Red Hat OpenShift Serverless 1.34

Eventing

Using event-driven architectures with OpenShift Serverless

Last Updated: 2024-10-10

Red Hat OpenShift Serverless 1.34 Eventing

Using event-driven architectures with OpenShift Serverless

Legal Notice

Copyright © 2024 Red Hat, Inc.

The text of and illustrations in this document are licensed by Red Hat under a Creative Commons Attribution–Share Alike 3.0 Unported license ("CC-BY-SA"). An explanation of CC-BY-SA is available at
http://creativecommons.org/licenses/by-sa/3.0/
. In accordance with CC-BY-SA, if you distribute this document or an adaptation of it, you must provide the URL for the original version.

Red Hat, as the licensor of this document, waives the right to enforce, and agrees not to assert, Section 4d of CC-BY-SA to the fullest extent permitted by applicable law.

Red Hat, Red Hat Enterprise Linux, the Shadowman logo, the Red Hat logo, JBoss, OpenShift, Fedora, the Infinity logo, and RHCE are trademarks of Red Hat, Inc., registered in the United States and other countries.

Linux ® is the registered trademark of Linus Torvalds in the United States and other countries.

Java ® is a registered trademark of Oracle and/or its affiliates.

XFS ® is a trademark of Silicon Graphics International Corp. or its subsidiaries in the United States and/or other countries.

MySQL ® is a registered trademark of MySQL AB in the United States, the European Union and other countries.

Node.js ® is an official trademark of Joyent. Red Hat is not formally related to or endorsed by the official Joyent Node.js open source or commercial project.

The OpenStack ® Word Mark and OpenStack logo are either registered trademarks/service marks or trademarks/service marks of the OpenStack Foundation, in the United States and other countries and are used with the OpenStack Foundation's permission. We are not affiliated with, endorsed or sponsored by the OpenStack Foundation, or the OpenStack community.

All other trademarks are the property of their respective owners.

Abstract

This document provides information about Eventing features such as event sources and sinks, brokers, triggers, channels, and subscriptions.

Table of Contents

CHAPTER 1. KNATIVE EVENTING **6**
1.1. KNATIVE EVENTING USE CASES: 6

CHAPTER 2. EVENT SOURCES **7**
2.1. EVENT SOURCES 7
2.2. EVENT SOURCE IN THE ADMINISTRATOR PERSPECTIVE 7
2.2.1. Creating an event source by using the Administrator perspective 7
2.3. CREATING AN API SERVER SOURCE 8
2.3.1. Creating an API server source by using the web console 8
2.3.2. Creating an API server source by using the Knative CLI 11
2.3.2.1. Knative CLI sink flag 14
2.3.3. Creating an API server source by using YAML files 15
2.4. CREATING A PING SOURCE 19
2.4.1. Creating a ping source by using the web console 19
2.4.2. Creating a ping source by using the Knative CLI 22
2.4.2.1. Knative CLI sink flag 23
2.4.3. Creating a ping source by using YAML 24
2.5. SOURCE FOR APACHE KAFKA 27
2.5.1. Creating an Apache Kafka event source by using the web console 27
2.5.2. Creating an Apache Kafka event source by using the Knative CLI 28
2.5.2.1. Knative CLI sink flag 30
2.5.3. Creating an Apache Kafka event source by using YAML 30
2.5.4. Configuring SASL authentication for Apache Kafka sources 32
2.5.5. Configuring KEDA autoscaling for KafkaSource 33
2.6. CUSTOM EVENT SOURCES 34
2.6.1. Sink binding 34
2.6.1.1. Creating a sink binding by using YAML 35
2.6.1.2. Creating a sink binding by using the Knative CLI 38
2.6.1.2.1. Knative CLI sink flag 41
2.6.1.3. Creating a sink binding by using the web console 41
2.6.1.4. Sink binding reference 45
2.6.1.4.1. Subject parameter 46
2.6.1.4.2. CloudEvent overrides 48
2.6.1.4.3. The include label 49
2.6.1.5. Integrating Service Mesh with a sink binding 49
2.6.2. Container source 51
2.6.2.1. Guidelines for creating a container image 52
2.6.2.2. Creating and managing container sources by using the Knative CLI 55
2.6.2.3. Creating a container source by using the web console 56
2.6.2.4. Container source reference 56
2.6.2.4.1. CloudEvent overrides 58
2.6.2.5. Integrating Service Mesh with ContainerSource 58
2.7. CONNECTING AN EVENT SOURCE TO AN EVENT SINK BY USING THE DEVELOPER PERSPECTIVE 60
2.7.1. Connect an event source to an event sink by using the Developer perspective 60

CHAPTER 3. EVENT SINKS **62**
3.1. EVENT SINKS 62
3.1.1. Knative CLI sink flag 62
3.2. CREATING EVENT SINKS 62
3.3. SINK FOR APACHE KAFKA 63
3.3.1. Creating an Apache Kafka sink by using YAML 63

3.3.2. Creating an event sink for Apache Kafka by using the OpenShift Container Platform web console 64
3.3.3. Configuring security for Apache Kafka sinks 65

CHAPTER 4. BROKERS 68
4.1. BROKERS 68
4.2. BROKER TYPES 68
4.2.1. Default broker implementation for development purposes 68
4.2.2. Production-ready Knative broker implementation for Apache Kafka 68
4.3. CREATING BROKERS 69
4.3.1. Creating a broker by using the Knative CLI 69
4.3.2. Creating a broker by annotating a trigger 70
4.3.3. Creating a broker by labeling a namespace 71
4.3.4. Deleting a broker that was created by injection 73
4.3.5. Creating a broker by using the web console 73
4.3.6. Creating a broker by using the Administrator perspective 74
4.3.7. Next steps 75
4.3.8. Additional resources 75
4.4. CONFIGURING THE DEFAULT BROKER BACKING CHANNEL 75
4.5. CONFIGURING THE DEFAULT BROKER CLASS 76
4.6. KNATIVE BROKER IMPLEMENTATION FOR APACHE KAFKA 78
4.6.1. Creating an Apache Kafka broker when it is not configured as the default broker type 78
4.6.1.1. Creating an Apache Kafka broker by using YAML 78
4.6.1.2. Creating an Apache Kafka broker that uses an externally managed Kafka topic 79
4.6.1.3. Knative Broker implementation for Apache Kafka with isolated data plane 80
4.6.1.4. Creating a Knative broker for Apache Kafka that uses an isolated data plane 81
4.6.2. Configuring Apache Kafka broker settings 82
4.6.3. Security configuration for the Knative broker implementation for Apache Kafka 84
4.6.3.1. Configuring TLS authentication for Apache Kafka brokers 84
4.6.3.2. Configuring SASL authentication for Apache Kafka brokers 85
4.6.4. Additional resources 86
4.7. MANAGING BROKERS 87
4.7.1. Managing brokers using the CLI 87
4.7.1.1. Listing existing brokers by using the Knative CLI 87
4.7.1.2. Describing an existing broker by using the Knative CLI 87
4.7.2. Connect a broker to a sink using the Developer perspective 88

CHAPTER 5. TRIGGERS 90
5.1. TRIGGERS OVERVIEW 90
5.1.1. Configuring event delivery ordering for triggers 90
5.1.2. Next steps 91
5.2. CREATING TRIGGERS 91
5.2.1. Creating a trigger by using the Administrator perspective 91
5.2.2. Creating a trigger by using the Developer perspective 92
5.2.3. Creating a trigger by using the Knative CLI 93
5.3. LIST TRIGGERS FROM THE COMMAND LINE 93
5.3.1. Listing triggers by using the Knative CLI 94
5.4. DESCRIBE TRIGGERS FROM THE COMMAND LINE 94
5.4.1. Describing a trigger by using the Knative CLI 94
5.5. CONNECTING A TRIGGER TO A SINK 95
5.6. FILTERING TRIGGERS FROM THE COMMAND LINE 95
5.6.1. Filtering events with triggers by using the Knative CLI 96
5.7. ADVANCED TRIGGER FILTERS 96
5.7.1. Advanced trigger filters overview 96

5.7.2. Supported filter dialects 97
5.7.2.1. exact filter dialect 97
5.7.2.2. prefix filter dialect 97
5.7.2.3. suffix filter dialect 98
5.7.2.4. all filter dialect 98
5.7.2.5. any filter dialect 99
5.7.2.6. not filter dialect 99
5.7.2.7. cesql filter dialect 99
5.7.3. Conflict with the existing filter field 100
5.8. UPDATING TRIGGERS FROM THE COMMAND LINE 100
5.8.1. Updating a trigger by using the Knative CLI 100
5.9. DELETING TRIGGERS FROM THE COMMAND LINE 101
5.9.1. Deleting a trigger by using the Knative CLI 101

CHAPTER 6. CHANNELS 103
6.1. CHANNELS AND SUBSCRIPTIONS 103
6.1.1. Channel implementation types 104
6.2. CREATING CHANNELS 104
6.2.1. Creating a channel by using the Administrator perspective 104
6.2.2. Creating a channel by using the Developer perspective 105
6.2.3. Creating a channel by using the Knative CLI 106
6.2.4. Creating a default implementation channel by using YAML 106
6.2.5. Creating a channel for Apache Kafka by using YAML 107
6.2.6. Next steps 108
6.3. CONNECTING CHANNELS TO SINKS 108
6.3.1. Creating a subscription by using the Developer perspective 108
6.3.2. Creating a subscription by using YAML 109
6.3.3. Creating a subscription by using the Knative CLI 110
6.3.4. Creating a subscription by using the Administrator perspective 112
6.3.5. Next steps 113
6.4. DEFAULT CHANNEL IMPLEMENTATION 113
6.4.1. Configuring the default channel implementation 113
6.5. SECURITY CONFIGURATION FOR CHANNELS 114
6.5.1. Configuring TLS authentication for Knative channels for Apache Kafka 114
6.5.2. Configuring SASL authentication for Knative channels for Apache Kafka 115

CHAPTER 7. SUBSCRIPTIONS 118
7.1. CREATING SUBSCRIPTIONS 118
7.1.1. Creating a subscription by using the Administrator perspective 118
7.1.2. Creating a subscription by using the Developer perspective 118
7.1.3. Creating a subscription by using YAML 120
7.1.4. Creating a subscription by using the Knative CLI 121
7.1.5. Next steps 122
7.2. MANAGING SUBSCRIPTIONS 123
7.2.1. Describing subscriptions by using the Knative CLI 123
7.2.2. Listing subscriptions by using the Knative CLI 123
7.2.3. Updating subscriptions by using the Knative CLI 124

CHAPTER 8. EVENT DELIVERY 125
8.1. CONFIGURABLE EVENT DELIVERY PARAMETERS 125
8.2. EXAMPLES OF CONFIGURING EVENT DELIVERY PARAMETERS 125
8.3. CONFIGURING EVENT DELIVERY ORDERING FOR TRIGGERS 127

CHAPTER 9. EVENT DISCOVERY 129

9.1. LISTING EVENT SOURCES AND EVENT SOURCE TYPES 129
9.2. LISTING EVENT SOURCE TYPES FROM THE COMMAND LINE 129
9.2.1. Listing available event source types by using the Knative CLI 129
9.3. LISTING EVENT SOURCE TYPES FROM THE DEVELOPER PERSPECTIVE 129
9.3.1. Viewing available event source types within the Developer perspective 130
9.4. LISTING EVENT SOURCES FROM THE COMMAND LINE 130
9.4.1. Listing available event sources by using the Knative CLI 130

CHAPTER 10. TUNING EVENTING CONFIGURATION 132
10.1. OVERRIDING KNATIVE EVENTING SYSTEM DEPLOYMENT CONFIGURATIONS 132
10.1.1. Overriding deployment configurations 132
10.1.2. Modifying consumer group IDs and topic names 133
10.2. HIGH AVAILABILITY 135
10.2.1. Configuring high availability replicas for Knative Eventing 135
10.2.2. Configuring high availability replicas for the Knative broker implementation for Apache Kafka 137
10.2.3. Overriding disruption budgets 138

CHAPTER 11. CONFIGURING TLS ENCRYPTION IN EVENTING 140
11.1. CREATING A SELFSIGNED CLUSTERISSUER RESOURCE FOR EVENTING 140
11.2. CREATING A CLUSTERISSUER RESOURCE FOR EVENTING 142
11.3. ENABLING TRANSPORT ENCRPTION FOR KNATIVE EVENTING 143
11.4. CONFIGURING ADDITIONAL CA TRUST BUNDLES 143
11.5. CONFIGURE CUSTOM EVENT SOURCES TO TRUST THE EVENTING CA 144
11.6. ADDING A SELFSIGNED CLUSTERISSUER RESOURCE TO CA TRUST BUNDLES 145
11.7. ENSURING SEAMLESS CA ROTATION 145
11.8. VERIFYING TRANSPORT ENCRYPTION IN EVENTING 146

CHAPTER 12. CONFIGURING KUBE-RBAC-PROXY FOR EVENTING 148
12.1. CONFIGURING KUBE-RBAC-PROXY RESOURCES FOR EVENTING 148
12.2. CONFIGURING KUBE-RBAC-PROXY RESOURCES FOR KNATIVE FOR APACHE KAFKA 148

CHAPTER 13. USING CONTAINERSOURCE WITH SERVICE MESH 150
13.1. CONFIGURING CONTAINERSOURCE WITH SERVICE MESH 150

CHAPTER 14. USING A SINK BINDING WITH SERVICE MESH 153
14.1. CONFIGURING A SINK BINDING WITH SERVICE MESH 153

CHAPTER 1. KNATIVE EVENTING

Knative Eventing on OpenShift Container Platform enables developers to use an event-driven architecture with serverless applications. An event-driven architecture is based on the concept of decoupled relationships between event producers and event consumers.

Event producers create events, and event *sinks*, or consumers, receive events. Knative Eventing uses standard HTTP POST requests to send and receive events between event producers and sinks. These events conform to the CloudEvents specifications, which enables creating, parsing, sending, and receiving events in any programming language.

1.1. KNATIVE EVENTING USE CASES:

Knative Eventing supports the following use cases:

Publish an event without creating a consumer

You can send events to a broker as an HTTP POST, and use binding to decouple the destination configuration from your application that produces events.

Consume an event without creating a publisher

You can use a trigger to consume events from a broker based on event attributes. The application receives events as an HTTP POST.

To enable delivery to multiple types of sinks, Knative Eventing defines the following generic interfaces that can be implemented by multiple Kubernetes resources:

Addressable resources

Able to receive and acknowledge an event delivered over HTTP to an address defined in the **status.address.url** field of the event. The Kubernetes **Service** resource also satisfies the addressable interface.

Callable resources

Able to receive an event delivered over HTTP and transform it, returning **0** or **1** new events in the HTTP response payload. These returned events may be further processed in the same way that events from an external event source are processed.

CHAPTER 2. EVENT SOURCES

2.1. EVENT SOURCES

A Knative *event source* can be any Kubernetes object that generates or imports cloud events, and relays those events to another endpoint, known as a *sink*. Sourcing events is critical to developing a distributed system that reacts to events.

You can create and manage Knative event sources by using the **Developer** perspective in the OpenShift Container Platform web console, the Knative (**kn**) CLI, or by applying YAML files.

Currently, OpenShift Serverless supports the following event source types:

API server source
: Brings Kubernetes API server events into Knative. The API server source sends a new event each time a Kubernetes resource is created, updated or deleted.

Ping source
: Produces events with a fixed payload on a specified cron schedule.

Kafka event source
: Connects an Apache Kafka cluster to a sink as an event source.

You can also create a custom event source.

2.2. EVENT SOURCE IN THE ADMINISTRATOR PERSPECTIVE

Sourcing events is critical to developing a distributed system that reacts to events.

2.2.1. Creating an event source by using the Administrator perspective

A Knative *event source* can be any Kubernetes object that generates or imports cloud events, and relays those events to another endpoint, known as a *sink*.

Prerequisites

- The OpenShift Serverless Operator and Knative Eventing are installed on your OpenShift Container Platform cluster.
- You have logged in to the web console and are in the **Administrator** perspective.
- You have cluster administrator permissions on OpenShift Container Platform, or you have cluster or dedicated administrator permissions on Red Hat OpenShift Service on AWS or OpenShift Dedicated.

Procedure

1. In the **Administrator** perspective of the OpenShift Container Platform web console, navigate to **Serverless** → **Eventing**.
2. In the **Create** list, select **Event Source**. You will be directed to the **Event Sources** page.
3. Select the event source type that you want to create.

2.3. CREATING AN API SERVER SOURCE

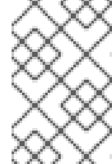
The API server source is an event source that can be used to connect an event sink, such as a Knative service, to the Kubernetes API server. The API server source watches for Kubernetes events and forwards them to the Knative Eventing broker.

2.3.1. Creating an API server source by using the web console

After Knative Eventing is installed on your cluster, you can create an API server source by using the web console. Using the OpenShift Container Platform web console provides a streamlined and intuitive user interface to create an event source.

Prerequisites

- You have logged in to the OpenShift Container Platform web console.
- The OpenShift Serverless Operator and Knative Eventing are installed on the cluster.
- 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.
- You have installed the OpenShift CLI (**oc**).



PROCEDURE

If you want to re-use an existing service account, you can modify your existing **ServiceAccount** resource to include the required permissions instead of creating a new resource.

1. Create a service account, role, and role binding for the event source as a YAML file:

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: events-sa
  namespace: default (1)

---
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: event-watcher
  namespace: default (2)
rules:
  - apiGroups:
      - ""
    resources:
      - events
    verbs:
      - get
      - list
      - watch

---
```

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: k8s-ra-event-watcher
  namespace: default (3)
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: event-watcher
subjects:
  - kind: ServiceAccount
    name: events-sa
    namespace: default (4)
```

1 2 3 4 Change this namespace to the namespace that you have selected for installing the event source.

2. Apply the YAML file:

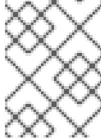
```
$ oc apply -f <filename>
```

3. In the **Developer** perspective, navigate to **+Add** → **Event Source**. The **Event Sources** page is displayed.

4. Optional: If you have multiple providers for your event sources, select the required provider from the **Providers** list to filter the available event sources from the provider.

5. Select **ApiServerSource** and then click **Create Event Source**. The **Create Event Source** page is displayed.

6. Configure the **ApiServerSource** settings by using the **Form view** or **YAML view**:



NOTE

You can switch between the **Form view** and **YAML view**. The data is persisted when switching between the views.

a. Enter **v1** as the **APIVERSION** and **Event** as the **KIND**.

b. Select the **Service Account Name** for the service account that you created.

c. In the **Target** section, select your event sink. This can be either a **Resource** or a **URI**:

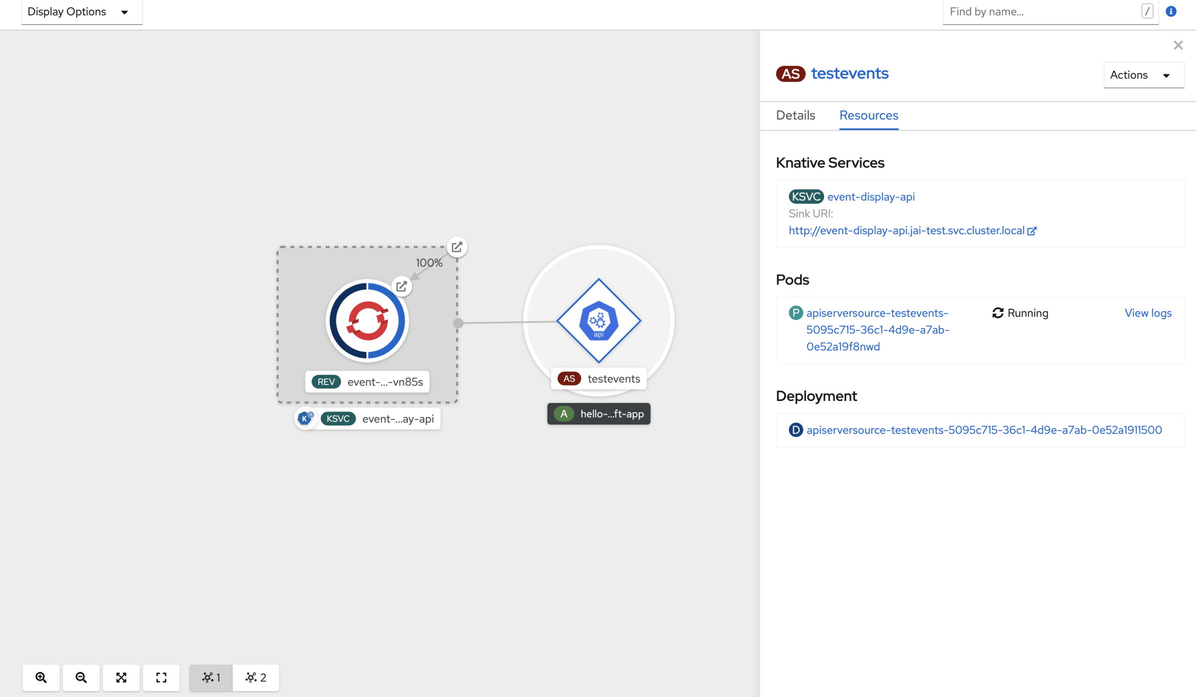
i. Select **Resource** to use a channel, broker, or service as an event sink for the event source.

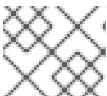
ii. Select **URI** to specify a Uniform Resource Identifier (URI) where the events are routed to.

7. Click **Create**.

Verification

- After you have created the API server source, check that it is connected to the event sink by viewing it in the **Topology** view.



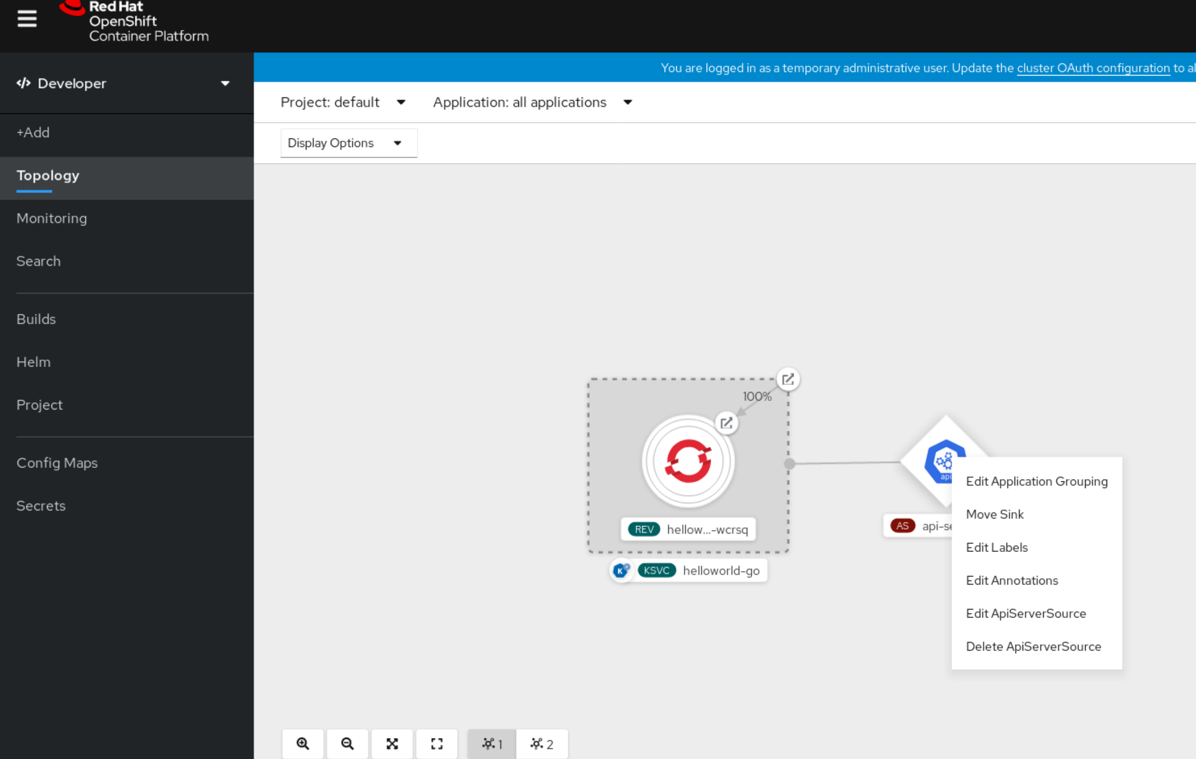


NOTE

If a URI sink is used, you can modify the URI by right-clicking on **URI sink → Edit URI**.

Deleting the API server source

1. Navigate to the **Topology** view.
2. Right-click the API server source and select **Delete ApiServerSource**.

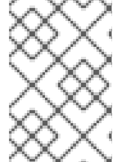


2.3.2. Creating an API server source by using the Knative CLI

You can use the **kn source apiserver create** command to create an API server source by using the **kn** CLI. Using the **kn** CLI to create an API server source provides a more streamlined and intuitive user interface than modifying YAML files directly.

Prerequisites

- The OpenShift Serverless Operator and Knative Eventing are installed on the cluster.
- 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.
- You have installed the OpenShift CLI (**oc**).
- You have installed the Knative (**kn**) CLI.



PROCEDURE

If you want to re-use an existing service account, you can modify your existing **ServiceAccount** resource to include the required permissions instead of creating a new resource.

1. Create a service account, role, and role binding for the event source as a YAML file:

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: events-sa
  namespace: default (1)

---
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: event-watcher
  namespace: default (2)
rules:
  - apiGroups:
      - ""
    resources:
      - events
    verbs:
      - get
      - list
      - watch

---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: k8s-ra-event-watcher
  namespace: default (3)
roleRef:
  apiGroup: rbac.authorization.k8s.io
```

```
  kind: Role
  name: event-watcher
subjects:
  - kind: ServiceAccount
    name: events-sa
    namespace: default 4
```

1 2 3 4 Change this namespace to the namespace that you have selected for installing the event source.

2. Apply the YAML file:

```
$ oc apply -f <filename>
```

3. Create an API server source that has an event sink. In the following example, the sink is a broker:

```
$ kn source apiserver create <event_source_name> --sink broker:<broker_name> --
resource "event:v1" --service-account <service_account_name> --mode Resource
```

4. To check that the API server source is set up correctly, create a Knative service that dumps incoming messages to its log:

```
$ kn service create event-display --image quay.io/openshift-knative/showcase
```

5. If you used a broker as an event sink, create a trigger to filter events from the **default** broker to the service:

```
$ kn trigger create <trigger_name> --sink ksvc:event-display
```

6. Create events by launching a pod in the default namespace:

```
$ oc create deployment event-origin --image quay.io/openshift-knative/showcase
```

7. Check that the controller is mapped correctly by inspecting the output generated by the following command:

```
$ kn source apiserver describe <source_name>
```

Example output

```
Name:                mysource
Namespace:           default
Annotations:         sources.knative.dev/creator=developer,
sources.knative.dev/lastModifier=developer
Age:                 3m
ServiceAccountName:  events-sa
Mode:                Resource
Sink:
  Name:       default
  Namespace:  default
  Kind:       Broker (eventing.knative.dev/v1)
Resources:
```

```
    Kind:       event (v1)
    Controller:  false
  Conditions:
    OK TYPE                  AGE REASON
    ++ Ready                  3m
    ++ Deployed               3m
    ++ SinkProvided           3m
    ++ SufficientPermissions  3m
    ++ EventTypesProvided     3m
```

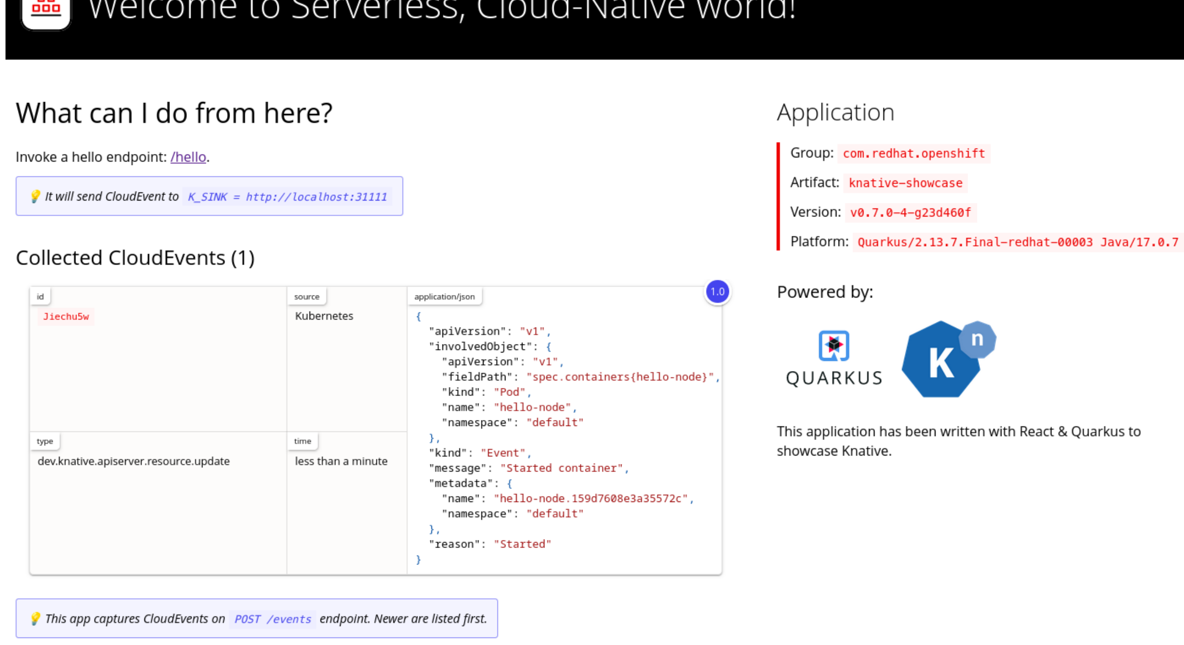
Verification

To verify that the Kubernetes events were sent to Knative, look at the event-display logs or use web browser to see the events.

- To view the events in a web browser, open the link returned by the following command:

```
$ kn service describe event-display -o url
```

Figure 2.1. Example browser page



- Alternatively, to see the logs in the terminal, view the event-display logs for the pods by entering the following command:

```
$ oc logs $(oc get pod -o name | grep event-display) -c user-container
```

Example output

```
☁  cloudevents.Event
Validation: valid
Context Attributes,
  specversion: 1.0
  type: dev.knative.apiserver.resource.update
  datacontenttype: application/json
  ...
```

```
Data,
 {
   "apiVersion": "v1",
   "involvedObject": {
     "apiVersion": "v1",
     "fieldPath": "spec.containers{event-origin}",
     "kind": "Pod",
     "name": "event-origin",
     "namespace": "default",
      .....
   },
   "kind": "Event",
   "message": "Started container",
   "metadata": {
     "name": "event-origin.159d7608e3a3572c",
     "namespace": "default",
     ....
   },
   "reason": "Started",
   ...
 }
```

Deleting the API server source

1. Delete the trigger:

```
$ kn trigger delete <trigger_name>
```

2. Delete the event source:

```
$ kn source apiserver delete <source_name>
```

3. Delete the service account, cluster role, and cluster binding:

```
$ oc delete -f authentication.yaml
```

2.3.2.1. Knative CLI sink flag

When you create an event source by using the Knative (**kn**) CLI, you can specify a sink where events are sent to from that resource by using the **--sink** flag. The sink can be any addressable or callable resource that can receive incoming events from other resources.

The following example creates a sink binding that uses a service, **http://event-display.svc.cluster.local**, as the sink:

Example command using the sink flag

```
$ kn source binding create bind-heartbeat \
  --namespace sinkbinding-example \
  --subject "Job:batch/v1:app=heartbeat-cron" \
  --sink http://event-display.svc.cluster.local \ (1)
  --ce-override "sink=bound"
```

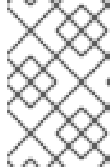
1. **svc** in **http://event-display.svc.cluster.local** determines that the sink is a Knative service. Other default sink prefixes include **channel**, and **broker**.

2.3.3. Creating an API server source by using YAML files

Creating Knative resources by using YAML files uses a declarative API, which enables you to describe event sources declaratively and in a reproducible manner. To create an API server source by using YAML, you must create a YAML file that defines an **ApiServerSource** object, then apply it by using the **oc apply** command.

Prerequisites

- The OpenShift Serverless Operator and Knative Eventing are installed on the cluster.
- 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.
- You have created the **default** broker in the same namespace as the one defined in the API server source YAML file.
- Install the OpenShift CLI (**oc**).



PROCEDURE

If you want to re-use an existing service account, you can modify your existing **ServiceAccount** resource to include the required permissions instead of creating a new resource.

1. Create a service account, role, and role binding for the event source as a YAML file:

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: events-sa
  namespace: default (1)

---
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: event-watcher
  namespace: default (2)
rules:
  - apiGroups:
      - ""
    resources:
      - events
    verbs:
      - get
      - list
      - watch

---
apiVersion: rbac.authorization.k8s.io/v1
```

```
kind: RoleBinding
metadata:
  name: k8s-ra-event-watcher
  namespace: default 3
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: event-watcher
subjects:
  - kind: ServiceAccount
    name: events-sa
    namespace: default 4
```

1 2 3 4 Change this namespace to the namespace that you have selected for installing the event source.

