OpenShift Container Platform 4.7

Serverless

OpenShift Serverless installation, usage, and release notes

Last Updated: 2021-05-21

OpenShift Container Platform 4.7 Serverless

OpenShift Serverless installation, usage, and release notes

Abstract

This document provides information on how to use OpenShift Serverless in OpenShift Container Platform.

Table of Contents

CHAPTER 1. OPENSHIFT SERVERLESS RELEASE NOTES

For an overview of OpenShift Serverless functionality, see Getting started with OpenShift Serverless.

1.1. RELEASE NOTES FOR RED HAT OPENSHIFT SERVERLESS 1.14.0

1.1.1. New features

- OpenShift Serverless now uses Knative Serving 0.20.0.
- OpenShift Serverless uses Knative Eventing 0.20.0.
- OpenShift Serverless now uses Kourier 0.20.0.
- OpenShift Serverless now uses Knative **kn** CLI 0.20.0.
- OpenShift Serverless now uses Knative Kafka 0.20.0.
- Knative Kafka on OpenShift Serverless is now Generally Available (GA).

  IMPORTANT

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

- The Operator channel for installing and upgrading OpenShift Serverless has been updated to **stable** for OpenShift Container Platform 4.6 and newer versions.
- OpenShift Serverless is now supported on IBM Power Systems, IBM Z, and LinuxONE, except for the following features, which are not yet supported:
  - Knative Kafka functionality.
  - OpenShift Serverless Functions developer preview.

1.1.2. Known issues

- Subscriptions for the Kafka channel sometimes fail to become marked as **READY** and remain in the **SubscriptionNotMarkedReadyByChannel** state. You can fix this by restarting the dispatcher for the Kafka channel.

1.2. RELEASE NOTES FOR RED HAT OPENSHIFT SERVERLESS 1.13.0

1.2.1. New features

- OpenShift Serverless now uses Knative Serving 0.19.0.
- OpenShift Serverless uses Knative Eventing 0.19.2.
- OpenShift Serverless now uses Kourier 0.19.0.
- OpenShift Serverless now uses Knative **kn** CLI 0.19.1.

- OpenShift Serverless now uses Knative Kafka 0.19.1.
- A **DomainMapping** custom resource (CR) has been added to OpenShift Serverless to enable users to map a custom domain name to a Knative Service. See the Knative documentation on Creating a mapping between a custom domain name and a Knative Service .
- In Knative Serving 0.19.0, **v1alpha1** and **v1beta1** versions of the **Service**, **Route**, **Configuration**, and **Revision** resources have been removed. The OpenShift Serverless Operator automatically upgrades older resources to **v1**, so no user action is required.

NOTE

New resources must not be created as **v1alpha1** or **v1beta1** versions, since this can cause errors and these resources will not be upgraded automatically.

1.3. RELEASE NOTES FOR RED HAT OPENSHIFT SERVERLESS 1.12.0

1.3.1. New features

- OpenShift Serverless now uses Knative Serving 0.18.2.
- OpenShift Serverless uses Knative Eventing 0.18.6.
- OpenShift Serverless now uses Kourier 0.18.0.
- OpenShift Serverless now uses Knative **kn** CLI 0.18.4.
- OpenShift Serverless now uses Knative Kafka 0.18.0.

1.3.2. Fixed issues

- In previous versions, if you used a ping source with OpenShift Serverless, after you uninstalled and deleted all other Knative Eventing components, the **pingsource-jobrunner** deployment was not deleted. This issue is now fixed, and the **pingsource-jobrunner** deployment has been renamed to **pingsource-mt-adapter**.
- In previous versions, deleting a sink before you delete the **SinkBinding** resource connected to it caused the resource deletion to hang. This issue is now fixed.

1.3.3. Known issues

- Using the **eventing.knative.dev/scope: namespace** annotation for the **KafkaChannel** objects is not supported.

1.4. RELEASE NOTES FOR RED HAT OPENSHIFT SERVERLESS 1.11.0

1.4.1. New features

- Knative Eventing on OpenShift Serverless is now Generally Available (GA).
- Knative Kafka features such as Kafka channel and Kafka event source are now available as a Technology Preview on OpenShift Serverless. Kafka integration is delivered through the OpenShift Serverless Operator and does not require a separate community Operator

installation.

- OpenShift Serverless Functions is now delivered as a Developer Preview through the standard Knative **kn** CLI installation. This feature is not yet supported by Red Hat for production deployments, but can be used for development and testing. For more information about using OpenShift Serverless Functions through the **kn func** CLI, see the OpenShift Serverless Functions Developer Preview documentation.
- OpenShift Serverless now uses Knative Serving 0.17.3.
- OpenShift Serverless uses Knative Eventing 0.17.2.
- OpenShift Serverless now uses Kourier 0.17.0.
- OpenShift Serverless now uses Knative **kn** CLI 0.17.3.
- OpenShift Serverless now uses Knative Kafka 0.17.1.

1.4.2. Known issues

- When the horizontal pod autoscaler (HPA) scales up the **broker-ingress** pod, the **imc-dispatcher** pod sometimes fails to forward replies. This is because the new **broker-ingress** pods are **Ready** before accepting connections, because they lack a readiness probe. If you are using HPA autoscaling and do not want to scale the **broker-ingress** pod manually, you must configure retries in the **Broker.Spec.Delivery**.
- Using the **eventing.knative.dev/scope: namespace** annotation with Kafka channels is not supported.

1.5. RELEASE NOTES FOR RED HAT OPENSHIFT SERVERLESS 1.10.0

1.5.1. New features

- OpenShift Serverless now uses Knative Operator 0.16.0.
- OpenShift Serverless now uses Knative Serving 0.16.0.
- OpenShift Serverless uses Knative Eventing 0.16.0.
- OpenShift Serverless now uses Kourier 0.16.0.
- OpenShift Serverless now uses Knative **kn** CLI 0.16.1.
- The annotation **knative-eventing-injection=enabled** that was previously used to label namespaces for broker creation is now deprecated. The new annotation is **eventing.knative.dev/injection=enabled**. For more information, see the documentation on *Brokers and triggers*.
- Multi-container support is now available on Knative as a Technology Preview feature. You can enable multi-container support in the **config-features** config map. For more information, see the Knative documentation.

1.5.2. Fixed issues

- In previous releases, Knative Serving had a fixed, minimum CPU request of 25m for **queue-**

proxy. If your cluster had any value set that conflicted with this, for example, if you had set a minimum CPU request for **defaultRequest** of more than **25m**, the Knative Service failed to deploy. This issue is fixed in 1.10.0.

1.6. ADDITIONAL RESOURCES

OpenShift Serverless is based on the open source Knative project.

- For details about the latest Knative Serving release, see the Knative Serving releases page.
- For details about the latest Knative Serving Operator release, see the Knative Serving Operator releases page.
- For details about the latest Knative CLI release, see the Knative CLI releases page.
- For details about the latest Knative Eventing release, see the Knative Eventing releases page.
- OpenShift Serverless Functions Developer Preview documentation.

CHAPTER 2. OPENSHIFT SERVERLESS SUPPORT

2.1. GETTING SUPPORT

If you experience difficulty with a procedure described in this documentation, visit the Red Hat Customer Portal at http://access.redhat.com. Through the customer portal, you can:

- Search or browse through the Red Hat Knowledgebase of technical support articles about Red Hat products
- Submit a support case to Red Hat Global Support Services (GSS)
- Access other product documentation

If you have a suggestion for improving this guide or have found an error, please submit a Bugzilla report at http://bugzilla.redhat.com against **Product** for the **Documentation** component. Please provide specific details, such as the section number, guide name, and OpenShift Serverless version so we can easily locate the content.

2.2. GATHERING DIAGNOSTIC INFORMATION FOR SUPPORT

When opening a support case, it is helpful to provide debugging information about your cluster to Red Hat Support.

The **must-gather** tool enables you to collect diagnostic information about your OpenShift Container Platform cluster, including data related to OpenShift Serverless.

For prompt support, supply diagnostic information for both OpenShift Container Platform and OpenShift Serverless.

2.2.1. About the must-gather tool

The **oc adm must-gather** CLI command collects the information from your cluster that is most likely needed for debugging issues, such as:

- Resource definitions
- Audit logs
- Service logs

You can specify one or more images when you run the command by including the **--image** argument. When you specify an image, the tool collects data related to that feature or product.

When you run **oc adm must-gather**, a new pod is created on the cluster. The data is collected on that pod and saved in a new directory that starts with **must-gather.local**. This directory is created in the current working directory.

2.2.2. About collecting OpenShift Serverless data

You can use the **oc adm must-gather** CLI command to collect information about your cluster, including features and objects associated with OpenShift Serverless. To collect OpenShift Serverless data with **must-gather**, you must specify the OpenShift Serverless image and the image tag for your installed version of OpenShift Serverless.

Procedure

- Collect data by using the **oc adm must-gather** command:

```
$ oc adm must-gather --image=registry.redhat.io/openshift-serverless-1/svls-must-gather-
rhel8:<image_version_tag>
```

Example command

```
$ oc adm must-gather --image=registry.redhat.io/openshift-serverless-1/svls-must-gather-
rhel8:1.14.0
```

CHAPTER 3. GETTING STARTED WITH OPENSHIFT SERVERLESS

OpenShift Serverless simplifies the process of delivering code from development into production by reducing the need for infrastructure set up or back-end development by developers.

3.1. HOW OPENSHIFT SERVERLESS WORKS

Developers on OpenShift Serverless can use the provided Kubernetes native APIs, as well as familiar languages and frameworks, to deploy applications and container workloads.

OpenShift Serverless on OpenShift Container Platform enables stateless, serverless workloads to all run on a single multi-cloud container platform with automated operations. Developers can use a single platform for hosting their microservices, legacy, and serverless applications.

OpenShift Serverless is based on the open source Knative project, which provides portability and consistency across hybrid and multi-cloud environments by enabling an enterprise-grade serverless platform.

3.2. SUPPORTED CONFIGURATIONS

The set of supported features, configurations, and integrations for OpenShift Serverless, current and past versions, are available at the Supported Configurations page .

3.3. NEXT STEPS

- Install the OpenShift Serverless Operator on your OpenShift Container Platform cluster to get started.
- View the OpenShift Serverless release notes .

CHAPTER 4. INSTALLING THE KNATIVE CLI

NOTE

The Knative CLI (**kn**) does not have its own login mechanism. To log in to the cluster, you must install the **oc** CLI and use the **oc login** command.

Installation options for the **oc** CLI will vary depending on your operating system.

For more information on installing the **oc** CLI for your operating system and logging in with **oc**, see the OpenShift CLI getting started documentation.

