a subscriber in any order.
- If a subscriber rejects an event, there are no re-delivery attempts by default. You can configure re-delivery attempts by modifying the delivery spec in the Subscription object.

Prerequisites

- The OpenShift Serverless operator, Knative Eventing, and Knative Serving components are installed on your OpenShift Container Platform cluster.
- You have installed the OpenShift Serverless CLI (**kn**).
- You have created a project or have access to a project with the appropriate roles and permissions to create applications and other workloads in OpenShift Container Platform.

Procedure

1. Log in to your OpenShift cluster.
2. Open the project in which you want to create your integration application. For example:
    oc project camel-k-knative

3. Create a channel by using the Knative (**kn**) CLI command
   kn channel create <channel_name> --type <channel_type>

   For example, to create a channel named **mychannel**:

   kn channel create mychannel --type messaging.knative.dev:v1:InMemoryChannel

4. To confirm that the channel now exists, type the following command to list all existing channels:
   kn channel list

   You should see your channel in the list.

Next steps

- Connecting a data source to a Knative destination in a kamelet binding
- Connecting a Knative destination to a data sink in a kamelet binding

3.2.3. Creating a Knative broker

A Knative broker is a custom resource that defines an event mesh for collecting a pool of CloudEvents. OpenShift Serverless provides a default Knative broker that you can create by using the **kn** CLI.

You can use a broker in a kamelet binding, for example, when your application handles multiple event types and you do not want to create a channel for each event type.

Prerequisites

- The OpenShift Serverless operator, Knative Eventing, and Knative Serving components are installed on your OpenShift Container Platform cluster.
- You have installed the OpenShift Serverless CLI (**kn**).
- You have created a project or have access to a project with the appropriate roles and permissions to create applications and other workloads in OpenShift Container Platform.

Procedure

1. Log in to your OpenShift cluster.
2. Open the project in which you want to create your integration application. For example:
   oc project camel-k-knative
3. Create the broker by using this Knative (**kn**) CLI command:
   kn broker create default
4. To confirm that the broker now exists, type the following command to list all existing brokers:
   kn broker list

You should see the default broker in the list.

Next steps

- Connecting a data source to a Knative destination in a kamelet binding

- Connecting a Knative destination to a data sink in a kamelet binding

3.3. CONNECTING A DATA SOURCE TO A KNATIVE DESTINATION IN A KAMELET BINDING

To connect a data source to a Knative destination (channel or broker), you create a kamelet binding as illustrated in *Figure 3.2*.