2. Apply the YAML file:

```
$ oc apply -f <filename>
```

3. Create an API server source as a YAML file:

```
apiVersion: sources.knative.dev/v1alpha1
kind: ApiServerSource
metadata:
  name: testevents
spec:
  serviceAccountName: events-sa
  mode: Resource
  resources:
    - apiVersion: v1
      kind: Event
  sink:
    ref:
      apiVersion: eventing.knative.dev/v1
      kind: Broker
      name: default
```

4. Apply the **ApiServerSource** YAML file:

```
$ oc apply -f <filename>
```

5. To check that the API server source is set up correctly, create a Knative service as a YAML file that dumps incoming messages to its log:

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: event-display
  namespace: default
spec:
  template:
```

```
    spec:
      containers:
        - image: quay.io/openshift-knative/showcase
```

6. Apply the **Service** YAML file:

```
$ oc apply -f <filename>
```

7. Create a **Trigger** object as a YAML file that filters events from the **default** broker to the service created in the previous step:

```
apiVersion: eventing.knative.dev/v1
kind: Trigger
metadata:
  name: event-display-trigger
  namespace: default
spec:
  broker: default
  subscriber:
    ref:
      apiVersion: serving.knative.dev/v1
      kind: Service
      name: event-display
```

8. Apply the **Trigger** YAML file:

```
$ oc apply -f <filename>
```

9. Create events by launching a pod in the default namespace:

```
$ oc create deployment event-origin --image=quay.io/openshift-knative/showcase
```

10. Check that the controller is mapped correctly, by entering the following command and inspecting the output:

```
$ oc get apiserversource.sources.knative.dev testevents -o yaml
```

Example output

```
apiVersion: sources.knative.dev/v1alpha1
kind: ApiServerSource
metadata:
  annotations:
  creationTimestamp: "2020-04-07T17:24:54Z"
  generation: 1
  name: testevents
  namespace: default
  resourceVersion: "62868"
  selfLink:
/apis/sources.knative.dev/v1alpha1/namespaces/default/apiserversources/testevents2
  uid: 1603d863-bb06-4d1c-b371-f580b4db99fa
spec:
  mode: Resource
```

```
resources:
- apiVersion: v1
  controller: false
  controllerSelector:
    apiVersion: ""
    kind: ""
    name: ""
    uid: ""
  kind: Event
  labelSelector: {}
serviceAccountName: events-sa
sink:
  ref:
    apiVersion: eventing.knative.dev/v1
    kind: Broker
    name: default
```

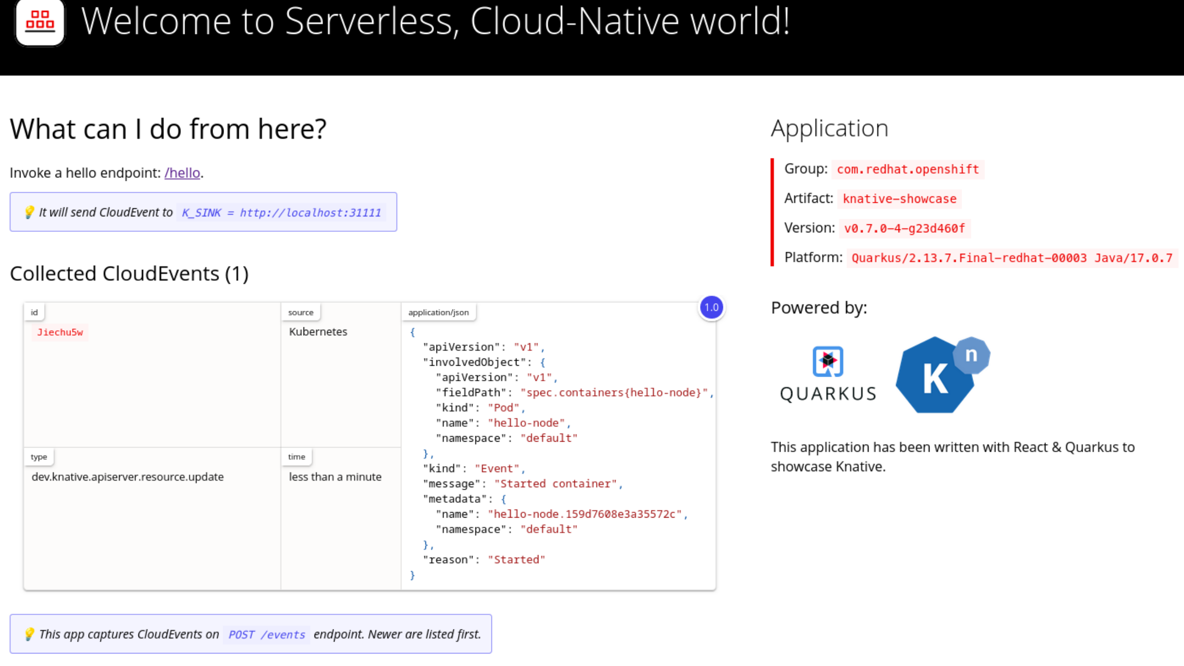
Verification

To verify that the Kubernetes events were sent to Knative, you can look at the event-display logs or use web browser to see the events.

- To view the events in a web browser, open the link returned by the following command:

```
$ oc get ksvc event-display -o jsonpath='{.status.url}'
```

Figure 2.2. Example browser page



- To see the logs in the terminal, view the event-display logs for the pods by entering the following command:

```
$ oc logs $(oc get pod -o name | grep event-display) -c user-container
```

Example output

```
☁  cloudevents.Event
Validation: valid
Context Attributes,
  specversion: 1.0
  type: dev.knative.apiserver.resource.update
  datacontenttype: application/json
  ...
Data,
  {
    "apiVersion": "v1",
    "involvedObject": {
      "apiVersion": "v1",
      "fieldPath": "spec.containers{event-origin}",
      "kind": "Pod",
      "name": "event-origin",
      "namespace": "default",
       .....
    },
    "kind": "Event",
    "message": "Started container",
    "metadata": {
      "name": "event-origin.159d7608e3a3572c",
      "namespace": "default",
      ....
    },
    "reason": "Started",
    ...
  }
```


Deleting the API server source

1. Delete the trigger:

```
$ oc delete -f trigger.yaml
```

2. Delete the event source:

```
$ oc delete -f k8s-events.yaml
```

3. Delete the service account, cluster role, and cluster binding:

```
$ oc delete -f authentication.yaml
```

2.4. CREATING A PING SOURCE

A ping source is an event source that can be used to periodically send ping events with a constant payload to an event consumer. A ping source can be used to schedule sending events, similar to a timer.

2.4.1. Creating a ping source by using the web console

After Knative Eventing is installed on your cluster, you can create a ping source by using the web console. Using the OpenShift Container Platform web console provides a streamlined and intuitive user interface to create an event source.

Prerequisites

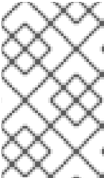
- You have logged in to the OpenShift Container Platform web console.
- The OpenShift Serverless Operator, Knative Serving and Knative Eventing are installed on the cluster.
- You have created a project or have access to a project with the appropriate roles and permissions to create applications and other workloads in OpenShift Container Platform.

Procedure

1. To verify that the ping source is working, create a simple Knative service that dumps incoming messages to the logs of the service.
   a. In the **Developer** perspective, navigate to **+Add → YAML**.
   b. Copy the example YAML:

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: event-display
spec:
  template:
    spec:
      containers:
        - image: quay.io/openshift-knative/showcase
```

   c. Click **Create**.
2. Create a ping source in the same namespace as the service created in the previous step, or any other sink that you want to send events to.
   a. In the **Developer** perspective, navigate to **+Add → Event Source**. The **Event Sources** page is displayed.
   b. Optional: If you have multiple providers for your event sources, select the required provider from the **Providers** list to filter the available event sources from the provider.
   c. Select **Ping Source** and then click **Create Event Source**. The **Create Event Source** page is displayed.



NOTE

You can configure the **PingSource** settings by using the **Form view** or **YAML view** and can switch between the views. The data is persisted when switching between the views.

   d. Enter a value for **Schedule**. In this example, the value is ***/2 * * * ***, which creates a PingSource that sends a message every two minutes.
   e. Optional: You can enter a value for **Data**, which is the message payload.
   f. In the **Target** section, select your event sink. This can be either a **Resource** or a **URI**:

i. Select **Resource** to use a channel, broker, or service as an event sink for the event source. In this example, the **event-display** service created in the previous step is used as the target **Resource**.

ii. Select **URI** to specify a Uniform Resource Identifier (URI) where the events are routed to.

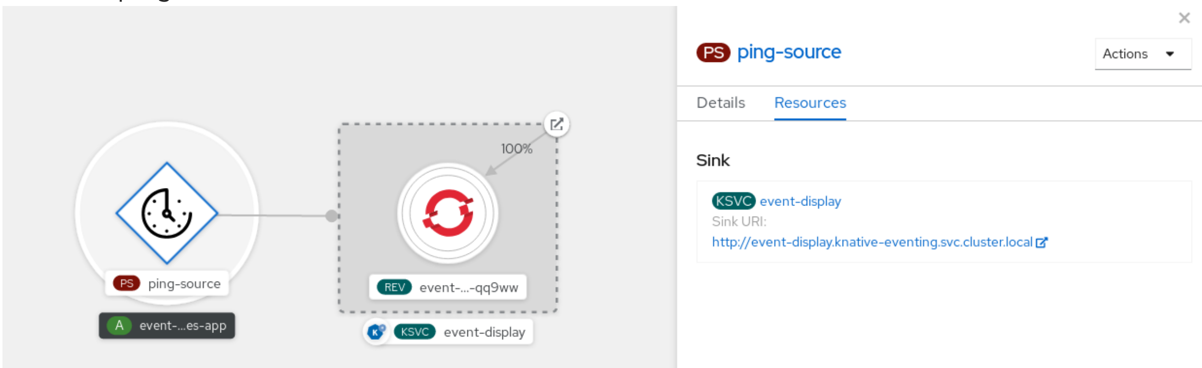
g. Click **Create**.

Verification

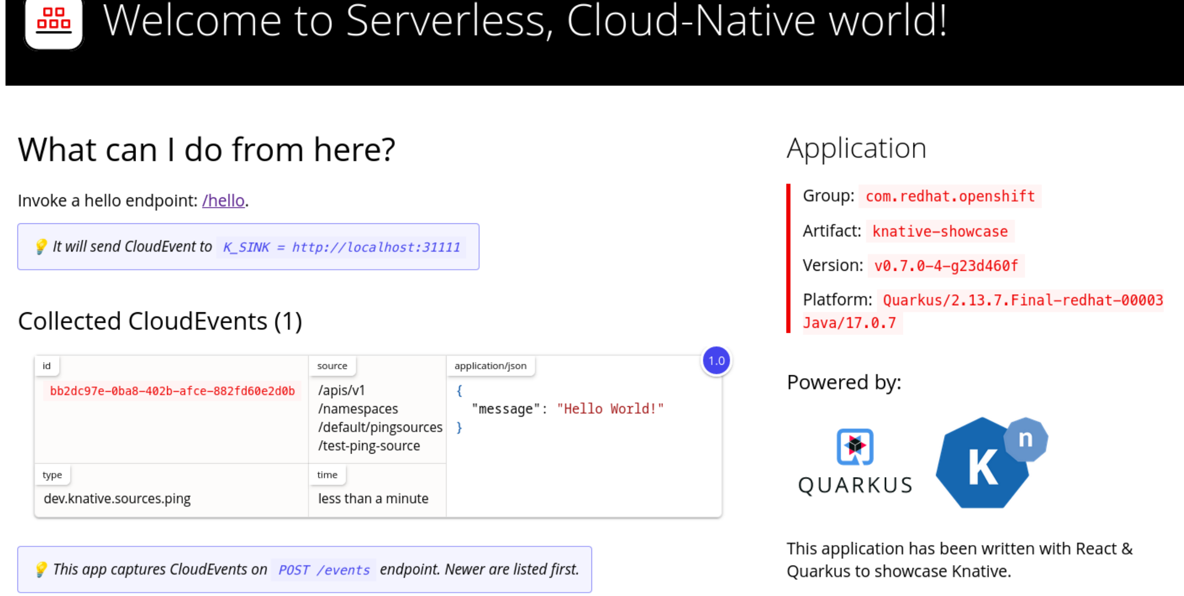
You can verify that the ping source was created and is connected to the sink by viewing the **Topology** page.

1. In the **Developer** perspective, navigate to **Topology**.

2. View the ping source and sink.



3. View the event-display service in the web browser. You should see the ping source events in the web UI.



Deleting the ping source

1. Navigate to the **Topology** view.

2. Right-click the API server source and select **Delete Ping Source**.

2.4.2. Creating a ping source by using the Knative CLI

You can use the **kn source ping create** command to create a ping source by using the Knative (**kn**) CLI. Using the Knative CLI to create event sources provides a more streamlined and intuitive user interface than modifying YAML files directly.

Prerequisites

- The OpenShift Serverless Operator, Knative Serving and Knative Eventing are installed on the cluster.
- You have installed the Knative (**kn**) CLI.
- 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.
- Optional: If you want to use the verification steps for this procedure, install the OpenShift CLI (**oc**).

Procedure

1. To verify that the ping source is working, create a simple Knative service that dumps incoming messages to the service logs:

```
$ kn service create event-display \
    --image quay.io/openshift-knative/showcase
```

2. For each set of ping events that you want to request, create a ping source in the same namespace as the event consumer:

```
$ kn source ping create test-ping-source \
    --schedule "*/2 * * * *" \
    --data '{"message": "Hello world!"}' \
    --sink ksvc:event-display
```

3. Check that the controller is mapped correctly by entering the following command and inspecting the output:

```
$ kn source ping describe test-ping-source
```

Example output

```
Name:         test-ping-source
Namespace:    default
Annotations:  sources.knative.dev/creator=developer,
sources.knative.dev/lastModifier=developer
Age:          15s
Schedule:     */2 * * * *
Data:         {"message": "Hello world!"}

Sink:
  Name:       event-display
  Namespace:  default
  Resource:   Service (serving.knative.dev/v1)
```

```
Conditions:
  OK TYPE                  AGE REASON
  ++ Ready                  8s
  ++ Deployed               8s
  ++ SinkProvided          15s
  ++ ValidSchedule         15s
  ++ EventTypeProvided    15s
  ++ ResourcesCorrect     15s
```

Verification

You can verify that the Kubernetes events were sent to the Knative event sink by looking at the logs of the sink pod.

By default, Knative services terminate their pods if no traffic is received within a 60 second period. The example shown in this guide creates a ping source that sends a message every 2 minutes, so each message should be observed in a newly created pod.

1. Watch for new pods created:

```
$ watch oc get pods
```

2. Cancel watching the pods using Ctrl+C, then look at the logs of the created pod:

```
$ oc logs $(oc get pod -o name | grep event-display) -c user-container
```

Example output

```
☁  cloudevents.Event
Validation: valid
Context Attributes,
  specversion: 1.0
  type: dev.knative.sources.ping
  source: /apis/v1/namespaces/default/pingsources/test-ping-source
  id: 99e4f4f6-08ff-4bff-acf1-47f61ded68c9
  time: 2020-04-07T16:16:00.000601161Z
  datacontenttype: application/json
Data,
  {
    "message": "Hello world!"
  }
```

Deleting the ping source

- Delete the ping source:

```
$ kn delete pingsources.sources.knative.dev <ping_source_name>
```

2.4.2.1. Knative CLI sink flag

When you create an event source by using the Knative (**kn**) CLI, you can specify a sink where events are sent to from that resource by using the **--sink** flag. The sink can be any addressable or callable resource that can receive incoming events from other resources.

The following example creates a sink binding that uses a service, **http://event-display.svc.cluster.local**, as the sink:

Example command using the sink flag

```
$ kn source binding create bind-heartbeat \
  --namespace sinkbinding-example \
  --subject "Job:batch/v1:app=heartbeat-cron" \
  --sink http://event-display.svc.cluster.local \ (1)
  --ce-override "sink=bound"
```

1. **svc** in **http://event-display.svc.cluster.local** determines that the sink is a Knative service. Other default sink prefixes include **channel**, and **broker**.

2.4.3. Creating a ping source by using YAML

Creating Knative resources by using YAML files uses a declarative API, which enables you to describe event sources declaratively and in a reproducible manner. To create a serverless ping source by using YAML, you must create a YAML file that defines a **PingSource** object, then apply it by using **oc apply**.

Example PingSource object

```
apiVersion: sources.knative.dev/v1
kind: PingSource
metadata:
  name: test-ping-source
spec:
  schedule: "*/2 * * * *" (1)
  data: '{"message": "Hello world!"}' (2)
  sink: (3)
    ref:
      apiVersion: serving.knative.dev/v1
      kind: Service
      name: event-display
```

1. The schedule of the event specified using CRON expression.
2. The event message body expressed as a JSON encoded data string.
3. These are the details of the event consumer. In this example, we are using a Knative service named **event-display**.

Prerequisites

- The OpenShift Serverless Operator, Knative Serving and Knative Eventing are installed on the cluster.
- Install the OpenShift CLI (**oc**).
- 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. To verify that the ping source is working, create a simple Knative service that dumps incoming messages to the service's logs.

   a. Create a service YAML file:

   ```
   apiVersion: serving.knative.dev/v1
   kind: Service
   metadata:
     name: event-display
   spec:
     template:
       spec:
         containers:
           - image: quay.io/openshift-knative/showcase
   ```

   b. Create the service:

   ```
   $ oc apply -f <filename>
   ```

2. For each set of ping events that you want to request, create a ping source in the same namespace as the event consumer.

   a. Create a YAML file for the ping source:

   ```
   apiVersion: sources.knative.dev/v1
   kind: PingSource
   metadata:
     name: test-ping-source
   spec:
     schedule: "*/2 * * * *"
     data: '{"message": "Hello world!"}'
     sink:
       ref:
         apiVersion: serving.knative.dev/v1
         kind: Service
         name: event-display
   ```

   b. Create the ping source:

   ```
   $ oc apply -f <filename>
   ```

3. Check that the controller is mapped correctly by entering the following command:

   ```
   $ oc get pingsource.sources.knative.dev <ping_source_name> -oyaml
   ```

   Example output

   ```
   apiVersion: sources.knative.dev/v1
   kind: PingSource
   metadata:
     annotations:
       sources.knative.dev/creator: developer
       sources.knative.dev/lastModifier: developer
   ```

```
  creationTimestamp: "2020-04-07T16:11:14Z"
  generation: 1
  name: test-ping-source
  namespace: default
  resourceVersion: "55257"
  selfLink: /apis/sources.knative.dev/v1/namespaces/default/pingsources/test-ping-source
  uid: 3d80d50b-f8c7-4c1b-99f7-3ec00e0a8164
spec:
  data: '{ value: "hello" }'
  schedule: '*/2 * * * *'
  sink:
    ref:
      apiVersion: serving.knative.dev/v1
      kind: Service
      name: event-display
      namespace: default
```


Verification

You can verify that the Kubernetes events were sent to the Knative event sink by looking at the sink pod's logs.

By default, Knative services terminate their pods if no traffic is received within a 60 second period. The example shown in this guide creates a PingSource that sends a message every 2 minutes, so each message should be observed in a newly created pod.

1. Watch for new pods created:

```
$ watch oc get pods
```

2. Cancel watching the pods using Ctrl+C, then look at the logs of the created pod:

```
$ oc logs $(oc get pod -o name | grep event-display) -c user-container
```

Example output

```
☁  cloudevents.Event
Validation: valid
Context Attributes,
  specversion: 1.0
  type: dev.knative.sources.ping
  source: /apis/v1/namespaces/default/pingsources/test-ping-source
  id: 042ff529-240e-45ee-b40c-3a908129853e
  time: 2020-04-07T16:22:00.000791674Z
  datacontenttype: application/json
Data,
  {
    "message": "Hello world!"
  }
```

Deleting the ping source

- Delete the ping source:

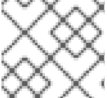
```
$ oc delete -f <filename>
```

Example command

```
$ oc delete -f ping-source.yaml
```

2.5. SOURCE FOR APACHE KAFKA

You can create an Apache Kafka source that reads events from an Apache Kafka cluster and passes these events to a sink. You can create a Kafka source by using the OpenShift Container Platform web console, the Knative (**kn**) CLI, or by creating a **KafkaSource** object directly as a YAML file and using the OpenShift CLI (**oc**) to apply it.



NOTE

See the documentation for Installing Knative broker for Apache Kafka .

2.5.1. Creating an Apache Kafka event source by using the web console

After the Knative broker implementation for Apache Kafka is installed on your cluster, you can create an Apache Kafka source by using the web console. Using the OpenShift Container Platform web console provides a streamlined and intuitive user interface to create a Kafka source.

Prerequisites

- The OpenShift Serverless Operator, Knative Eventing, and the **KnativeKafka** custom resource are installed on your cluster.
- You have logged in to the web console.
- You have access to a Red Hat AMQ Streams (Kafka) cluster that produces the Kafka messages you want to import.
- 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. In the **Developer** perspective, navigate to the **+Add** page and select **Event Source**.
2. In the **Event Sources** page, select **Kafka Source** in the **Type** section.
3. Configure the **Kafka Source** settings:
    a. Add a comma-separated list of **Bootstrap Servers**.
    b. Add a comma-separated list of **Topics**.
    c. Add a **Consumer Group**.
    d. Select the **Service Account Name** for the service account that you created.
    e. In the **Target** section, select your event sink. This can be either a **Resource** or a **URI**:

i. Select **Resource** to use a channel, broker, or service as an event sink for the event source.

ii. Select **URI** to specify a Uniform Resource Identifier (URI) where the events are routed to.

f. Enter a **Name** for the Kafka event source.

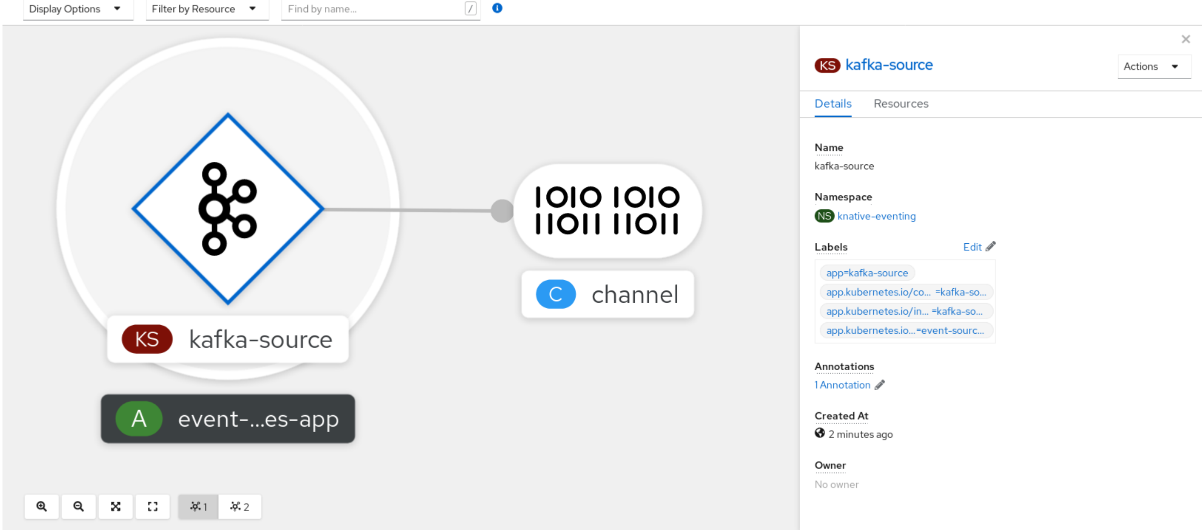
4. Click **Create**.

Verification

You can verify that the Kafka event source was created and is connected to the sink by viewing the **Topology** page.

1. In the **Developer** perspective, navigate to **Topology**.

2. View the Kafka event source and sink.



2.5.2. Creating an Apache Kafka event source by using the Knative CLI

You can use the **kn source kafka create** command to create a Kafka source by using the Knative (**kn**) CLI. Using the Knative CLI to create event sources provides a more streamlined and intuitive user interface than modifying YAML files directly.

Prerequisites

- The OpenShift Serverless Operator, Knative Eventing, Knative Serving, and the **KnativeKafka** custom resource (CR) are installed on your cluster.

- 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.

- You have access to a Red Hat AMQ Streams (Kafka) cluster that produces the Kafka messages you want to import.

- You have installed the Knative (**kn**) CLI.

- Optional: You have installed the OpenShift CLI (**oc**) if you want to use the verification steps in this procedure.

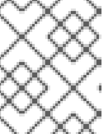
Procedure

1. To verify that the Kafka event source is working, create a Knative service that dumps incoming events into the service logs:

```
$ kn service create event-display \
    --image quay.io/openshift-knative/showcase
```

2. Create a **KafkaSource** CR:

```
$ kn source kafka create <kafka_source_name> \
    --servers <cluster_kafka_bootstrap>.kafka.svc:9092 \
    --topics <topic_name> --consumergroup my-consumer-group \
    --sink event-display
```



NOTE

Replace the placeholder values in this command with values for your source name, bootstrap servers, and topics.

The **--servers**, **--topics**, and **--consumergroup** options specify the connection parameters to the Kafka cluster. The **--consumergroup** option is optional.

3. Optional: View details about the **KafkaSource** CR you created:

```
$ kn source kafka describe <kafka_source_name>
```

Example output

```
Name:              example-kafka-source
Namespace:         kafka
Age:               1h
BootstrapServers:  example-cluster-kafka-bootstrap.kafka.svc:9092
Topics:            example-topic
ConsumerGroup:     example-consumer-group

Sink:
  Name:       event-display
  Namespace:  default
  Resource:   Service (serving.knative.dev/v1)

Conditions:
  OK TYPE            AGE REASON
  ++ Ready           1h
  ++ Deployed        1h
  ++ SinkProvided    1h
```

Verification steps

1. Trigger the Kafka instance to send a message to the topic:

```
$ oc -n kafka run kafka-producer \
    -ti --image=quay.io/strimzi/kafka:latest-kafka-2.7.0 --rm=true \
```

```
--restart=Never -- bin/kafka-console-producer.sh \
--broker-list <cluster_kafka_bootstrap>:9092 --topic my-topic
```

Enter the message in the prompt. This command assumes that:

- The Kafka cluster is installed in the **kafka** namespace.
- The **KafkaSource** object has been configured to use the **my-topic** topic.

2. Verify that the message arrived by viewing the logs:

```
$ oc logs $(oc get pod -o name | grep event-display) -c user-container
```

Example output

```
☁  cloudevents.Event
Validation: valid
Context Attributes,
  specversion: 1.0
  type: dev.knative.kafka.event
  source: /apis/v1/namespaces/default/kafkasources/example-kafka-source#example-topic
  subject: partition:46#0
  id: partition:46/offset:0
  time: 2021-03-10T11:21:49.4Z
Extensions,
  traceparent: 00-161ff3815727d8755848ec01c866d1cd-7ff3916c44334678-00
Data,
  Hello!
```

2.5.2.1. Knative CLI sink flag

When you create an event source by using the Knative (**kn**) CLI, you can specify a sink where events are sent to from that resource by using the **--sink** flag. The sink can be any addressable or callable resource that can receive incoming events from other resources.

The following example creates a sink binding that uses a service, **http://event-display.svc.cluster.local**, as the sink:

Example command using the sink flag

```
$ kn source binding create bind-heartbeat \
  --namespace sinkbinding-example \
  --subject "Job:batch/v1:app=heartbeat-cron" \
  --sink http://event-display.svc.cluster.local \ (1)
  --ce-override "sink=bound"
```

(1) **svc** in **http://event-display.svc.cluster.local** determines that the sink is a Knative service. Other default sink prefixes include **channel**, and **broker**.

2.5.3. Creating an Apache Kafka event source by using YAML

Creating Knative resources by using YAML files uses a declarative API, which enables you to describe applications declaratively and in a reproducible manner. To create a Kafka source by using YAML, you

must create a YAML file that defines a **KafkaSource** object, then apply it by using the **oc apply** command.

Prerequisites

- The OpenShift Serverless Operator, Knative Eventing, and the **KnativeKafka** custom resource are installed on your cluster.
- 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.
- You have access to a Red Hat AMQ Streams (Kafka) cluster that produces the Kafka messages you want to import.
- Install the OpenShift CLI (**oc**).

Procedure

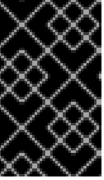
1. Create a **KafkaSource** object as a YAML file:

```
apiVersion: sources.knative.dev/v1beta1
kind: KafkaSource
metadata:
  name: <source_name>
spec:
  consumerGroup: <group_name> (1)
  bootstrapServers:
  - <list_of_bootstrap_servers>
  topics:
  - <list_of_topics> (2)
  sink:
  - <list_of_sinks> (3)
```

(1) A consumer group is a group of consumers that use the same group ID, and consume data from a topic.

(2) A topic provides a destination for the storage of data. Each topic is split into one or more partitions.

(3) A sink specifies where events are sent to from a source.



IMPORTANT

Only the **v1beta1** version of the API for **KafkaSource** objects on OpenShift Serverless is supported. Do not use the **v1alpha1** version of this API, as this version is now deprecated.

Example KafkaSource object

```
apiVersion: sources.knative.dev/v1beta1
kind: KafkaSource
metadata:
  name: kafka-source
```

```
spec:
  consumerGroup: knative-group
  bootstrapServers:
  - my-cluster-kafka-bootstrap.kafka:9092
  topics:
  - knative-demo-topic
  sink:
    ref:
      apiVersion: serving.knative.dev/v1
      kind: Service
      name: event-display
```

2. Apply the **KafkaSource** YAML file:

```
$ oc apply -f <filename>
```

Verification

- Verify that the Kafka event source was created by entering the following command:

```
$ oc get pods
```

Example output

```
NAME                              READY   STATUS    RESTARTS   AGE
kafkasource-kafka-source-5ca0248f-...   1/1     Running   0          13m
```

2.5.4. Configuring SASL authentication for Apache Kafka sources

Simple Authentication and Security Layer (SASL) is used by Apache Kafka for authentication. If you use SASL authentication on your cluster, users must provide credentials to Knative for communicating with the Kafka cluster; otherwise events cannot be produced or consumed.

Prerequisites

- You have cluster or dedicated administrator permissions on OpenShift Container Platform.
- The OpenShift Serverless Operator, Knative Eventing, and the **KnativeKafka** CR are installed on your OpenShift Container Platform cluster.
- 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.
- You have a username and password for a Kafka cluster.
- You have chosen the SASL mechanism to use, for example, **PLAIN**, **SCRAM-SHA-256**, or **SCRAM-SHA-512**.
- If TLS is enabled, you also need the **ca.crt** certificate file for the Kafka cluster.
- You have installed the OpenShift (**oc**) CLI.

Procedure

1. Create the certificate files as secrets in your chosen namespace:

```
$ oc create secret -n <namespace> generic <kafka_auth_secret> \
  --from-file=ca.crt=caroot.pem \
  --from-literal=password="SecretPassword" \
  --from-literal=saslType="SCRAM-SHA-512" \ 1
  --from-literal=user="my-sasl-user"
```

1 The SASL type can be **PLAIN**, **SCRAM-SHA-256**, or **SCRAM-SHA-512**.

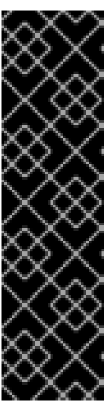
2. Create or modify your Kafka source so that it contains the following **spec** configuration:

```
apiVersion: sources.knative.dev/v1beta1
kind: KafkaSource
metadata:
  name: example-source
spec:
...
  net:
    sasl:
      enable: true
      user:
        secretKeyRef:
          name: <kafka_auth_secret>
          key: user
      password:
        secretKeyRef:
          name: <kafka_auth_secret>
          key: password
      type:
        secretKeyRef:
          name: <kafka_auth_secret>
          key: saslType
    tls:
      enable: true
      caCert: 1
        secretKeyRef:
          name: <kafka_auth_secret>
          key: ca.crt
...
```

1 The **caCert** spec is not required if you are using a public cloud Kafka service.

2.5.5. Configuring KEDA autoscaling for KafkaSource

You can configure Knative Eventing sources for Apache Kafka (KafkaSource) to be autoscaled using the Custom Metrics Autoscaler Operator, which is based on the Kubernetes Event Driven Autoscaler (KEDA).



IMPORTANT

Configuring KEDA autoscaling for KafkaSource is a Technology Preview feature only. Technology Preview features are not supported with Red Hat production service level agreements (SLAs) and might not be functionally complete. Red Hat does not recommend using them in production. These features provide early access to upcoming product features, enabling customers to test functionality and provide feedback during the development process.

For more information about the support scope of Red Hat Technology Preview features, see Technology Preview Features Support Scope .

Prerequisites

- The OpenShift Serverless Operator, Knative Eventing, and the **KnativeKafka** custom resource are installed on your cluster.

Procedure

1. In the **KnativeKafka** custom resource, enable KEDA scaling:

   Example YAML

```
apiVersion: operator.serverless.openshift.io/v1alpha1
kind: KnativeKafka
metadata:
  name: knative-kafka
  namespace: knative-eventing
spec:
  config:
    kafka-features:
      controller-autoscaler-keda: enabled
```

2. Apply the **KnativeKafka** YAML file:

```
$ oc apply -f <filename>
```

2.6. CUSTOM EVENT SOURCES

If you need to ingress events from an event producer that is not included in Knative, or from a producer that emits events which are not in the **CloudEvent** format, you can do this by creating a custom event source. You can create a custom event source by using one of the following methods:

- Use a **PodSpecable** object as an event source, by creating a sink binding.
- Use a container as an event source, by creating a container source.