4.1. INSTALLING THE KNATIVE CLI USING THE OPENSHIFT CONTAINER PLATFORM WEB CONSOLE

Once the OpenShift Serverless Operator is installed, you will see a link to download the Knative CLI (**kn**) for Linux (x86_64, amd64, s390x, ppc64le), macOS, or Windows from the **Command Line Tools** page in the OpenShift Container Platform web console.

You can access the **Command Line Tools** page by clicking the ? icon in the top right corner of the web console and selecting **Command Line Tools** in the drop down menu.

Procedure

1. Download the **kn** CLI from the **Command Line Tools** page.
2. Unpack the archive:

```
$ tar -xf <file>
```

3. Move the **kn** binary to a directory on your **PATH**.
4. To check your **PATH**, run:

```
$ echo $PATH
```

NOTE

If you do not use RHEL or Fedora, ensure that **libc** is installed in a directory on your library path. If **libc** is not available, you might see the following error when you run CLI commands:

```
$ kn: No such file or directory
```

4.2. INSTALLING THE KNATIVE CLI FOR LINUX USING AN RPM

For Red Hat Enterprise Linux (RHEL), you can install the Knative CLI (**kn**) as an RPM if you have an active OpenShift Container Platform subscription on your Red Hat account.

Procedure

1. Enter the command:

   ```
   # subscription-manager register
   ```

2. Enter the command:

   ```
   # subscription-manager refresh
   ```

3. Enter the command:

   ```
   # subscription-manager attach --pool=<pool_id> 1
   ```

   1 Pool ID for an active OpenShift Container Platform subscription

4. Enter the command:

   ```
   # subscription-manager repos --enable="openshift-serverless-1-for-rhel-8-x86_64-rpms"
   ```

5. Enter the command:

   ```
   # yum install openshift-serverless-clients
   ```

4.3. INSTALLING THE KNATIVE CLI FOR LINUX

For Linux distributions, you can download the Knative CLI (**kn**) directly as a **tar.gz** archive.

Procedure

1. Download the **kn** CLI.

2. Unpack the archive:

   ```
   $ tar -xf <file>
   ```

3. Move the **kn** binary to a directory on your **PATH**.

4. To check your **PATH**, run:

   ```
   $ echo $PATH
   ```

   NOTE

   If you do not use RHEL or Fedora, ensure that **libc** is installed in a directory on your library path. If **libc** is not available, you might see the following error when you run CLI commands:

   ```
   $ kn: No such file or directory
   ```

4.4. INSTALLING THE KNATIVE CLI FOR LINUX ON IBM POWER SYSTEMS USING AN RPM

For Red Hat Enterprise Linux (RHEL), you can install the Knative CLI (**kn**) as an RPM if you have an active OpenShift Container Platform subscription on your Red Hat account.

Procedure

1. Register with a Red Hat Subscription Management (RHSM) service during the firstboot process:

   ```
   # subscription-manager register
   ```

2. Refresh the RHSM:

   ```
   # subscription-manager refresh
   ```

3. Attach the subscription to a system by specifying ID of the subscription pool, using the **--pool** option:

   ```
   # subscription-manager attach --pool=<pool_id> 1
   ```

   1 Pool ID for an active OpenShift Container Platform subscription

4. Enable the repository using Red Hat Subscription Manager:

   ```
   # subscription-manager repos --enable="openshift-serverless-1-for-rhel-8-ppc64le-rpms"
   ```

5. Install the **openshift-serverless-clients** on the system:

   ```
   # yum install openshift-serverless-clients
   ```

4.5. INSTALLING THE KNATIVE CLI FOR LINUX ON IBM POWER SYSTEMS

For Linux distributions, you can download the Knative CLI (**kn**) directly as a **tar.gz** archive.

Procedure

1. Download the **kn** CLI.

2. Unpack the archive:

   ```
   $ tar -xf <file>
   ```

3. Move the **kn** binary to a directory on your **PATH**.

4. To check your **PATH**, run:

   ```
   $ echo $PATH
   ```

NOTE

If you do not use RHEL, ensure that **libc** is installed in a directory on your library path.

If **libc** is not available, you might see the following error when you run CLI commands:

```
$ kn: No such file or directory
```

4.6. INSTALLING THE KNATIVE CLI FOR LINUX ON IBM Z AND LINUXONE USING AN RPM

For Red Hat Enterprise Linux (RHEL), you can install the Knative CLI (**kn**) as an RPM if you have an active OpenShift Container Platform subscription on your Red Hat account.

Procedure

1. Register with a Red Hat Subscription Management (RHSM) service during the firstboot process:

```
# subscription-manager register
```

2. Refresh the RHSM:

```
# subscription-manager refresh
```

3. Attach the subscription to a system by specifying ID of the subscription pool, using the **--pool** option:

```
# subscription-manager attach --pool=<pool_id> 1
```

1 Pool ID for an active OpenShift Container Platform subscription

4. Enable the repository using Red Hat Subscription Manager:

```
# subscription-manager repos --enable="openshift-serverless-1-for-rhel-8-s390x-rpms"
```

5. Install the **openshift-serverless-clients** on the system:

```
# yum install openshift-serverless-clients
```

4.7. INSTALLING THE KNATIVE CLI FOR LINUX ON IBM Z AND LINUXONE

For Linux distributions, you can download the Knative CLI (**kn**) directly as a **tar.gz** archive.

Procedure

1. Download the **kn** CLI.

2. Unpack the archive:

```
$ tar -xf <file>
```

3. Move the **kn** binary to a directory on your **PATH**.

4. To check your **PATH**, run:

```
$ echo $PATH
```

NOTE

If you do not use RHEL, ensure that **libc** is installed in a directory on your library path.

If **libc** is not available, you might see the following error when you run CLI commands:

```
$ kn: No such file or directory
```

4.8. INSTALLING THE KNATIVE CLI FOR MACOS

The Knative CLI (**kn**) for macOS is provided as a **tar.gz** archive.

Procedure

1. Download the **kn** CLI.

2. Unpack and unzip the archive.

3. Move the **kn** binary to a directory on your **PATH**.

4. To check your **PATH**, open a terminal window and run:

```
$ echo $PATH
```

4.9. INSTALLING THE KNATIVE CLI FOR WINDOWS

The Knative CLI (**kn**) for Windows is provided as a zip archive.

Procedure

1. Download the **kn** CLI.

2. Extract the archive with a ZIP program.

3. Move the **kn** binary to a directory on your **PATH**.

4. To check your **PATH**, open the command prompt and run the command:

```
C:\> path
```

4.10. CUSTOMIZING KN

You can customize your **kn** CLI setup by creating a **config.yaml** configuration file. You can provide this configuration by using the **--config** flag, otherwise the configuration is picked up from a default location. The default configuration location conforms to the XDG Base Directory Specification, and is different for Unix systems and Windows systems.

For Unix systems:

- If the **XDG_CONFIG_HOME** environment variable is set, the default configuration location that the **kn** CLI looks for is **$XDG_CONFIG_HOME/kn**.
- If the **XDG_CONFIG_HOME** environment variable is not set, the **kn** CLI looks for the configuration in the home directory of the user at **$HOME/.config/kn/config.yaml**.

For Windows systems, the default **kn** CLI configuration location is **%APPDATA%\kn**.

Example configuration file

```
plugins:
  path-lookup: true (1)
  directory: ~/.config/kn/plugins (2)
eventing:
  sink-mappings: (3)
  - prefix: svc (4)
    group: core (5)
    version: v1 (6)
    resource: services (7)
```

(1) Specifies whether the **kn** CLI should look for plug-ins in the **PATH** environment variable. This is a boolean configuration option. The default value is **false**.

(2) Specifies the directory where the **kn** CLI will look for plug-ins. The default path depends on the operating system, as described above. This can be any directory that is visible to the user.

(3) The **sink-mappings** spec defines the Kubernetes addressable resource that is used when you use the **--sink** flag with a **kn** CLI command.

(4) The prefix you want to use to describe your sink. **svc** for a service, **channel**, and **broker** are predefined prefixes in **kn**.

(5) The API group of the Kubernetes resource.

(6) The version of the Kubernetes resource.

(7) The plural name of the Kubernetes resource type. For example, **services** or **brokers**.

4.11. KN PLUG-INS

The **kn** CLI supports the use of plug-ins, which enable you to extend the functionality of your **kn** installation by adding custom commands and other shared commands that are not part of the core distribution. **kn** CLI plug-ins are used in the same way as the main **kn** functionality.

Currently, Red Hat supports the **kn-source-kafka** plug-in.

CHAPTER 5. ADMINISTRATION GUIDE

5.1. INSTALLING OPENSHIFT SERVERLESS

This guide walks cluster administrators through installing the OpenShift Serverless Operator to an OpenShift Container Platform cluster.

NOTE

OpenShift Serverless is supported for installation in a restricted network environment. For more information, see Using Operator Lifecycle Manager on restricted networks .

IMPORTANT

Before upgrading to the latest Serverless release, you must remove the community Knative Eventing Operator if you have previously installed it. Having the Knative Eventing Operator installed will prevent you from being able to install the latest version of Knative Eventing using the OpenShift Serverless Operator.

5.1.1. Defining cluster size requirements for an OpenShift Serverless installation

To install and use OpenShift Serverless, the OpenShift Container Platform cluster must be sized correctly. The minimum requirement for OpenShift Serverless is a cluster with 10 CPUs and 40GB memory. The total size requirements to run OpenShift Serverless are dependent on the applications deployed. By default, each pod requests approximately 400m of CPU, so the minimum requirements are based on this value. In the size requirement provided, an application can scale up to 10 replicas. Lowering the actual CPU request of applications can increase the number of possible replicas.

NOTE

The requirements provided relate only to the pool of worker machines of the OpenShift Container Platform cluster. Master nodes are not used for general scheduling and are omitted from the requirements.

NOTE

The following limitations apply to all OpenShift Serverless deployments:

- Maximum number of Knative services: 1000
- Maximum number of Knative revisions: 1000

5.1.2. Additional requirements for advanced use cases

For more advanced use cases such as logging or metering on OpenShift Container Platform, you must deploy more resources. Recommended requirements for such use cases are 24 CPUs and 96GB of memory.

If you have high availability (HA) enabled on your cluster, this requires between 0.5 - 1.5 cores and between 200MB - 2GB of memory for each replica of the Knative Serving control plane. HA is enabled for some Knative Serving components by default. You can disable HA by following the documentation on Configuring high availability replicas on OpenShift Serverless .

5.1.3. Scaling your cluster using machine sets

You can use the OpenShift Container Platform **MachineSet** API to manually scale your cluster up to the desired size. The minimum requirements usually mean that you must scale up one of the default machine sets by two additional machines. See Manually scaling a machine set .