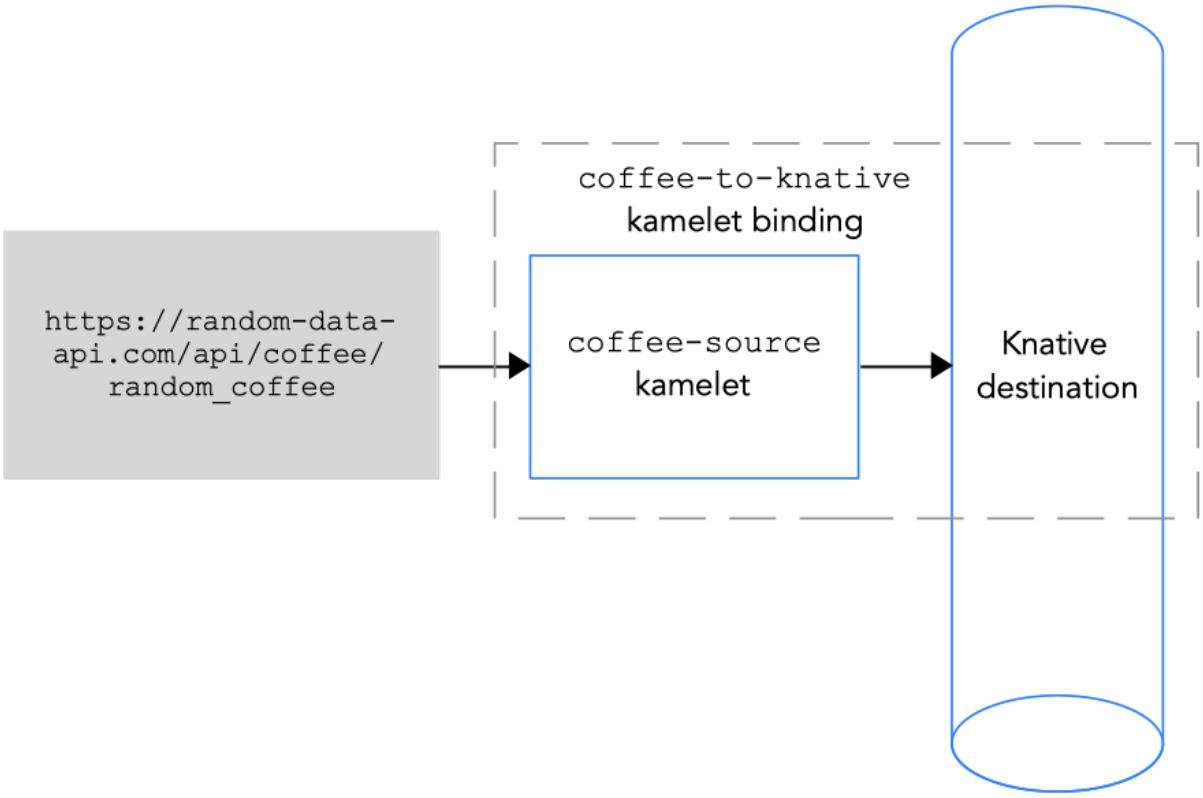


Figure 3.2 Connecting a data source to a Knative destination

The Knative destination can be a Knative channel or a Knative broker.

When you send data to a channel, there is only one event type for the channel. You do not need to specify any property values for the channel in a kamelet binding.

When you send data to a broker, because the broker can handle more than one event type, you must specify a value for the type property when you reference the broker in a kamelet binding.

Prerequisites

- You know the name and type of the Knative channel or broker to which you want to send events. The example in this procedure uses the **InMemoryChannel** channel named **mychannel** or the broker named **default**. For the broker example, the **type** property value is **coffee** for coffee events.
- You know which kamelet you want to add to your Camel integration and the required instance parameters.
  The example kamelet for this procedure is the **coffee-source** kamelet. It has an optional parameter, **period**, that specifies how often to send each event. You can copy the code from

Example source kamelet to a file named **coffee-source.kamelet.yaml** file and then run the following command to add it as a resource to your namespace:

oc apply -f coffee-source.kamelet.yaml

Procedure

To connect a data source to a Knative destination, create a kamelet binding:

1. In an editor of your choice, create a YAML file with the following basic structure:

```
apiVersion: camel.apache.org/v1alpha1
kind: KameletBinding
metadata:
  name:
spec:
  source:
  sink:
```

2. Add a name for the kamelet binding. For this example, the name is **coffees-to-knative** because the binding connects the **coffee-source** kamelet to a Knative destination.

```
apiVersion: camel.apache.org/v1alpha1
kind: KameletBinding
metadata:
  name: coffees-to-knative
spec:
  source:
  sink:
```

3. For the kamelet binding's source, specify a data source kamelet (for example, the **coffee-source** kamelet produces events that contain data about coffee) and configure any parameters for the kamelet.

```
apiVersion: camel.apache.org/v1alpha1
kind: KameletBinding
metadata:
  name: coffees-to-knative
spec:
  source:
    ref:
      kind: Kamelet
      apiVersion: camel.apache.org/v1alpha1
      name: coffee-source
    properties:
      period: 5000
  sink:
```

4. For the kamelet binding's sink specify the Knative channel or broker and the required parameters.
This example specifies a Knative channel as the sink:

```
apiVersion: camel.apache.org/v1alpha1
kind: KameletBinding
metadata:
```

```
  name: coffees-to-knative
spec:
  source:
    ref:
      kind: Kamelet
      apiVersion: camel.apache.org/v1alpha1
      name: coffee-source
    properties:
      period: 5000
  sink:
    ref:
      apiVersion: messaging.knative.dev/v1
      kind: InMemoryChannel
      name: mychannel
```