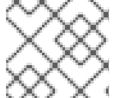
2.6.1. Sink binding

The **SinkBinding** object supports decoupling event production from delivery addressing. Sink binding is used to connect *event producers* to an event consumer, or *sink*. An event producer is a Kubernetes resource that embeds a **PodSpec** template and produces events. A sink is an addressable Kubernetes object that can receive events.

The **SinkBinding** object injects environment variables into the **PodTemplateSpec** of the sink, which means that the application code does not need to interact directly with the Kubernetes API to locate the event destination. These environment variables are as follows:

K_SINK
The URL of the resolved sink.

K_CE_OVERRIDES
A JSON object that specifies overrides to the outbound event.



NOTE

The **SinkBinding** object currently does not support custom revision names for services.

2.6.1.1. Creating a sink binding by using YAML

Creating Knative resources by using YAML files uses a declarative API, which enables you to describe event sources declaratively and in a reproducible manner. To create a sink binding by using YAML, you must create a YAML file that defines an **SinkBinding** object, then apply it by using the **oc apply** command.

Prerequisites

- The OpenShift Serverless Operator, Knative Serving and Knative Eventing are installed on the cluster.
- Install the OpenShift CLI (**oc**).
- 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. To check that sink binding is set up correctly, create a Knative event display service, or event sink, that dumps incoming messages to its log.
    a. Create a service YAML file:

    Example service YAML file

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: event-display
spec:
  template:
    spec:
      containers:
        - image: quay.io/openshift-knative/showcase
```

    b. Create the service:

```
$ oc apply -f <filename>
```

2. Create a sink binding instance that directs events to the service.

a. Create a sink binding YAML file:

Example service YAML file

```
apiVersion: sources.knative.dev/v1alpha1
kind: SinkBinding
metadata:
  name: bind-heartbeat
spec:
  subject:
    apiVersion: batch/v1
    kind: Job (1)
    selector:
      matchLabels:
        app: heartbeat-cron

  sink:
    ref:
      apiVersion: serving.knative.dev/v1
      kind: Service
      name: event-display
```

1. In this example, any Job with the label **app: heartbeat-cron** will be bound to the event sink.

b. Create the sink binding:

```
$ oc apply -f <filename>
```

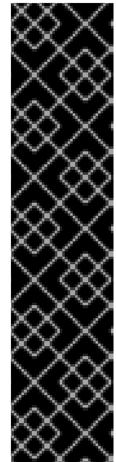
3. Create a **CronJob** object.

a. Create a cron job YAML file:

Example cron job YAML file

```
apiVersion: batch/v1
kind: CronJob
metadata:
  name: heartbeat-cron
spec:
  # Run every minute
  schedule: "* * * * *"
  jobTemplate:
    metadata:
      labels:
        app: heartbeat-cron
        bindings.knative.dev/include: "true"
    spec:
      template:
        spec:
          restartPolicy: Never
          containers:
            - name: single-heartbeat
              image: quay.io/openshift-knative/heartbeats:latest
```

```
        args:
          - --period=1
        env:
          - name: ONE_SHOT
            value: "true"
          - name: POD_NAME
            valueFrom:
              fieldRef:
                fieldPath: metadata.name
          - name: POD_NAMESPACE
            valueFrom:
              fieldRef:
                fieldPath: metadata.namespace
```



IMPORTANT

To use sink binding, you must manually add a **bindings.knative.dev/include=true** label to your Knative resources.

For example, to add this label to a **CronJob** resource, add the following lines to the **Job** resource YAML definition:

```
jobTemplate:
  metadata:
    labels:
      app: heartbeat-cron
      bindings.knative.dev/include: "true"
```

b. Create the cron job:

```
$ oc apply -f <filename>
```

4. Check that the controller is mapped correctly by entering the following command and inspecting the output:

```
$ oc get sinkbindings.sources.knative.dev bind-heartbeat -oyaml
```

Example output

```
spec:
  sink:
    ref:
      apiVersion: serving.knative.dev/v1
      kind: Service
      name: event-display
      namespace: default
  subject:
    apiVersion: batch/v1
    kind: Job
    namespace: default
    selector:
      matchLabels:
        app: heartbeat-cron
```

Verification

You can verify that the Kubernetes events were sent to the Knative event sink by looking at the message dumper function logs.

1. Enter the command:

```
$ oc get pods
```

2. Enter the command:

```
$ oc logs $(oc get pod -o name | grep event-display) -c user-container
```

Example output

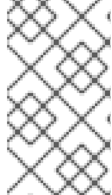
```
☁  cloudevents.Event
Validation: valid
Context Attributes,
  specversion: 1.0
  type: dev.knative.eventing.samples.heartbeat
  source: https://knative.dev/eventing-contrib/cmd/heartbeats/#event-test/mypod
  id: 2b72d7bf-c38f-4a98-a433-608fbcdd2596
  time: 2019-10-18T15:23:20.809775386Z
  contenttype: application/json
Extensions,
  beats: true
  heart: yes
  the: 42
Data,
  {
    "id": 1,
    "label": ""
  }
```

2.6.1.2. Creating a sink binding by using the Knative CLI

You can use the **kn source binding create** command to create a sink binding by using the Knative (**kn**) CLI. Using the Knative CLI to create event sources provides a more streamlined and intuitive user interface than modifying YAML files directly.

Prerequisites

- The OpenShift Serverless Operator, Knative Serving and Knative Eventing are installed on the cluster.
- 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.
- Install the Knative (**kn**) CLI.
- Install the OpenShift CLI (**oc**).



NOTE

The following procedure requires you to create YAML files.

If you change the names of the YAML files from those used in the examples, you must ensure that you also update the corresponding CLI commands.

Procedure

1. To check that sink binding is set up correctly, create a Knative event display service, or event sink, that dumps incoming messages to its log:

```
$ kn service create event-display --image quay.io/openshift-knative/showcase
```

2. Create a sink binding instance that directs events to the service:

```
$ kn source binding create bind-heartbeat --subject Job:batch/v1:app=heartbeat-cron --sink
ksvc:event-display
```

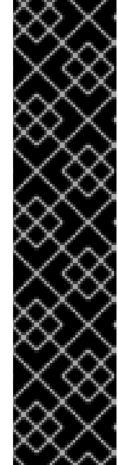
3. Create a **CronJob** object.

   a. Create a cron job YAML file:

Example cron job YAML file

```
apiVersion: batch/v1
kind: CronJob
metadata:
  name: heartbeat-cron
spec:
  # Run every minute
  schedule: "* * * * *"
  jobTemplate:
    metadata:
      labels:
        app: heartbeat-cron
        bindings.knative.dev/include: "true"
    spec:
      template:
        spec:
          restartPolicy: Never
          containers:
            - name: single-heartbeat
              image: quay.io/openshift-knative/heartbeats:latest
              args:
                - --period=1
              env:
                - name: ONE_SHOT
                  value: "true"
                - name: POD_NAME
                  valueFrom:
                    fieldRef:
                      fieldPath: metadata.name
                - name: POD_NAMESPACE
```

```
          valueFrom:
            fieldRef:
              fieldPath: metadata.namespace
```



IMPORTANT

To use sink binding, you must manually add a **bindings.knative.dev/include=true** label to your Knative CRs.

For example, to add this label to a **CronJob** CR, add the following lines to the **Job** CR YAML definition:

```
jobTemplate:
  metadata:
    labels:
      app: heartbeat-cron
      bindings.knative.dev/include: "true"
```

b. Create the cron job:

```
$ oc apply -f <filename>
```

4. Check that the controller is mapped correctly by entering the following command and inspecting the output:

```
$ kn source binding describe bind-heartbeat
```

Example output

```
Name:         bind-heartbeat
Namespace:    demo-2
Annotations:  sources.knative.dev/creator=minikube-user,
sources.knative.dev/lastModifier=minikub ...
Age:          2m
Subject:
  Resource:   job (batch/v1)
  Selector:
    app:      heartbeat-cron
Sink:
  Name:       event-display
  Resource:   Service (serving.knative.dev/v1)

Conditions:
  OK TYPE     AGE REASON
  ++ Ready    2m
```

Verification

You can verify that the Kubernetes events were sent to the Knative event sink by looking at the message dumper function logs.

- View the message dumper function logs by entering the following commands:

```
$ oc get pods
```

```
$ oc logs $(oc get pod -o name | grep event-display) -c user-container
```

Example output

```
☁  cloudevents.Event
Validation: valid
Context Attributes,
  specversion: 1.0
  type: dev.knative.eventing.samples.heartbeat
  source: https://knative.dev/eventing-contrib/cmd/heartbeats/#event-test/mypod
  id: 2b72d7bf-c38f-4a98-a433-608fbcdd2596
  time: 2019-10-18T15:23:20.809775386Z
  contenttype: application/json
Extensions,
  beats: true
  heart: yes
  the: 42
Data,
  {
    "id": 1,
    "label": ""
  }
```

2.6.1.2.1. Knative CLI sink flag

When you create an event source by using the Knative (**kn**) CLI, you can specify a sink where events are sent to from that resource by using the **--sink** flag. The sink can be any addressable or callable resource that can receive incoming events from other resources.

The following example creates a sink binding that uses a service, **http://event-display.svc.cluster.local**, as the sink:

Example command using the sink flag

```
$ kn source binding create bind-heartbeat \
  --namespace sinkbinding-example \
  --subject "Job:batch/v1:app=heartbeat-cron" \
  --sink http://event-display.svc.cluster.local \ (1)
  --ce-override "sink=bound"
```

1. **svc** in **http://event-display.svc.cluster.local** determines that the sink is a Knative service. Other default sink prefixes include **channel**, and **broker**.

2.6.1.3. Creating a sink binding by using the web console

After Knative Eventing is installed on your cluster, you can create a sink binding by using the web console. Using the OpenShift Container Platform web console provides a streamlined and intuitive user interface to create an event source.

Prerequisites

- You have logged in to the OpenShift Container Platform web console.
- The OpenShift Serverless Operator, Knative Serving, and Knative Eventing are installed on your OpenShift Container Platform cluster.
- 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. Create a Knative service to use as a sink:
    a. In the **Developer** perspective, navigate to **+Add → YAML**.
    b. Copy the example YAML:

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: event-display
spec:
  template:
    spec:
      containers:
        - image: quay.io/openshift-knative/showcase
```

    c. Click **Create**.

2. Create a **CronJob** resource that is used as an event source and sends an event every minute.
    a. In the **Developer** perspective, navigate to **+Add → YAML**.
    b. Copy the example YAML:

```
apiVersion: batch/v1
kind: CronJob
metadata:
  name: heartbeat-cron
spec:
  # Run every minute
  schedule: "*/1 * * * *"
  jobTemplate:
    metadata:
      labels:
        app: heartbeat-cron
        bindings.knative.dev/include: true 1
    spec:
      template:
        spec:
          restartPolicy: Never
          containers:
            - name: single-heartbeat
              image: quay.io/openshift-knative/heartbeats
              args:
              - --period=1
              env:
```

```
        - name: ONE_SHOT
          value: "true"
        - name: POD_NAME
          valueFrom:
            fieldRef:
              fieldPath: metadata.name
        - name: POD_NAMESPACE
          valueFrom:
            fieldRef:
              fieldPath: metadata.namespace
```

1 Ensure that you include the **bindings.knative.dev/include: true** label. The default namespace selection behavior of OpenShift Serverless uses inclusion mode.

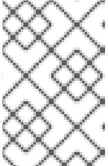
c. Click **Create**.

3. Create a sink binding in the same namespace as the service created in the previous step, or any other sink that you want to send events to.

a. In the **Developer** perspective, navigate to **+Add → Event Source**. The **Event Sources** page is displayed.

b. Optional: If you have multiple providers for your event sources, select the required provider from the **Providers** list to filter the available event sources from the provider.

c. Select **Sink Binding** and then click **Create Event Source**. The **Create Event Source** page is displayed.

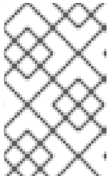


NOTE

You can configure the **Sink Binding** settings by using the **Form view** or **YAML view** and can switch between the views. The data is persisted when switching between the views.

d. In the **apiVersion** field enter **batch/v1**.

e. In the **Kind** field enter **Job**.



NOTE

The **CronJob** kind is not supported directly by OpenShift Serverless sink binding, so the **Kind** field must target the **Job** objects created by the cron job, rather than the cron job object itself.

f. In the **Target** section, select your event sink. This can be either a **Resource** or a **URI**:

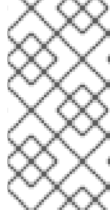
i. Select **Resource** to use a channel, broker, or service as an event sink for the event source. In this example, the **event-display** service created in the previous step is used as the target **Resource**.

ii. Select **URI** to specify a Uniform Resource Identifier (URI) where the events are routed to.

g. In the **Match labels** section:

i. Enter **app** in the **Name** field.

ii. Enter **heartbeat-cron** in the **Value** field.



NOTE

The label selector is required when using cron jobs with sink binding, rather than the resource name. This is because jobs created by a cron job do not have a predictable name, and contain a randomly generated string in their name. For example, **hearthbeat-cron-1cc23f**.

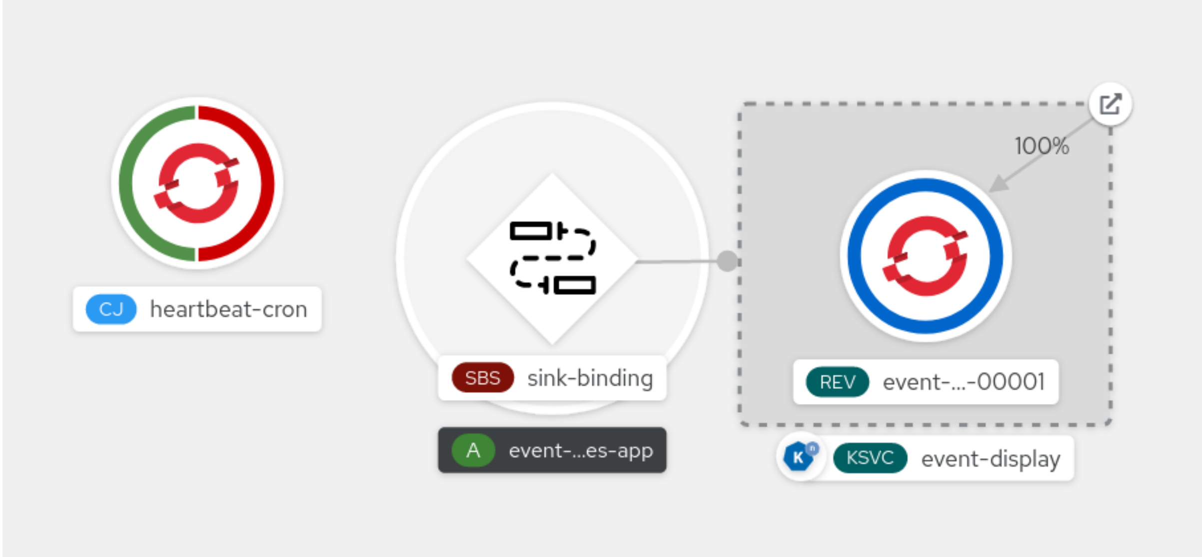
h. Click **Create**.

Verification

You can verify that the sink binding, sink, and cron job have been created and are working correctly by viewing the **Topology** page and pod logs.

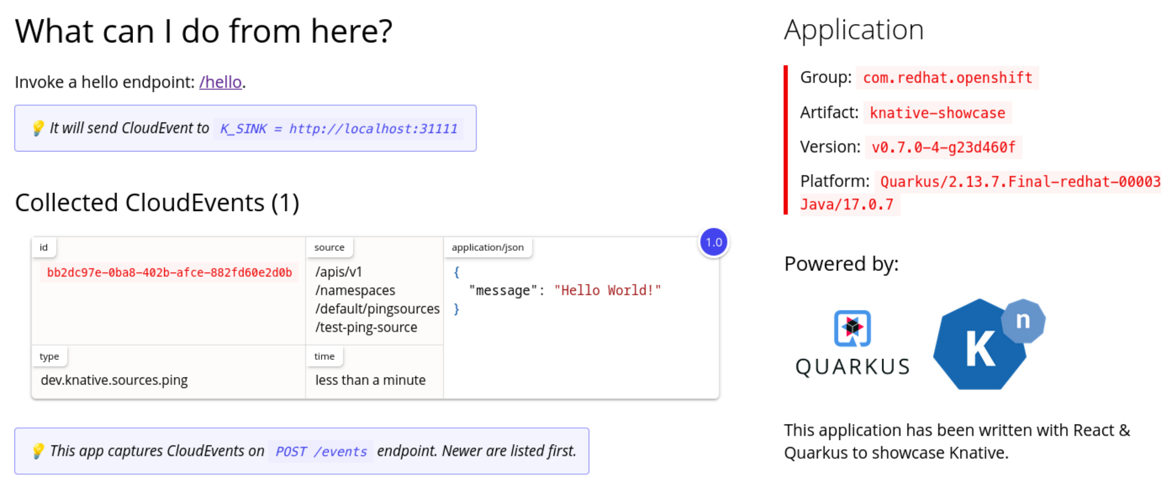
1. In the **Developer** perspective, navigate to **Topology**.

2. View the sink binding, sink, and heartbeats cron job.



3. Observe that successful jobs are being registered by the cron job once the sink binding is added. This means that the sink binding is successfully reconfiguring the jobs created by the cron job.

4. Browse the **event-display** service to see events produced by the heartbeats cron job.



2.6.1.4. Sink binding reference

You can use a **PodSpecable** object as an event source by creating a sink binding. You can configure multiple parameters when creating a **SinkBinding** object.

SinkBinding objects support the following parameters:

Field	Description	Required or optional
apiVersion	Specifies the API version, for example **sources.knative.dev/v1**.	Required
kind	Identifies this resource object as a **SinkBinding** object.	Required
metadata	Specifies metadata that uniquely identifies the **SinkBinding** object. For example, a **name**.	Required
spec	Specifies the configuration information for this **SinkBinding** object.	Required
spec.sink	A reference to an object that resolves to a URI to use as the sink.	Required
spec.subject	References the resources for which the runtime contract is augmented by binding implementations.	Required

Field	Description	Required or optional
spec.ceOverrides	Defines overrides to control the output format and modifications to the event sent to the sink.	Optional

2.6.1.4.1. Subject parameter

The **Subject** parameter references the resources for which the runtime contract is augmented by binding implementations. You can configure multiple fields for a **Subject** definition.

The **Subject** definition supports the following fields:

Field	Description	Required or optional
apiVersion	API version of the referent.	Required
kind	Kind of the referent.	Required
namespace	Namespace of the referent. If omitted, this defaults to the namespace of the object.	Optional
name	Name of the referent.	Do not use if you configure **selector**.
selector	Selector of the referents.	Do not use if you configure **name**.
selector.matchExpressions	A list of label selector requirements.	Only use one of either **matchExpressions** or **matchLabels**.
selector.matchExpressions.key	The label key that the selector applies to.	Required if using **matchExpressions**.
selector.matchExpressions.operator	Represents a key’s relationship to a set of values. Valid operators are **In**, **NotIn**, **Exists** and **DoesNotExist**.	Required if using **matchExpressions**.
selector.matchExpressions.values	An array of string values. If the **operator** parameter value is **In** or **NotIn**, the values array must be non-empty. If the **operator** parameter value is **Exists** or **DoesNotExist**, the values array must be empty. This array is replaced during a strategic merge patch.	Required if using **matchExpressions**.

Field	Description	Required or optional
selector.matchLabels	A map of key-value pairs. Each key-value pair in the **matchLabels** map is equivalent to an element of **matchExpressions**, where the key field is **matchLabels.<key>**, the **operator** is **In**, and the **values** array contains only **matchLabels.<value>**.	Only use one of either **matchExpressions** or **matchLabels**.

Subject parameter examples

Given the following YAML, the **Deployment** object named **mysubject** in the **default** namespace is selected:

```
apiVersion: sources.knative.dev/v1
kind: SinkBinding
metadata:
  name: bind-heartbeat
spec:
  subject:
    apiVersion: apps/v1
    kind: Deployment
    namespace: default
    name: mysubject
  ...
```

Given the following YAML, any **Job** object with the label **working=example** in the **default** namespace is selected:

```
apiVersion: sources.knative.dev/v1
kind: SinkBinding
metadata:
  name: bind-heartbeat
spec:
  subject:
    apiVersion: batch/v1
    kind: Job
    namespace: default
    selector:
      matchLabels:
        working: example
  ...
```

Given the following YAML, any **Pod** object with the label **working=example** or **working=sample** in the **default** namespace is selected:

```
apiVersion: sources.knative.dev/v1
kind: SinkBinding
metadata:
  name: bind-heartbeat
```

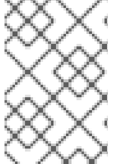
```
spec:
  subject:
    apiVersion: v1
    kind: Pod
    namespace: default
    selector:
      - matchExpression:
        key: working
        operator: In
        values:
          - example
          - sample
  ...
```

2.6.1.4.2. CloudEvent overrides

A **ceOverrides** definition provides overrides that control the CloudEvent's output format and modifications sent to the sink. You can configure multiple fields for the **ceOverrides** definition.

A **ceOverrides** definition supports the following fields:

Field	Description	Required or optional
extensions	Specifies which attributes are added or overridden on the outbound event. Each **extensions** key-value pair is set independently on the event as an attribute extension.	Optional



NOTE

Only valid **CloudEvent** attribute names are allowed as extensions. You cannot set the spec defined attributes from the extensions override configuration. For example, you can not modify the **type** attribute.

CloudEvent Overrides example

```
apiVersion: sources.knative.dev/v1
kind: SinkBinding
metadata:
  name: bind-heartbeat
spec:
  ...
  ceOverrides:
    extensions:
      extra: this is an extra attribute
      additional: 42
```

This sets the **K_CE_OVERRIDES** environment variable on the **subject**:

Example output

```
{ "extensions": { "extra": "this is an extra attribute", "additional": "42" } }
```

2.6.1.4.3. The include label

To use a sink binding, you need to do assign the **bindings.knative.dev/include: "true"** label to either the resource or the namespace that the resource is included in. If the resource definition does not include the label, a cluster administrator can attach it to the namespace by running:

```
$ oc label namespace <namespace> bindings.knative.dev/include=true
```

2.6.1.5. Integrating Service Mesh with a sink binding

Prerequisites

- You have integrated Service Mesh with OpenShift Serverless.

Procedure

1. Create a **Service** in a namespace that is a member of the **ServiceMeshMemberRoll**.

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: event-display
  namespace: <namespace> 1
spec:
  template:
    metadata:
      annotations:
        sidecar.istio.io/inject: "true" 2
        sidecar.istio.io/rewriteAppHTTPProbers: "true"
    spec:
      containers:
      - image: quay.io/openshift-knative/showcase
```

1. A namespace that is a member of the **ServiceMeshMemberRoll**.
2. Injects Service Mesh sidecars into the Knative service pods.

2. Apply the **Service** resource.

```
$ oc apply -f <filename>
```

3. Create a **SinkBinding** resource.

```
apiVersion: sources.knative.dev/v1
kind: SinkBinding
metadata:
  name: bind-heartbeat
  namespace: <namespace> 1
spec:
  subject:
```

```
    apiVersion: batch/v1
    kind: Job 2
    selector:
      matchLabels:
        app: heartbeat-cron

  sink:
    ref:
      apiVersion: serving.knative.dev/v1
      kind: Service
      name: event-display
```

1. A namespace that is a member of the **ServiceMeshMemberRoll**.
2. In this example, any Job with the label **app: heartbeat-cron** is bound to the event sink.

4. Apply the **SinkBinding** resource.

```
$ oc apply -f <filename>
```

5. Create a **CronJob**:

```
apiVersion: batch/v1
kind: CronJob
metadata:
  name: heartbeat-cron
  namespace: <namespace> 1
spec:
  # Run every minute
  schedule: "* * * * *"
  jobTemplate:
    metadata:
      labels:
        app: heartbeat-cron
        bindings.knative.dev/include: "true"
    spec:
      template:
        metadata:
          annotations:
            sidecar.istio.io/inject: "true" 2
            sidecar.istio.io/rewriteAppHTTPProbers: "true"
        spec:
          restartPolicy: Never
          containers:
            - name: single-heartbeat
              image: quay.io/openshift-knative/heartbeats:latest
              args:
                - --period=1
              env:
                - name: ONE_SHOT
                  value: "true"
                - name: POD_NAME
                  valueFrom:
                    fieldRef:
```

```
              fieldPath: metadata.name
          - name: POD_NAMESPACE
            valueFrom:
              fieldRef:
                fieldPath: metadata.namespace
```

1. A namespace that is a member of the **ServiceMeshMemberRoll**.
2. Injects Service Mesh sidecars into the **CronJob** pods.

6. Apply the **CronJob** resource.

```
$ oc apply -f <filename>
```

Verification

To verify that the events were sent to the Knative event sink, look at the message dumper function logs.

1. Enter the following command:

```
$ oc get pods
```

2. Enter the following command:

```
$ oc logs $(oc get pod -o name | grep event-display) -c user-container
```

Example output

```
☁  cloudevents.Event
Validation: valid
Context Attributes,
  specversion: 1.0
  type: dev.knative.eventing.samples.heartbeat
  source: https://knative.dev/eventing/test/heartbeats/#event-test/mypod
  id: 2b72d7bf-c38f-4a98-a433-608fbcdd2596
  time: 2019-10-18T15:23:20.809775386Z
  contenttype: application/json
Extensions,
  beats: true
  heart: yes
  the: 42
Data,
  {
    "id": 1,
    "label": ""
  }
```

Additional resources

- IntegratingService Mesh with OpenShift Serverless

2.6.2. Container source

Container sources create a container image that generates events and sends events to a sink. You can use a container source to create a custom event source, by creating a container image and a **ContainerSource** object that uses your image URI.

2.6.2.1. Guidelines for creating a container image

Two environment variables are injected by the container source controller: **K_SINK** and **K_CE_OVERRIDES**. These variables are resolved from the **sink** and **ceOverrides** spec, respectively. Events are sent to the sink URI specified in the **K_SINK** environment variable. The message must be sent as a **POST** using the CloudEvent HTTP format.

Example container images

The following is an example of a heartbeats container image:

```
package main

import (
 "context"
 "encoding/json"
 "flag"
 "fmt"
 "log"
 "os"
 "strconv"
 "time"

 duckv1 "knative.dev/pkg/apis/duck/v1"

 cloudevents "github.com/cloudevents/sdk-go/v2"
 "github.com/kelseyhightower/envconfig"
)

type Heartbeat struct {
 Sequence int    `json:"id"`
 Label    string `json:"label"`
}

var (
 eventSource string
 eventType   string
 sink        string
 label       string
 periodStr   string
)

func init() {
 flag.StringVar(&eventSource, "eventSource", "", "the event-source (CloudEvents)")
 flag.StringVar(&eventType, "eventType", "dev.knative.eventing.samples.heartbeat", "the event-type
(CloudEvents)")
 flag.StringVar(&sink, "sink", "", "the host url to heartbeat to")
 flag.StringVar(&label, "label", "", "a special label")
 flag.StringVar(&periodStr, "period", "5", "the number of seconds between heartbeats")
}

type envConfig struct {
```

```
 // Sink URL where to send heartbeat cloud events
 Sink string `envconfig:"K_SINK"`

 // CEOverrides are the CloudEvents overrides to be applied to the outbound event.
 CEOverrides string `envconfig:"K_CE_OVERRIDES"`

 // Name of this pod.
 Name string `envconfig:"POD_NAME" required:"true"`

 // Namespace this pod exists in.
 Namespace string `envconfig:"POD_NAMESPACE" required:"true"`

 // Whether to run continuously or exit.
 OneShot bool `envconfig:"ONE_SHOT" default:"false"`
}

func main() {
 flag.Parse()

 var env envConfig
 if err := envconfig.Process("", &env); err != nil {
  log.Printf("[ERROR] Failed to process env var: %s", err)
  os.Exit(1)
 }

 if env.Sink != "" {
  sink = env.Sink
 }

 var ceOverrides *duckv1.CloudEventOverrides
 if len(env.CEOverrides) > 0 {
  overrides := duckv1.CloudEventOverrides{}
  err := json.Unmarshal([]byte(env.CEOverrides), &overrides)
  if err != nil {
   log.Printf("[ERROR] Unparseable CloudEvents overrides %s: %v", env.CEOverrides, err)
   os.Exit(1)
  }
  ceOverrides = &overrides
 }

 p, err := cloudevents.NewHTTP(cloudevents.WithTarget(sink))
 if err != nil {
  log.Fatalf("failed to create http protocol: %s", err.Error())
 }

 c, err := cloudevents.NewClient(p, cloudevents.WithUUIDs(), cloudevents.WithTimeNow())
 if err != nil {
  log.Fatalf("failed to create client: %s", err.Error())
 }

 var period time.Duration
 if p, err := strconv.Atoi(periodStr); err != nil {
  period = time.Duration(5) * time.Second
 } else {
  period = time.Duration(p) * time.Second
 }
```

```
 if eventSource == "" {
  eventSource = fmt.Sprintf("https://knative.dev/eventing-contrib/cmd/heartbeats/#%s/%s",
env.Namespace, env.Name)
  log.Printf("Heartbeats Source: %s", eventSource)
 }

 if len(label) > 0 && label[0] == '"' {
  label, _ = strconv.Unquote(label)
 }
 hb := &Heartbeat{
  Sequence: 0,
  Label:    label,
 }
 ticker := time.NewTicker(period)
 for {
  hb.Sequence++

  event := cloudevents.NewEvent("1.0")
  event.SetType(eventType)
  event.SetSource(eventSource)
  event.SetExtension("the", 42)
  event.SetExtension("heart", "yes")
  event.SetExtension("beats", true)

  if ceOverrides != nil && ceOverrides.Extensions != nil {
   for n, v := range ceOverrides.Extensions {
    event.SetExtension(n, v)
   }
  }

  if err := event.SetData(cloudevents.ApplicationJSON, hb); err != nil {
   log.Printf("failed to set cloudevents data: %s", err.Error())
  }

  log.Printf("sending cloudevent to %s", sink)
  if res := c.Send(context.Background(), event); !cloudevents.IsACK(res) {
   log.Printf("failed to send cloudevent: %v", res)
  }

  if env.OneShot {
   return
  }

  // Wait for next tick
  <-ticker.C
 }
}
```

The following is an example of a container source that references the previous heartbeats container image:

```
apiVersion: sources.knative.dev/v1
kind: ContainerSource
metadata:
  name: test-heartbeats
```

```
spec:
  template:
    spec:
      containers:
        # This corresponds to a heartbeats image URI that you have built and published
        - image: gcr.io/knative-releases/knative.dev/eventing/cmd/heartbeats
          name: heartbeats
          args:
            - --period=1
          env:
            - name: POD_NAME
              value: "example-pod"
            - name: POD_NAMESPACE
              value: "event-test"
  sink:
    ref:
      apiVersion: serving.knative.dev/v1
      kind: Service
      name: showcase
...
```

2.6.2.2. Creating and managing container sources by using the Knative CLI

You can use the **kn source container** commands to create and manage container sources by using the Knative (**kn**) CLI. Using the Knative CLI to create event sources provides a more streamlined and intuitive user interface than modifying YAML files directly.

Create a container source

```
$ kn source container create <container_source_name> --image <image_uri> --sink <sink>
```

Delete a container source

```
$ kn source container delete <container_source_name>
```

Describe a container source

```
$ kn source container describe <container_source_name>
```

List existing container sources

```
$ kn source container list
```

List existing container sources in YAML format

```
$ kn source container list -o yaml
```

Update a container source

This command updates the image URI for an existing container source:

```
$ kn source container update <container_source_name> --image <image_uri>
```

2.6.2.3. Creating a container source by using the web console

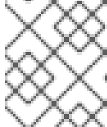
After Knative Eventing is installed on your cluster, you can create a container source by using the web console. Using the OpenShift Container Platform web console provides a streamlined and intuitive user interface to create an event source.

Prerequisites

- You have logged in to the OpenShift Container Platform web console.
- The OpenShift Serverless Operator, Knative Serving, and Knative Eventing are installed on your OpenShift Container Platform cluster.
- 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. In the **Developer** perspective, navigate to **+Add → Event Source**. The **Event Sources** page is displayed.
2. Select **Container Source** and then click **Create Event Source**. The **Create Event Source** page is displayed.
3. Configure the **Container Source** settings by using the **Form view** or **YAML view**:



NOTE

You can switch between the **Form view** and **YAML view**. The data is persisted when switching between the views.

   a. In the **Image** field, enter the URI of the image that you want to run in the container created by the container source.
   b. In the **Name** field, enter the name of the image.
   c. Optional: In the **Arguments** field, enter any arguments to be passed to the container.
   d. Optional: In the **Environment variables** field, add any environment variables to set in the container.
   e. In the **Target** section, select your event sink. This can be either a **Resource** or a **URI**:
      i. Select **Resource** to use a channel, broker, or service as an event sink for the event source.
      ii. Select **URI** to specify a Uniform Resource Identifier (URI) where the events are routed to.
4. After you have finished configuring the container source, click **Create**.

2.6.2.4. Container source reference

You can use a container as an event source, by creating a **ContainerSource** object. You can configure multiple parameters when creating a **ContainerSource** object.

ContainerSource objects support the following fields:

Field	Description	Required or optional
apiVersion	Specifies the API version, for example **sources.knative.dev/v1**.	Required
kind	Identifies this resource object as a **ContainerSource** object.	Required
metadata	Specifies metadata that uniquely identifies the **ContainerSource** object. For example, a **name**.	Required
spec	Specifies the configuration information for this **ContainerSource** object.	Required
spec.sink	A reference to an object that resolves to a URI to use as the sink.	Required
spec.template	A **template** spec for the **ContainerSource** object.	Required
spec.ceOverrides	Defines overrides to control the output format and modifications to the event sent to the sink.	Optional

Template parameter example

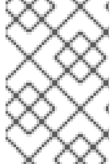
```
apiVersion: sources.knative.dev/v1
kind: ContainerSource
metadata:
  name: test-heartbeats
spec:
  template:
    spec:
      containers:
        - image: quay.io/openshift-knative/heartbeats:latest
          name: heartbeats
          args:
            - --period=1
          env:
            - name: POD_NAME
              value: "mypod"
            - name: POD_NAMESPACE
              value: "event-test"
  ...
```

2.6.2.4.1. CloudEvent overrides

A **ceOverrides** definition provides overrides that control the CloudEvent's output format and modifications sent to the sink. You can configure multiple fields for the **ceOverrides** definition.

A **ceOverrides** definition supports the following fields:

Field	Description	Required or optional
extensions	Specifies which attributes are added or overridden on the outbound event. Each **extensions** key-value pair is set independently on the event as an attribute extension.	Optional



NOTE

Only valid **CloudEvent** attribute names are allowed as extensions. You cannot set the spec defined attributes from the extensions override configuration. For example, you can not modify the **type** attribute.

CloudEvent Overrides example

```
apiVersion: sources.knative.dev/v1
kind: ContainerSource
metadata:
  name: test-heartbeats
spec:
  ...
  ceOverrides:
    extensions:
      extra: this is an extra attribute
      additional: 42
```

This sets the **K_CE_OVERRIDES** environment variable on the **subject**:

Example output

```
{ "extensions": { "extra": "this is an extra attribute", "additional": "42" } }
```

2.6.2.5. Integrating Service Mesh with ContainerSource

Prerequisites

- You have integrated Service Mesh with OpenShift Serverless.

Procedure

1. Create a **Service** in a namespace that is a member of the **ServiceMeshMemberRoll**.