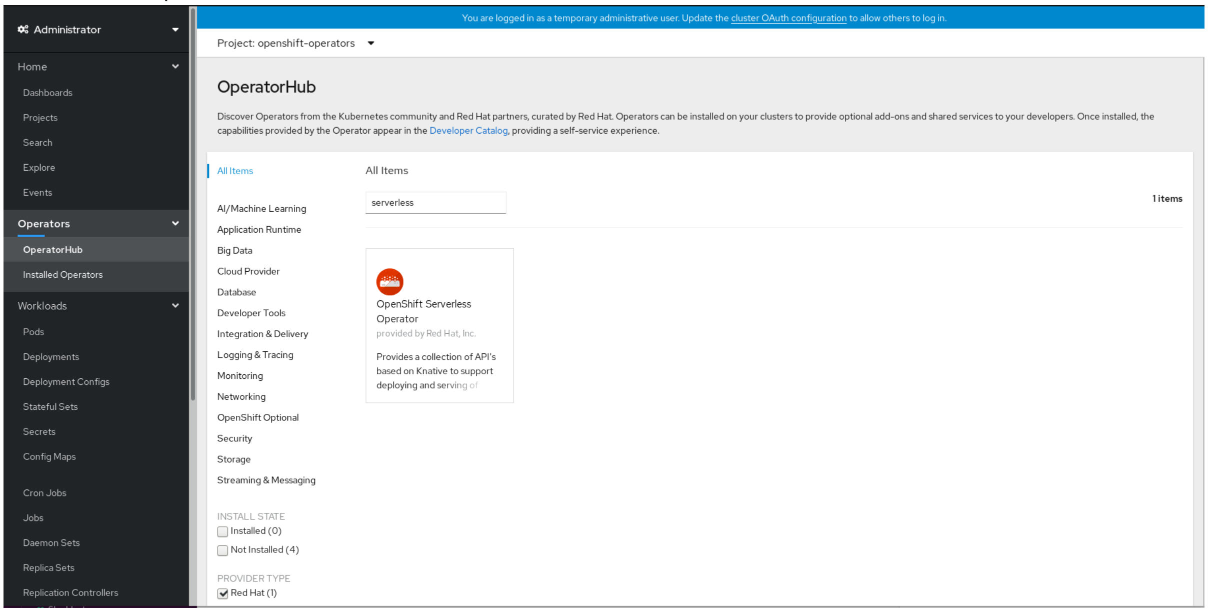
5.1.4. Installing the OpenShift Serverless Operator

This procedure describes how to install and subscribe to the OpenShift Serverless Operator from the OperatorHub using the OpenShift Container Platform web console.

Procedure

1. In the OpenShift Container Platform web console, navigate to the **Operators → OperatorHub** page.

2. Scroll, or type they keyword **Serverless** into the **Filter by keyword box** to find the OpenShift Serverless Operator.



You are logged in as a temporary administrative user. Update the cluster OAuth configuration to allow others to log in.

Discover Operators from the Kubernetes community and Red Hat partners, curated by Red Hat. Operators can be installed on your clusters to provide optional add-ons and shared services to your developers. Once installed, the capabilities provided by the Operator appear in the Developer Catalog, providing a self-service experience.

Provides a collection of API's based on Knative to support deploying and serving of

3. Review the information about the Operator and click **Install**.

4. On the **Install Operator** page:

    a. The **Installation Mode** is **All namespaces on the cluster (default)** This mode installs the Operator in the default **openshift-serverless** namespace to watch and be made available to all namespaces in the cluster.

    b. The **Installed Namespace** will be **openshift-serverless**.

    c. Select the **stable** channel as the **Update Channel**. The **stable** channel will enable installation of the latest stable release of the OpenShift Serverless Operator.

    d. Select **Automatic** or **Manual** approval strategy.

5. Click **Install** to make the Operator available to the selected namespaces on this OpenShift Container Platform cluster.

6. From the **Catalog → Operator Management** page, you can monitor the OpenShift Serverless Operator subscription's installation and upgrade progress.

a. If you selected a **Manual** approval strategy, the subscription's upgrade status will remain **Upgrading** until you review and approve its install plan. After approving on the **Install Plan** page, the subscription upgrade status moves to **Up to date**.

b. If you selected an **Automatic** approval strategy, the upgrade status should resolve to **Up to date** without intervention.

Verification

After the Subscription's upgrade status is **Up to date**, select **Catalog → Installed Operators** to verify that the OpenShift Serverless Operator eventually shows up and its **Status** ultimately resolves to **InstallSucceeded** in the relevant namespace.

If it does not:

1. Switch to the **Catalog → Operator Management** page and inspect the **Operator Subscriptions** and **Install Plans** tabs for any failure or errors under **Status**.

2. Check the logs in any pods in the **openshift-serverless** project on the **Workloads → Pods** page that are reporting issues to troubleshoot further.

5.1.5. Next steps

- After the OpenShift Serverless Operator is installed, you can install the Knative Serving component. See the documentation on Installing Knative Serving.

- After the OpenShift Serverless Operator is installed, you can install the Knative Eventing component. See the documentation on Installing Knative Eventing.

5.2. INSTALLING KNATIVE SERVING

After you install the OpenShift Serverless Operator, you can install Knative Serving. This guide provides information about installing Knative Serving using the default settings. However, you can configure more advanced settings in the **KnativeServing** custom resource definition (CRD). For more information about configuration options for the **KnativeServing** CRD, see Advanced installation configuration options.

5.2.1. Prerequisites

- You have access to an OpenShift Container Platform account with cluster administrator access.

- You have installed the OpenShift Serverless Operator.

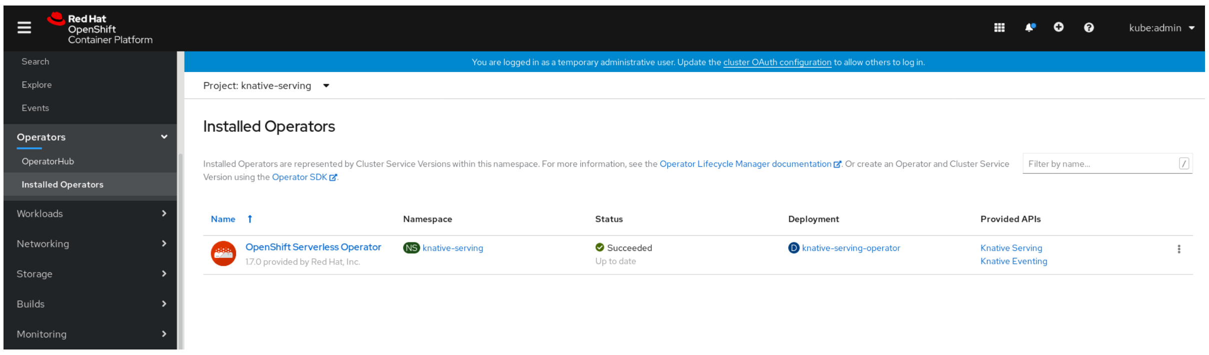
5.2.2. Installing Knative Serving using the web console

Procedure

1. In the **Administrator** perspective of the OpenShift Container Platform web console, navigate to **Operators → Installed Operators**.

2. Check that the **Project** dropdown at the top of the page is set to **Project: knative-serving**.

3. Click **Knative Serving** in the list of **Provided APIs** for the OpenShift Serverless Operator to go to the **Knative Serving** tab.



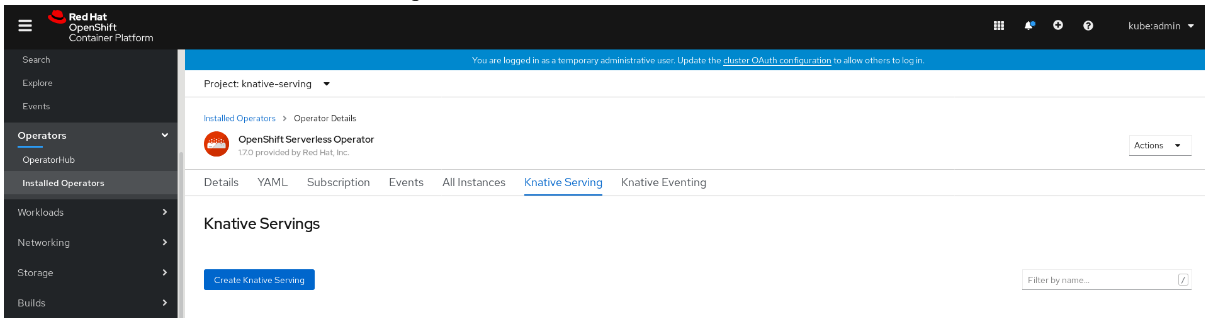
You are logged in as a temporary administrative user. Update the cluster OAuth configuration to allow others to log in.

Installed Operators

Installed Operators are represented by Cluster Service Versions within this namespace. For more information, see the Operator Lifecycle Manager documentation. Or create an Operator and Cluster Service Version using the Operator SDK.

Name	Namespace	Status	Deployment	Provided APIs
OpenShift Serverless Operator 1.7.0 provided by Red Hat, Inc.	NS knative-serving	Succeeded Up to date	D knative-serving-operator	Knative Serving Knative Eventing

4. Click the **Create Knative Serving** button.



You are logged in as a temporary administrative user. Update the cluster OAuth configuration to allow others to log in.

OpenShift Serverless Operator
1.7.0 provided by Red Hat, Inc.

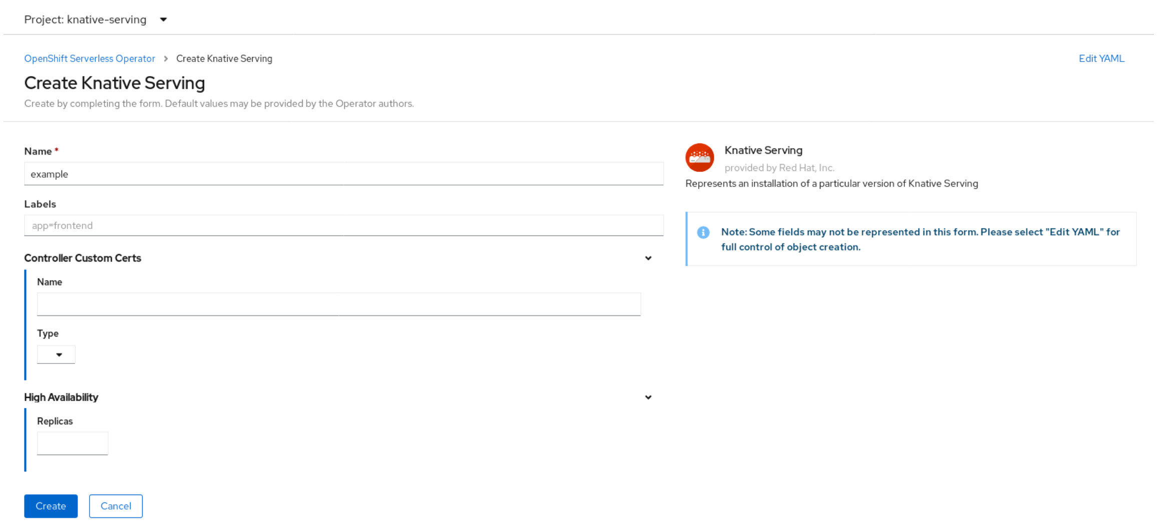
Knative Servings

5. In the **Create Knative Serving** page, you can install Knative Serving using the default settings by clicking **Create**.
   You can also modify settings for the Knative Serving installation by editing the **KnativeServing** object using either the form provided, or by editing the YAML.