This example specifies a Knative broker as the sink:

```
apiVersion: camel.apache.org/v1alpha1
kind: KameletBinding
metadata:
  name: coffees-to-knative
spec:
  source:
    ref:
      kind: Kamelet
      apiVersion: camel.apache.org/v1alpha1
      name: coffee-source
    properties:
      period: 5000
  sink:
      ref:
        kind: Broker
        apiVersion: eventing.knative.dev/v1
        name: default
      properties:
        type: coffee
```

5. Save the YAML file (for example, **coffees-to-knative.yaml**).
6. Log into your OpenShift project.
7. Add the kamelet binding as a resource to your OpenShift namespace:
   oc apply -f <kamelet binding filename>

   For example:

   oc apply -f coffees-to-knative.yaml

   The Camel K operator generates and runs a Camel K integration by using the **KameletBinding** resource. It might take a few minutes to build.
8. To see the status of the **KameletBinding**:
   oc get kameletbindings
9. To see the status of their integrations:
   oc get integrations

10. To view the integration's log:
kamel logs <integration> -n <project>

    For example:

    kamel logs coffees-to-knative -n my-camel-knative

    The result is similar to the following output:

```
...
[1] INFO  [io.quarkus] (main) camel-k-integration 1.4.0 on JVM (powered by Quarkus
1.13.0.Final) started in 2.790s.
```

Next steps

- Connecting a Knative destination to a data sink in a kamelet binding

See also

- Applying operations to data within a connection
- Handling errors within a connection

3.4. CONNECTING A KNATIVE DESTINATION TO A DATA SINK IN A KAMELET BINDING

To connect a Knative destination to a data sink, you create a kamelet binding as illustrated in *Figure 3.3*.

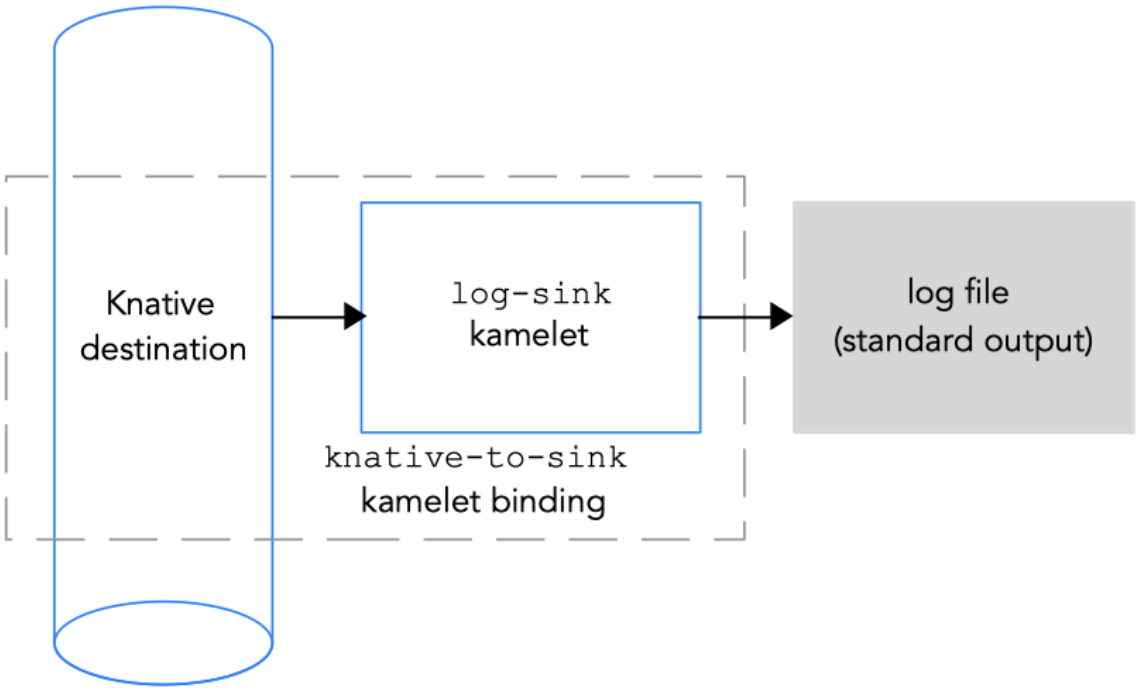


Figure 3.3 Connecting a Knative destination to a data sink

The Knative destination can be a Knative channel or a Knative broker.