```
apiVersion: serving.knative.dev/v1
```

```
kind: Service
metadata:
  name: event-display
  namespace: <namespace> 1
spec:
  template:
    metadata:
      annotations:
        sidecar.istio.io/inject: "true" 2
        sidecar.istio.io/rewriteAppHTTPProbers: "true"
    spec:
      containers:
      - image: quay.io/openshift-knative/showcase
```

1. A namespace that is a member of the **ServiceMeshMemberRoll**.

2. Injects Service Mesh sidecars into the Knative service pods.

2. Apply the **Service** resource.

```
$ oc apply -f <filename>
```

3. Create a **ContainerSource** object in a namespace that is a member of the **ServiceMeshMemberRoll** and sink set to the **event-display**.

```
apiVersion: sources.knative.dev/v1
kind: ContainerSource
metadata:
  name: test-heartbeats
  namespace: <namespace> 1
spec:
  template:
    metadata: 2
      annotations:
        sidecar.istio.io/inject: "true"
        sidecar.istio.io/rewriteAppHTTPProbers: "true"
    spec:
      containers:
        - image: quay.io/openshift-knative/heartbeats:latest
          name: heartbeats
          args:
            - --period=1s
          env:
            - name: POD_NAME
              value: "example-pod"
            - name: POD_NAMESPACE
              value: "event-test"
  sink:
    ref:
      apiVersion: serving.knative.dev/v1
      kind: Service
      name: event-display
```

1. A namespace is part of the **ServiceMeshMemberRoll**.

2 Enables Service Mesh integration with a **ContainerSource** object.

4. Apply the **ContainerSource** resource.

```
$ oc apply -f <filename>
```

Verification

To verify that the events were sent to the Knative event sink, look at the message dumper function logs.

1. Enter the following command:

```
$ oc get pods
```

2. Enter the following command:

```
$ oc logs $(oc get pod -o name | grep event-display) -c user-container
```

Example output

```
☁  cloudevents.Event
Validation: valid
Context Attributes,
  specversion: 1.0
  type: dev.knative.eventing.samples.heartbeat
  source: https://knative.dev/eventing/test/heartbeats/#event-test/mypod
  id: 2b72d7bf-c38f-4a98-a433-608fbcdd2596
  time: 2019-10-18T15:23:20.809775386Z
  contenttype: application/json
Extensions,
  beats: true
  heart: yes
  the: 42
Data,
  {
    "id": 1,
    "label": ""
  }
```

Additional resources

- IntegratingService Mesh with OpenShift Serverless

2.7. CONNECTING AN EVENT SOURCE TO AN EVENT SINK BY USING THE DEVELOPER PERSPECTIVE

When you create an event source by using the OpenShift Container Platform web console, you can specify a target event sink that events are sent to from that source. The event sink can be any addressable or callable resource that can receive incoming events from other resources.

2.7.1. Connect an event source to an event sink by using the Developer perspective

Prerequisites

- The OpenShift Serverless Operator, Knative Serving, and Knative Eventing are installed on your OpenShift Container Platform cluster.
- You have logged in to the web console and are in the **Developer** perspective.
- 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.
- You have created an event sink, such as a Knative service, channel or broker.

Procedure

1. Create an event source of any type, by navigating to **+Add → Event Source** and selecting the event source type that you want to create.
2. In the **Target** section of the **Create Event Source** form view, select your event sink. This can be either a **Resource** or a **URI**:
   a. Select **Resource** to use a channel, broker, or service as an event sink for the event source.
   b. Select **URI** to specify a Uniform Resource Identifier (URI) where the events are routed to.
3. Click **Create**.

Verification

You can verify that the event source was created and is connected to the sink by viewing the **Topology** page.

1. In the **Developer** perspective, navigate to **Topology**.
2. View the event source and click the connected event sink to see the sink details in the right panel.

CHAPTER 3. EVENT SINKS

3.1. EVENT SINKS

When you create an event source, you can specify an event sink where events are sent to from the source. An event sink is an addressable or a callable resource that can receive incoming events from other resources. Knative services, channels, and brokers are all examples of event sinks. There is also a specific Apache Kafka sink type available.

Addressable objects receive and acknowledge an event delivered over HTTP to an address defined in their **status.address.url** field. As a special case, the core Kubernetes **Service** object also fulfills the addressable interface.

Callable objects are able to receive an event delivered over HTTP and transform the event, returning **0** or **1** new events in the HTTP response. These returned events may be further processed in the same way that events from an external event source are processed.

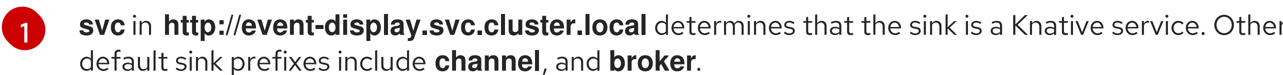
3.1.1. Knative CLI sink flag

When you create an event source by using the Knative (**kn**) CLI, you can specify a sink where events are sent to from that resource by using the **--sink** flag. The sink can be any addressable or callable resource that can receive incoming events from other resources.

The following example creates a sink binding that uses a service, **http://event-display.svc.cluster.local**, as the sink:

Example command using the sink flag

```
$ kn source binding create bind-heartbeat \
  --namespace sinkbinding-example \
  --subject "Job:batch/v1:app=heartbeat-cron" \
  --sink http://event-display.svc.cluster.local \ 1
  --ce-override "sink=bound"
```



1. **svc** in **http://event-display.svc.cluster.local** determines that the sink is a Knative service. Other default sink prefixes include **channel**, and **broker**.

TIP

You can configure which CRs can be used with the **--sink** flag for Knative (**kn**) CLI commands by Customizing **kn**.

3.2. CREATING EVENT SINKS

When you create an event source, you can specify an event sink where events are sent to from the source. An event sink is an addressable or a callable resource that can receive incoming events from other resources. Knative services, channels, and brokers are all examples of event sinks. There is also a specific Apache Kafka sink type available.

For information about creating resources that can be used as event sinks, see the following documentation:

- Serverless applications
- Creating brokers
- Creating channels
- Kafka sink

3.3. SINK FOR APACHE KAFKA

Apache Kafka sinks are a type of event sink that are available if a cluster administrator has enabled Apache Kafka on your cluster. You can send events directly from an event source to a Kafka topic by using a Kafka sink.

3.3.1. Creating an Apache Kafka sink by using YAML

You can create a Kafka sink that sends events to a Kafka topic. By default, a Kafka sink uses the binary content mode, which is more efficient than the structured mode. To create a Kafka sink by using YAML, you must create a YAML file that defines a **KafkaSink** object, then apply it by using the **oc apply** command.

Prerequisites

- The OpenShift Serverless Operator, Knative Eventing, and the **KnativeKafka** custom resource (CR) are installed on your cluster.
- 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.
- You have access to a Red Hat AMQ Streams (Kafka) cluster that produces the Kafka messages you want to import.
- Install the OpenShift CLI (**oc**).

Procedure

1. Create a **KafkaSink** object definition as a YAML file:

   Kafka sink YAML

   ```
   apiVersion: eventing.knative.dev/v1alpha1
   kind: KafkaSink
   metadata:
     name: <sink-name>
     namespace: <namespace>
   spec:
     topic: <topic-name>
     bootstrapServers:
      - <bootstrap-server>
   ```

2. To create the Kafka sink, apply the **KafkaSink** YAML file:

   ```
   $ oc apply -f <filename>
   ```

3. Configure an event source so that the sink is specified in its spec:

Example of a Kafka sink connected to an API server source

```
apiVersion: sources.knative.dev/v1alpha2
kind: ApiServerSource
metadata:
  name: <source-name> 1
  namespace: <namespace> 2
spec:
  serviceAccountName: <service-account-name> 3
  mode: Resource
  resources:
  - apiVersion: v1
    kind: Event
  sink:
    ref:
      apiVersion: eventing.knative.dev/v1alpha1
      kind: KafkaSink
      name: <sink-name> 4
```

1 The name of the event source.

2 The namespace of the event source.

3 The service account for the event source.

4 The Kafka sink name.

3.3.2. Creating an event sink for Apache Kafka by using the OpenShift Container Platform web console

You can create a Kafka sink that sends events to a Kafka topic by using the **Developer** perspective in the OpenShift Container Platform web console. By default, a Kafka sink uses the binary content mode, which is more efficient than the structured mode.

As a developer, you can create an event sink to receive events from a particular source and send them to a Kafka topic.

Prerequisites

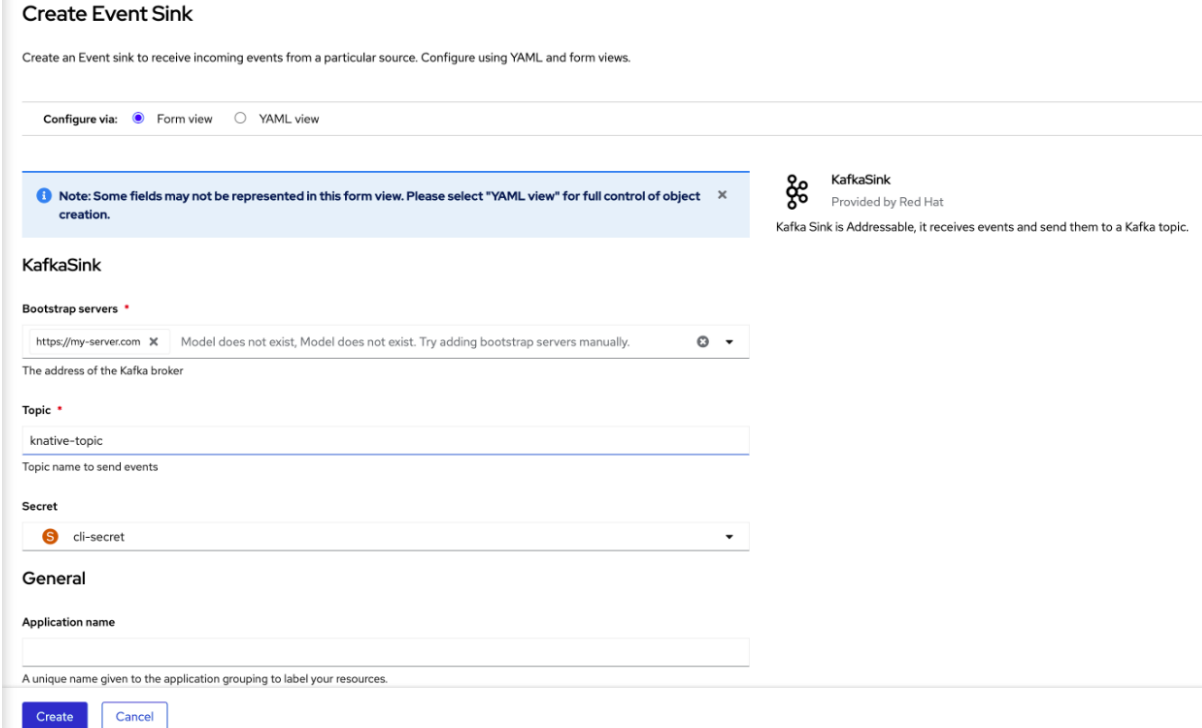
- You have installed the OpenShift Serverless Operator, with Knative Serving, Knative Eventing, and Knative broker for Apache Kafka APIs, from the OperatorHub.
- You have created a Kafka topic in your Kafka environment.

Procedure

1. In the **Developer** perspective, navigate to the **+Add** view.
2. Click **Event Sink** in the **Eventing catalog**.
3. Search for **KafkaSink** in the catalog items and click it.

4. Click **Create Event Sink**

5. In the form view, type the URL of the bootstrap server, which is a combination of host name and port.



6. Type the name of the topic to send event data.

7. Type the name of the event sink.

8. Click **Create**.

Verification

1. In the **Developer** perspective, navigate to the **Topology** view.

2. Click the created event sink to view its details in the right panel.

3.3.3. Configuring security for Apache Kafka sinks

Transport Layer Security (TLS) is used by Apache Kafka clients and servers to encrypt traffic between Knative and Kafka, as well as for authentication. TLS is the only supported method of traffic encryption for the Knative broker implementation for Apache Kafka.

Simple Authentication and Security Layer (SASL) is used by Apache Kafka for authentication. If you use SASL authentication on your cluster, users must provide credentials to Knative for communicating with the Kafka cluster; otherwise events cannot be produced or consumed.

Prerequisites

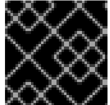
- The OpenShift Serverless Operator, Knative Eventing, and the **KnativeKafka** custom resources (CRs) are installed on your OpenShift Container Platform cluster.

- Kafka sink is enabled in the **KnativeKafka** CR.

- 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.
- You have a Kafka cluster CA certificate stored as a **.pem** file.
- You have a Kafka cluster client certificate and a key stored as **.pem** files.
- You have installed the OpenShift (**oc**) CLI.
- You have chosen the SASL mechanism to use, for example, **PLAIN**, **SCRAM-SHA-256**, or **SCRAM-SHA-512**.

Procedure

1. Create the certificate files as a secret in the same namespace as your **KafkaSink** object:



IMPORTANT

Certificates and keys must be in PEM format.

- For authentication using SASL without encryption:

```
$ oc create secret -n <namespace> generic <secret_name> \
  --from-literal=protocol=SASL_PLAINTEXT \
  --from-literal=sasl.mechanism=<sasl_mechanism> \
  --from-literal=user=<username> \
  --from-literal=password=<password>
```

- For authentication using SASL and encryption using TLS:

```
$ oc create secret -n <namespace> generic <secret_name> \
  --from-literal=protocol=SASL_SSL \
  --from-literal=sasl.mechanism=<sasl_mechanism> \
  --from-file=ca.crt=<my_caroot.pem_file_path> \ 1
  --from-literal=user=<username> \
  --from-literal=password=<password>
```

1 The **ca.crt** can be omitted to use the system's root CA set if you are using a public cloud managed Kafka service.

- For authentication and encryption using TLS:

```
$ oc create secret -n <namespace> generic <secret_name> \
  --from-literal=protocol=SSL \
  --from-file=ca.crt=<my_caroot.pem_file_path> \ 1
  --from-file=user.crt=<my_cert.pem_file_path> \
  --from-file=user.key=<my_key.pem_file_path>
```

1 The **ca.crt** can be omitted to use the system's root CA set if you are using a public cloud managed Kafka service.

2. Create or modify a **KafkaSink** object and add a reference to your secret in the **auth** spec:

```
apiVersion: eventing.knative.dev/v1alpha1
kind: KafkaSink
metadata:
  name: <sink_name>
  namespace: <namespace>
spec:
...
  auth:
    secret:
      ref:
        name: <secret_name>
...
```

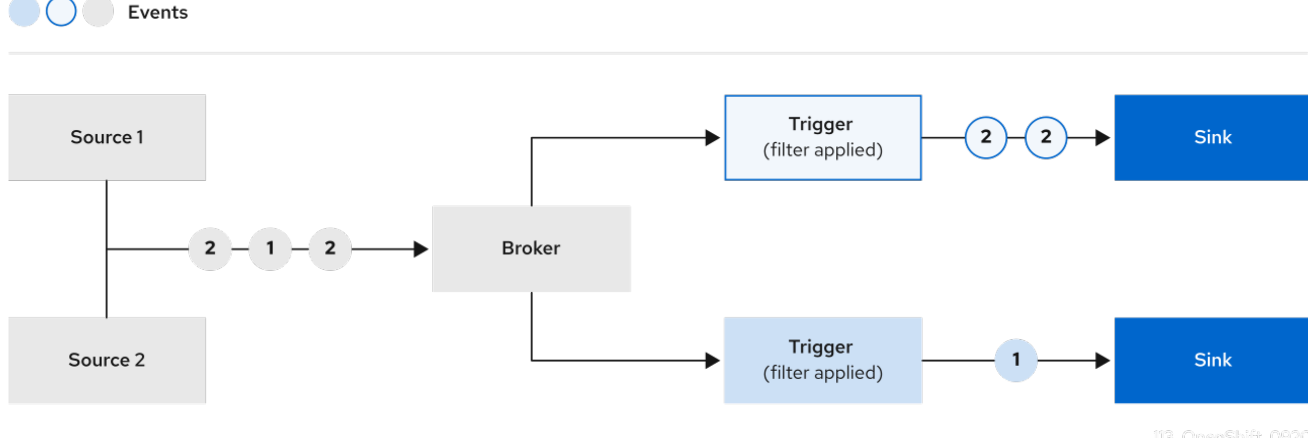
3. Apply the **KafkaSink** object:

```
$ oc apply -f <filename>
```

CHAPTER 4. BROKERS

4.1. BROKERS

Brokers can be used in combination with triggers to deliver events from an event source to an event sink. Events are sent from an event source to a broker as an HTTP **POST** request. After events have entered the broker, they can be filtered by CloudEvent attributes using triggers, and sent as an HTTP **POST** request to an event sink.



4.2. BROKER TYPES

Cluster administrators can set the default broker implementation for a cluster. When you create a broker, the default broker implementation is used, unless you provide set configurations in the **Broker** object.

4.2.1. Default broker implementation for development purposes

Knative provides a default, channel-based broker implementation. This channel-based broker can be used for development and testing purposes, but does not provide adequate event delivery guarantees for production environments. The default broker is backed by the **InMemoryChannel** channel implementation by default.

If you want to use Apache Kafka to reduce network hops, use the Knative broker implementation for Apache Kafka. Do not configure the channel-based broker to be backed by the **KafkaChannel** channel implementation.

4.2.2. Production-ready Knative broker implementation for Apache Kafka

For production-ready Knative Eventing deployments, Red Hat recommends using the Knative broker implementation for Apache Kafka. The broker is an Apache Kafka native implementation of the Knative broker, which sends CloudEvents directly to the Kafka instance.

The Knative broker has a native integration with Kafka for storing and routing events. This allows better integration with Kafka for the broker and trigger model over other broker types, and reduces network hops. Other benefits of the Knative broker implementation include:

- At-least-once delivery guarantees
- Ordered delivery of events, based on the CloudEvents partitioning extension

- Control plane high availability
- A horizontally scalable data plane

The Knative broker implementation for Apache Kafka stores incoming CloudEvents as Kafka records, using the binary content mode. This means that all CloudEvent attributes and extensions are mapped as headers on the Kafka record, while the **data** spec of the CloudEvent corresponds to the value of the Kafka record.

4.3. CREATING BROKERS

Knative provides a default, channel-based broker implementation. This channel-based broker can be used for development and testing purposes, but does not provide adequate event delivery guarantees for production environments.

If a cluster administrator has configured your OpenShift Serverless deployment to use Apache Kafka as the default broker type, creating a broker by using the default settings creates a Knative broker for Apache Kafka.

If your OpenShift Serverless deployment is not configured to use the Knative broker for Apache Kafka as the default broker type, the channel-based broker is created when you use the default settings in the following procedures.

4.3.1. Creating a broker by using the Knative CLI

Brokers can be used in combination with triggers to deliver events from an event source to an event sink. Using the Knative (**kn**) CLI to create brokers provides a more streamlined and intuitive user interface over modifying YAML files directly. You can use the **kn broker create** command to create a broker.

Prerequisites

- The OpenShift Serverless Operator and Knative Eventing are installed on your OpenShift Container Platform cluster.
- You have installed the Knative (**kn**) CLI.
- 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

- Create a broker:

```
$ kn broker create <broker_name>
```

Verification

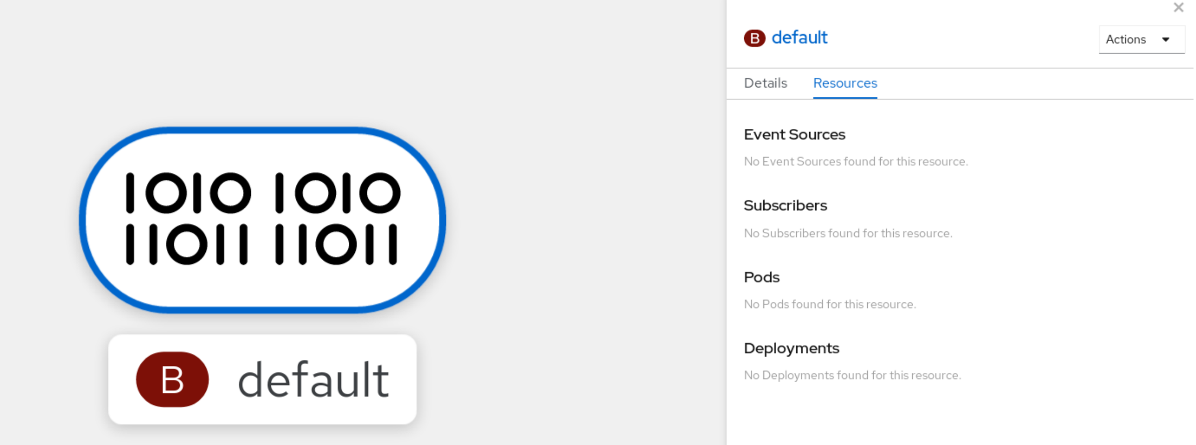
1. Use the **kn** command to list all existing brokers:

```
$ kn broker list
```

Example output

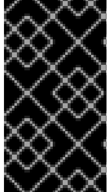
```
NAME      URL                                                          AGE   CONDITIONS   READY
REASON
default   http://broker-ingress.knative-eventing.svc.cluster.local/test/default   45s    5 OK / 5
True
```

2. Optional: If you are using the OpenShift Container Platform web console, you can navigate to the **Topology** view in the **Developer** perspective, and observe that the broker exists:



4.3.2. Creating a broker by annotating a trigger

Brokers can be used in combination with triggers to deliver events from an event source to an event sink. You can create a broker by adding the **eventing.knative.dev/injection: enabled** annotation to a **Trigger** object.



IMPORTANT

If you create a broker by using the **eventing.knative.dev/injection: enabled** annotation, you cannot delete this broker without cluster administrator permissions. If you delete the broker without having a cluster administrator remove this annotation first, the broker is created again after deletion.

Prerequisites

- The OpenShift Serverless Operator and Knative Eventing are installed on your OpenShift Container Platform cluster.
- Install the OpenShift CLI (**oc**).
- 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. Create a **Trigger** object as a YAML file that has the **eventing.knative.dev/injection: enabled** annotation:

```
apiVersion: eventing.knative.dev/v1
kind: Trigger
metadata:
  annotations:
    eventing.knative.dev/injection: enabled
```

```
  name: <trigger_name>
spec:
  broker: default
  subscriber: (1)
    ref:
      apiVersion: serving.knative.dev/v1
      kind: Service
      name: <service_name>
```

1. Specify details about the event sink, or *subscriber*, that the trigger sends events to.

2. Apply the **Trigger** YAML file:

```
$ oc apply -f <filename>
```

Verification

You can verify that the broker has been created successfully by using the **oc** CLI, or by observing it in the **Topology** view in the web console.

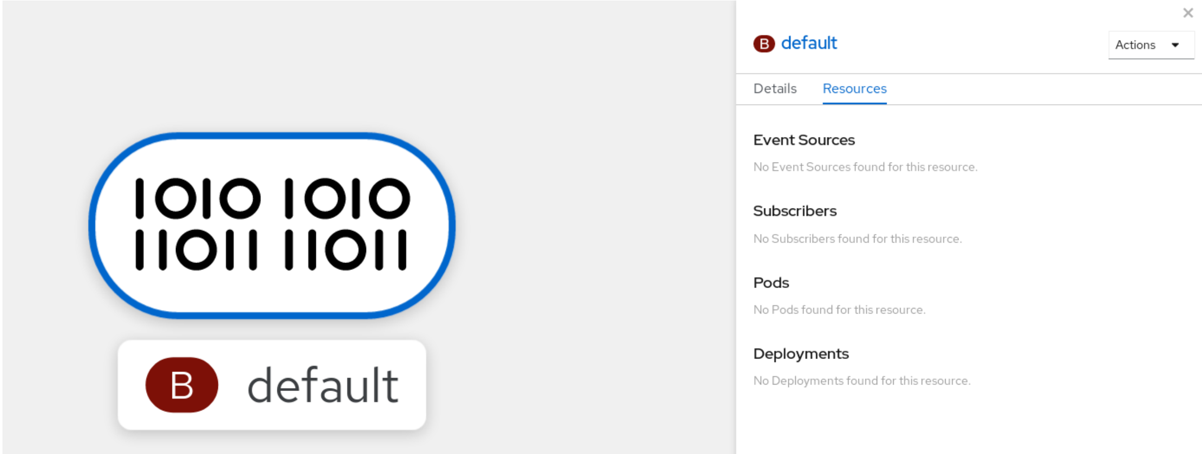
1. Enter the following **oc** command to get the broker:

```
$ oc -n <namespace> get broker default
```

Example output

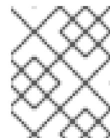
```
NAME      READY     REASON    URL                                                          AGE
default   True                http://broker-ingress.knative-eventing.svc.cluster.local/test/default
3m56s
```

2. Optional: If you are using the OpenShift Container Platform web console, you can navigate to the **Topology** view in the **Developer** perspective, and observe that the broker exists:



4.3.3. Creating a broker by labeling a namespace

Brokers can be used in combination with triggers to deliver events from an event source to an event sink. You can create the **default** broker automatically by labelling a namespace that you own or have write permissions for.



NOTE

Brokers created using this method are not removed if you remove the label. You must manually delete them.

Prerequisites

- The OpenShift Serverless Operator and Knative Eventing are installed on your OpenShift Container Platform cluster.
- Install the OpenShift CLI (**oc**).
- 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.
- You have cluster or dedicated administrator permissions if you are using Red Hat OpenShift Service on AWS or OpenShift Dedicated.

Procedure

- Label a namespace with **eventing.knative.dev/injection=enabled**:

```
$ oc label namespace <namespace> eventing.knative.dev/injection=enabled
```

Verification

You can verify that the broker has been created successfully by using the **oc** CLI, or by observing it in the **Topology** view in the web console.

1. Use the **oc** command to get the broker:

```
$ oc -n <namespace> get broker <broker_name>
```

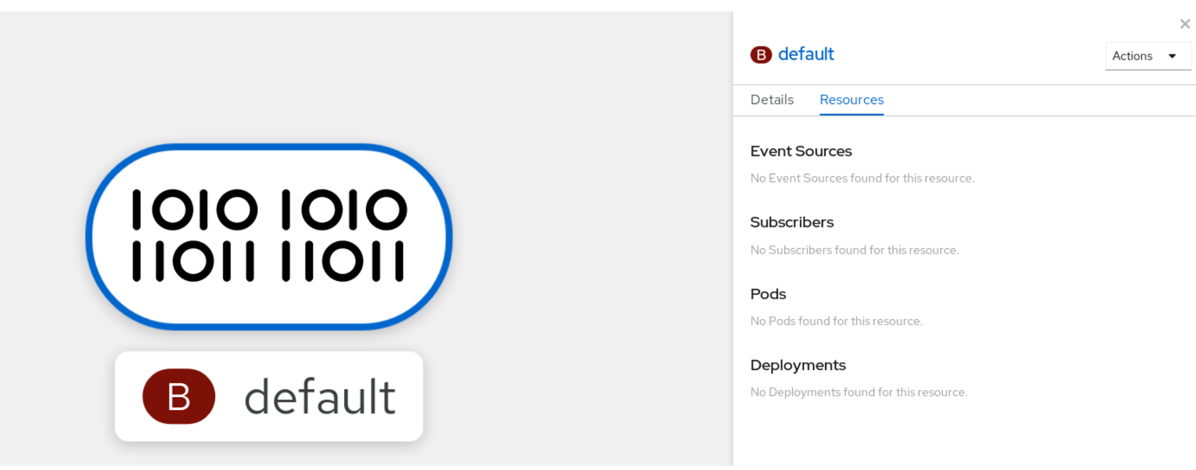
Example command

```
$ oc -n default get broker default
```

Example output

```
NAME      READY     REASON    URL                                                                AGE
default   True                http://broker-ingress.knative-eventing.svc.cluster.local/test/default
3m56s
```

2. Optional: If you are using the OpenShift Container Platform web console, you can navigate to the **Topology** view in the **Developer** perspective, and observe that the broker exists:



4.3.4. Deleting a broker that was created by injection

If you create a broker by injection and later want to delete it, you must delete it manually. Brokers created by using a namespace label or trigger annotation are not deleted permanently if you remove the label or annotation.

Prerequisites

- Install the OpenShift CLI (**oc**).

Procedure

1. Remove the **eventing.knative.dev/injection=enabled** label from the namespace:

```
$ oc label namespace <namespace> eventing.knative.dev/injection-
```

   Removing the annotation prevents Knative from recreating the broker after you delete it.

2. Delete the broker from the selected namespace:

```
$ oc -n <namespace> delete broker <broker_name>
```

Verification

- Use the **oc** command to get the broker:

```
$ oc -n <namespace> get broker <broker_name>
```

  Example command

```
$ oc -n default get broker default
```

  Example output

```
No resources found.
Error from server (NotFound): brokers.eventing.knative.dev "default" not found
```

4.3.5. Creating a broker by using the web console

After Knative Eventing is installed on your cluster, you can create a broker by using the web console. Using the OpenShift Container Platform web console provides a streamlined and intuitive user interface to create a broker.

Prerequisites

- You have logged in to the OpenShift Container Platform web console.
- The OpenShift Serverless Operator, Knative Serving and Knative Eventing are installed on the cluster.
- 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.

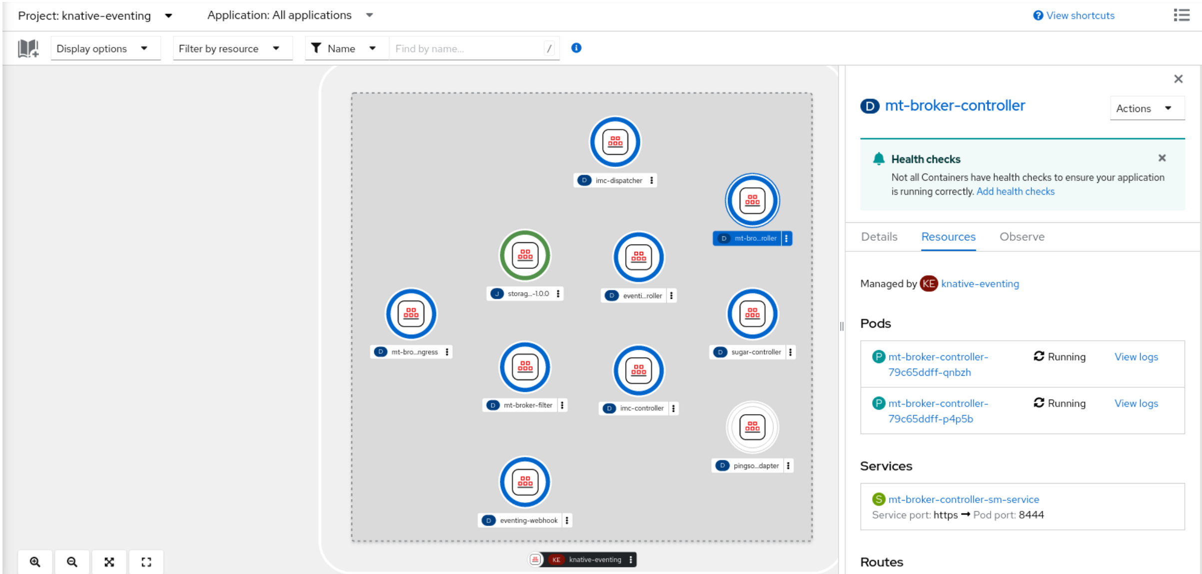
Procedure

1. In the **Developer** perspective, navigate to **+Add → Broker**. The **Broker** page is displayed.
2. Optional. Update the **Name** of the broker. If you do not update the name, the generated broker is named **default**.
3. Click **Create**.

Verification

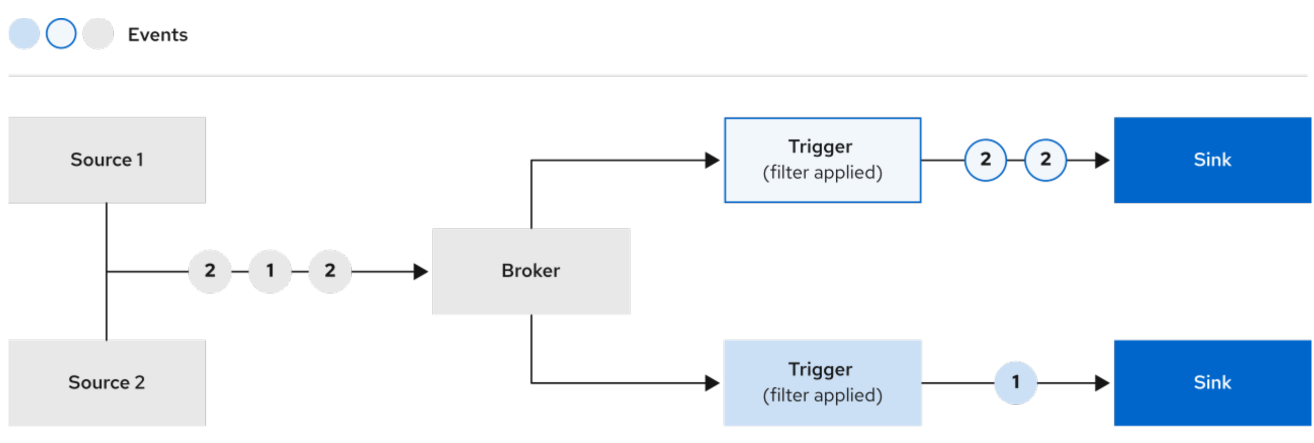
You can verify that the broker was created by viewing broker components in the **Topology** page.

1. In the **Developer** perspective, navigate to **Topology**.
2. View the **mt-broker-ingress**, **mt-broker-filter**, and **mt-broker-controller** components.



4.3.6. Creating a broker by using the Administrator perspective

Brokers can be used in combination with triggers to deliver events from an event source to an event sink. Events are sent from an event source to a broker as an HTTP **POST** request. After events have entered the broker, they can be filtered by CloudEvent attributes using triggers, and sent as an HTTP **POST** request to an event sink.



Prerequisites

- The OpenShift Serverless Operator and Knative Eventing are installed on your OpenShift Container Platform cluster.
- You have logged in to the web console and are in the **Administrator** perspective.
- You have cluster administrator permissions on OpenShift Container Platform, or you have cluster or dedicated administrator permissions on Red Hat OpenShift Service on AWS or OpenShift Dedicated.

Procedure

1. In the **Administrator** perspective of the OpenShift Container Platform web console, navigate to **Serverless** → **Eventing**.
2. In the **Create** list, select **Broker**. You will be directed to the **Create Broker** page.
3. Optional: Modify the YAML configuration for the broker.
4. Click **Create**.

4.3.7. Next steps

- Configure event delivery parameters that are applied in cases where an event fails to be delivered to an event sink.

4.3.8. Additional resources

- Configuring the default broker class
- Triggers
- Connect a broker to a sink using the Developer perspective

4.4. CONFIGURING THE DEFAULT BROKER BACKING CHANNEL

If you are using a channel-based broker, you can set the default backing channel type for the broker to either **InMemoryChannel** or **KafkaChannel**.

Prerequisites

- You have administrator permissions on OpenShift Container Platform.
- You have installed the OpenShift Serverless Operator and Knative Eventing on your cluster.
- You have installed the OpenShift (**oc**) CLI.
- If you want to use Apache Kafka channels as the default backing channel type, you must also install the **KnativeKafka** CR on your cluster.

Procedure

1. Modify the **KnativeEventing** custom resource (CR) to add configuration details for the **config-br-default-channel** config map:

```
apiVersion: operator.knative.dev/v1beta1
kind: KnativeEventing
metadata:
  name: knative-eventing
  namespace: knative-eventing
spec:
  config: 1
    config-br-default-channel:
      channel-template-spec: |
        apiVersion: messaging.knative.dev/v1beta1
        kind: KafkaChannel 2
        spec:
          numPartitions: 6 3
          replicationFactor: 3 4
```

   1. In **spec.config**, you can specify the config maps that you want to add modified configurations for.
   2. The default backing channel type configuration. In this example, the default channel implementation for the cluster is **KafkaChannel**.
   3. The number of partitions for the Kafka channel that backs the broker.
   4. The replication factor for the Kafka channel that backs the broker.

2. Apply the updated **KnativeEventing** CR:

```
$ oc apply -f <filename>
```

4.5. CONFIGURING THE DEFAULT BROKER CLASS

You can use the **config-br-defaults** config map to specify default broker class settings for Knative Eventing. You can specify the default broker class for the entire cluster or for one or more namespaces. Currently the **MTChannelBasedBroker** and **Kafka** broker types are supported.

Prerequisites

- You have administrator permissions on OpenShift Container Platform.

- You have installed the OpenShift Serverless Operator and Knative Eventing on your cluster.
- If you want to use the Knative broker for Apache Kafka as the default broker implementation, you must also install the **KnativeKafka** CR on your cluster.

Procedure

- Modify the **KnativeEventing** custom resource to add configuration details for the **config-br-defaults** config map:

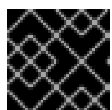
```
apiVersion: operator.knative.dev/v1beta1
kind: KnativeEventing
metadata:
  name: knative-eventing
  namespace: knative-eventing
spec:
  defaultBrokerClass: Kafka (1)
  config: (2)
    config-br-defaults: (3)
      default-br-config: |
        clusterDefault: (4)
          brokerClass: Kafka
          apiVersion: v1
          kind: ConfigMap
          name: kafka-broker-config (5)
          namespace: knative-eventing (6)
        namespaceDefaults: (7)
          my-namespace:
            brokerClass: MTChannelBasedBroker
            apiVersion: v1
            kind: ConfigMap
            name: config-br-default-channel (8)
            namespace: knative-eventing (9)
...
```

1. The default broker class for Knative Eventing.
2. In **spec.config**, you can specify the config maps that you want to add modified configurations for.
3. The **config-br-defaults** config map specifies the default settings for any broker that does not specify **spec.config** settings or a broker class.
4. The cluster-wide default broker class configuration. In this example, the default broker class implementation for the cluster is **Kafka**.
5. The **kafka-broker-config** config map specifies default settings for the Kafka broker. See "Configuring Knative broker for Apache Kafka settings" in the "Additional resources" section.
6. The namespace where the **kafka-broker-config** config map exists.
7. The namespace-scoped default broker class configuration. In this example, the default broker class implementation for the **my-namespace** namespace is **MTChannelBasedBroker**. You can specify default broker class implementations for

multiple namespaces.

8. The **config-br-default-channel** config map specifies the default backing channel for the broker. See "Configuring the default broker backing channel" in the "Additional resources" section.

9. The namespace where the **config-br-default-channel** config map exists.



IMPORTANT

Configuring a namespace-specific default overrides any cluster-wide settings.

4.6. KNATIVE BROKER IMPLEMENTATION FOR APACHE KAFKA

For production-ready Knative Eventing deployments, Red Hat recommends using the Knative broker implementation for Apache Kafka. The broker is an Apache Kafka native implementation of the Knative broker, which sends CloudEvents directly to the Kafka instance.

The Knative broker has a native integration with Kafka for storing and routing events. This allows better integration with Kafka for the broker and trigger model over other broker types, and reduces network hops. Other benefits of the Knative broker implementation include:

- At-least-once delivery guarantees
- Ordered delivery of events, based on the CloudEvents partitioning extension
- Control plane high availability
- A horizontally scalable data plane

The Knative broker implementation for Apache Kafka stores incoming CloudEvents as Kafka records, using the binary content mode. This means that all CloudEvent attributes and extensions are mapped as headers on the Kafka record, while the **data** spec of the CloudEvent corresponds to the value of the Kafka record.

4.6.1. Creating an Apache Kafka broker when it is not configured as the default broker type

If your OpenShift Serverless deployment is not configured to use Kafka broker as the default broker type, you can use one of the following procedures to create a Kafka-based broker.

4.6.1.1. Creating an Apache Kafka broker by using YAML

Creating Knative resources by using YAML files uses a declarative API, which enables you to describe applications declaratively and in a reproducible manner. To create a Kafka broker by using YAML, you must create a YAML file that defines a **Broker** object, then apply it by using the **oc apply** command.

Prerequisites

- The OpenShift Serverless Operator, Knative Eventing, and the **KnativeKafka** custom resource are installed on your OpenShift Container Platform cluster.
- 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.

- You have installed the OpenShift CLI (**oc**).

Procedure

1. Create a Kafka-based broker as a YAML file:

```
apiVersion: eventing.knative.dev/v1
kind: Broker
metadata:
  annotations:
    eventing.knative.dev/broker.class: Kafka (1)
  name: example-kafka-broker
spec:
  config:
    apiVersion: v1
    kind: ConfigMap
    name: kafka-broker-config (2)
    namespace: knative-eventing
```

(1) The broker class. If not specified, brokers use the default class as configured by cluster administrators. To use the Kafka broker, this value must be **Kafka**.

(2) The default config map for Knative brokers for Apache Kafka. This config map is created when the Kafka broker functionality is enabled on the cluster by a cluster administrator.

2. Apply the Kafka-based broker YAML file:

```
$ oc apply -f <filename>
```

4.6.1.2. Creating an Apache Kafka broker that uses an externally managed Kafka topic

If you want to use a Kafka broker without allowing it to create its own internal topic, you can use an externally managed Kafka topic instead. To do this, you must create a Kafka **Broker** object that uses the **kafka.eventing.knative.dev/external.topic** annotation.

Prerequisites

- The OpenShift Serverless Operator, Knative Eventing, and the **KnativeKafka** custom resource are installed on your OpenShift Container Platform cluster.
- You have access to a Kafka instance such as Red Hat AMQ Streams , and have created a Kafka topic.
- 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.
- You have installed the OpenShift CLI (**oc**).

Procedure

1. Create a Kafka-based broker as a YAML file:

```
apiVersion: eventing.knative.dev/v1
```

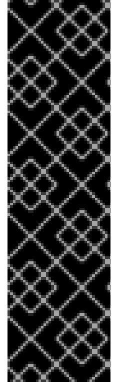
```
kind: Broker
metadata:
  annotations:
    eventing.knative.dev/broker.class: Kafka (1)
    kafka.eventing.knative.dev/external.topic: <topic_name> (2)
...
```

1. The broker class. If not specified, brokers use the default class as configured by cluster administrators. To use the Kafka broker, this value must be **Kafka**.
2. The name of the Kafka topic that you want to use.

2. Apply the Kafka-based broker YAML file:

```
$ oc apply -f <filename>
```

4.6.1.3. Knative Broker implementation for Apache Kafka with isolated data plane



IMPORTANT

The Knative Broker implementation for Apache Kafka with isolated data plane is a Technology Preview feature only. Technology Preview features are not supported with Red Hat production service level agreements (SLAs) and might not be functionally complete. Red Hat does not recommend using them in production. These features provide early access to upcoming product features, enabling customers to test functionality and provide feedback during the development process.

For more information about the support scope of Red Hat Technology Preview features, see Technology Preview Features Support Scope .

The Knative Broker implementation for Apache Kafka has 2 planes:

Control plane

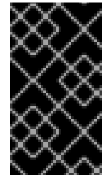
Consists of controllers that talk to the Kubernetes API, watch for custom objects, and manage the data plane.

Data plane

The collection of components that listen for incoming events, talk to Apache Kafka, and send events to the event sinks. The Knative Broker implementation for Apache Kafka data plane is where events flow. The implementation consists of **kafka-broker-receiver** and **kafka-broker-dispatcher** deployments.

When you configure a Broker class of **Kafka**, the Knative Broker implementation for Apache Kafka uses a shared data plane. This means that the **kafka-broker-receiver** and **kafka-broker-dispatcher** deployments in the **knative-eventing** namespace are used for all Apache Kafka Brokers in the cluster.

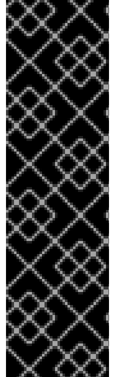
However, when you configure a Broker class of **KafkaNamespaced**, the Apache Kafka broker controller creates a new data plane for each namespace where a broker exists. This data plane is used by all **KafkaNamespaced** brokers in that namespace. This provides isolation between the data planes, so that the **kafka-broker-receiver** and **kafka-broker-dispatcher** deployments in the user namespace are only used for the broker in that namespace.



IMPORTANT

As a consequence of having separate data planes, this security feature creates more deployments and uses more resources. Unless you have such isolation requirements, use a **regular** Broker with a class of **Kafka**.

4.6.1.4. Creating a Knative broker for Apache Kafka that uses an isolated data plane



IMPORTANT

The Knative Broker implementation for Apache Kafka with isolated data plane is a Technology Preview feature only. Technology Preview features are not supported with Red Hat production service level agreements (SLAs) and might not be functionally complete. Red Hat does not recommend using them in production. These features provide early access to upcoming product features, enabling customers to test functionality and provide feedback during the development process.

For more information about the support scope of Red Hat Technology Preview features, see Technology Preview Features Support Scope.

To create a **KafkaNamespaced** broker, you must set the **eventing.knative.dev/broker.class** annotation to **KafkaNamespaced**.

Prerequisites

- The OpenShift Serverless Operator, Knative Eventing, and the **KnativeKafka** custom resource are installed on your OpenShift Container Platform cluster.
- You have access to an Apache Kafka instance, such as Red Hat AMQ Streams, and have created a Kafka topic.
- 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.
- You have installed the OpenShift CLI (**oc**).

Procedure

1. Create an Apache Kafka-based broker by using a YAML file:

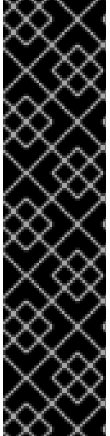
```
apiVersion: eventing.knative.dev/v1
kind: Broker
metadata:
  annotations:
    eventing.knative.dev/broker.class: KafkaNamespaced (1)
  name: default
  namespace: my-namespace (2)
spec:
  config:
    apiVersion: v1
    kind: ConfigMap
    name: my-config (3)
...
```

1 To use the Apache Kafka broker with isolated data planes, the broker class value must be **KafkaNamespaced**.

2 3 The referenced **ConfigMap** object **my-config** must be in the same namespace as the **Broker** object, in this case **my-namespace**.

2. Apply the Apache Kafka-based broker YAML file:

```
$ oc apply -f <filename>
```



IMPORTANT

The **ConfigMap** object in **spec.config** must be in the same namespace as the **Broker** object:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: my-config
  namespace: my-namespace
data:
  ...
```

After the creation of the first **Broker** object with the **KafkaNamespaced** class, the **kafka-broker-receiver** and **kafka-broker-dispatcher** deployments are created in the namespace. Subsequently, all brokers with the **KafkaNamespaced** class in the same namespace will use the same data plane. If no brokers with the **KafkaNamespaced** class exist in the namespace, the data plane in the namespace is deleted.

4.6.2. Configuring Apache Kafka broker settings

You can configure the replication factor, bootstrap servers, and the number of topic partitions for a Kafka broker, by creating a config map and referencing this config map in the Kafka **Broker** object.

Knative Eventing supports the full set of topic config options that Kafka supports. To set these options, you must add a key to the ConfigMap with the **default.topic.config.** prefix.

Prerequisites

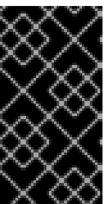
- You have cluster or dedicated administrator permissions on OpenShift Container Platform.
- The OpenShift Serverless Operator, Knative Eventing, and the **KnativeKafka** custom resource (CR) are installed on your OpenShift Container Platform cluster.
- You have created a project or have access to a project that has the appropriate roles and permissions to create applications and other workloads in OpenShift Container Platform.
- You have installed the OpenShift CLI (**oc**).

Procedure

1. Modify the **kafka-broker-config** config map, or create your own config map that contains the following configuration:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: <config_map_name> (1)
  namespace: <namespace> (2)
data:
  default.topic.partitions: <integer> (3)
  default.topic.replication.factor: <integer> (4)
  bootstrap.servers: <list_of_servers> (5)
  default.topic.config.<config_option>: <value> (6)
```

1. The config map name.
2. The namespace where the config map exists.
3. The number of topic partitions for the Kafka broker. This controls how quickly events can be sent to the broker. A higher number of partitions requires greater compute resources.
4. The replication factor of topic messages. This prevents against data loss. A higher replication factor requires greater compute resources and more storage.
5. A comma separated list of bootstrap servers. This can be inside or outside of the OpenShift Container Platform cluster, and is a list of Kafka clusters that the broker receives events from and sends events to.
6. A topic config option. For more information, see the full set of possible options and values .



IMPORTANT

The **default.topic.replication.factor** value must be less than or equal to the number of Kafka broker instances in your cluster. For example, if you only have one Kafka broker, the **default.topic.replication.factor** value should not be more than **"1"**.

Example Kafka broker config map

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: kafka-broker-config
  namespace: knative-eventing
data:
  default.topic.partitions: "10"
  default.topic.replication.factor: "3"
  bootstrap.servers: "my-cluster-kafka-bootstrap.kafka:9092"
  default.topic.config.retention.ms: "3600"
```

2. Apply the config map:

```
$ oc apply -f <config_map_filename>
```

3. Specify the config map for the Kafka **Broker** object:

Example Broker object

```
apiVersion: eventing.knative.dev/v1
kind: Broker
metadata:
  name: <broker_name> 1
  namespace: <namespace> 2
  annotations:
    eventing.knative.dev/broker.class: Kafka 3
spec:
  config:
    apiVersion: v1
    kind: ConfigMap
    name: <config_map_name> 4
    namespace: <namespace> 5
...
```

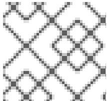
1. The broker name.
2. The namespace where the broker exists.
3. The broker class annotation. In this example, the broker is a Kafka broker that uses the class value **Kafka**.
4. The config map name.
5. The namespace where the config map exists.

4. Apply the broker:

```
$ oc apply -f <broker_filename>
```

4.6.3. Security configuration for the Knative broker implementation for Apache Kafka

Kafka clusters are generally secured by using the TLS or SASL authentication methods. You can configure a Kafka broker or channel to work against a protected Red Hat AMQ Streams cluster by using TLS or SASL.



NOTE

Red Hat recommends that you enable both SASL and TLS together.

4.6.3.1. Configuring TLS authentication for Apache Kafka brokers

Transport Layer Security (TLS) is used by Apache Kafka clients and servers to encrypt traffic between Knative and Kafka, as well as for authentication. TLS is the only supported method of traffic encryption for the Knative broker implementation for Apache Kafka.

Prerequisites

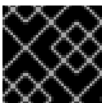
- You have cluster or dedicated administrator permissions on OpenShift Container Platform.

- The OpenShift Serverless Operator, Knative Eventing, and the **KnativeKafka** CR are installed on your OpenShift Container Platform cluster.
- 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.
- You have a Kafka cluster CA certificate stored as a **.pem** file.
- You have a Kafka cluster client certificate and a key stored as **.pem** files.
- Install the OpenShift CLI (**oc**).

Procedure

1. Create the certificate files as a secret in the **knative-eventing** namespace:

```
$ oc create secret -n knative-eventing generic <secret_name> \
  --from-literal=protocol=SSL \
  --from-file=ca.crt=caroot.pem \
  --from-file=user.crt=certificate.pem \
  --from-file=user.key=key.pem
```



IMPORTANT

Use the key names **ca.crt**, **user.crt**, and **user.key**. Do not change them.

2. Edit the **KnativeKafka** CR and add a reference to your secret in the **broker** spec:

```
apiVersion: operator.serverless.openshift.io/v1alpha1
kind: KnativeKafka
metadata:
  namespace: knative-eventing
  name: knative-kafka
spec:
  broker:
    enabled: true
    defaultConfig:
      authSecretName: <secret_name>
...
```

4.6.3.2. Configuring SASL authentication for Apache Kafka brokers

Simple Authentication and Security Layer (SASL) is used by Apache Kafka for authentication. If you use SASL authentication on your cluster, users must provide credentials to Knative for communicating with the Kafka cluster; otherwise events cannot be produced or consumed.

Prerequisites

- You have cluster or dedicated administrator permissions on OpenShift Container Platform.
- The OpenShift Serverless Operator, Knative Eventing, and the **KnativeKafka** CR are installed on your OpenShift Container Platform cluster.

- 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.
- You have a username and password for a Kafka cluster.
- You have chosen the SASL mechanism to use, for example, **PLAIN**, **SCRAM-SHA-256**, or **SCRAM-SHA-512**.
- If TLS is enabled, you also need the **ca.crt** certificate file for the Kafka cluster.
- Install the OpenShift CLI (**oc**).

Procedure

1. Create the certificate files as a secret in the **knative-eventing** namespace:

```
$ oc create secret -n knative-eventing generic <secret_name> \
  --from-literal=protocol=SASL_SSL \
  --from-literal=sasl.mechanism=<sasl_mechanism> \
  --from-file=ca.crt=caroot.pem \
  --from-literal=password="SecretPassword" \
  --from-literal=user="my-sasl-user"
```

   - Use the key names **ca.crt**, **password**, and **sasl.mechanism**. Do not change them.
   - If you want to use SASL with public CA certificates, you must use the **tls.enabled=true** flag, rather than the **ca.crt** argument, when creating the secret. For example:

```
$ oc create secret -n <namespace> generic <kafka_auth_secret> \
  --from-literal=tls.enabled=true \
  --from-literal=password="SecretPassword" \
  --from-literal=saslType="SCRAM-SHA-512" \
  --from-literal=user="my-sasl-user"
```

2. Edit the **KnativeKafka** CR and add a reference to your secret in the **broker** spec:

```
apiVersion: operator.serverless.openshift.io/v1alpha1
kind: KnativeKafka
metadata:
  namespace: knative-eventing
  name: knative-kafka
spec:
  broker:
    enabled: true
    defaultConfig:
      authSecretName: <secret_name>
...
```

4.6.4. Additional resources

- Red Hat AMQ Streams documentation
- TLS and SASL on Kafka

4.7. MANAGING BROKERS

After you have created a broker, you can manage your broker by using Knative (**kn**) CLI commands, or by modifying it in the OpenShift Container Platform web console.

4.7.1. Managing brokers using the CLI

The Knative (**kn**) CLI provides commands that can be used to describe and list existing brokers.

4.7.1.1. Listing existing brokers by using the Knative CLI

Using the Knative (**kn**) CLI to list brokers provides a streamlined and intuitive user interface. You can use the **kn broker list** command to list existing brokers in your cluster by using the Knative CLI.

Prerequisites

- The OpenShift Serverless Operator and Knative Eventing are installed on your OpenShift Container Platform cluster.
- You have installed the Knative (**kn**) CLI.

Procedure

- List all existing brokers:

```
$ kn broker list
```

Example output

```
NAME      URL                                                                    AGE   CONDITIONS   READY
REASON
default   http://broker-ingress.knative-eventing.svc.cluster.local/test/default   45s   5 OK / 5
True
```

4.7.1.2. Describing an existing broker by using the Knative CLI

Using the Knative (**kn**) CLI to describe brokers provides a streamlined and intuitive user interface. You can use the **kn broker describe** command to print information about existing brokers in your cluster by using the Knative CLI.

Prerequisites

- The OpenShift Serverless Operator and Knative Eventing are installed on your OpenShift Container Platform cluster.
- You have installed the Knative (**kn**) CLI.

Procedure

- Describe an existing broker:

```
$ kn broker describe <broker_name>
```

Example command using default broker

```
$ kn broker describe default
```

Example output

```
Name:         default
Namespace:    default
Annotations:  eventing.knative.dev/broker.class=MTChannelBasedBroker,
eventing.knative.dev/creato ...
Age:          22s

Address:
  URL:    http://broker-ingress.knative-eventing.svc.cluster.local/default/default

Conditions:
  OK TYPE                   AGE REASON
  ++ Ready                  22s
  ++ Addressable            22s
  ++ FilterReady            22s
  ++ IngressReady           22s
  ++ TriggerChannelReady    22s
```

4.7.2. Connect a broker to a sink using the Developer perspective

You can connect a broker to an event sink in the OpenShift Container Platform **Developer** perspective by creating a trigger.

Prerequisites

- The OpenShift Serverless Operator, Knative Serving, and Knative Eventing are installed on your OpenShift Container Platform cluster.
- You have logged in to the web console and are in the **Developer** perspective.
- 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.
- You have created a sink, such as a Knative service or channel.
- You have created a broker.

Procedure

1. In the **Topology** view, point to the broker that you have created. An arrow appears. Drag the arrow to the sink that you want to connect to the broker. This action opens the **Add Trigger** dialog box.
2. In the **Add Trigger** dialog box, enter a name for the trigger and click **Add**.

Verification

You can verify that the broker is connected to the sink by viewing the **Topology** page.

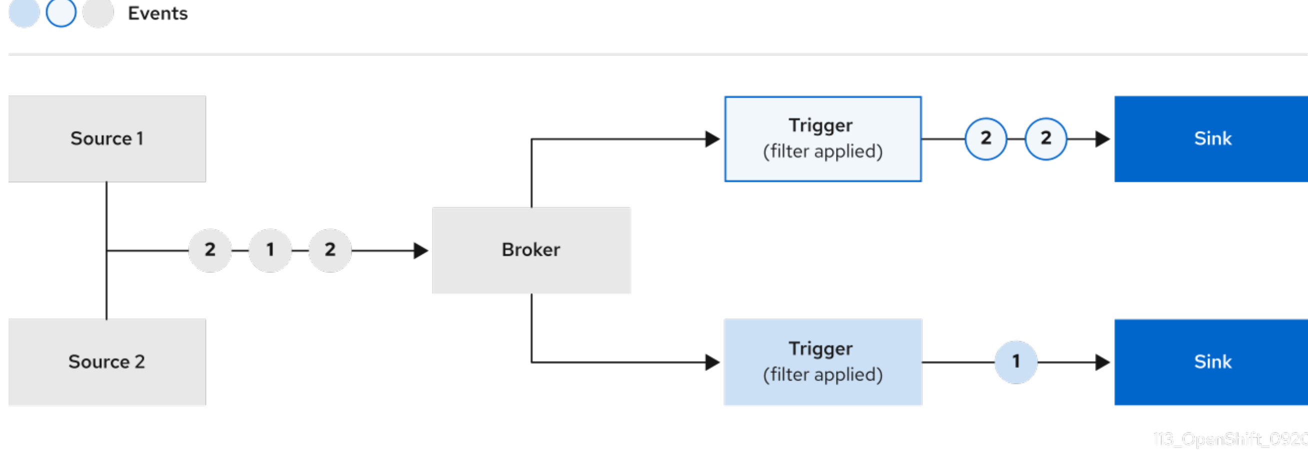
1. In the **Developer** perspective, navigate to **Topology**.

2. Click the line that connects the broker to the sink to see details about the trigger in the **Details** panel.

CHAPTER 5. TRIGGERS

5.1. TRIGGERS OVERVIEW

Brokers can be used in combination with triggers to deliver events from an event source to an event sink. Events are sent from an event source to a broker as an HTTP **POST** request. After events have entered the broker, they can be filtered by CloudEvent attributes using triggers, and sent as an HTTP **POST** request to an event sink.



If you are using a Knative broker for Apache Kafka, you can configure the delivery order of events from triggers to event sinks. See Configuring event delivery ordering for triggers.

5.1.1. Configuring event delivery ordering for triggers

If you are using a Kafka broker, you can configure the delivery order of events from triggers to event sinks.

Prerequisites

- The OpenShift Serverless Operator, Knative Eventing, and Knative Kafka are installed on your OpenShift Container Platform cluster.
- Kafka broker is enabled for use on your cluster, and you have created a Kafka broker.
- 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.
- You have installed the OpenShift (**oc**) CLI.

Procedure

1. Create or modify a **Trigger** object and set the **kafka.eventing.knative.dev/delivery.order** annotation:

```
apiVersion: eventing.knative.dev/v1
kind: Trigger
metadata:
  name: <trigger_name>
```

```
annotations:
  kafka.eventing.knative.dev/delivery.order: ordered
# ...
```

The supported consumer delivery guarantees are:

unordered

An unordered consumer is a non-blocking consumer that delivers messages unordered, while preserving proper offset management.

ordered

An ordered consumer is a per-partition blocking consumer that waits for a successful response from the CloudEvent subscriber before it delivers the next message of the partition.
The default ordering guarantee is **unordered**.

2. Apply the **Trigger** object:

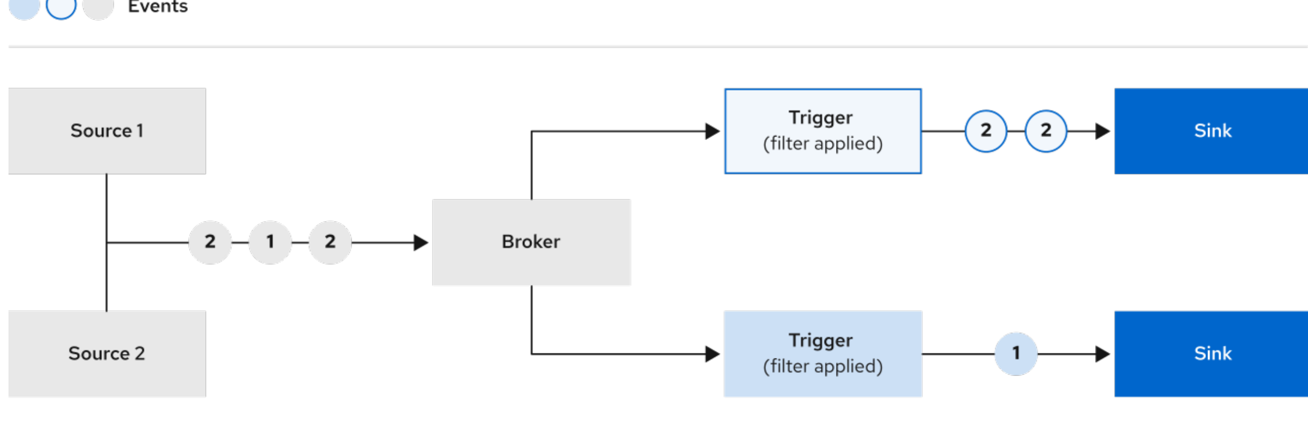
```
$ oc apply -f <filename>
```

5.1.2. Next steps

- Configure event delivery parameters that are applied in cases where an event fails to be delivered to an event sink.

5.2. CREATING TRIGGERS

Brokers can be used in combination with triggers to deliver events from an event source to an event sink. Events are sent from an event source to a broker as an HTTP **POST** request. After events have entered the broker, they can be filtered by CloudEvent attributes using triggers, and sent as an HTTP **POST** request to an event sink.



5.2.1. Creating a trigger by using the Administrator perspective

Using the OpenShift Container Platform web console provides a streamlined and intuitive user interface to create a trigger. After Knative Eventing is installed on your cluster and you have created a broker, you can create a trigger by using the web console.

Prerequisites

- The OpenShift Serverless Operator and Knative Eventing are installed on your OpenShift Container Platform cluster.
- You have logged in to the web console and are in the **Administrator** perspective.
- You have cluster administrator permissions on OpenShift Container Platform, or you have cluster or dedicated administrator permissions on Red Hat OpenShift Service on AWS or OpenShift Dedicated.
- You have created a Knative broker.
- You have created a Knative service to use as a subscriber.

Procedure

1. In the **Administrator** perspective of the OpenShift Container Platform web console, navigate to **Serverless** → **Eventing**.
2. In the **Broker** tab, select the Options menu ⋮ for the broker that you want to add a trigger to.
3. Click **Add Trigger** in the list.
4. In the **Add Trigger** dialogue box, select a **Subscriber** for the trigger. The subscriber is the Knative service that will receive events from the broker.
5. Click **Add**.

5.2.2. Creating a trigger by using the Developer perspective

Using the OpenShift Container Platform web console provides a streamlined and intuitive user interface to create a trigger. After Knative Eventing is installed on your cluster and you have created a broker, you can create a trigger by using the web console.

Prerequisites

- The OpenShift Serverless Operator, Knative Serving, and Knative Eventing are installed on your OpenShift Container Platform cluster.
- You have logged in to the web console.
- 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.
- You have created a broker and a Knative service or other event sink to connect to the trigger.

Procedure

1. In the **Developer** perspective, navigate to the **Topology** page.
2. Hover over the broker that you want to create a trigger for, and drag the arrow. The **Add Trigger** option is displayed.

3. Click **Add Trigger**.

4. Select your sink in the **Subscriber** list.

5. Click **Add**.

Verification

- After the subscription has been created, you can view it in the **Topology** page, where it is represented as a line that connects the broker to the event sink.