   - Using the form is recommended for simpler configurations that do not require full control of **KnativeServing** object creation.
   - Editing the YAML is recommended for more complex configurations that require full control of **KnativeServing** object creation. You can access the YAML by clicking the **edit YAML** link in the top right of the **Create Knative Serving** page.
     After you complete the form, or have finished modifying the YAML, click **Create**.

NOTE

For more information about configuration options for the KnativeServing custom resource definition, see the documentation on *Advanced installation configuration options*.



Create Knative Serving

Create by completing the form. Default values may be provided by the Operator authors.

Name *: example

Labels: app=frontend

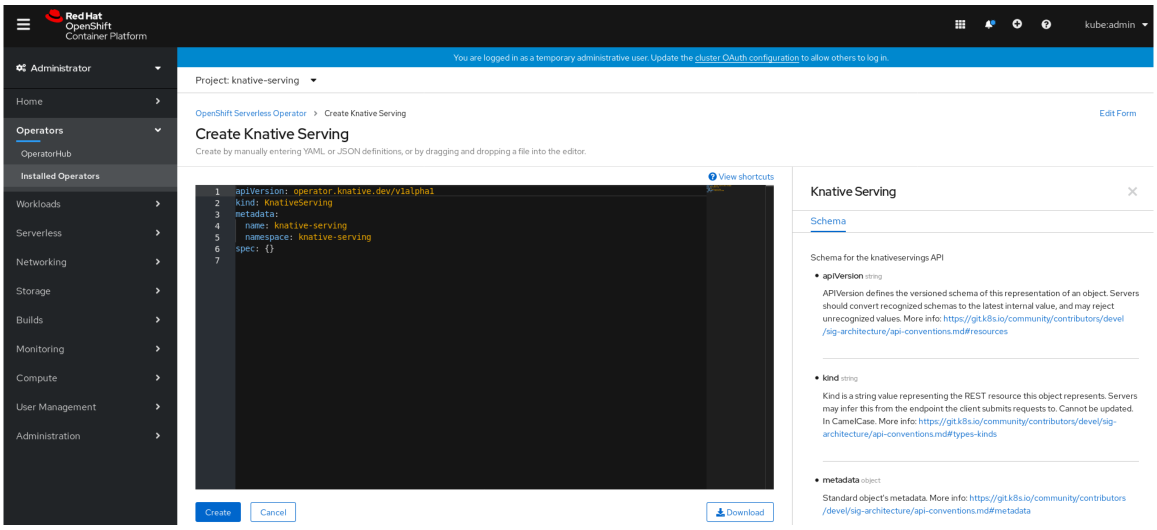
Name:

Type:

Replicas:

Knative Serving
provided by Red Hat, Inc.
Represents an installation of a particular version of Knative Serving

Note: Some fields may not be represented in this form. Please select "Edit YAML" for full control of object creation.



You are logged in as a temporary administrative user. Update the cluster OAuth configuration to allow others to log in.

Project: knative-serving

OpenShift Serverless Operator > Create Knative Serving

Create Knative Serving

Create by manually entering YAML or JSON definitions, or by dragging and dropping a file into the editor.

```
apiVersion: operator.knative.dev/v1alpha1
kind: KnativeServing
metadata:
  name: knative-serving
  namespace: knative-serving
spec: {}
```

Knative Serving

Schema

Schema for the knativeservings API

- **apiVersion** string

  APIVersion defines the versioned schema of this representation of an object. Servers should convert recognized schemas to the latest internal value, and may reject unrecognized values. More info: https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#resources

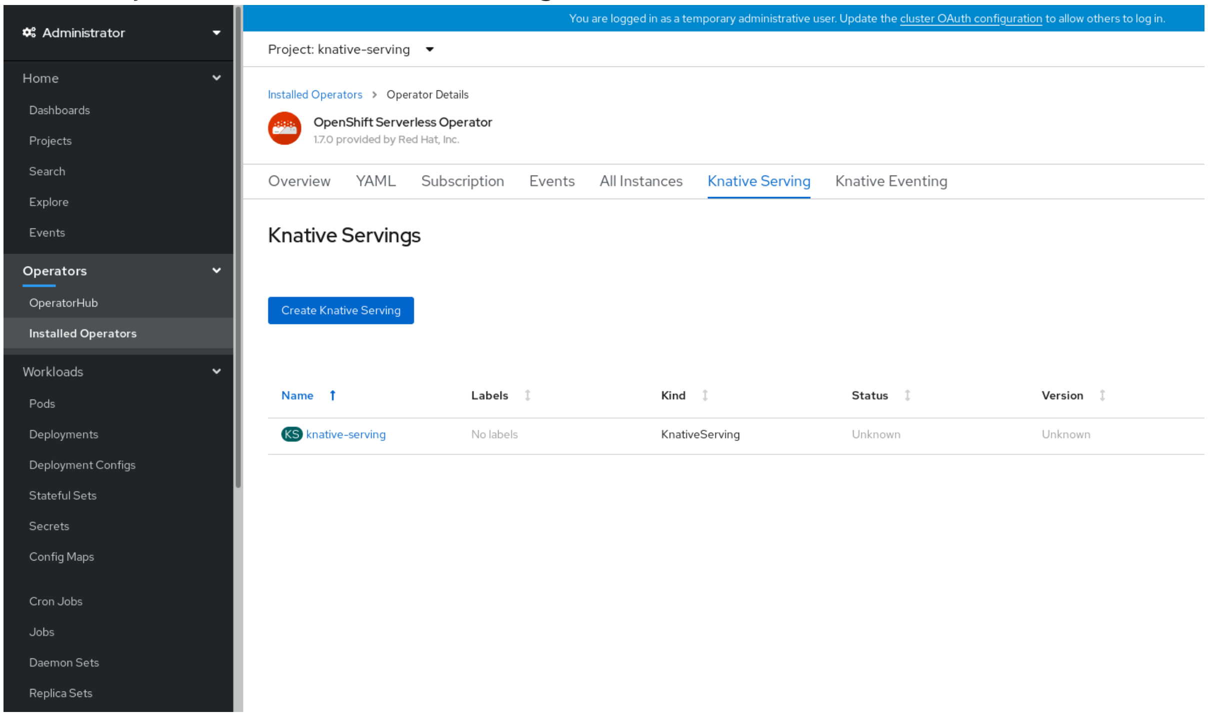
- **kind** string

  Kind is a string value representing the REST resource this object represents. Servers may infer this from the endpoint the client submits requests to. Cannot be updated. In CamelCase. More info: https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#types-kinds

- **metadata** object

  Standard object's metadata. More info: https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#metadata

6. After you have installed Knative Serving, the **KnativeServing** object is created, and you will be automically directed to the **Knative Serving** tab.



You are logged in as a temporary administrative user. Update the cluster OAuth configuration to allow others to log in.

Project: knative-serving

Installed Operators > Operator Details

OpenShift Serverless Operator
1.7.0 provided by Red Hat, Inc.

Knative Servings

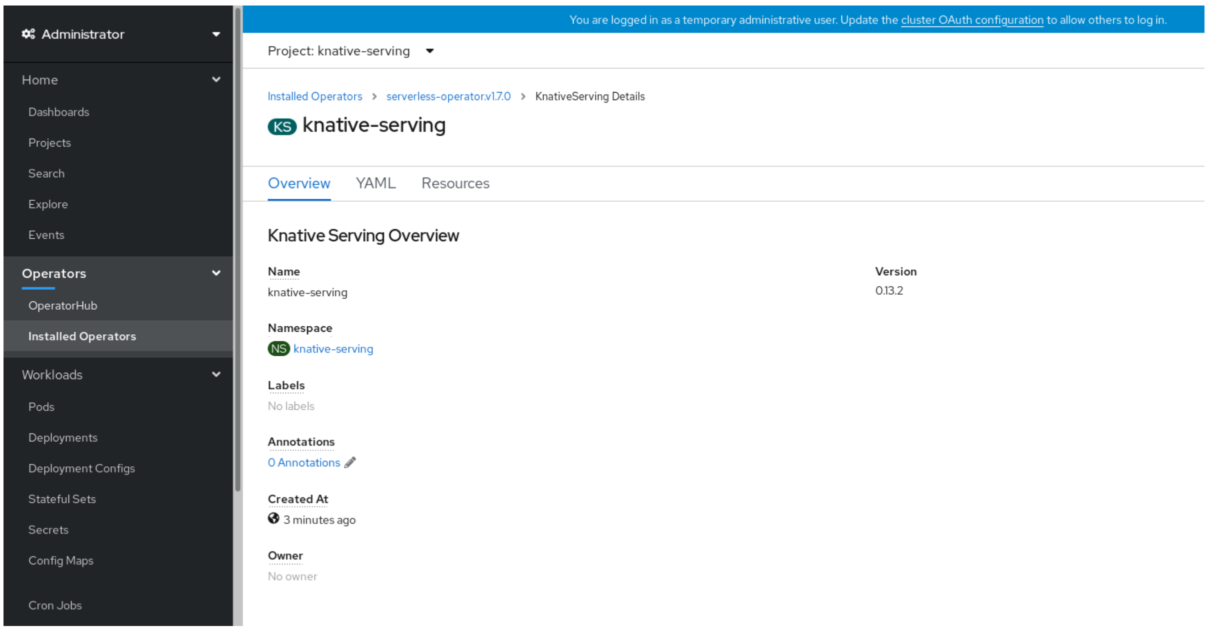
Name	Labels	Kind	Status	Version
KS knative-serving	No labels	KnativeServing	Unknown	Unknown

You will see the **knative-serving** custom resource in the list of resources.