When you send data from a channel, there is only one event type for the channel. You do not need to specify any property values for the channel in a kamelet binding.

When you send data from a broker, because the broker can handle more than one event type, you must specify a value for the type property when you reference the broker in a kamelet binding.

Prerequisites

- You know the name and type of the Knative channel or the name of the broker from which you want to receive events. For a broker, you also know the type of events that you want to receive. The example in this procedure uses the InMemoryChannel channel named mychannel or the broker named mybroker and coffee events (for the type property). These are the same example destinations that are used to receive events from the coffee source in Connecting a data source to a Knative channel in a kamelet binding.

- You know which kamelet you want to add to your Camel integration and the required instance parameters.
  The example kamelet for this procedure is the **log-sink** kamelet. You can copy the code from the Example sink kamelet to a file named **log-sink.kamelet.yaml** file and then run the following command to add it as a resource to your namespace:

  oc apply -f log-sink.kamelet.yaml

Procedure

To connect a Knative channel to a data sink, create a kamelet binding:

1. In an editor of your choice, create a YAML file with the following basic structure:

```
apiVersion: camel.apache.org/v1alpha1
kind: KameletBinding
metadata:
  name:
spec:
  source:
  sink:
```

2. Add a name for the kamelet binding. For this example, the name is **knative-to-log** because the binding connects the Knative destination to the **log-sink** kamelet.

```
apiVersion: camel.apache.org/v1alpha1
kind: KameletBinding
metadata:
  name: knative-to-log
spec:
  source:
  sink:
```

3. For the kamelet binding's source, specify the Knative channel or broker and the required parameters.
   This example specifies a Knative channel as the source:

```
apiVersion: camel.apache.org/v1alpha1
kind: KameletBinding
metadata:
  name: knative-to-log
spec:
  source:
    ref:
          apiVersion: messaging.knative.dev/v1
          kind: InMemoryChannel
          name: mychannel
  sink:
```

This example specifies a Knative broker as the source:

```
apiVersion: camel.apache.org/v1alpha1
kind: KameletBinding
metadata:
  name: knative-to-log
spec:
  source:
    ref:
         kind: Broker
         apiVersion: eventing.knative.dev/v1
         name: default
        properties:
        type: coffee
sink:
```

4. For the kamelet binding's sink, specify the data consumer kamelet (for example, the log-sink kamelet) and configure any parameters for the kamelet, for example:

```
apiVersion: camel.apache.org/v1alpha1
kind: KameletBinding
metadata:
  name: knative-to-log
spec:
  source:
    ref:
          apiVersion: messaging.knative.dev/v1
          kind: InMemoryChannel
          name: mychannel
  sink:
    ref:
      apiVersion: camel.apache.org/v1alpha1
      kind: Kamelet
      name: log-sink
```

5. Save the YAML file (for example, **knative-to-log.yaml**).

6. Log into your OpenShift project.

7. Add the kamelet binding as a resource to your OpenShift namespace: **oc apply -f <kamelet binding filename>**
   For example:

oc apply -f knative-to-log.yaml

The Camel K operator generates and runs a Camel K integration by using the **KameletBinding** resource. It might take a few minutes to build.

8. To see the status of the **KameletBinding**:
oc get kameletbindings

9. To see the status of the integration:
oc get integrations

10. To view the integration's log:
kamel logs <integration> -n <project>

For example:

kamel logs knative-to-log -n my-camel-knative

In the output, you should see coffee events, for example:

```
[1] 2021-07-23 13:06:38,111 INFO [sink] (vert.x-worker-thread-1) {"id":254,"uid":"8e180ef7-8924-4fc7-ab81-d6058618cc42","blend_name":"Good-morning Star","origin":"Santander, Colombia","variety":"Kaffa","notes":"delicate, creamy, lemongrass, granola, soil","intensifier":"sharp"}
[1] 2021-07-23 13:06:43,273 INFO [sink] (vert.x-worker-thread-2) {"id":8169,"uid":"3733c3a5-4ad9-43a3-9acc-d4cd43de6f3d","blend_name":"Caf? Java","origin":"Nayarit, Mexico","variety":"Red Bourbon","notes":"unbalanced, full, granola, bittersweet chocolate, nougat","intensifier":"delicate"}
```