Deleting a trigger

1. In the **Developer** perspective, navigate to the **Topology** page.

2. Click on the trigger that you want to delete.

3. In the **Actions** context menu, select **Delete Trigger**.

5.2.3. Creating a trigger by using the Knative CLI

You can use the **kn trigger create** command to create a trigger.

Prerequisites

- The OpenShift Serverless Operator and Knative Eventing are installed on your OpenShift Container Platform cluster.

- You have installed the Knative (**kn**) CLI.

- 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

- Create a trigger:

  ```
  $ kn trigger create <trigger_name> --broker <broker_name> --filter <key=value> --sink <sink_name>
  ```

  Alternatively, you can create a trigger and simultaneously create the **default** broker using broker injection:

  ```
  $ kn trigger create <trigger_name> --inject-broker --filter <key=value> --sink <sink_name>
  ```

  By default, triggers forward all events sent to a broker to sinks that are subscribed to that broker. Using the **--filter** attribute for triggers allows you to filter events from a broker, so that subscribers will only receive a subset of events based on your defined criteria.

5.3. LIST TRIGGERS FROM THE COMMAND LINE

Using the Knative (**kn**) CLI to list triggers provides a streamlined and intuitive user interface.

5.3.1. Listing triggers by using the Knative CLI

You can use the **kn trigger list** command to list existing triggers in your cluster.

Prerequisites

- The OpenShift Serverless Operator and Knative Eventing are installed on your OpenShift Container Platform cluster.
- You have installed the Knative (**kn**) CLI.

Procedure

1. Print a list of available triggers:

```
$ kn trigger list
```

Example output

```
NAME    BROKER    SINK           AGE   CONDITIONS   READY   REASON
email   default   ksvc:edisplay   4s    5 OK / 5     True
ping    default   ksvc:edisplay   32s   5 OK / 5     True
```

2. Optional: Print a list of triggers in JSON format:

```
$ kn trigger list -o json
```

5.4. DESCRIBE TRIGGERS FROM THE COMMAND LINE

Using the Knative (**kn**) CLI to describe triggers provides a streamlined and intuitive user interface.

5.4.1. Describing a trigger by using the Knative CLI

You can use the **kn trigger describe** command to print information about existing triggers in your cluster by using the Knative CLI.

Prerequisites

- The OpenShift Serverless Operator and Knative Eventing are installed on your OpenShift Container Platform cluster.
- You have installed the Knative (**kn**) CLI.
- You have created a trigger.

Procedure

- Enter the command:

```
$ kn trigger describe <trigger_name>
```

Example output

```
Name:         ping
Namespace:    default
Labels:       eventing.knative.dev/broker=default
Annotations:  eventing.knative.dev/creator=kube:admin,
eventing.knative.dev/lastModifier=kube:admin
Age:          2m
Broker:       default
Filter:
  type:       dev.knative.event

Sink:
  Name:       edisplay
  Namespace:  default
  Resource:   Service (serving.knative.dev/v1)

Conditions:
  OK TYPE                  AGE REASON
  ++ Ready                  2m
  ++ BrokerReady            2m
  ++ DependencyReady        2m
  ++ Subscribed             2m
  ++ SubscriberResolved     2m
```

5.5. CONNECTING A TRIGGER TO A SINK

You can connect a trigger to a sink, so that events from a broker are filtered before they are sent to the sink. A sink that is connected to a trigger is configured as a **subscriber** in the **Trigger** object's resource spec.

Example of a Trigger object connected to an Apache Kafka sink

```
apiVersion: eventing.knative.dev/v1
kind: Trigger
metadata:
  name: <trigger_name> (1)
spec:
...
  subscriber:
    ref:
      apiVersion: eventing.knative.dev/v1alpha1
      kind: KafkaSink
      name: <kafka_sink_name> (2)
```

1. The name of the trigger being connected to the sink.
2. The name of a **KafkaSink** object.

5.6. FILTERING TRIGGERS FROM THE COMMAND LINE

Using the Knative (**kn**) CLI to filter events by using triggers provides a streamlined and intuitive user interface. You can use the **kn trigger create** command, along with the appropriate flags, to filter events by using triggers.

5.6.1. Filtering events with triggers by using the Knative CLI

In the following trigger example, only events with the attribute **type: dev.knative.samples.helloworld** are sent to the event sink:

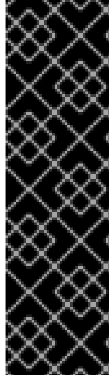
```
$ kn trigger create <trigger_name> --broker <broker_name> --filter
type=dev.knative.samples.helloworld --sink ksvc:<service_name>
```

You can also filter events by using multiple attributes. The following example shows how to filter events using the type, source, and extension attributes:

```
$ kn trigger create <trigger_name> --broker <broker_name> --sink ksvc:<service_name> \
--filter type=dev.knative.samples.helloworld \
--filter source=dev.knative.samples/helloworldsource \
--filter myextension=my-extension-value
```

5.7. ADVANCED TRIGGER FILTERS

The advanced trigger filters give you advanced options for more precise event routing. You can filter events by exact matches, prefixes, or suffixes, as well as by CloudEvent extensions. This added control makes it easier to fine-tune how events flow ensuring that only relevant events trigger specific actions.



IMPORTANT

Advanced trigger filters feature is a Technology Preview feature only. Technology Preview features are not supported with Red Hat production service level agreements (SLAs) and might not be functionally complete. Red Hat does not recommend using them in production. These features provide early access to upcoming product features, enabling customers to test functionality and provide feedback during the development process.

For more information about the support scope of Red Hat Technology Preview features, see Technology Preview Features Support Scope .

5.7.1. Advanced trigger filters overview

The advanced trigger filters feature adds a new **filters** field to triggers that aligns with the filters API field defined in the **CloudEvents Subscriptions** API. You can specify filter expressions, where each expression evaluates to **true** or **false** for each event.

The following example shows a trigger using the advanced filters field:

```
apiVersion: eventing.knative.dev/v1
kind: Trigger
metadata:
  name: my-service-trigger
spec:
  broker: default
  filters:
    - cesql: "source LIKE '%commerce%' AND type IN ('order.created', 'order.updated',
'order.canceled')"
  subscriber:
    ref:
```

```
apiVersion: serving.knative.dev/v1
kind: Service
name: my-service
```

The **filters** field contains an array of filter expressions, each evaluating to either **true** or **false**. If any expression evaluates to **false**, the event is not sent to the subscriber. Each filter expression uses a specific dialect that determines the type of filter and the set of allowed additional properties within the expression.

5.7.2. Supported filter dialects

You can use dialects to define flexible filter expressions to target specific events.

The advanced trigger filters support the following dialects that offer different ways to match and filter events:

- **exact**
- **prefix**
- **suffix**
- **all**
- **any**
- **not**
- **cesql**

Each dialect provides a different method for filtering events based on a specific criteria, enabling precise event selection for processing.

5.7.2.1. exact filter dialect

The **exact** dialect filters events by comparing a string value of the CloudEvent attribute to exactly match the specified string. The comparison is case sensitive. If the attribute is not a string, the filter converts the attribute to its string representation before comparing it to the specified value.

Example of the exact filter dialect

```
apiVersion: eventing.knative.dev/v1
kind: Trigger
metadata:
  ...
spec:
  ...
  filters:
    - exact:
        type: com.github.push
```

5.7.2.2. prefix filter dialect

The **prefix** dialect filters events by comparing a string value of the CloudEvent attribute that starts with the specified string. This comparison is case sensitive. If the attribute is not a string, the filter converts the attribute to its string representation before matching it against the specified value.

Example of the prefix filter dialect

```
apiVersion: eventing.knative.dev/v1
kind: Trigger
metadata:
  ...
spec:
  ...
  filters:
    - prefix:
        type: com.github.
```

5.7.2.3. suffix filter dialect

The **suffix** dialect filters events by comparing a string value of the CloudEvent attribute that ends with the specified string. This comparison is case-sensitive. If the attribute is not a string, the filter converts the attribute to its string representation before matching it to the specified value.

Example of the suffix filter dialect

```
apiVersion: eventing.knative.dev/v1
kind: Trigger
metadata:
  ...
spec:
  ...
  filters:
    - suffix:
        type: .created
```

5.7.2.4. all filter dialect

The **all** filter dialect needs that all nested filter expressions evaluate to **true** to process the event. If any of the nested expressions return **false**, the event is not sent to the subscriber.

Example of the all filter dialect

```
apiVersion: eventing.knative.dev/v1
kind: Trigger
metadata:
  ...
spec:
  ...
  filters:
    - all:
        - exact:
            type: com.github.push
        - exact:
            subject: https://github.com/cloudevents/spec
```

5.7.2.5. **any** filter dialect

The **any** filter dialect requires at least one of the nested filter expressions to evaluate to **true**. If none of the nested expressions return **true**, the event is not sent to the subscriber.

Example of the any filter dialect

```
apiVersion: eventing.knative.dev/v1
kind: Trigger
metadata:
  ...
spec:
  ...
  filters:
    - any:
        - exact:
            type: com.github.push
        - exact:
            subject: https://github.com/cloudevents/spec
```

5.7.2.6. **not** filter dialect

The **not** filter dialect requires that the nested filter expression evaluates to **false** for the event to be processed. If the nested expression evaluates to **true**, the event is not sent to the subscriber.

Example of the not filter dialect

```
apiVersion: eventing.knative.dev/v1
kind: Trigger
metadata:
  ...
spec:
  ...
  filters:
    - not:
        exact:
          type: com.github.push
```

5.7.2.7. **cesql** filter dialect

CloudEvents SQL expressions (cesql) allow computing values and matching of CloudEvent attributes against complex expressions that lean on the syntax of Structured Query Language (SQL) **WHERE** clauses.

The **cesql** filter dialect uses CloudEvents SQL expressions to filter events. The provided CESQL expression must evaluate to **true** for the event to be processed.

Example of the cesql filter dialect

```
apiVersion: eventing.knative.dev/v1
kind: Trigger
metadata:
  ...
spec:
```

```
...
filters:
  - cesql: "source LIKE '%commerce%' AND type IN ('order.created', 'order.updated',
'order.canceled')"
```

For more information about the syntax and the features of the **cesql** filter dialect, see CloudEvents SQL Expression Language.

5.7.3. Conflict with the existing filter field

You can use the **filters** and the existing **filter** field at the same time. If you enable the new **new-trigger-filters** feature and an object contains both **filter** and **filters**, the **filters** field overrides. This setup allows you to test the new **filters** field while maintaining support for existing filters. You can gradually introduce the new field into existing trigger objects.

Example of filters field overriding the filter field:

```
apiVersion: eventing.knative.dev/v1
kind: Trigger
metadata:
  name: my-service-trigger
spec:
  broker: default
  # Existing filter field. This will be ignored when the new filters field is present.
  filter:
    attributes:
      type: dev.knative.foo.bar
      myextension: my-extension-value
  # New filters field. This takes precedence over the old filter field.
  filters:
    - cesql: "type = 'dev.knative.foo.bar' AND myextension = 'my-extension-value'"
  subscriber:
    ref:
      apiVersion: serving.knative.dev/v1
      kind: Service
      name: my-service
```

5.8. UPDATING TRIGGERS FROM THE COMMAND LINE

Using the Knative (**kn**) CLI to update triggers provides a streamlined and intuitive user interface.

5.8.1. Updating a trigger by using the Knative CLI

You can use the **kn trigger update** command with certain flags to update attributes for a trigger.

Prerequisites

- The OpenShift Serverless Operator and Knative Eventing are installed on your OpenShift Container Platform cluster.
- You have installed the Knative (**kn**) CLI.
- 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

- Update a trigger:

```
$ kn trigger update <trigger_name> --filter <key=value> --sink <sink_name> [flags]
```

  - You can update a trigger to filter exact event attributes that match incoming events. For example, using the **type** attribute:

```
$ kn trigger update <trigger_name> --filter type=knative.dev.event
```

  - You can remove a filter attribute from a trigger. For example, you can remove the filter attribute with key **type**:

```
$ kn trigger update <trigger_name> --filter type-
```

  - You can use the **--sink** parameter to change the event sink of a trigger:

```
$ kn trigger update <trigger_name> --sink ksvc:my-event-sink
```

5.9. DELETING TRIGGERS FROM THE COMMAND LINE

Using the Knative (**kn**) CLI to delete a trigger provides a streamlined and intuitive user interface.

5.9.1. Deleting a trigger by using the Knative CLI

You can use the **kn trigger delete** command to delete a trigger.

Prerequisites

- The OpenShift Serverless Operator and Knative Eventing are installed on your OpenShift Container Platform cluster.
- You have installed the Knative (**kn**) CLI.
- 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

- Delete a trigger:

```
$ kn trigger delete <trigger_name>
```

Verification

1. List existing triggers:

```
$ kn trigger list
```

2. Verify that the trigger no longer exists:

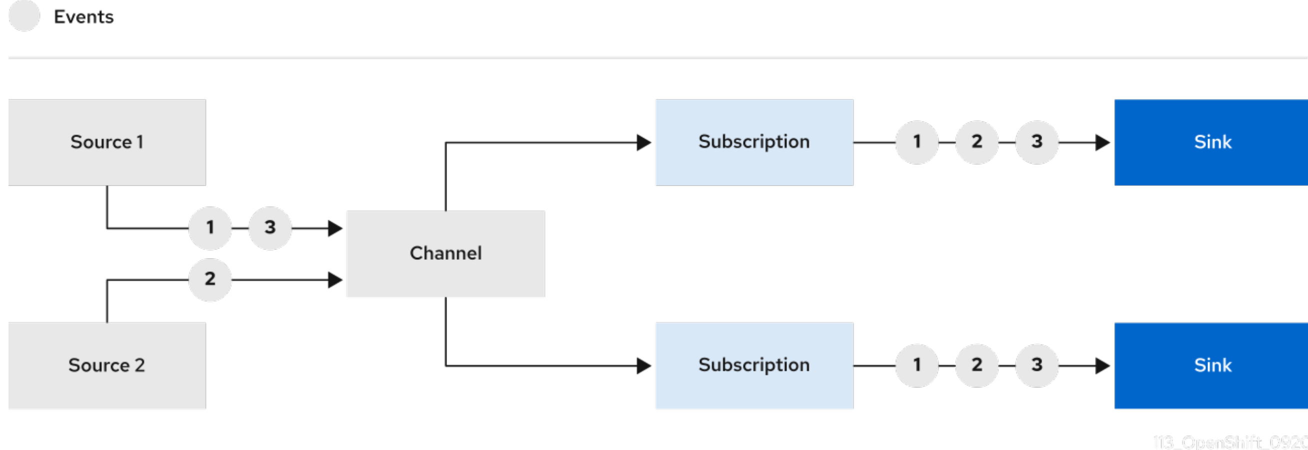
Example output

```
No triggers found.
```

CHAPTER 6. CHANNELS

6.1. CHANNELS AND SUBSCRIPTIONS

Channels are custom resources that define a single event-forwarding and persistence layer. 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.

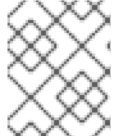


You can create channels by instantiating a supported **Channel** object, and configure re-delivery attempts by modifying the **delivery** spec in a **Subscription** object.

After you create a **Channel** object, a mutating admission webhook adds a set of **spec.channelTemplate** properties for the **Channel** object based on the default channel implementation. For example, for an **InMemoryChannel** default implementation, the **Channel** object looks as follows:

```
apiVersion: messaging.knative.dev/v1
kind: Channel
metadata:
  name: example-channel
  namespace: default
spec:
  channelTemplate:
    apiVersion: messaging.knative.dev/v1
    kind: InMemoryChannel
```

The channel controller then creates the backing channel instance based on the **spec.channelTemplate** configuration.



NOTE

The **spec.channelTemplate** properties cannot be changed after creation, because they are set by the default channel mechanism rather than by the user.

When this mechanism is used with the preceding example, two objects are created: a generic backing channel and an **InMemoryChannel** channel. If you are using a different default channel implementation, the **InMemoryChannel** is replaced with one that is specific to your implementation. For example, with the Knative broker for Apache Kafka, the **KafkaChannel** channel is created.

The backing channel acts as a proxy that copies its subscriptions to the user-created channel object, and sets the user-created channel object status to reflect the status of the backing channel.

6.1.1. Channel implementation types

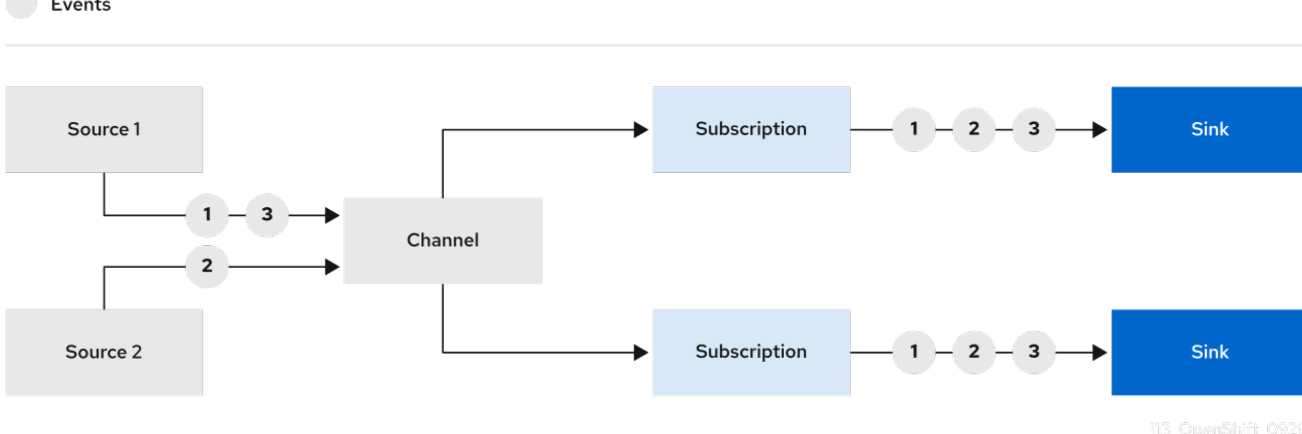
OpenShift Serverless supports the **InMemoryChannel** and **KafkaChannel** channels implementations. The **InMemoryChannel** channel is recommended for development use only due to its limitations. You can use the **KafkaChannel** channel for a production environment.

The following are limitations of **InMemoryChannel** type channels:

- 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.

6.2. CREATING CHANNELS

Channels are custom resources that define a single event-forwarding and persistence layer. 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.



You can create channels by instantiating a supported **Channel** object, and configure re-delivery attempts by modifying the **delivery** spec in a **Subscription** object.

6.2.1. Creating a channel by using the Administrator perspective

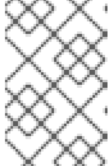
After Knative Eventing is installed on your cluster, you can create a channel by using the Administrator perspective.

Prerequisites

- The OpenShift Serverless Operator and Knative Eventing are installed on your OpenShift Container Platform cluster.
- You have logged in to the web console and are in the **Administrator** perspective.
- You have cluster administrator permissions on OpenShift Container Platform, or you have cluster or dedicated administrator permissions on Red Hat OpenShift Service on AWS or OpenShift Dedicated.

Procedure

1. In the **Administrator** perspective of the OpenShift Container Platform web console, navigate to **Serverless** → **Eventing**.
2. In the **Create** list, select **Channel**. You will be directed to the **Channel** page.
3. Select the type of **Channel** object that you want to create in the **Type** list.



NOTE

Currently only **InMemoryChannel** channel objects are supported by default. Knative channels for Apache Kafka are available if you have installed the Knative broker implementation for Apache Kafka on OpenShift Serverless.

4. Click **Create**.

6.2.2. Creating a channel by using the Developer perspective

Using the OpenShift Container Platform web console provides a streamlined and intuitive user interface to create a channel. After Knative Eventing is installed on your cluster, you can create a channel by using the web console.

Prerequisites

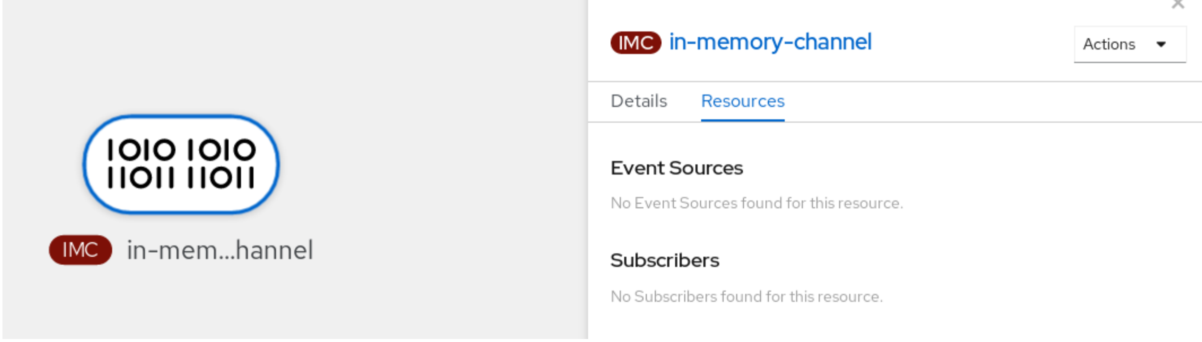
- You have logged in to the OpenShift Container Platform web console.
- The OpenShift Serverless Operator and Knative Eventing are installed on your OpenShift Container Platform cluster.
- You have created a project or have access to a project with the appropriate roles and permissions to create applications and other workloads in OpenShift Container Platform.

Procedure

1. In the **Developer** perspective, navigate to **+Add** → **Channel**.
2. Select the type of **Channel** object that you want to create in the **Type** list.
3. Click **Create**.

Verification

- Confirm that the channel now exists by navigating to the **Topology** page.



6.2.3. Creating a channel by using the Knative CLI

Using the Knative (**kn**) CLI to create channels provides a more streamlined and intuitive user interface than modifying YAML files directly. You can use the **kn channel create** command to create a channel.

Prerequisites

- The OpenShift Serverless Operator and Knative Eventing are installed on the cluster.
- You have installed the Knative (**kn**) CLI.
- 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

- Create a channel:

```
$ kn channel create <channel_name> --type <channel_type>
```

  The channel type is optional, but where specified, must be given in the format **Group:Version:Kind**. For example, you can create an **InMemoryChannel** object:

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

  Example output

```
Channel 'mychannel' created in namespace 'default'.
```

Verification

- To confirm that the channel now exists, list the existing channels and inspect the output:

```
$ kn channel list
```

  Example output

```
kn channel list
NAME        TYPE              URL                                                     AGE   READY   REASON
mychannel   InMemoryChannel   http://mychannel-kn-channel.default.svc.cluster.local   93s
True
```

Deleting a channel

- Delete a channel:

```
$ kn channel delete <channel_name>
```

6.2.4. Creating a default implementation channel by using YAML

Creating Knative resources by using YAML files uses a declarative API, which enables you to describe channels declaratively and in a reproducible manner. To create a serverless channel by using YAML, you must create a YAML file that defines a **Channel** object, then apply it by using the **oc apply** command.

Prerequisites

- The OpenShift Serverless Operator and Knative Eventing are installed on the cluster.
- Install the OpenShift CLI (**oc**).
- 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. Create a **Channel** object as a YAML file:

```
apiVersion: messaging.knative.dev/v1
kind: Channel
metadata:
  name: example-channel
  namespace: default
```

2. Apply the YAML file:

```
$ oc apply -f <filename>
```

6.2.5. Creating a channel for Apache Kafka by using YAML

Creating Knative resources by using YAML files uses a declarative API, which enables you to describe channels declaratively and in a reproducible manner. You can create a Knative Eventing channel that is backed by Kafka topics by creating a Kafka channel. To create a Kafka channel by using YAML, you must create a YAML file that defines a **KafkaChannel** object, then apply it by using the **oc apply** command.

Prerequisites

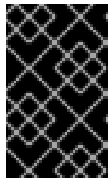
- The OpenShift Serverless Operator, Knative Eventing, and the **KnativeKafka** custom resource are installed on your OpenShift Container Platform cluster.
- Install the OpenShift CLI (**oc**).
- You have created a project or have access to a project with the appropriate roles and permissions to create applications and other workloads in OpenShift Container Platform.

Procedure

1. Create a **KafkaChannel** object as a YAML file:

```
apiVersion: messaging.knative.dev/v1beta1
kind: KafkaChannel
metadata:
  name: example-channel
  namespace: default
```

```
spec:
  numPartitions: 3
  replicationFactor: 1
```



IMPORTANT

Only the **v1beta1** version of the API for **KafkaChannel** objects on OpenShift Serverless is supported. Do not use the **v1alpha1** version of this API, as this version is now deprecated.

2. Apply the **KafkaChannel** YAML file:

```
$ oc apply -f <filename>
```

6.2.6. Next steps

- After you have created a channel, you can connect the channel to a sink so that the sink can receive events.
- Configure event delivery parameters that are applied in cases where an event fails to be delivered to an event sink.

6.3. CONNECTING CHANNELS TO SINKS

Events that have been sent to a channel from an event source or producer can be forwarded to one or more sinks by using *subscriptions*. You can create subscriptions by configuring a **Subscription** object, which specifies the channel and the sink (also known as a *subscriber*) that consumes the events sent to that channel.

6.3.1. Creating a subscription by using the Developer perspective

After you have created a channel and an event sink, you can create a subscription to enable event delivery. Using the OpenShift Container Platform web console provides a streamlined and intuitive user interface to create a subscription.

Prerequisites

- The OpenShift Serverless Operator, Knative Serving, and Knative Eventing are installed on your OpenShift Container Platform cluster.
- You have logged in to the web console.
- You have created an event sink, such as a Knative service, and a channel.
- 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. In the **Developer** perspective, navigate to the **Topology** page.
2. Create a subscription using one of the following methods:

a. Hover over the channel that you want to create a subscription for, and drag the arrow. The **Add Subscription** option is displayed.

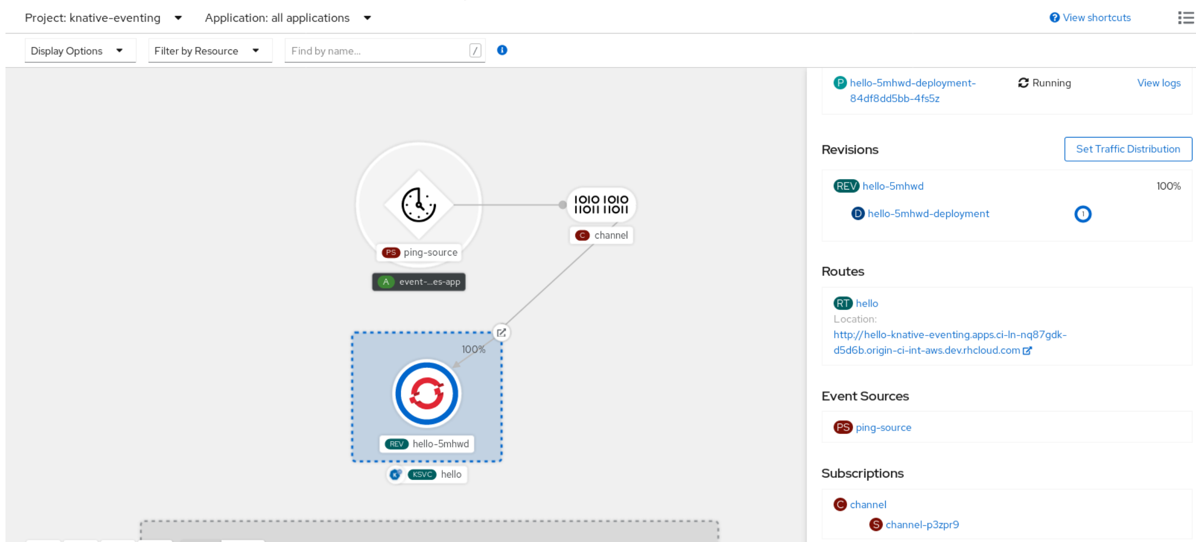
i. Select your sink in the **Subscriber** list.

ii. Click **Add**.

b. If the service is available in the **Topology** view under the same namespace or project as the channel, click on the channel that you want to create a subscription for, and drag the arrow directly to a service to immediately create a subscription from the channel to that service.

Verification

- After the subscription has been created, you can see it represented as a line that connects the channel to the service in the **Topology** view:



6.3.2. Creating a subscription by using YAML

After you have created a channel and an event sink, you can create a subscription to enable event delivery. Creating Knative resources by using YAML files uses a declarative API, which enables you to describe subscriptions declaratively and in a reproducible manner. To create a subscription by using YAML, you must create a YAML file that defines a **Subscription** object, then apply it by using the **oc apply** command.

Prerequisites

- The OpenShift Serverless Operator and Knative Eventing are installed on the cluster.
- Install the OpenShift CLI (**oc**).
- 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

- Create a **Subscription** object:
  - Create a YAML file and copy the following sample code into it:

```
apiVersion: messaging.knative.dev/v1
kind: Subscription
metadata:
  name: my-subscription (1)
  namespace: default
spec:
  channel: (2)
    apiVersion: messaging.knative.dev/v1
    kind: Channel
    name: example-channel
  delivery: (3)
    deadLetterSink:
      ref:
        apiVersion: serving.knative.dev/v1
        kind: Service
        name: error-handler
  subscriber: (4)
    ref:
      apiVersion: serving.knative.dev/v1
      kind: Service
      name: event-display
```

(1) Name of the subscription.

(2) Configuration settings for the channel that the subscription connects to.

(3) Configuration settings for event delivery. This tells the subscription what happens to events that cannot be delivered to the subscriber. When this is configured, events that failed to be consumed are sent to the **deadLetterSink**. The event is dropped, no re-delivery of the event is attempted, and an error is logged in the system. The **deadLetterSink** value must be a Destination.

(4) Configuration settings for the subscriber. This is the event sink that events are delivered to from the channel.

  - Apply the YAML file:

```
$ oc apply -f <filename>
```

6.3.3. Creating a subscription by using the Knative CLI

After you have created a channel and an event sink, you can create a subscription to enable event delivery. Using the Knative (**kn**) CLI to create subscriptions provides a more streamlined and intuitive user interface than modifying YAML files directly. You can use the **kn subscription create** command with the appropriate flags to create a subscription.

Prerequisites

- The OpenShift Serverless Operator and Knative Eventing are installed on your OpenShift Container Platform cluster.
- You have installed the Knative (**kn**) CLI.
- 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

- Create a subscription to connect a sink to a channel:

```
$ kn subscription create <subscription_name> \
  --channel <group:version:kind>:<channel_name> \ 1
  --sink <sink_prefix>:<sink_name> \ 2
  --sink-dead-letter <sink_prefix>:<sink_name> 3
```

1. **--channel** specifies the source for cloud events that should be processed. You must provide the channel name. If you are not using the default **InMemoryChannel** channel that is backed by the **Channel** custom resource, you must prefix the channel name with the **<group:version:kind>** for the specified channel type. For example, this will be **messaging.knative.dev:v1beta1:KafkaChannel** for an Apache Kafka backed channel.

2. **--sink** specifies the target destination to which the event should be delivered. By default, the **<sink_name>** is interpreted as a Knative service of this name, in the same namespace as the subscription. You can specify the type of the sink by using one of the following prefixes:

ksvc
: A Knative service.

channel
: A channel that should be used as destination. Only default channel types can be referenced here.

broker
: An Eventing broker.

3. Optional: **--sink-dead-letter** is an optional flag that can be used to specify a sink which events should be sent to in cases where events fail to be delivered. For more information, see the OpenShift Serverless *Event delivery* documentation.

Example command

```
$ kn subscription create mysubscription --channel mychannel --sink ksvc:event-display
```

Example output

```
Subscription 'mysubscription' created in namespace 'default'.
```

Verification

- To confirm that the channel is connected to the event sink, or *subscriber*, by a subscription, list the existing subscriptions and inspect the output:

```
$ kn subscription list
```

Example output

```
NAME           CHANNEL            SUBSCRIBER           REPLY   DEAD LETTER SINK
READY   REASON
mysubscription   Channel:mychannel   ksvc:event-display                        True
```

Deleting a subscription

- Delete a subscription:

```
$ kn subscription delete <subscription_name>
```

6.3.4. Creating a subscription by using the Administrator perspective

After you have created a channel and an event sink, also known as a *subscriber*, you can create a subscription to enable event delivery. Subscriptions are created by configuring a **Subscription** object, which specifies the channel and the subscriber to deliver events to. You can also specify some subscriber-specific options, such as how to handle failures.

Prerequisites

- The OpenShift Serverless Operator and Knative Eventing are installed on your OpenShift Container Platform cluster.
- You have logged in to the web console and are in the **Administrator** perspective.
- You have cluster administrator permissions on OpenShift Container Platform, or you have cluster or dedicated administrator permissions on Red Hat OpenShift Service on AWS or OpenShift Dedicated.
- You have created a Knative channel.
- You have created a Knative service to use as a subscriber.

Procedure

1. In the **Administrator** perspective of the OpenShift Container Platform web console, navigate to **Serverless → Eventing**.
2. In the **Channel** tab, select the Options menu ⋮ for the channel that you want to add a subscription to.
3. Click **Add Subscription** in the list.

4. In the **Add Subscription** dialogue box, select a **Subscriber** for the subscription. The subscriber is the Knative service that receives events from the channel.

5. Click **Add**.

6.3.5. Next steps

- Configure event delivery parameters that are applied in cases where an event fails to be delivered to an event sink.