Verification

1. Click on **knative-serving** custom resource in the **Knative Serving** tab.

2. You will be automatically directed to the **Knative Serving Overview** page.



Knative Serving Overview

Name: knative-serving

Version: 0.13.2

Namespace: NS knative-serving

Labels: No labels

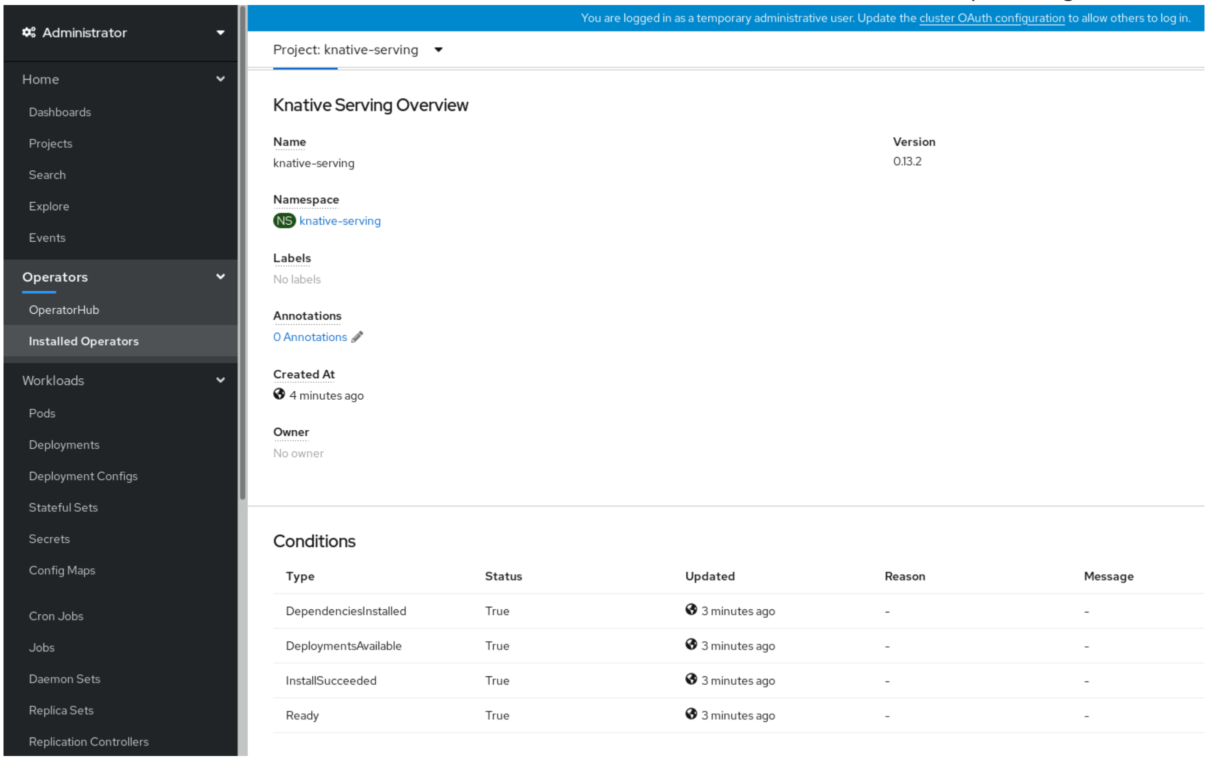
Annotations: 0 Annotations

Created At: 3 minutes ago

Owner: No owner

3. Scroll down to look at the list of **Conditions**.

4. You should see a list of conditions with a status of **True**, as shown in the example image.



Knative Serving Overview

Name: knative-serving

Version: 0.13.2

Namespace: NS knative-serving

Labels: No labels

Annotations: 0 Annotations

Created At: 4 minutes ago

Owner: No owner

Conditions

Type	Status	Updated	Reason	Message
DependenciesInstalled	True	3 minutes ago	-	-
DeploymentsAvailable	True	3 minutes ago	-	-
InstallSucceeded	True	3 minutes ago	-	-
Ready	True	3 minutes ago	-	-

NOTE

It may take a few seconds for the Knative Serving resources to be created. You can check their status in the **Resources** tab.

5. If the conditions have a status of **Unknown** or **False**, wait a few moments and then check again after you have confirmed that the resources have been created.

5.2.3. Installing Knative Serving using YAML

Procedure

1. Create a file named **serving.yaml** and copy the following example YAML into it:

```
apiVersion: operator.knative.dev/v1alpha1
kind: KnativeServing
metadata:
    name: knative-serving
    namespace: knative-serving
```

2. Apply the **serving.yaml** file:

```
$ oc apply -f serving.yaml
```

Verification

1. To verify the installation is complete, enter the following command:

```
$ oc get knativeserving.operator.knative.dev/knative-serving -n knative-serving --
template='{{range .status.conditions}}{{printf "%s=%s\n" .type .status}}{{end}}'
```

Example output

```
DependenciesInstalled=True
DeploymentsAvailable=True
InstallSucceeded=True
Ready=True
```

NOTE

It may take a few seconds for the Knative Serving resources to be created.

2. If the conditions have a status of **Unknown** or **False**, wait a few moments and then check again after you have confirmed that the resources have been created.

3. Check that the Knative Serving resources have been created by entering:

```
$ oc get pods -n knative-serving
```

Example output

```
NAME                               READY   STATUS    RESTARTS   AGE
activator-5c596cf8d6-5l86c         1/1     Running   0          9m37s
activator-5c596cf8d6-gkn5k         1/1     Running   0          9m22s
autoscaler-5854f586f6-gj597        1/1     Running   0          9m36s
autoscaler-hpa-78665569b8-qmlmn    1/1     Running   0          9m26s
autoscaler-hpa-78665569b8-tqwvw    1/1     Running   0          9m26s
controller-7fd5655f49-9gxz5        1/1     Running   0          9m32s
controller-7fd5655f49-pncv5        1/1     Running   0          9m14s
kn-cli-downloads-8c65d4cbf-mt4t7   1/1     Running   0          9m42s
webhook-5c7d878c7c-n267j           1/1     Running   0          9m35s
```

5.2.4. Next steps

- For cloud events functionality on OpenShift Serverless, you can install the Knative Eventing component. See the documentation on Installing Knative Eventing .
- Install the Knative CLI to use **kn** commands with Knative Serving. For example, **kn service** commands. See the documentation on Installing the Knative CLI (**kn**).

5.3. INSTALLING KNATIVE EVENTING

After you install the OpenShift Serverless Operator, you can install Knative Eventing by following the procedures described in this guide.

This guide provides information about installing Knative Eventing using the default settings.

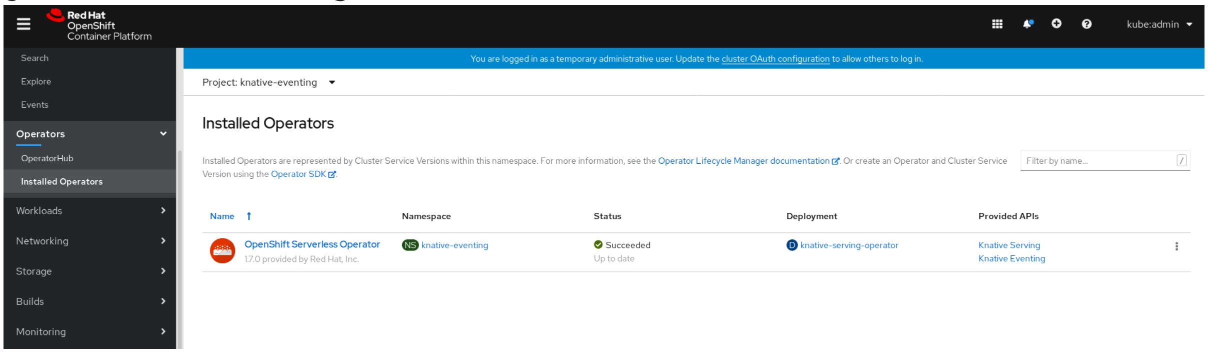
5.3.1. Prerequisites

- You have access to an OpenShift Container Platform account with cluster administrator access.
- You have installed OpenShift Serverless Operator.

5.3.2. Installing Knative Eventing using the web console

Procedure

1. In the **Administrator** perspective of the OpenShift Container Platform web console, navigate to **Operators** → **Installed Operators**.
2. Check that the **Project** dropdown at the top of the page is set to **Project: knative-eventing**.
3. Click **Knative Eventing** in the list of **Provided APIs** for the OpenShift Serverless Operator to go to the **Knative Eventing** tab.



You are logged in as a temporary administrative user. Update the cluster OAuth configuration to allow others to log in.

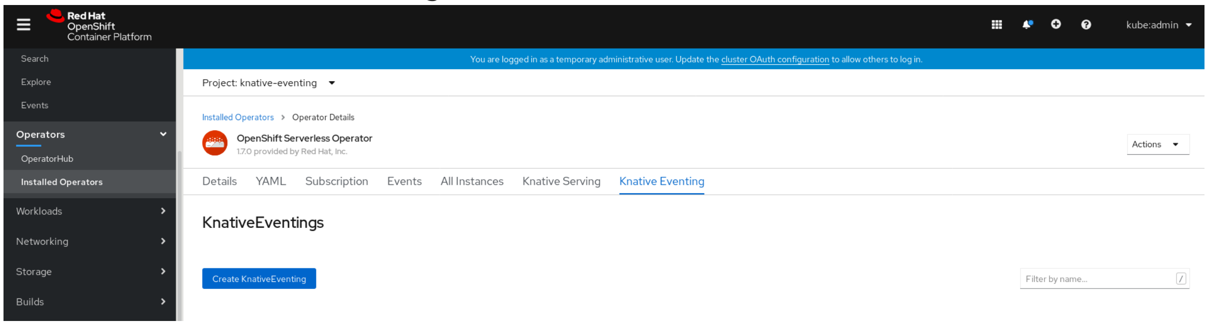
Project: knative-eventing

Installed Operators

Installed Operators are represented by Cluster Service Versions within this namespace. For more information, see the Operator Lifecycle Manager documentation. Or create an Operator and Cluster Service Version using the Operator SDK.

Name	Namespace	Status	Deployment	Provided APIs
OpenShift Serverless Operator 1.7.0 provided by Red Hat, Inc.	NS knative-eventing	Succeeded Up to date	D knative-serving-operator	Knative Serving Knative Eventing

4. Click the **Create Knative Eventing** button.



You are logged in as a temporary administrative user. Update the cluster OAuth configuration to allow others to log in.