11. To stop a running integration, delete the associated kamelet binding resource:
oc delete kameletbindings/<kameletbinding-name>

For example:

oc delete kameletbindings/knative-to-log

See also

- Applying operations to data within a connection
- Handling errors within a connection

CHAPTER 4. KAMELETS REFERENCE

4.1. KAMELET STRUCTURE

A kamelet is typically coded in the YAML domain-specific language. The file name prefix is the name of the kamelet. For example, a kamelet with the name **FTP sink** has the filename **ftp-sink.kamelet.yaml**.

Note that in OpenShift, a kamelet is a resource that shows the name of the kamelet (not the filename).

At a high level, a kamelet resource describes:

- A metadata section containing the ID of the kamelet and other information, such as the type of kamelet (**source**, **sink**, or **action**).
- A definition (JSON-schema specification) that contains a set of parameters that you can use to configure the kamelet.
- An optional **types** section containing information about input and output expected by the kamelet.
- A Camel flow in YAML DSL that defines the implementation of the kamelet.

The following diagram shows an example of a kamelet and its parts.

Example kamelet structure

```
telegram-text-source.kamelet.yaml
apiVersion: camel.apache.org/v1alpha1
kind: Kamelet
metadata:
  name: telegram-source 1
  annotations: 2
    camel.apache.org/catalog.version: "master-SNAPSHOT"
    camel.apache.org/kamelet.icon: "data:image/..."
    camel.apache.org/provider: "Red Hat"
    camel.apache.org/kamelet.group: "Telegram"
  labels: 3
    camel.apache.org/kamelet.type: "source"
spec:
  definition: 4
    title: "Telegram Source"
    description: |-
      Receive all messages that people send to your telegram bot.
    required:
      - authorizationToken
    type: object
    properties:
      authorizationToken:
        title: Token
        description: The token to access your bot on Telegram, that you
              can obtain from the Telegram "Bot Father".
        type: string
        format: password
      x-descriptors:
      - urn:alm:descriptor:com.tectonic.ui:password
```

```
types: 5
  out:
    mediaType: application/json
dependencies:
- "camel:jackson"
- "camel:kamelet"
- "camel:telegram"
flow: 6
  from:
     uri: telegram:bots
     parameters:
        authorizationToken: "{{authorizationToken}}"
     steps:
     - marshal:
       json: {}
     - to: "kamelet:sink"
```

1. The kamelet ID - Use this ID in Camel K integrations when you want to reference the kamelet.
2. Annotations, such as icon, provide display features for the kamelet.
3. Labels allow a user to query kamelets (for example, by kind: "source", "sink", or “action”)
4. Description of the kamelet and parameters in JSON-schema specification format.
5. The media type of the output (can include a schema).
6. The route template that defines the behavior of the kamelet.

4.2. EXAMPLE SOURCE KAMELET

Here is the content of the example **coffee-source** kamelet:

```
apiVersion: camel.apache.org/v1alpha1
kind: Kamelet
metadata:
  name: coffee-source
  labels:
    camel.apache.org/kamelet.type: "source"
spec:
  definition:
    title: "Coffee Source"
    description: "Retrieve a random coffee from a catalog of coffees"
    properties:
      period:
        title: Period
        description: The interval between two events in seconds
        type: integer
        default: 1000
  types:
    out:
      mediaType: application/json
  flow:
    from:
      uri: timer:tick
```

```
    parameters:
      period: "{{period}}"
    steps:
    - to: "https://random-data-api.com/api/coffee/random_coffee"
    - to: "kamelet:sink"
```

4.3. EXAMPLE SINK KAMELET

Here is the content of the example **log-sink** kamelet:

```
apiVersion: camel.apache.org/v1alpha1
kind: Kamelet
metadata:
  name: log-sink
  labels:
    camel.apache.org/kamelet.type: "sink"
spec:
  definition:
    title: "Log Sink"
    description: "Consume events"
  flow:
    from:
      uri: "kamelet:source"
      steps:
      - convert-body-to: 'java.lang.String'
      - log: "${body}"
```