6.4. DEFAULT CHANNEL IMPLEMENTATION

You can use the **default-ch-webhook** config map to specify the default channel implementation of Knative Eventing. You can specify the default channel implementation for the entire cluster or for one or more namespaces. Currently the **InMemoryChannel** and **KafkaChannel** channel types are supported.

6.4.1. Configuring the default channel implementation

Prerequisites

- You have administrator permissions on OpenShift Container Platform.
- You have installed the OpenShift Serverless Operator and Knative Eventing on your cluster.
- If you want to use Knative channels for Apache Kafka as the default channel implementation, you must also install the **KnativeKafka** CR on your cluster.

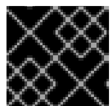
Procedure

- Modify the **KnativeEventing** custom resource to add configuration details for the **default-ch-webhook** config map:

```
apiVersion: operator.knative.dev/v1beta1
kind: KnativeEventing
metadata:
  name: knative-eventing
  namespace: knative-eventing
spec:
  config: 1
    default-ch-webhook: 2
      default-ch-config: |
        clusterDefault: 3
          apiVersion: messaging.knative.dev/v1
          kind: InMemoryChannel
          spec:
            delivery:
              backoffDelay: PT0.5S
              backoffPolicy: exponential
              retry: 5
        namespaceDefaults: 4
          my-namespace:
            apiVersion: messaging.knative.dev/v1beta1
            kind: KafkaChannel
```

```
spec:
  numPartitions: 1
  replicationFactor: 1
```

1. In **spec.config**, you can specify the config maps that you want to add modified configurations for.
2. The **default-ch-webhook** config map can be used to specify the default channel implementation for the cluster or for one or more namespaces.
3. The cluster-wide default channel type configuration. In this example, the default channel implementation for the cluster is **InMemoryChannel**.
4. The namespace-scoped default channel type configuration. In this example, the default channel implementation for the **my-namespace** namespace is **KafkaChannel**.



IMPORTANT

Configuring a namespace-specific default overrides any cluster-wide settings.

6.5. SECURITY CONFIGURATION FOR CHANNELS

6.5.1. Configuring TLS authentication for Knative channels for Apache Kafka

Transport Layer Security (TLS) is used by Apache Kafka clients and servers to encrypt traffic between Knative and Kafka, as well as for authentication. TLS is the only supported method of traffic encryption for the Knative broker implementation for Apache Kafka.

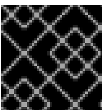
Prerequisites

- You have cluster or dedicated administrator permissions on OpenShift Container Platform.
- The OpenShift Serverless Operator, Knative Eventing, and the **KnativeKafka** CR are installed on your OpenShift Container Platform cluster.
- 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.
- You have a Kafka cluster CA certificate stored as a **.pem** file.
- You have a Kafka cluster client certificate and a key stored as **.pem** files.
- Install the OpenShift CLI (**oc**).

Procedure

1. Create the certificate files as secrets in your chosen namespace:

```
$ oc create secret -n <namespace> generic <kafka_auth_secret> \
  --from-file=ca.crt=caroot.pem \
  --from-file=user.crt=certificate.pem \
  --from-file=user.key=key.pem
```



IMPORTANT

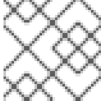
Use the key names **ca.crt**, **user.crt**, and **user.key**. Do not change them.

2. Start editing the **KnativeKafka** custom resource:

```
$ oc edit knativekafka
```

3. Reference your secret and the namespace of the secret:

```
apiVersion: operator.serverless.openshift.io/v1alpha1
kind: KnativeKafka
metadata:
  namespace: knative-eventing
  name: knative-kafka
spec:
  channel:
    authSecretName: <kafka_auth_secret>
    authSecretNamespace: <kafka_auth_secret_namespace>
    bootstrapServers: <bootstrap_servers>
    enabled: true
  source:
    enabled: true
```



NOTE

Make sure to specify the matching port in the bootstrap server.

For example:

```
apiVersion: operator.serverless.openshift.io/v1alpha1
kind: KnativeKafka
metadata:
  namespace: knative-eventing
  name: knative-kafka
spec:
  channel:
    authSecretName: tls-user
    authSecretNamespace: kafka
    bootstrapServers: eventing-kafka-bootstrap.kafka.svc:9094
    enabled: true
  source:
    enabled: true
```

6.5.2. Configuring SASL authentication for Knative channels for Apache Kafka

Simple Authentication and Security Layer (SASL) is used by Apache Kafka for authentication. If you use SASL authentication on your cluster, users must provide credentials to Knative for communicating with the Kafka cluster; otherwise events cannot be produced or consumed.

Prerequisites

- You have cluster or dedicated administrator permissions on OpenShift Container Platform.

- The OpenShift Serverless Operator, Knative Eventing, and the **KnativeKafka** CR are installed on your OpenShift Container Platform cluster.
- 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.
- You have a username and password for a Kafka cluster.
- You have chosen the SASL mechanism to use, for example, **PLAIN**, **SCRAM-SHA-256**, or **SCRAM-SHA-512**.
- If TLS is enabled, you also need the **ca.crt** certificate file for the Kafka cluster.
- Install the OpenShift CLI (**oc**).

Procedure

1. Create the certificate files as secrets in your chosen namespace:

```
$ oc create secret -n <namespace> generic <kafka_auth_secret> \
  --from-file=ca.crt=caroot.pem \
  --from-literal=password="SecretPassword" \
  --from-literal=saslType="SCRAM-SHA-512" \
  --from-literal=user="my-sasl-user"
```

   - Use the key names **ca.crt**, **password**, and **sasl.mechanism**. Do not change them.
   - If you want to use SASL with public CA certificates, you must use the **tls.enabled=true** flag, rather than the **ca.crt** argument, when creating the secret. For example:

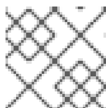
```
$ oc create secret -n <namespace> generic <kafka_auth_secret> \
  --from-literal=tls.enabled=true \
  --from-literal=password="SecretPassword" \
  --from-literal=saslType="SCRAM-SHA-512" \
  --from-literal=user="my-sasl-user"
```

2. Start editing the **KnativeKafka** custom resource:

```
$ oc edit knativekafka
```

3. Reference your secret and the namespace of the secret:

```
apiVersion: operator.serverless.openshift.io/v1alpha1
kind: KnativeKafka
metadata:
  namespace: knative-eventing
  name: knative-kafka
spec:
  channel:
    authSecretName: <kafka_auth_secret>
    authSecretNamespace: <kafka_auth_secret_namespace>
    bootstrapServers: <bootstrap_servers>
    enabled: true
  source:
    enabled: true
```



NOTE

Make sure to specify the matching port in the bootstrap server.

For example:

```
apiVersion: operator.serverless.openshift.io/v1alpha1
kind: KnativeKafka
metadata:
  namespace: knative-eventing
  name: knative-kafka
spec:
  channel:
    authSecretName: scram-user
    authSecretNamespace: kafka
    bootstrapServers: eventing-kafka-bootstrap.kafka.svc:9093
    enabled: true
  source:
    enabled: true
```

CHAPTER 7. SUBSCRIPTIONS

7.1. CREATING SUBSCRIPTIONS

After you have created a channel and an event sink, you can create a subscription to enable event delivery. Subscriptions are created by configuring a **Subscription** object, which specifies the channel and the sink (also known as a *subscriber*) to deliver events to.

7.1.1. Creating a subscription by using the Administrator perspective

After you have created a channel and an event sink, also known as a *subscriber*, you can create a subscription to enable event delivery. Subscriptions are created by configuring a **Subscription** object, which specifies the channel and the subscriber to deliver events to. You can also specify some subscriber-specific options, such as how to handle failures.

Prerequisites

- The OpenShift Serverless Operator and Knative Eventing are installed on your OpenShift Container Platform cluster.
- You have logged in to the web console and are in the **Administrator** perspective.
- You have cluster administrator permissions on OpenShift Container Platform, or you have cluster or dedicated administrator permissions on Red Hat OpenShift Service on AWS or OpenShift Dedicated.
- You have created a Knative channel.
- You have created a Knative service to use as a subscriber.

Procedure

1. In the **Administrator** perspective of the OpenShift Container Platform web console, navigate to **Serverless → Eventing**.
2. In the **Channel** tab, select the Options menu for the channel that you want to add a subscription to.
3. Click **Add Subscription** in the list.
4. In the **Add Subscription** dialogue box, select a **Subscriber** for the subscription. The subscriber is the Knative service that receives events from the channel.
5. Click **Add**.

7.1.2. Creating a subscription by using the Developer perspective

After you have created a channel and an event sink, you can create a subscription to enable event delivery. Using the OpenShift Container Platform web console provides a streamlined and intuitive user interface to create a subscription.

Prerequisites

- The OpenShift Serverless Operator, Knative Serving, and Knative Eventing are installed on your OpenShift Container Platform cluster.
- You have logged in to the web console.
- You have created an event sink, such as a Knative service, and a channel.
- 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

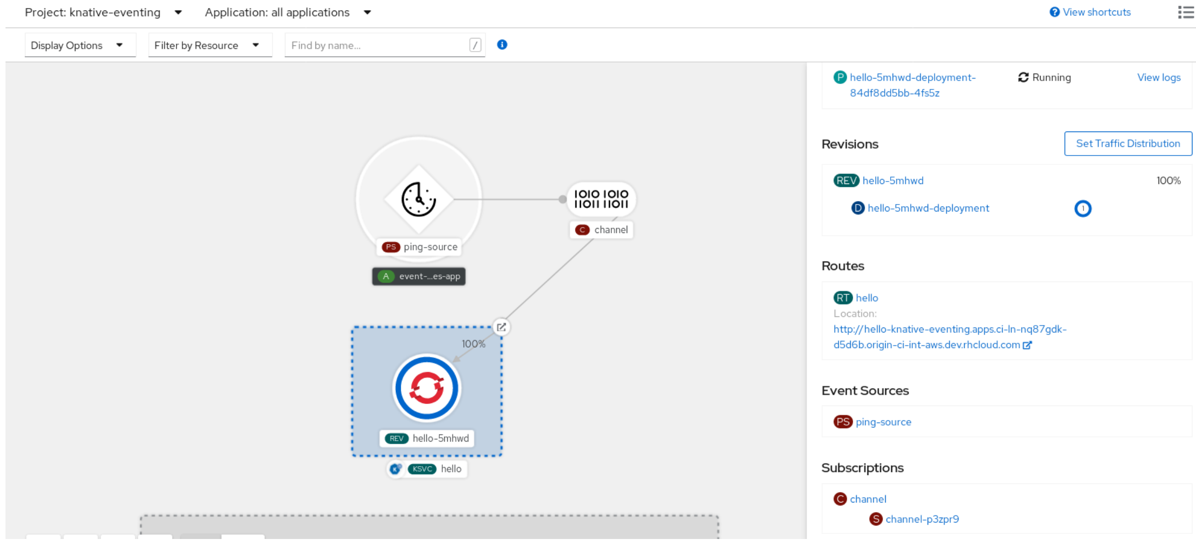
1. In the **Developer** perspective, navigate to the **Topology** page.
2. Create a subscription using one of the following methods:
    a. Hover over the channel that you want to create a subscription for, and drag the arrow. The **Add Subscription** option is displayed.

        i. Select your sink in the **Subscriber** list.
        ii. Click **Add**.

    b. If the service is available in the **Topology** view under the same namespace or project as the channel, click on the channel that you want to create a subscription for, and drag the arrow directly to a service to immediately create a subscription from the channel to that service.

Verification

- After the subscription has been created, you can see it represented as a line that connects the channel to the service in the **Topology** view:



7.1.3. Creating a subscription by using YAML

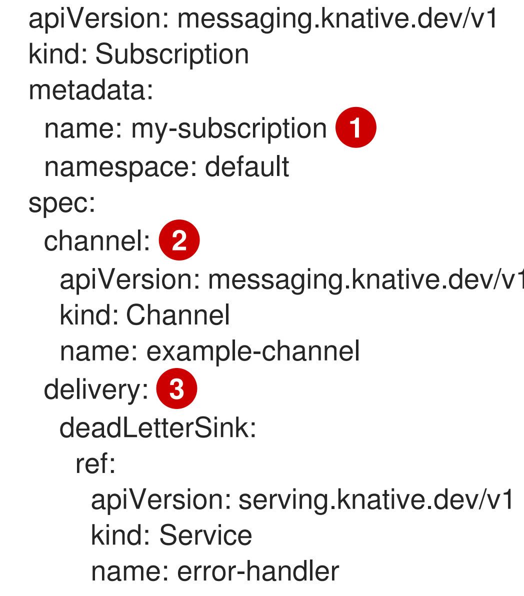
After you have created a channel and an event sink, you can create a subscription to enable event delivery. Creating Knative resources by using YAML files uses a declarative API, which enables you to describe subscriptions declaratively and in a reproducible manner. To create a subscription by using YAML, you must create a YAML file that defines a **Subscription** object, then apply it by using the **oc apply** command.

Prerequisites

- The OpenShift Serverless Operator and Knative Eventing are installed on the cluster.
- Install the OpenShift CLI (**oc**).
- 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

- Create a **Subscription** object:
  - Create a YAML file and copy the following sample code into it:



```
apiVersion: messaging.knative.dev/v1
kind: Subscription
metadata:
  name: my-subscription 1
  namespace: default
spec:
  channel: 2
    apiVersion: messaging.knative.dev/v1
    kind: Channel
    name: example-channel
  delivery: 3
    deadLetterSink:
      ref:
        apiVersion: serving.knative.dev/v1
        kind: Service
        name: error-handler
```

```
subscriber: (4)
  ref:
    apiVersion: serving.knative.dev/v1
    kind: Service
    name: event-display
```

1. Name of the subscription.
2. Configuration settings for the channel that the subscription connects to.
3. Configuration settings for event delivery. This tells the subscription what happens to events that cannot be delivered to the subscriber. When this is configured, events that failed to be consumed are sent to the **deadLetterSink**. The event is dropped, no re-delivery of the event is attempted, and an error is logged in the system. The **deadLetterSink** value must be a Destination.
4. Configuration settings for the subscriber. This is the event sink that events are delivered to from the channel.

- Apply the YAML file:

```
$ oc apply -f <filename>
```

7.1.4. Creating a subscription by using the Knative CLI

After you have created a channel and an event sink, you can create a subscription to enable event delivery. Using the Knative (**kn**) CLI to create subscriptions provides a more streamlined and intuitive user interface than modifying YAML files directly. You can use the **kn subscription create** command with the appropriate flags to create a subscription.

Prerequisites

- The OpenShift Serverless Operator and Knative Eventing are installed on your OpenShift Container Platform cluster.
- You have installed the Knative (**kn**) CLI.
- 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

- Create a subscription to connect a sink to a channel:

```
$ kn subscription create <subscription_name> \
  --channel <group:version:kind>:<channel_name> \ (1)
  --sink <sink_prefix>:<sink_name> \ (2)
  --sink-dead-letter <sink_prefix>:<sink_name> (3)
```

1. **--channel** specifies the source for cloud events that should be processed. You must provide the channel name. If you are not using the default **InMemoryChannel** channel that is backed by the **Channel** custom resource, you must prefix the channel name with the **<group:version:kind>** for the specified channel type. For example, this will be

messaging.knative.dev:v1beta1:KafkaChannel for an Apache Kafka backed channel.

--sink specifies the target destination to which the event should be delivered. By default, the **<sink_name>** is interpreted as a Knative service of this name, in the same namespace as the subscription. You can specify the type of the sink by using one of the following prefixes:

ksvc
: A Knative service.

channel
: A channel that should be used as destination. Only default channel types can be referenced here.

broker
: An Eventing broker.

Optional: **--sink-dead-letter** is an optional flag that can be used to specify a sink which events should be sent to in cases where events fail to be delivered. For more information, see the OpenShift Serverless *Event delivery* documentation.

Example command

```
$ kn subscription create mysubscription --channel mychannel --sink ksvc:event-display
```

Example output

```
Subscription 'mysubscription' created in namespace 'default'.
```

Verification

- To confirm that the channel is connected to the event sink, or *subscriber*, by a subscription, list the existing subscriptions and inspect the output:

```
$ kn subscription list
```

Example output

```
NAME            CHANNEL             SUBSCRIBER           REPLY   DEAD LETTER SINK
READY   REASON
mysubscription   Channel:mychannel   ksvc:event-display                    True
```

Deleting a subscription

- Delete a subscription:

```
$ kn subscription delete <subscription_name>
```

7.1.5. Next steps

- Configure event delivery parameters that are applied in cases where an event fails to be delivered to an event sink.

7.2. MANAGING SUBSCRIPTIONS

7.2.1. Describing subscriptions by using the Knative CLI

You can use the **kn subscription describe** command to print information about a subscription in the terminal by using the Knative (**kn**) CLI. Using the Knative CLI to describe subscriptions provides a more streamlined and intuitive user interface than viewing YAML files directly.

Prerequisites

- You have installed the Knative (**kn**) CLI.
- You have created a subscription in your cluster.

Procedure

- Describe a subscription:

```
$ kn subscription describe <subscription_name>
```

Example output

```
Name:            my-subscription
Namespace:       default
Annotations:     messaging.knative.dev/creator=openshift-user,
messaging.knative.dev/lastModifier=min ...
Age:             43s
Channel:         Channel:my-channel (messaging.knative.dev/v1)
Subscriber:
  URI:           http://edisplay.default.example.com
Reply:
  Name:          default
  Resource:      Broker (eventing.knative.dev/v1)
DeadLetterSink:
  Name:          my-sink
  Resource:      Service (serving.knative.dev/v1)

Conditions:
  OK TYPE                  AGE REASON
  ++ Ready                 43s
  ++ AddedToChannel        43s
  ++ ChannelReady          43s
  ++ ReferencesResolved    43s
```

7.2.2. Listing subscriptions by using the Knative CLI

You can use the **kn subscription list** command to list existing subscriptions on your cluster by using the Knative (**kn**) CLI. Using the Knative CLI to list subscriptions provides a streamlined and intuitive user interface.

Prerequisites

- You have installed the Knative (**kn**) CLI.

Procedure

- List subscriptions on your cluster:

```
$ kn subscription list
```

Example output

```
NAME           CHANNEL           SUBSCRIBER          REPLY   DEAD LETTER SINK
READY   REASON
mysubscription   Channel:mychannel   ksvc:event-display                        True
```

7.2.3. Updating subscriptions by using the Knative CLI

You can use the **kn subscription update** command as well as the appropriate flags to update a subscription from the terminal by using the Knative (**kn**) CLI. Using the Knative CLI to update subscriptions provides a more streamlined and intuitive user interface than updating YAML files directly.

Prerequisites

- You have installed the Knative (**kn**) CLI.
- You have created a subscription.

Procedure

- Update a subscription:

```
$ kn subscription update <subscription_name> \
  --sink <sink_prefix>:<sink_name> \ (1)
  --sink-dead-letter <sink_prefix>:<sink_name> (2)
```

(1) **--sink** specifies the updated target destination to which the event should be delivered. You can specify the type of the sink by using one of the following prefixes:

ksvc
: A Knative service.

channel
: A channel that should be used as destination. Only default channel types can be referenced here.

broker
: An Eventing broker.

(2) Optional: **--sink-dead-letter** is an optional flag that can be used to specify a sink which events should be sent to in cases where events fail to be delivered. For more information, see the OpenShift Serverless *Event delivery* documentation.

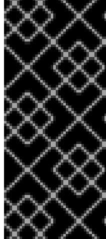
Example command

```
$ kn subscription update mysubscription --sink ksvc:event-display
```

CHAPTER 8. EVENT DELIVERY

You can configure event delivery parameters that are applied in cases where an event fails to be delivered to an event sink. Different channel and broker types have their own behavior patterns that are followed for event delivery.

Configuring event delivery parameters, including a dead letter sink, ensures that any events that fail to be delivered to an event sink are retried. Otherwise, undelivered events are dropped.



IMPORTANT

If an event is successfully delivered to a channel or broker receiver for Apache Kafka, the receiver responds with a **202** status code, which means that the event has been safely stored inside a Kafka topic and is not lost. If the receiver responds with any other status code, the event is not safely stored, and steps must be taken by the user to resolve the issue.

8.1. CONFIGURABLE EVENT DELIVERY PARAMETERS

The following parameters can be configured for event delivery:

Dead letter sink

You can configure the **deadLetterSink** delivery parameter so that if an event fails to be delivered, it is stored in the specified event sink. Undelivered events that are not stored in a dead letter sink are dropped. The dead letter sink be any addressable object that conforms to the Knative Eventing sink contract, such as a Knative service, a Kubernetes service, or a URI.

Retries

You can set a minimum number of times that the delivery must be retried before the event is sent to the dead letter sink, by configuring the **retry** delivery parameter with an integer value.

Back off delay

You can set the **backoffDelay** delivery parameter to specify the time delay before an event delivery retry is attempted after a failure. The duration of the **backoffDelay** parameter is specified using the ISO 8601 format. For example, **PT1S** specifies a 1 second delay.

Back off policy

The **backoffPolicy** delivery parameter can be used to specify the retry back off policy. The policy can be specified as either **linear** or **exponential**. When using the **linear** back off policy, the back off delay is equal to **backoffDelay * <numberOfRetries>**. When using the **exponential** backoff policy, the back off delay is equal to **backoffDelay*2^<numberOfRetries>**.

8.2. EXAMPLES OF CONFIGURING EVENT DELIVERY PARAMETERS

You can configure event delivery parameters for **Broker**, **Trigger**, **Channel**, and **Subscription** objects. If you configure event delivery parameters for a broker or channel, these parameters are propagated to triggers or subscriptions created for those objects. You can also set event delivery parameters for triggers or subscriptions to override the settings for the broker or channel.

Example Broker object

```
apiVersion: eventing.knative.dev/v1
kind: Broker
metadata:
# ...
```

```
spec:
  delivery:
    deadLetterSink:
      ref:
        apiVersion: eventing.knative.dev/v1alpha1
        kind: KafkaSink
        name: <sink_name>
    backoffDelay: <duration>
    backoffPolicy: <policy_type>
    retry: <integer>
# ...
```

Example Trigger object

```
apiVersion: eventing.knative.dev/v1
kind: Trigger
metadata:
# ...
spec:
  broker: <broker_name>
  delivery:
    deadLetterSink:
      ref:
        apiVersion: serving.knative.dev/v1
        kind: Service
        name: <sink_name>
    backoffDelay: <duration>
    backoffPolicy: <policy_type>
    retry: <integer>
# ...
```

Example Channel object

```
apiVersion: messaging.knative.dev/v1
kind: Channel
metadata:
# ...
spec:
  delivery:
    deadLetterSink:
      ref:
        apiVersion: serving.knative.dev/v1
        kind: Service
        name: <sink_name>
    backoffDelay: <duration>
    backoffPolicy: <policy_type>
    retry: <integer>
# ...
```

Example Subscription object

```
apiVersion: messaging.knative.dev/v1
kind: Subscription
metadata:
```

```
# ...
spec:
  channel:
    apiVersion: messaging.knative.dev/v1
    kind: Channel
    name: <channel_name>
  delivery:
    deadLetterSink:
      ref:
        apiVersion: serving.knative.dev/v1
        kind: Service
        name: <sink_name>
    backoffDelay: <duration>
    backoffPolicy: <policy_type>
    retry: <integer>
# ...
```

8.3. CONFIGURING EVENT DELIVERY ORDERING FOR TRIGGERS

If you are using a Kafka broker, you can configure the delivery order of events from triggers to event sinks.

Prerequisites

- The OpenShift Serverless Operator, Knative Eventing, and Knative Kafka are installed on your OpenShift Container Platform cluster.
- Kafka broker is enabled for use on your cluster, and you have created a Kafka broker.
- 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.
- You have installed the OpenShift (**oc**) CLI.

Procedure

1. Create or modify a **Trigger** object and set the **kafka.eventing.knative.dev/delivery.order** annotation:

   ```
   apiVersion: eventing.knative.dev/v1
   kind: Trigger
   metadata:
     name: <trigger_name>
     annotations:
        kafka.eventing.knative.dev/delivery.order: ordered
   # ...
   ```

   The supported consumer delivery guarantees are:

   unordered
   : An unordered consumer is a non-blocking consumer that delivers messages unordered, while preserving proper offset management.

   ordered

An ordered consumer is a per-partition blocking consumer that waits for a successful response from the CloudEvent subscriber before it delivers the next message of the partition.
The default ordering guarantee is **unordered**.

2. Apply the **Trigger** object:

```
$ oc apply -f <filename>
```

CHAPTER 9. EVENT DISCOVERY

9.1. LISTING EVENT SOURCES AND EVENT SOURCE TYPES

It is possible to view a list of all event sources or event source types that exist or are available for use on your OpenShift Container Platform cluster. You can use the Knative (**kn**) CLI or the **Developer** perspective in the OpenShift Container Platform web console to list available event sources or event source types.

9.2. LISTING EVENT SOURCE TYPES FROM THE COMMAND LINE

Using the Knative (**kn**) CLI provides a streamlined and intuitive user interface to view available event source types on your cluster.

9.2.1. Listing available event source types by using the Knative CLI

You can list event source types that can be created and used on your cluster by using the **kn source list-types** CLI command.

Prerequisites

- The OpenShift Serverless Operator and Knative Eventing are installed on the cluster.
- You have installed the Knative (**kn**) CLI.

Procedure

1. List the available event source types in the terminal:

```
$ kn source list-types
```

Example output

```
TYPE              NAME                                   DESCRIPTION
ApiServerSource   apiserversources.sources.knative.dev   Watch and send Kubernetes API events to a sink
PingSource        pingsources.sources.knative.dev        Periodically send ping events to a sink
SinkBinding       sinkbindings.sources.knative.dev       Binding for connecting a PodSpecable to a sink
```

2. Optional: On OpenShift Container Platform, you can also list the available event source types in YAML format:

```
$ kn source list-types -o yaml
```

9.3. LISTING EVENT SOURCE TYPES FROM THE DEVELOPER PERSPECTIVE

It is possible to view a list of all available event source types on your cluster. Using the OpenShift Container Platform web console provides a streamlined and intuitive user interface to view available event source types.

9.3.1. Viewing available event source types within the Developer perspective

Prerequisites

- You have logged in to the OpenShift Container Platform web console.
- The OpenShift Serverless Operator and Knative Eventing are installed on your OpenShift Container Platform cluster.
- 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. Access the **Developer** perspective.
2. Click **+Add**.
3. Click **Event Source**.
4. View the available event source types.

9.4. LISTING EVENT SOURCES FROM THE COMMAND LINE

Using the Knative (**kn**) CLI provides a streamlined and intuitive user interface to view existing event sources on your cluster.

9.4.1. Listing available event sources by using the Knative CLI

You can list existing event sources by using the **kn source list** command.

Prerequisites

- The OpenShift Serverless Operator and Knative Eventing are installed on the cluster.
- You have installed the Knative (**kn**) CLI.

Procedure

1. List the existing event sources in the terminal:

```
$ kn source list
```

Example output

```
NAME   TYPE              RESOURCE                               SINK         READY
a1     ApiServerSource   apiserversources.sources.knative.dev   ksvc:eshow2   True
b1     SinkBinding       sinkbindings.sources.knative.dev       ksvc:eshow3   False
p1     PingSource        pingsources.sources.knative.dev        ksvc:eshow1   True
```

2. Optional: You can list event sources of a specific type only, by using the **--type** flag:

```
$ kn source list --type <event_source_type>
```

Example command

```
$ kn source list --type PingSource
```

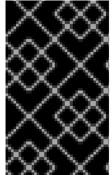
Example output

```
NAME   TYPE              RESOURCE                               SINK         READY
p1     PingSource       pingsources.sources.knative.dev       ksvc:eshow1   True
```

CHAPTER 10. TUNING EVENTING CONFIGURATION

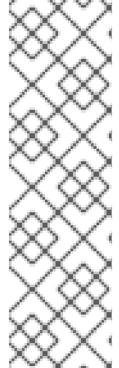
10.1. OVERRIDING KNATIVE EVENTING SYSTEM DEPLOYMENT CONFIGURATIONS

You can override the default configurations for some specific deployments by modifying the **workloads** spec in the **KnativeEventing** custom resource (CR). Currently, overriding default configuration settings is supported for the **eventing-controller**, **eventing-webhook**, and **imc-controller** fields, as well as for the **readiness** and **liveness** fields for probes.



IMPORTANT

The **replicas** spec cannot override the number of replicas for deployments that use the Horizontal Pod Autoscaler (HPA), and does not work for the **eventing-webhook** deployment.



NOTE

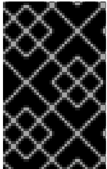
You can only override probes that are defined in the deployment by default.

All Knative Serving deployments define a readiness and a liveness probe by default, with these exceptions:

- **net-kourier-controller** and **3scale-kourier-gateway** only define a readiness probe.
- **net-istio-controller** and **net-istio-webhook** define no probes.

10.1.1. Overriding deployment configurations

Currently, overriding default configuration settings is supported for the **eventing-controller**, **eventing-webhook**, and **imc-controller** fields, as well as for the **readiness** and **liveness** fields for probes.



IMPORTANT

The **replicas** spec cannot override the number of replicas for deployments that use the Horizontal Pod Autoscaler (HPA), and does not work for the **eventing-webhook** deployment.

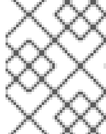
In the following example, a **KnativeEventing** CR overrides the **eventing-controller** deployment so that:

- The **readiness** probe timeout **eventing-controller** is set to be 10 seconds.
- The deployment has specified CPU and memory resource limits.
- The deployment has 3 replicas.
- The **example-label: label** label is added.
- The **example-annotation: annotation** annotation is added.
- The **nodeSelector** field is set to select nodes with the **disktype: hdd** label.

KnativeEventing CR example

```
apiVersion: operator.knative.dev/v1beta1
kind: KnativeEventing
metadata:
  name: knative-eventing
  namespace: knative-eventing
spec:
  workloads:
  - name: eventing-controller
    readinessProbes: 1
      - container: controller
        timeoutSeconds: 10
    resources:
    - container: eventing-controller
      requests:
        cpu: 300m
        memory: 100Mi
      limits:
        cpu: 1000m
        memory: 250Mi
    replicas: 3
    labels:
      example-label: label
    annotations:
      example-annotation: annotation
    nodeSelector:
      disktype: hdd
```

1. You can use the **readiness** and **liveness** probe overrides to override all fields of a probe in a container of a deployment as specified in the Kubernetes API except for the fields related to the probe handler: **exec**, **grpc**, **httpGet**, and **tcpSocket**.



NOTE

The **KnativeEventing** CR label and annotation settings override the deployment's labels and annotations for both the deployment itself and the resulting pods.

10.1.2. Modifying consumer group IDs and topic names

You can change templates for generating consumer group IDs and topic names used by your triggers, brokers, and channels.

Prerequisites

- You have cluster or dedicated administrator permissions on OpenShift Container Platform.
- The OpenShift Serverless Operator, Knative Eventing, and the **KnativeKafka** custom resource (CR) are installed on your OpenShift Container Platform cluster.
- You have created a project or have access to a project that has the appropriate roles and permissions to create applications and other workloads in OpenShift Container Platform.
- You have installed the OpenShift CLI (**oc**).

Procedure

1. To change templates for generating consumer group IDs and topic names used by your triggers, brokers, and channels, modify the **KnativeKafka** resource:

```
apiVersion: v1
kind: KnativeKafka
metadata:
  name: knative-kafka
  namespace: knative-eventing
# ...
spec:
  config:
    config-kafka-features:
      triggers.consumergroup.template: <template> (1)
      brokers.topic.template: <template> (2)
      channels.topic.template: <template> (3)
```

(1) The template for generating the consumer group ID used by your triggers. Use a valid Go **text/template** value. Defaults to **{% raw %}"knative-trigger-{{ .Namespace }}-{{ .Name }}"{% endraw %}**.

(2) The template for generating Kafka topic names used by your brokers. Use a valid Go **text/template** value. Defaults to **{% raw %}"knative-broker-{{ .Namespace }}-{{ .Name }}"{% endraw %}**.

(3) The template for generating Kafka topic names used by your channels. Use a valid Go **text/template** value. Defaults to **{% raw %}"messaging-kafka.{{ .Namespace }}.{{ .Name }}"{% endraw %}**.

Example template configuration

```
apiVersion: v1
kind: KnativeKafka
metadata:
  name: knative-kafka
  namespace: knative-eventing
# ...
spec:
  config:
    config-kafka-features:
      triggers.consumergroup.template: "{% raw %}"knative-trigger-{{ .Namespace }}-{{ .Name
}}-{{ .annotations.my-annotation }}"{% endraw %}"
      brokers.topic.template: "{% raw %}"knative-broker-{{ .Namespace }}-{{ .Name }}-{{
.annotations.my-annotation }}"{% endraw %}"
      channels.topic.template: "{% raw %}"messaging-kafka.{{ .Namespace }}.{{ .Name }}-{{
.annotations.my-annotation }}"{% endraw %}"
```

2. Apply the **KnativeKafka** YAML file:

```
$ oc apply -f <knative_kafka_filename>
```

Additional resources

- Probe configuration section of the Kubernetes API documentation

10.2. HIGH AVAILABILITY

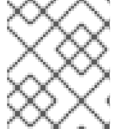
High availability (HA) is a standard feature of Kubernetes APIs that helps to ensure that APIs stay operational if a disruption occurs. In an HA deployment, if an active controller crashes or is deleted, another controller is readily available. This controller takes over processing of the APIs that were being serviced by the controller that is now unavailable.