Project: knative-eventing

Installed Operators > Operator Details

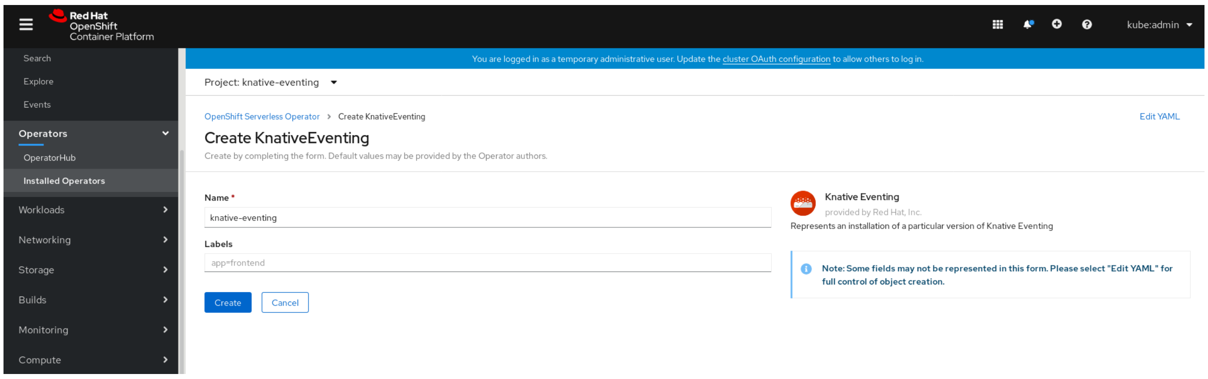
OpenShift Serverless Operator
1.7.0 provided by Red Hat, Inc.

KnativeEventings

5. In the **Create Knative Eventing** page, you can choose to configure the **KnativeEventing** object by using either the default form provided, or by editing the YAML.

- Using the form is recommended for simpler configurations that do not require full control of **KnativeEventing** object creation.
  Optional. If you are configuring the **KnativeEventing** object using the form, make any changes that you want to implement for your Knative Eventing deployment.

6. Click **Create**.



You are logged in as a temporary administrative user. Update the cluster OAuth configuration to allow others to log in.

Project: knative-eventing

OpenShift Serverless Operator > Create KnativeEventing

Create KnativeEventing

Create by completing the form. Default values may be provided by the Operator authors.

Name * knative-eventing

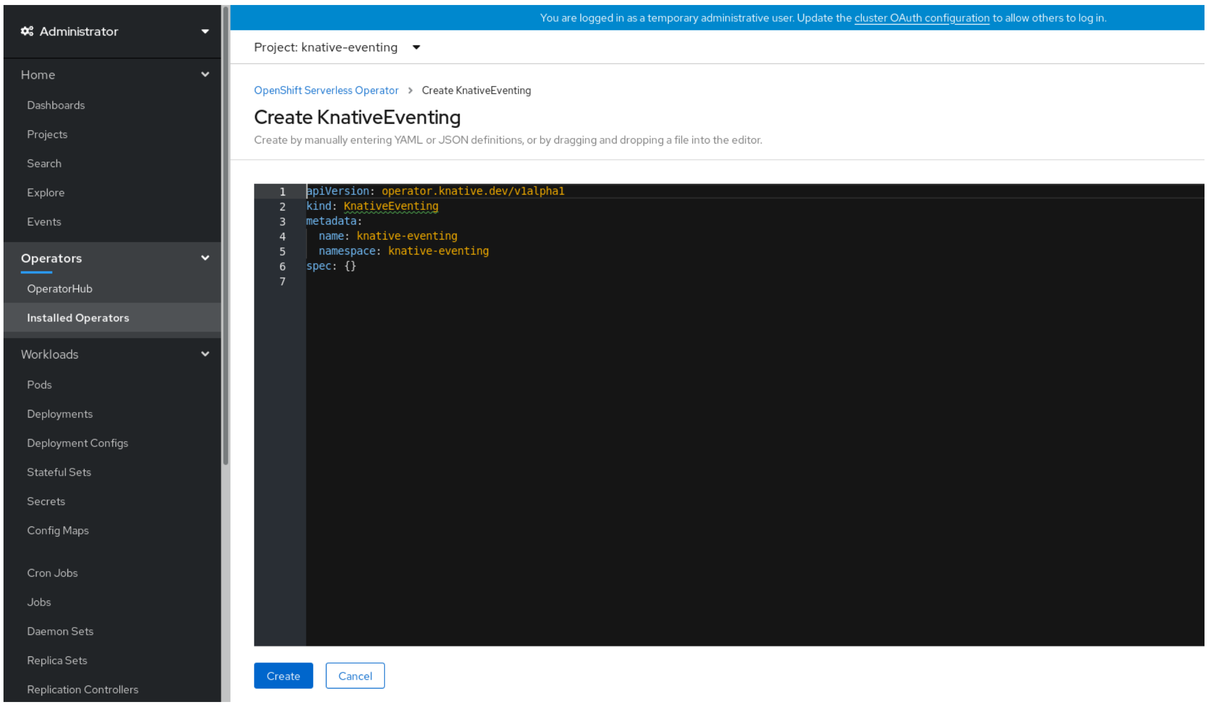
Labels app=frontend

Knative Eventing
provided by Red Hat, Inc.
Represents an installation of a particular version of Knative Eventing

Note: Some fields may not be represented in this form. Please select "Edit YAML" for full control of object creation.

- Editing the YAML is recommended for more complex configurations that require full control of **KnativeEventing** object creation. You can access the YAML by clicking the **edit YAML** link in the top right of the **Create Knative Eventing** page.
  Optional. If you are configuring the **KnativeEventing** object by editing the YAML, make any changes to the YAML that you want to implement for your Knative Eventing deployment.

7. Click **Create**.



You are logged in as a temporary administrative user. Update the cluster OAuth configuration to allow others to log in.

Project: knative-eventing

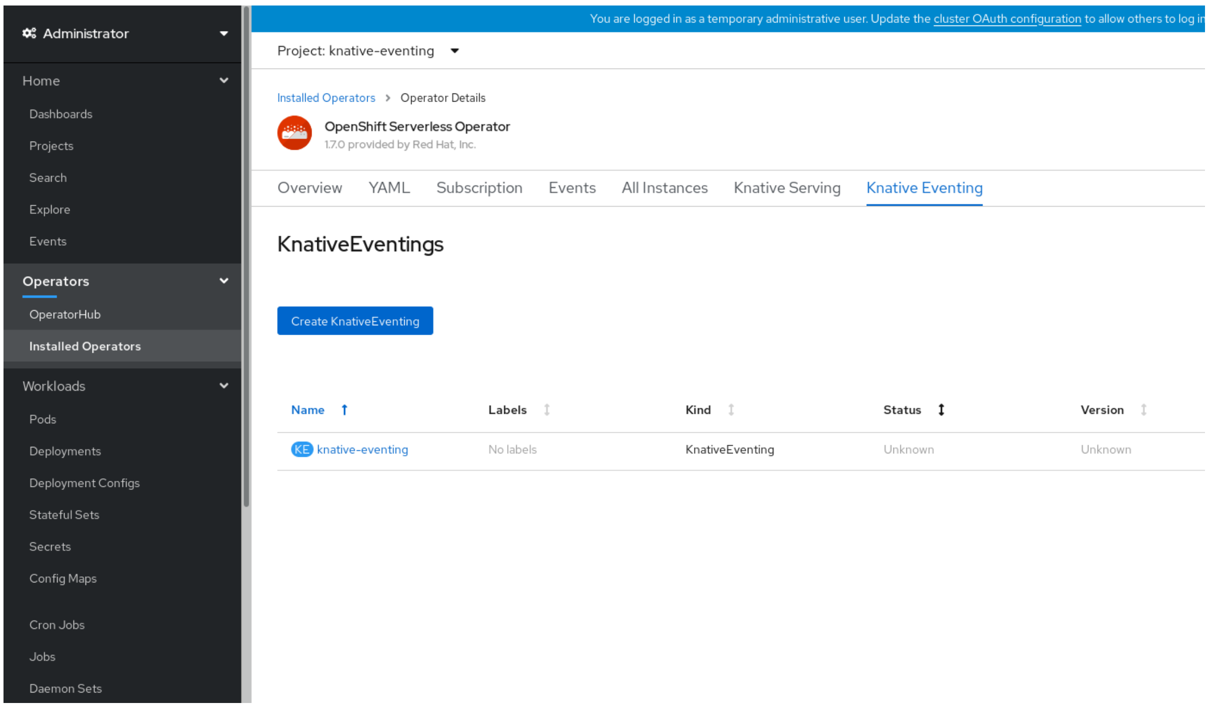
OpenShift Serverless Operator > Create KnativeEventing

Create KnativeEventing

Create by manually entering YAML or JSON definitions, or by dragging and dropping a file into the editor.

```
apiVersion: operator.knative.dev/v1alpha1
kind: KnativeEventing
metadata:
  name: knative-eventing
  namespace: knative-eventing
spec: {}
```

8. After you have installed Knative Eventing, the **KnativeEventing** object is created, and you will be automically directed to the **Knative Eventing** tab.

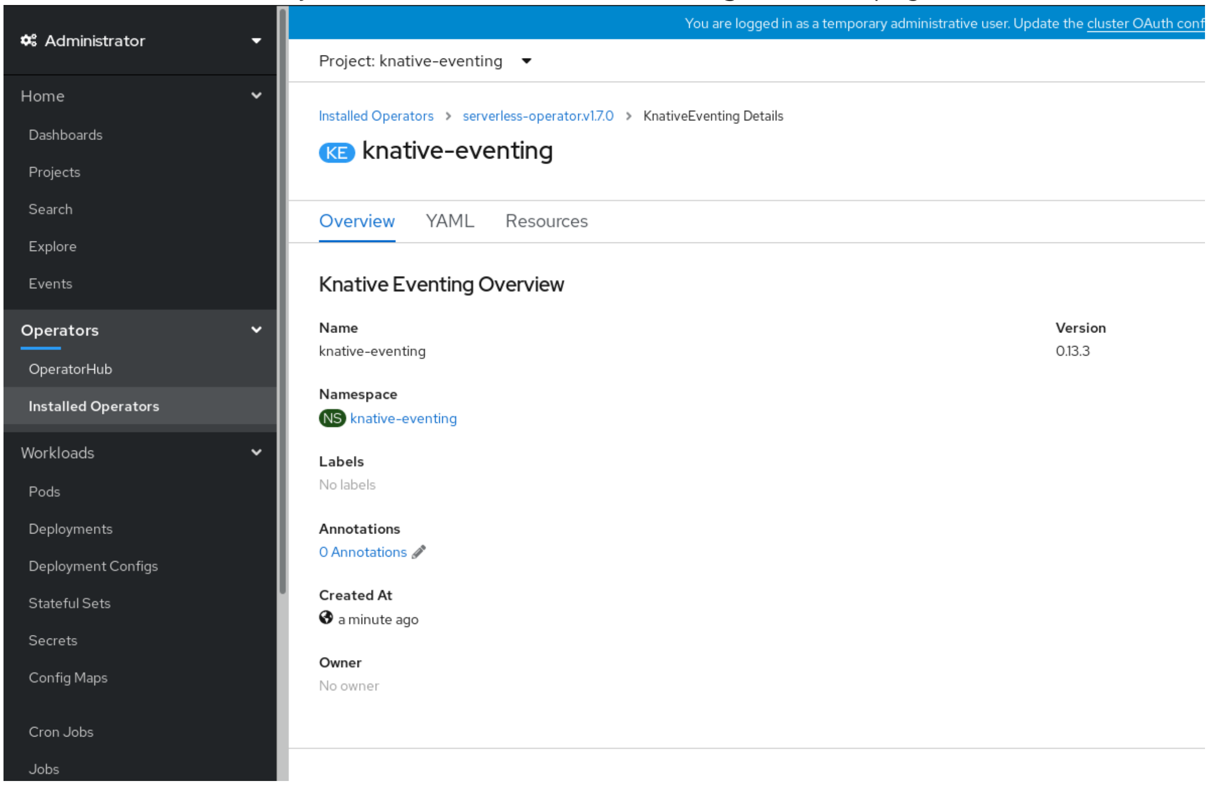


Name	Labels	Kind	Status	Version
KE knative-eventing	No labels	KnativeEventing	Unknown	Unknown

You will see the **knative-eventing** custom resource in the list of resources.

Verification

1. Click on the **knative-eventing** custom resource in the **Knative Eventing** tab.
2. You will be automatically directed to the **Knative Eventing Overview** page.



Name
knative-eventing

Version
0.13.3

Namespace
NS knative-eventing

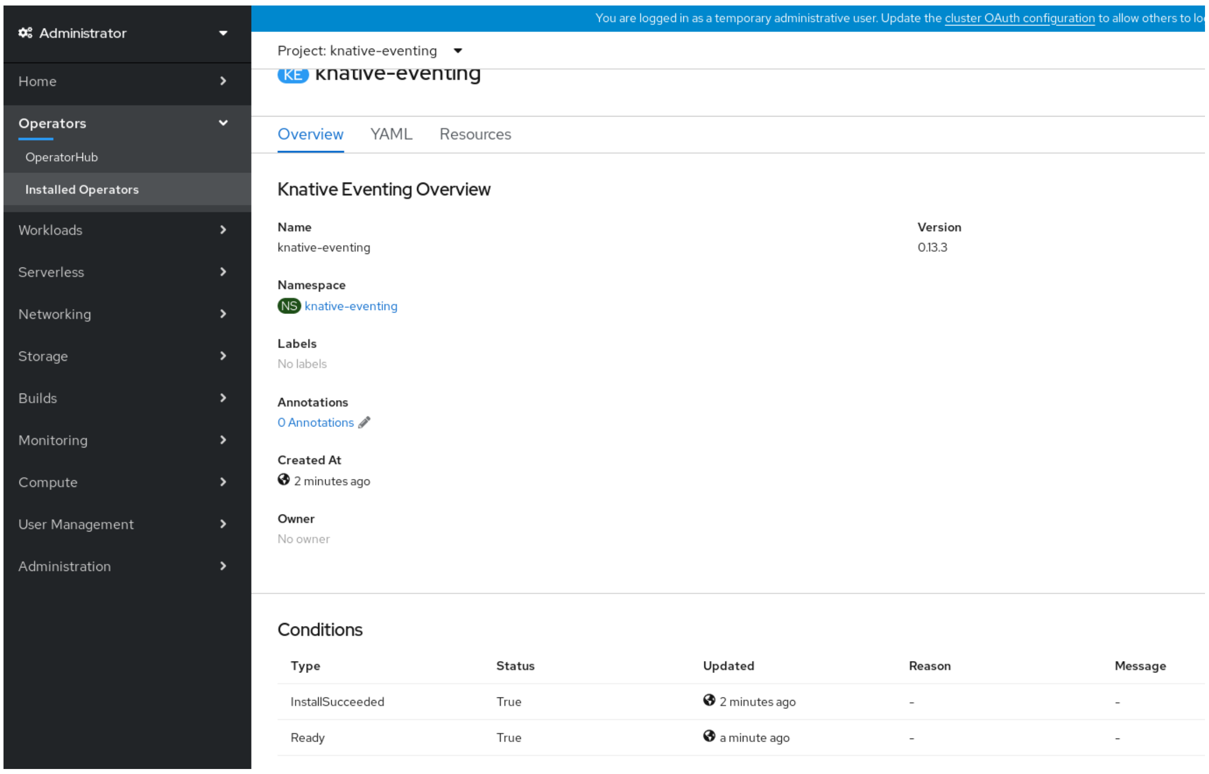
Labels
No labels

Annotations
0 Annotations

Created At
a minute ago

Owner
No owner

3. Scroll down to look at the list of **Conditions**.
4. You should see a list of conditions with a status of **True**, as shown in the example image.



Knative Eventing Overview

Name
knative-eventing

Version
0.13.3

Namespace
NS knative-eventing

Labels
No labels

Annotations
0 Annotations

Created At
2 minutes ago

Owner
No owner

Conditions

Type	Status	Updated	Reason	Message
InstallSucceeded	True	2 minutes ago	-	-
Ready	True	a minute ago	-	-

NOTE

It may take a few seconds for the Knative Eventing resources to be created. You can check their status in the **Resources** tab.

5. If the conditions have a status of **Unknown** or **False**, wait a few moments and then check again after you have confirmed that the resources have been created.

5.3.3. Installing Knative Eventing using YAML

Procedure

1. Create a file named **eventing.yaml**.

2. Copy the following sample YAML into **eventing.yaml**:

```
apiVersion: operator.knative.dev/v1alpha1
kind: KnativeEventing
metadata:
    name: knative-eventing
    namespace: knative-eventing
```

3. Optional. Make any changes to the YAML that you want to implement for your Knative Eventing deployment.

4. Apply the **eventing.yaml** file by entering:

```
$ oc apply -f eventing.yaml
```

Verification

1. Verify the installation is complete by entering the following command and observing the output:

```
$ oc get knativeeventing.operator.knative.dev/knative-eventing \
  -n knative-eventing \
  --template='{{range .status.conditions}}{{printf "%s=%s\n" .type .status}}{{end}}'
```

Example output

```
InstallSucceeded=True
Ready=True
```

NOTE

It may take a few seconds for the Knative Eventing resources to be created.

2. If the conditions have a status of **Unknown** or **False**, wait a few moments and then check again after you have confirmed that the resources have been created.

3. Check that the Knative Eventing resources have been created by entering:

```
$ oc get pods -n knative-eventing
```

Example output

```
NAME                                   READY   STATUS    RESTARTS   AGE
broker-controller-58765d9d49-g9zp6     1/1     Running   0          7m21s
eventing-controller-65fdd66b54-jw7bh   1/1     Running   0          7m31s
eventing-webhook-57fd74b5bd-kvhlz      1/1     Running   0          7m31s
imc-controller-5b75d458fc-ptvm2        1/1     Running   0          7m19s
imc-dispatcher-64f6d5fccb-kkc4c        1/1     Running   0          7m18s
```

5.3.4. Next steps

- Install the Knative CLI to use **kn** commands with Knative Eventing. For example, **kn source** commands. See Installing the Knative CLI.

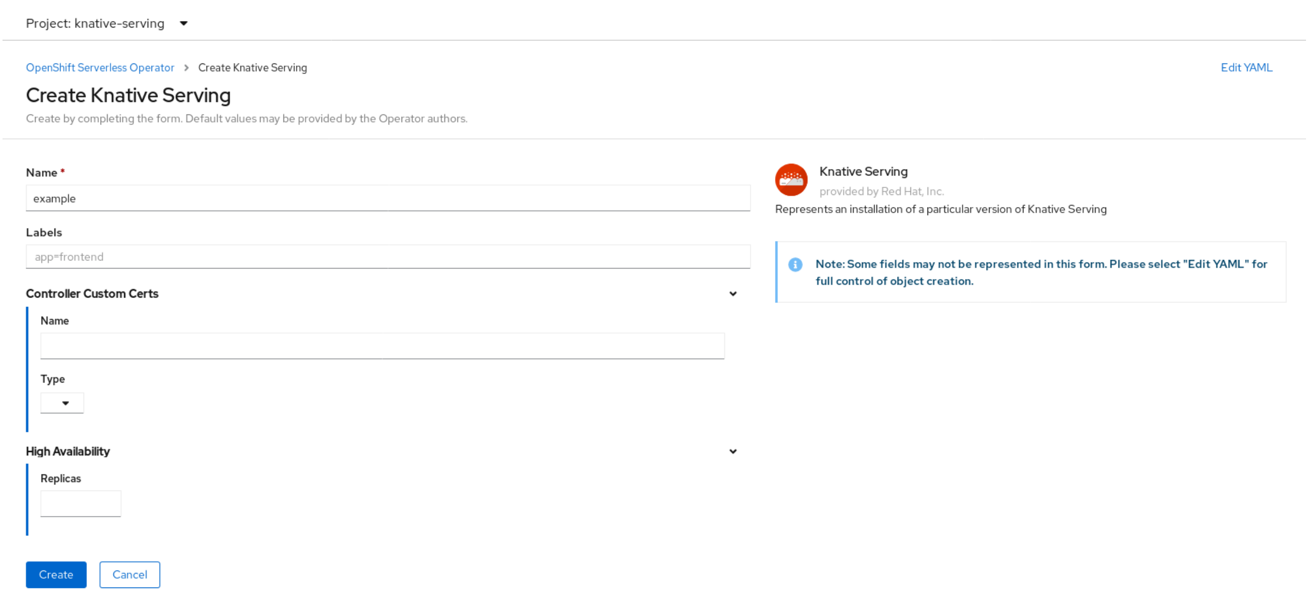
5.4. ADVANCED INSTALLATION CONFIGURATION OPTIONS

This guide provides information for cluster administrators about advanced installation configuration options for OpenShift Serverless components.

5.4.1. Knative Serving supported installation configuration options

IMPORTANT

Do not modify any YAML contained inside the **config** field. Some of the configuration values in this field are injected by the OpenShift Serverless Operator, and modifying them will cause your deployment to become unsupported.


Project: knative-serving

OpenShift Serverless Operator > Create Knative Serving

Edit YAML

Create Knative Serving

Create by completing the form. Default values may be provided by the Operator authors.

Name *

example

Labels

app=frontend

Controller Custom Certs

Name

Type

High Availability

Replicas

Create Cancel

Knative Serving

provided by Red Hat, Inc.

Represents an installation of a particular version of Knative Serving

Note: Some fields may not be represented in this form. Please select "Edit YAML" for full control of object creation.

5.4.1.1. Controller Custom Certs

If your registry uses a self-signed certificate, you must enable tag-to-digest resolution by creating a config map or secret. To enable tag-to-digest resolution, the Knative Serving controller requires access to the container registry.

The following example **KnativeServing** custom resource configuration uses a certificate in a config map named **certs** in the **knative-serving** namespace. This example triggers the OpenShift Serverless Operator to:

1. Create and mount a volume containing the certificate in the controller.
2. Set the required environment variable properly.