HA in OpenShift Serverless is available through leader election, which is enabled by default after the Knative Serving or Eventing control plane is installed. When using a leader election HA pattern, instances of controllers are already scheduled and running inside the cluster before they are required. These controller instances compete to use a shared resource, known as the leader election lock. The instance of the controller that has access to the leader election lock resource at any given time is called the leader.

HA in OpenShift Serverless is available through leader election, which is enabled by default after the Knative Serving or Eventing control plane is installed. When using a leader election HA pattern, instances of controllers are already scheduled and running inside the cluster before they are required. These controller instances compete to use a shared resource, known as the leader election lock. The instance of the controller that has access to the leader election lock resource at any given time is called the leader.

10.2.1. Configuring high availability replicas for Knative Eventing

High availability (HA) is available by default for the Knative Eventing **eventing-controller**, **eventing-webhook**, **imc-controller**, **imc-dispatcher**, and **mt-broker-controller** components, which are configured to have two replicas each by default. You can change the number of replicas for these components by modifying the **spec.high-availability.replicas** value in the **KnativeEventing** custom resource (CR).



NOTE

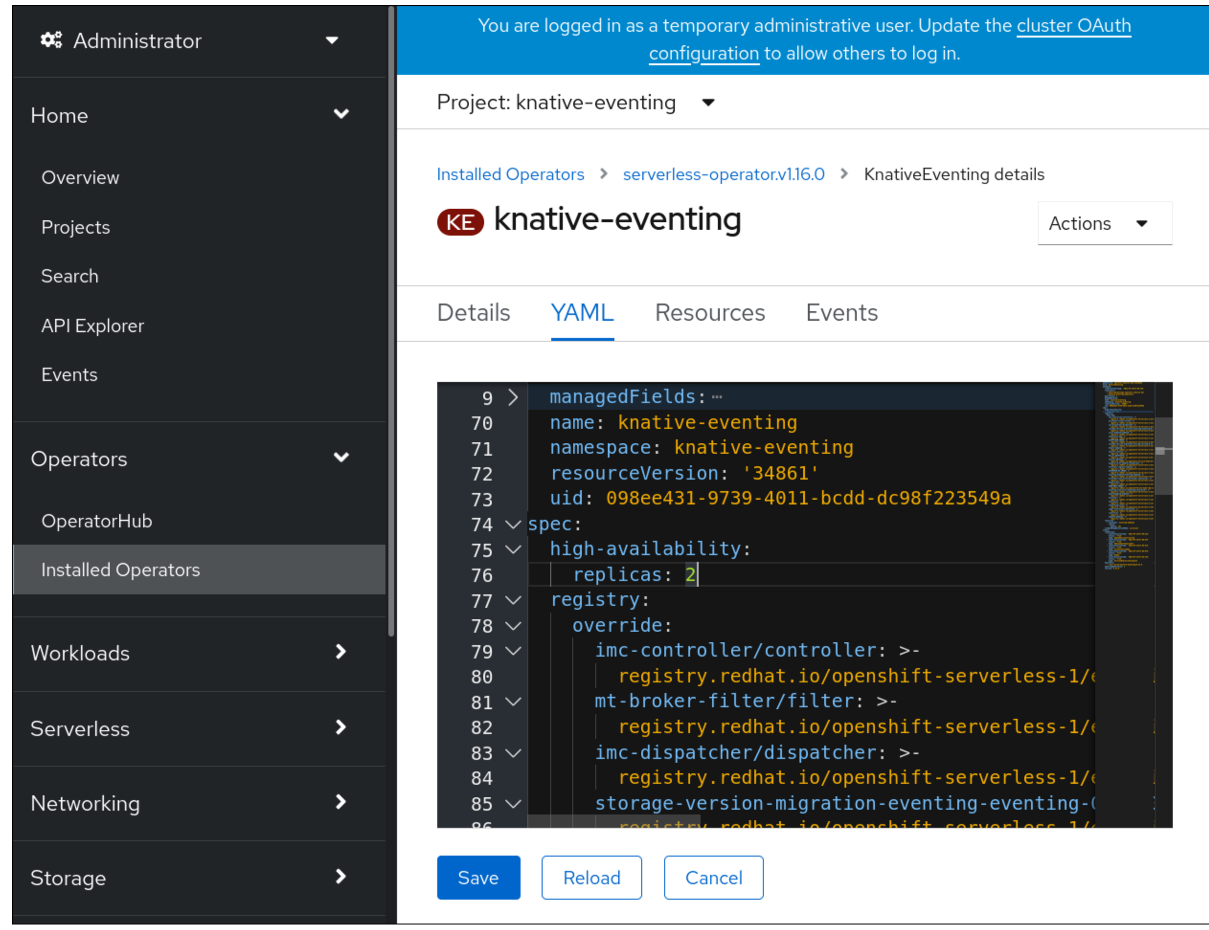
For Knative Eventing, the **mt-broker-filter** and **mt-broker-ingress** deployments are not scaled by HA. If multiple deployments are needed, scale these components manually.

Prerequisites

- You have cluster administrator permissions on OpenShift Container Platform, or you have cluster or dedicated administrator permissions on Red Hat OpenShift Service on AWS or OpenShift Dedicated.
- The OpenShift Serverless Operator and Knative Eventing are installed on your cluster.

Procedure

1. In the OpenShift Container Platform web console **Administrator** perspective, navigate to **OperatorHub → Installed Operators**.
2. Select the **knative-eventing** namespace.
3. Click **Knative Eventing** in the list of **Provided APIs** for the OpenShift Serverless Operator to go to the **Knative Eventing** tab.
4. Click **knative-eventing**, then go to the **YAML** tab in the **knative-eventing** page.

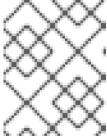


5. Modify the number of replicas in the **KnativeEventing** CR:

Example YAML

```
apiVersion: operator.knative.dev/v1beta1
kind: KnativeEventing
metadata:
  name: knative-eventing
  namespace: knative-eventing
spec:
  high-availability:
    replicas: 3
```

6. You can also specify the number of replicas for a specific workload.



NOTE

Workload-specific configuration overrides the global setting for Knative Eventing.

Example YAML

```
apiVersion: operator.knative.dev/v1beta1
kind: KnativeEventing
metadata:
  name: knative-eventing
  namespace: knative-eventing
spec:
```

```
high-availability:
  replicas: 3
workloads:
- name: mt-broker-filter
  replicas: 3
```

7. Verify that the high availability limits are respected:

Example command

```
$ oc get hpa -n knative-eventing
```

Example output

```
NAME               REFERENCE                    TARGETS   MINPODS   MAXPODS
REPLICAS   AGE
broker-filter-hpa    Deployment/mt-broker-filter    1%/70%    3         12        3          112s
broker-ingress-hpa   Deployment/mt-broker-ingress   1%/70%    3         12        3          112s
eventing-webhook     Deployment/eventing-webhook    4%/100%   3         7         3          115s
```

10.2.2. Configuring high availability replicas for the Knative broker implementation for Apache Kafka

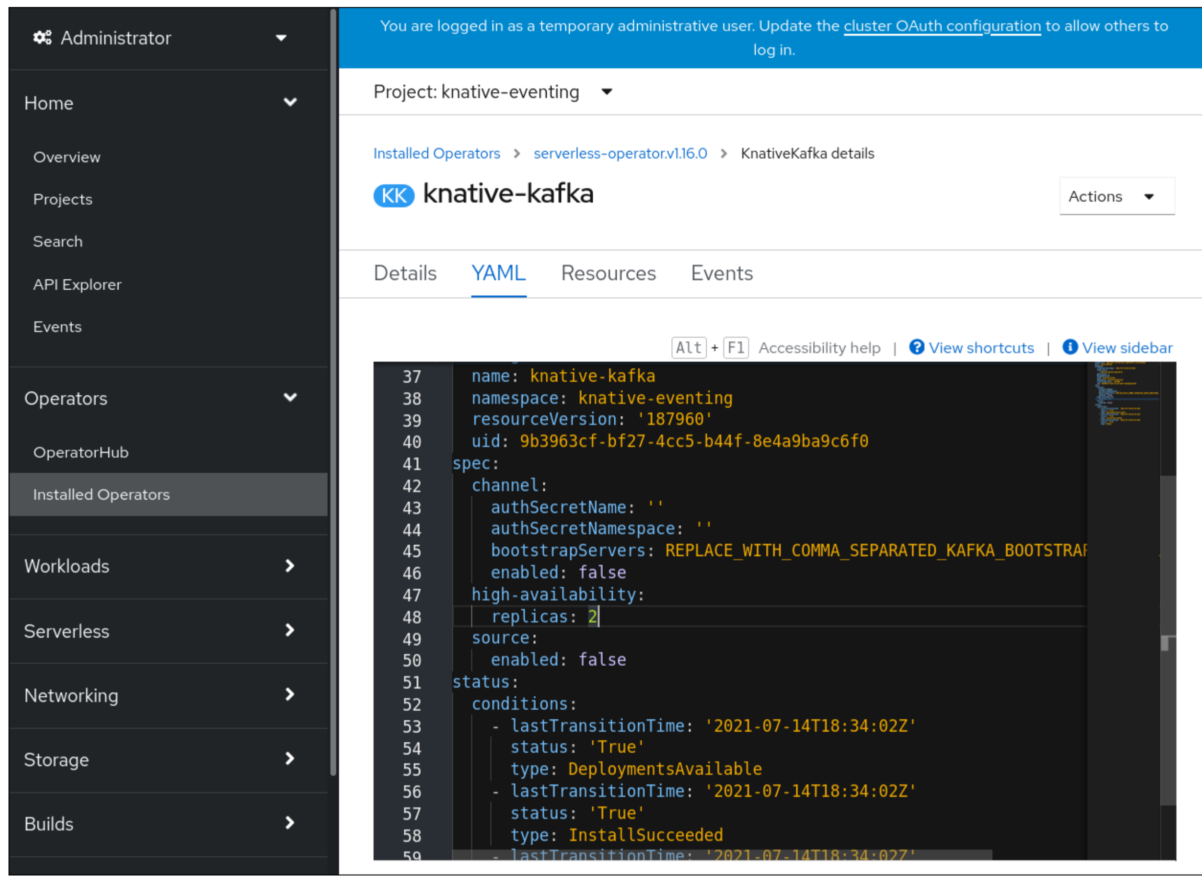
High availability (HA) is available by default for the Knative broker implementation for Apache Kafka components **kafka-controller** and **kafka-webhook-eventing**, which are configured to have two each replicas by default. You can change the number of replicas for these components by modifying the **spec.high-availability.replicas** value in the **KnativeKafka** custom resource (CR).

Prerequisites

- You have cluster administrator permissions on OpenShift Container Platform, or you have cluster or dedicated administrator permissions on Red Hat OpenShift Service on AWS or OpenShift Dedicated.
- The OpenShift Serverless Operator and Knative broker for Apache Kafka are installed on your cluster.

Procedure

1. In the OpenShift Container Platform web console **Administrator** perspective, navigate to **OperatorHub → Installed Operators**.
2. Select the **knative-eventing** namespace.
3. Click **Knative Kafka** in the list of **Provided APIs** for the OpenShift Serverless Operator to go to the **Knative Kafka** tab.
4. Click **knative-kafka**, then go to the **YAML** tab in the **knative-kafka** page.



5. Modify the number of replicas in the **KnativeKafka** CR:

Example YAML

```yaml
apiVersion: operator.serverless.openshift.io/v1alpha1
kind: KnativeKafka
metadata:
  name: knative-kafka
  namespace: knative-eventing
spec:
  high-availability:
    replicas: 3
```

10.2.3. Overriding disruption budgets

A Pod Disruption Budget (PDB) is a standard feature of Kubernetes APIs that helps limit the disruption to an application when its pods need to be rescheduled for maintenance reasons.

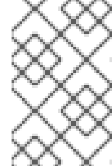
Procedure

- Override the default PDB for a specific resource by modifying the **minAvailable** configuration value in the **KnativeEventing** custom resource (CR).

Example PDB with a minAvailable seting of 70%

```yaml
apiVersion: operator.knative.dev/v1beta1
kind: KnativeEventing
metadata:
 name: knative-eventing
```

```
  namespace: knative-eventing
spec:
  podDisruptionBudgets:
  - name: eventing-webhook
    minAvailable: 70%
```

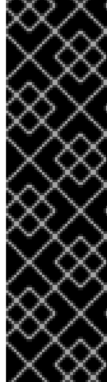


NOTE

If you disable high-availability, for example, by changing the **high-availability.replicas** value to **1**, make sure you also update the corresponding PDB **minAvailable** value to **0**. Otherwise, the pod disruption budget prevents automatic cluster or Operator updates.

CHAPTER 11. CONFIGURING TLS ENCRYPTION IN EVENTING

With the transport encryption feature, you can transport data and events over secured and encrypted HTTPS connections by using *Transport Layer Security* (TLS).



IMPORTANT

OpenShift Serverless transport encryption for Eventing is a Technology Preview feature only. Technology Preview features are not supported with Red Hat production service level agreements (SLAs) and might not be functionally complete. Red Hat does not recommend using them in production. These features provide early access to upcoming product features, enabling customers to test functionality and provide feedback during the development process.

For more information about the support scope of Red Hat Technology Preview features, see Technology Preview Features Support Scope .

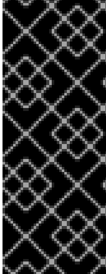
The **transport-encryption** feature flag is an **enum** configuration that defines how Addressables, such as Broker, Channel, and Sink, accept events. It controls whether Addressables must accept events over HTTP or HTTPS based on the selected setting.

The possible values for **transport-encryption** are as follows:

Value	Description
disabled	• Addressables can accept events to HTTPS endpoints. • Producers can send events to HTTPS endpoints.
permissive	• Addressables must accept events on both HTTP and HTTPS endpoints. • Addressables must advertise both HTTP and HTTPS endpoints. • Producers must send events to HTTPS endpoints, if available.
strict	• Addressables must not accept events to non-HTTPS endpoints. • Addressables must only advertise HTTPS endpoints.

11.1. CREATING A SELFSIGNED CLUSTERISSUER RESOURCE FOR EVENTING

ClusterIssuers are Kubernetes resources that represent certificate authorities (CAs) that can generate signed certificates by honoring certificate signing requests. All cert-manager certificates require a referenced issuer in a ready condition to attempt to honor the request. For more details, see Issuer.



IMPORTANT

For simplicity, this procedure uses a **SelfSigned** issuer as the root certificate authority. For more details about **SelfSigned** issuer implications and limitations, see SelfSigned issuers. If you are using a custom public key infrastructure (PKI), you must configure it so its privately signed CA certificates are recognized across the cluster. For more details about cert-manager, see certificate authorities (CAs). You can use any other issuer that is usable for cluster-local services.

Prerequisites

- You have cluster administrator permissions on OpenShift Container Platform, or you have cluster or dedicated administrator permissions on Red Hat OpenShift Service on AWS or OpenShift Dedicated.
- You have installed the OpenShift Serverless Operator.
- You have installed the cert-manager Operator for Red Hat OpenShift.
- You have installed the OpenShift (**oc**) CLI.

Procedure

1. Create a **SelfSigned ClusterIssuer** resource as follows:

```
apiVersion: cert-manager.io/v1
kind: ClusterIssuer
metadata:
  name: knative-eventing-selfsigned-issuer
spec:
  selfSigned: {}
```

2. Apply the **ClusterIssuer** resource by running the following command:

```
$ oc apply -f <filename>
```

3. Create a root certificate by using the **SelfSigned ClusterIssuer** resource as follows:

```
apiVersion: cert-manager.io/v1
kind: Certificate
metadata:
  name: knative-eventing-selfsigned-ca
  namespace: cert-manager (1)
spec:
  secretName: knative-eventing-ca (2)
  isCA: true
  commonName: selfsigned-ca
  privateKey:
    algorithm: ECDSA
    size: 256
```

```
issuerRef:
  name: knative-eventing-selfsigned-issuer
  kind: ClusterIssuer
  group: cert-manager.io
```

1. Specify the cert-manager Operator for Red Hat OpenShift that is used by default.
2. Specify the secret where the certificate is stored. The name is later used by the **ClusterIssuer** for Eventing.

4. Apply the **Certificate** resource by running the following:

```
$ oc apply -f <filename>
```

11.2. CREATING A CLUSTERISSUER RESOURCE FOR EVENTING

ClusterIssuers are Kubernetes resources that represent certificate authorities (CAs) that can generate signed certificates by honoring certificate signing requests.

Prerequisites

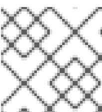
- You have cluster administrator permissions on OpenShift Container Platform, or you have cluster or dedicated administrator permissions on Red Hat OpenShift Service on AWS or OpenShift Dedicated.
- You have installed the OpenShift Serverless Operator.
- You have installed the cert-manager Operator for Red Hat OpenShift.
- You have installed the OpenShift (**oc**) CLI.

Procedure

1. Create the **knative-eventing-ca-issuer ClusterIssuer** resource as follows:
   Every Eventing component uses this issuer to issue their server's certs.

```
apiVersion: cert-manager.io/v1
kind: ClusterIssuer
metadata:
  name: knative-eventing-ca-issuer
spec:
  ca:
    secretName: knative-eventing-ca (1)
```

1. The **secretName** value in the **cert-manager** namespace (default for cert-manager Operator for Red Hat OpenShift) contains the certificate that can be used by Knative Eventing components.



NOTE

The **ClusterIssuer** name must be **knative-eventing-ca-issuer**.

2. Apply the **ClusterIssuer** resource by running the following command:

```
$ oc apply -f <filename>
```

11.3. ENABLING TRANSPORT ENCRPTION FOR KNATIVE EVENTING

You can enable transport encryption in **KnativeEventing** by setting the **transport-encryption** feature to **strict**.

Prerequisites

- You have cluster administrator permissions on OpenShift Container Platform, or you have cluster or dedicated administrator permissions on Red Hat OpenShift Service on AWS or OpenShift Dedicated.
- You have installed the OpenShift Serverless Operator.
- You have installed the cert-manager Operator for Red Hat OpenShift.
- You have installed the OpenShift (**oc**) CLI.

Procedure

1. Enable the **transport-encryption** in **KnativeEventing** as follows:

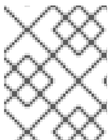
```
apiVersion: operator.knative.dev/v1beta1
kind: KnativeEventing
metadata:
  name: knative-eventing
  namespace: knative-eventing
spec:

  # Other spec fields omitted ...
  # ...

  config:
    features:
      transport-encryption: strict
```

2. Apply the **KnativeEventing** resource by running the following command:

```
$ oc apply -f <filename>
```

11.4. CONFIGURING ADDITIONAL CA TRUST BUNDLES

By default, Eventing clients trust the OpenShift CA bundle configured for custom PKI. For more details, see Configuring a custom PKI.



NOTE

When a new connection is established, Eventing clients automatically include these CA bundles in their trusted list.

Prerequisites

- You have cluster administrator permissions on OpenShift Container Platform, or you have cluster or dedicated administrator permissions on Red Hat OpenShift Service on AWS or OpenShift Dedicated.
- You have installed the OpenShift Serverless Operator.
- You have installed the cert-manager Operator for Red Hat OpenShift.

Procedure

- Create a CA bundle for Eventing as follows:

```
kind: ConfigMap
metadata:
  name: <my_org_eventing_bundle> (1)
  namespace: knative-eventing
  labels:
    networking.knative.dev/trust-bundle: "true"
data: (2)
  ca.crt: ...
  ca1.crt: ...
  tls.crt: ...
```

(1) Use a unique name to avoid conflicts with existing or future Eventing config maps.

(2) All keys with valid PEM-encoded CA bundles are trusted by Eventing clients.

11.5. CONFIGURE CUSTOM EVENT SOURCES TO TRUST THE EVENTING CA

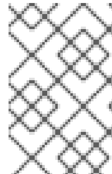
To create a custom event source, use a SinkBinding. The SinkBinding can inject the configured CA trust bundles as a projected volume into each container by using the **knative-custom-certs** directory.

In specific cases, you might inject company-specific CA trust bundles into base container images and automatically configure runtimes, such as OpenJDK or Node.js, and so on. to trust those CA bundles. In such cases, you might not need to configure your clients.

By using the **my_org_eventing_bundle** config map from the previous example, with the **ca.crt**, **ca1.crt**, and **tls.crt** data keys, the **knative-custom-certs** directory has the following layout:

```
/knative-custom-certs/ca.crt
/knative-custom-certs/ca1.crt
/knative-custom-certs/tls.crt
```

You can use these files to add CA trust bundles to HTTP clients that send events to Eventing.



NOTE

Depending on the runtime, programming language, or library you use, different methods exist for configuring custom CA cert files, such as using command-line flags, environment variables, or reading the content of the files.

11.6. ADDING A SELFSIGNED CLUSTERISSUER RESOURCE TO CA TRUST BUNDLES

If you are using a **SelfSigned ClusterIssuer** resource, you can add the CA to the Eventing CA trust bundles.

Prerequisites

- You have cluster administrator permissions on OpenShift Container Platform, or you have cluster or dedicated administrator permissions on Red Hat OpenShift Service on AWS or OpenShift Dedicated.
- You have installed the OpenShift Serverless Operator.
- You have installed the cert-manager Operator for Red Hat OpenShift.
- You have installed the OpenShift (**oc**) CLI.

Procedure

1. Export the CA from the **knative-eventing-ca** secret in the cert-manager Operator for Red Hat OpenShift namespace (default is **cert-manager** certificate) by running the following command:

```
$ oc get secret -n cert-manager knative-eventing-ca -o=jsonpath='{.data.ca\.crt}' | base64 -d
> ca.crt
```

2. Create a CA trust bundle in the **knative-eventing** namespace by running the following command:

```
$ oc create configmap -n knative-eventing my-org-selfsigned-ca-bundle --from-file=ca.crt
```

3. Label the **ConfigMap** by running the following command:

```
$ oc label configmap -n knative-eventing my-org-selfsigned-ca-bundle
networking.knative.dev/trust-bundle=true
```

11.7. ENSURING SEAMLESS CA ROTATION

Ensuring seamless CA rotation is essential to avoid service downtime or to handle emergencies.

Prerequisites

- You have cluster administrator permissions on OpenShift Container Platform, or you have cluster or dedicated administrator permissions on Red Hat OpenShift Service on AWS or OpenShift Dedicated.
- You have installed the OpenShift Serverless Operator.
- You have installed the cert-manager Operator for Red Hat OpenShift.
- You have installed the OpenShift (**oc**) CLI.

Procedure

1. Create a CA certificate.
2. Add the public key of the new CA certificate to the CA trust bundles.
   Ensure that you also keep the public key of the existing CA.
3. Ensure all clients use the latest CA trust bundles.
   Knative Eventing components automatically reload the updated CA trust bundles. For custom workloads that consume trust bundles, reload or restart them as needed.
4. Update the **knative-eventing-ca-issuer ClusterIssuer** to reference the secret containing the CA certificate that you created in step 1.
5. Force **cert-manager** to renew certificates in the **knative-eventing namespace**.
   For more information about **cert-manager**, see Reissuance triggered by user actions.
6. As soon as the CA rotation is fully completed, remove the public key of the old CA from the trust bundle config map.

11.8. VERIFYING TRANSPORT ENCRYPTION IN EVENTING

To confirm that transport encryption is correctly configured, you can create and test an **InMemoryChannel** resource. Follow the steps to ensure that it uses HTTPS as expected.

Prerequisites

- You have cluster administrator permissions on OpenShift Container Platform, or you have cluster or dedicated administrator permissions on Red Hat OpenShift Service on AWS or OpenShift Dedicated.
- You have installed the OpenShift Serverless Operator.
- You have installed the cert-manager Operator for Red Hat OpenShift.
- You have installed the OpenShift (**oc**) CLI.

Procedure

1. Create an **InMemoryChannel** resource as follows:

```
apiVersion: messaging.knative.dev/v1
kind: InMemoryChannel
metadata:
  name: transport-encryption-test
```

2. Apply the **InMemoryChannel** resource by running the following command:

```
$ oc apply -f <filename>
```

3. View the **InMemoryChannel** address by running the following command:

```
$ oc get inmemorychannels.messaging.knative.dev transport-encryption-test
```

Example output

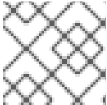
```
NAME                    URL                                                                                      AGE
READY   REASON
transport-encryption-test   https://imc-dispatcher.knative-
eventing.svc.cluster.local/default/transport-encryption-test   17s   True
```

CHAPTER 12. CONFIGURING KUBE-RBAC-PROXY FOR EVENTING

The **kube-rbac-proxy** component provides internal authentication and authorization capabilities for Knative Eventing.

12.1. CONFIGURING KUBE-RBAC-PROXY RESOURCES FOR EVENTING

You can globally override resource allocation for the **kube-rbac-proxy** container by using the OpenShift Serverless Operator CR.



NOTE

You can also override resource allocation for a specific deployment.

The following configuration sets Knative Eventing **kube-rbac-proxy** minimum and maximum CPU and memory allocation:

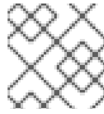
KnativeEventing CR example

```
apiVersion: operator.knative.dev/v1beta1
kind: KnativeEventing
metadata:
  name: knative-eventing
  namespace: knative-eventing
spec:
  config:
    deployment:
      "kube-rbac-proxy-cpu-request": "10m" (1)
      "kube-rbac-proxy-memory-request": "20Mi" (2)
      "kube-rbac-proxy-cpu-limit": "100m" (3)
      "kube-rbac-proxy-memory-limit": "100Mi" (4)
```

1. Sets minimum CPU allocation.
2. Sets minimum RAM allocation.
3. Sets maximum CPU allocation.
4. Sets maximum RAM allocation.

12.2. CONFIGURING KUBE-RBAC-PROXY RESOURCES FOR KNATIVE FOR APACHE KAFKA

You can globally override resource allocation for the **kube-rbac-proxy** container by using the OpenShift Serverless Operator CR.



NOTE

You can also override resource allocation for a specific deployment.

The following configuration sets Knative Kafka **kube-rbac-proxy** minimum and maximum CPU and memory allocation:

KnativeKafka CR example

```
apiVersion: operator.serverless.openshift.io/v1alpha1
kind: KnativeKafka
metadata:
  name: knative-kafka
  namespace: knative-kafka
spec:
  config:
    deployment:
      "kube-rbac-proxy-cpu-request": "10m" (1)
      "kube-rbac-proxy-memory-request": "20Mi" (2)
      "kube-rbac-proxy-cpu-limit": "100m" (3)
      "kube-rbac-proxy-memory-limit": "100Mi" (4)
```

1. Sets minimum CPU allocation.
2. Sets minimum RAM allocation.
3. Sets maximum CPU allocation.
4. Sets maximum RAM allocation.

CHAPTER 13. USING CONTAINERSOURCE WITH SERVICE MESH

You can use container source with Service Mesh.

13.1. CONFIGURING CONTAINERSOURCE WITH SERVICE MESH

This procedure describes how to configure container source with Service Mesh.

Prerequisites

- You have set up integration of Service Mesh and Serverless.

Procedure

1. Create a **Service** in a namespace that is member of the **ServiceMeshMemberRoll**:

   Example event-display-service.yaml configuration file

   ```
   apiVersion: serving.knative.dev/v1
   kind: Service
   metadata:
     name: event-display
     namespace: <namespace> 1
   spec:
     template:
       metadata:
         annotations:
           sidecar.istio.io/inject: "true" 2
           sidecar.istio.io/rewriteAppHTTPProbers: "true"
       spec:
         containers:
         - image: quay.io/openshift-knative/knative-eventing-sources-event-display:latest
   ```

   1 A namespace that is a member of the **ServiceMeshMemberRoll**.

   2 This annotation injects Service Mesh sidecars into the Knative service pods.

2. Apply the **Service** resource:

   ```
   $ oc apply -f event-display-service.yaml
   ```

3. Create a **ContainerSource** object in a namespace that is member of the **ServiceMeshMemberRoll** and sink set to the **event-display**:

   Example test-heartbeats-containersource.yaml configuration file

   ```
   apiVersion: sources.knative.dev/v1
   kind: ContainerSource
   metadata:
     name: test-heartbeats
     namespace: <namespace> 1
   ```

```
spec:
  template:
    metadata: (2)
      annotations:
        sidecar.istio.io/inject": "true"
        sidecar.istio.io/rewriteAppHTTPProbers: "true"
    spec:
      containers:
        # This corresponds to a heartbeats image URI that you have built and published
        - image: quay.io/openshift-knative/heartbeats
          name: heartbeats
          args:
            - --period=1s
          env:
            - name: POD_NAME
              value: "example-pod"
            - name: POD_NAMESPACE
              value: "event-test"
  sink:
    ref:
      apiVersion: serving.knative.dev/v1
      kind: Service
      name: event-display-service
```

1. A namespace that is part of the **ServiceMeshMemberRoll**.
2. These annotations enable Service Mesh integration with the **ContainerSource** object.

4. Apply the **ContainerSource** resource:

```
$ oc apply -f test-heartbeats-containersource.yaml
```

5. Optional: Verify that the events were sent to the Knative event sink by looking at the message dumper function logs:

Example command

```
$ oc logs $(oc get pod -o name | grep event-display) -c user-container
```

Example output

```
☁  cloudevents.Event
Validation: valid
Context Attributes,
  specversion: 1.0
  type: dev.knative.eventing.samples.heartbeat
  source: https://knative.dev/eventing-contrib/cmd/heartbeats/#event-test/mypod
  id: 2b72d7bf-c38f-4a98-a433-608fbcdd2596
  time: 2019-10-18T15:23:20.809775386Z
  contenttype: application/json
Extensions,
  beats: true
  heart: yes
  the: 42
```

```
Data,
  {
    "id": 1,
    "label": ""
  }
```

CHAPTER 14. USING A SINK BINDING WITH SERVICE MESH

You can use a sink binding with Service Mesh.

14.1. CONFIGURING A SINK BINDING WITH SERVICE MESH

This procedure describes how to configure a sink binding with Service Mesh.

Prerequisites

- You have set up integration of Service Mesh and Serverless.

Procedure

1. Create a **Service** object in a namespace that is member of the **ServiceMeshMemberRoll**:

   Example event-display-service.yaml configuration file

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: event-display
  namespace: <namespace> (1)
spec:
  template:
    metadata:
      annotations:
        sidecar.istio.io/inject: "true" (2)
        sidecar.istio.io/rewriteAppHTTPProbers: "true"
    spec:
      containers:
      - image: quay.io/openshift-knative/knative-eventing-sources-event-display:latest
```

   (1) A namespace that is a member of the **ServiceMeshMemberRoll**.

   (2) This annotation injects Service Mesh sidecars into the Knative service pods.

2. Apply the **Service** object:

```
$ oc apply -f event-display-service.yaml
```

3. Create a **SinkBinding** object:

   Example heartbeat-sinkbinding.yaml configuration file

```
apiVersion: sources.knative.dev/v1alpha1
kind: SinkBinding
metadata:
  name: bind-heartbeat
  namespace: <namespace> (1)
spec:
  subject:
```

```
    apiVersion: batch/v1
    kind: Job 2
    selector:
      matchLabels:
        app: heartbeat-cron

  sink:
    ref:
      apiVersion: serving.knative.dev/v1
      kind: Service
      name: event-display
```

1 A namespace that is part of the **ServiceMeshMemberRoll**.

2 Bind any Job with the label **app: heartbeat-cron** to the event sink.

4. Apply the **SinkBinding** object:

```
$ oc apply -f heartbeat-sinkbinding.yaml
```

5. Create a **CronJob** object:

Example heartbeat-cronjob.yaml configuration file

```
apiVersion: batch/v1
kind: CronJob
metadata:
  name: heartbeat-cron
  namespace: <namespace> 1
spec:
  # Run every minute
  schedule: "* * * * *"
  jobTemplate:
    metadata:
      labels:
        app: heartbeat-cron
        bindings.knative.dev/include: "true"
    spec:
      template:
        metadata:
          annotations:
            sidecar.istio.io/inject: "true" 2
            sidecar.istio.io/rewriteAppHTTPProbers: "true"
        spec:
          restartPolicy: Never
          containers:
            - name: single-heartbeat
              image: quay.io/openshift-knative/heartbeats:latest
              args:
                - --period=1
              env:
                - name: ONE_SHOT
                  value: "true"
                - name: POD_NAME
```

```
            valueFrom:
              fieldRef:
                fieldPath: metadata.name
          - name: POD_NAMESPACE
            valueFrom:
              fieldRef:
                fieldPath: metadata.namespace
```

1. A namespace that is part of the **ServiceMeshMemberRoll**.
2. Inject Service Mesh sidecars into the **CronJob** pods.

6. Apply the **CronJob** object:

```
$ oc apply -f heartbeat-cronjob.yaml
```

7. Optional: Verify that the events were sent to the Knative event sink by looking at the message dumper function logs:

Example command

```
$ oc logs $(oc get pod -o name | grep event-display) -c user-container
```

Example output

```
☁  cloudevents.Event
Validation: valid
Context Attributes,
  specversion: 1.0
  type: dev.knative.eventing.samples.heartbeat
  source: https://knative.dev/eventing-contrib/cmd/heartbeats/#event-test/mypod
  id: 2b72d7bf-c38f-4a98-a433-608fbcdd2596
  time: 2019-10-18T15:23:20.809775386Z
  contenttype: application/json
Extensions,
  beats: true
  heart: yes
  the: 42
Data,
  {
    "id": 1,
    "label": ""
  }
```