Example YAML

```
apiVersion: operator.knative.dev/v1alpha1
kind: KnativeServing
metadata:
  name: knative-serving
  namespace: knative-serving
spec:
  controller-custom-certs:
    name: config-service-ca
    type: ConfigMap (1)
```

(1) The supported types are **ConfigMap** and **Secret**.

If no controller custom cert is specified, this setting defaults to use the **config-service-ca** config map.

After tag-to-digest resolution is enabled, the OpenShift Serverless Operator automatically configures Knative Serving controller access to the registry.

IMPORTANT

The config map or secret must reside in the same namespace as the Knative Serving custom resource definition (CRD).

5.4.1.2. High availability

High availability, which can be configured using the **spec.high-availability** field, defaults to **2** replicas per controller if no number of replicas is specified by a user during the Knative Serving installation.

You can set this to **1** to disable high availability, or add more replicas by setting a higher integer.

Example YAML

```
apiVersion: operator.knative.dev/v1alpha1
kind: KnativeServing
metadata:
  name: knative-serving
  namespace: knative-serving
spec:
  high-availability:
    replicas: 2
```

5.4.2. Additional resources

- For more information about configuring high availability, see High availability on OpenShift Serverless.

5.5. UPGRADING OPENSHIFT SERVERLESS

If you have installed a previous version of OpenShift Serverless, follow the instructions in this guide to upgrade to the latest version.

IMPORTANT

Before upgrading to the latest Serverless release, you must remove the community Knative Eventing Operator if you have previously installed it. Having the Knative Eventing Operator installed will prevent you from being able to install the latest version of Knative Eventing.

5.5.1. Upgrading the Subscription Channel

Prerequisites

- You have installed a previous version of OpenShift Serverless Operator, and have selected Automatic updates during the installation process.

  NOTE

  If you have selected Manual updates, you will need to complete additional steps after updating the channel as described in this guide. The Subscription's upgrade status will remain **Upgrading** until you review and approve its Install Plan. Information about the Install Plan can be found in the OpenShift Container Platform Operators documentation.

- You have logged in to the OpenShift Container Platform web console.

Procedure

1. Select the **openshift-operators** namespace in the OpenShift Container Platform web console.
2. Navigate to the **Operators → Installed Operators** page.
3. Select the **OpenShift Serverless Operator Operator**.
4. Click **Subscription → Channel**.
5. In the **Change Subscription Update Channel** window, select **stable**, and then click **Save**.
6. Wait until all pods have been upgraded in the **knative-serving** namespace and the **KnativeServing** custom resource reports the latest Knative Serving version.

Verification

To verify that the upgrade has been successful, you can check the status of pods in the **knative-serving** namespace, and the version of the **KnativeServing** custom resource.

1. Check the status of the pods:

```
$ oc get knativeserving.operator.knative.dev knative-serving -n knative-serving -
o=jsonpath='{.status.conditions[?(@.type=="Ready")].status}'
```

   This command should return a status of **True**.

2. Check the version of the **KnativeServing** custom resource:

```
$ oc get knativeserving.operator.knative.dev knative-serving -n knative-serving -
o=jsonpath='{.status.version}'
```

   This command should return the latest version of Knative Serving. You can check the latest version in the OpenShift Serverless Operator release notes.

5.6. REMOVING OPENSHIFT SERVERLESS

This guide provides details of how to remove the OpenShift Serverless Operator and other OpenShift Serverless components.

NOTE

Before you can remove the OpenShift Serverless Operator, you must remove Knative Serving and Knative Eventing.

5.6.1. Uninstalling Knative Serving

To uninstall Knative Serving, you must remove its custom resource and delete the **knative-serving** namespace.

Procedure

1. Delete the **knative-serving** custom resource:

```
$ oc delete knativeservings.operator.knative.dev knative-serving -n knative-serving
```

2. After the command has completed and all pods have been removed from the **knative-serving** namespace, delete the namespace:

```
$ oc delete namespace knative-serving
```

5.6.2. Uninstalling Knative Eventing

To uninstall Knative Eventing, you must remove its custom resource and delete the **knative-eventing** namespace.

Procedure

1. Delete the **knative-eventing** custom resource:

```
$ oc delete knativeeventings.operator.knative.dev knative-eventing -n knative-eventing
```

2. After the command has completed and all pods have been removed from the **knative-eventing** namespace, delete the namespace:

```
$ oc delete namespace knative-eventing
```

5.6.3. Removing the OpenShift Serverless Operator

You can remove the OpenShift Serverless Operator from the host cluster by following the documentation on Deleting Operators from a cluster .

5.6.4. Deleting OpenShift Serverless custom resource definitions

After uninstalling the OpenShift Serverless, the Operator and API custom resource definitions (CRDs) remain on the cluster. You can use the following procedure to remove the remaining CRDs.

IMPORTANT

Removing the Operator and API CRDs also removes all resources that were defined using them, including Knative services.

5.6.5. Prerequisites

- You uninstalled Knative Serving and removed the OpenShift Serverless Operator.

Procedure

- To delete the remaining OpenShift Serverless CRDs, enter the following command:

```
$ oc get crd -oname | grep 'knative.dev' | xargs oc delete
```

5.7. CREATING EVENTING COMPONENTS IN THE ADMINISTRATOR PERSPECTIVE

You can create Knative Eventing components with OpenShift Serverless in the **Administrator** perspective of the web console.

5.7.1. Creating an event source by using the Administrator perspective

If you have cluster administrator permissions, you can create an event source by using the Administrator perspective in the web console.

Prerequisites

- The OpenShift Serverless Operator and Knative Eventing are installed on your OpenShift Container Platform cluster.
- You have cluster administrator permissions for OpenShift Container Platform.

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.

See Getting started with event sources for more information on which event source types are supported and can be created using by OpenShift Serverless.

5.7.2. Creating a broker by using the Administrator perspective

If you have cluster administrator permissions, you can create a broker by using the Administrator perspective in the web console.

Prerequisites

- The OpenShift Serverless Operator and Knative Eventing are installed on your OpenShift Container Platform cluster.
- You have cluster administrator permissions for OpenShift Container Platform.

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**.

5.7.3. Creating a trigger by using the Administrator perspective

If you have cluster administrator permissions and have created a broker, you can create a trigger to connect your broker to a subscriber by using the Administrator perspective in the web console.

Prerequisites

- The OpenShift Serverless Operator and Knative Eventing are installed on your OpenShift Container Platform cluster.
- You have cluster administrator permissions for OpenShift Container Platform.
- You have created a 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**.

5.7.4. Creating a channel by using the Administrator perspective

If you have cluster administrator permissions, you can create a channel by using the Administrator perspective in the web console.

Prerequisites

- The OpenShift Serverless Operator and Knative Eventing are installed on your OpenShift Container Platform cluster.
- You have cluster administrator permissions for OpenShift Container Platform.

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 from the **Type** drop-down.

   NOTE

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

4. Click **Create**.

5.7.5. Creating a subscription by using the Administrator perspective

If you have cluster administrator permissions and have created a channel, you can create a subscription to connect your broker to a subscriber by using the Administrator perspective in the web console.

Prerequisites

- The OpenShift Serverless Operator and Knative Eventing are installed on your OpenShift Container Platform cluster.
- You have cluster administrator permissions for OpenShift Container Platform.
- You have created a 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 will receive events from the channel.
5. Click **Add**.

5.7.6. Additional resources

- See Brokers.
- See Subscriptions.
- See Triggers.
- See Channels.
- See Knative Kafka.

5.8. CREATING KNATIVE SERVING COMPONENTS IN THE ADMINISTRATOR PERSPECTIVE

If you have cluster administrator permissions on a OpenShift Container Platform cluster, you can create Knative Serving components with OpenShift Serverless in the **Administrator** perspective of the web console or by using the **kn** and **oc** CLIs.

5.8.1. Creating serverless applications using the Administrator perspective

Prerequisites

To create serverless applications using the **Administrator** perspective, ensure that you have completed the following steps.

- The OpenShift Serverless Operator and Knative Serving are installed.
- You have logged in to the web console and are in the **Administrator** perspective.

Procedure

1. Navigate to the **Serverless** → **Serving** page.
2. In the **Create** list, select **Service**.
3. Manually enter YAML or JSON definitions, or by dragging and dropping a file into the editor.
4. Click **Create**.

5.9. USING METERING WITH OPENSHIFT SERVERLESS

As a cluster administrator, you can use metering to analyze what is happening in your OpenShift Serverless cluster.

For more information about metering on OpenShift Container Platform, see About metering.

NOTE

Metering is not currently supported for IBM Z and IBM Power Systems.

5.9.1. Installing metering

For information about installing metering on OpenShift Container Platform, see Installing Metering.

5.9.2. Datasources for Knative Serving metering

The following **ReportDataSources** are examples of how Knative Serving can be used with OpenShift Container Platform metering.

5.9.2.1. Datasource for CPU usage in Knative Serving

This datasource provides the accumulated CPU seconds used per Knative service over the report time period.

YAML file

```
apiVersion: metering.openshift.io/v1
kind: ReportDataSource
metadata:
  name: knative-service-cpu-usage
spec:
  prometheusMetricsImporter:
    query: >
      sum
          by(namespace,
             label_serving_knative_dev_service,
            label_serving_knative_dev_revision)
          (

label_replace(rate(container_cpu_usage_seconds_total{container!="POD",container!="",pod!=""}
[1m]), "pod", "$1", "pod", "(.*)")
            *
            on(pod, namespace)
            group_left(label_serving_knative_dev_service, label_serving_knative_dev_revision)
            kube_pod_labels{label_serving_knative_dev_service!=""}
          )
```

5.9.2.2. Datasource for memory usage in Knative Serving

This datasource provides the average memory consumption per Knative service over the report time period.

YAML file

```
apiVersion: metering.openshift.io/v1
kind: ReportDataSource
metadata:
  name: knative-service-memory-usage
spec:
  prometheusMetricsImporter:
    query: >
      sum
          by(namespace,
            label_serving_knative_dev_service,
            label_serving_knative_dev_revision)
          (
            label_replace(container_memory_usage_bytes{container!="POD", container!="",pod!=""},
"pod", "$1", "pod", "(.*)")
            *
            on(pod, namespace)
            group_left(label_serving_knative_dev_service, label_serving_knative_dev_revision)
            kube_pod_labels{label_serving_knative_dev_service!=""}
          )
```

5.9.2.3. Applying Datasources for Knative Serving metering

You can apply the **ReportDataSources** by using the following command:

```
$ oc apply -f <datasource_name>.yaml
```

Example

```
$ oc apply -f knative-service-memory-usage.yaml
```

5.9.3. Queries for Knative Serving metering

The following **ReportQuery** resources reference the example **DataSources** provided.

5.9.3.1. Query for CPU usage in Knative Serving

YAML file

```
apiVersion: metering.openshift.io/v1
kind: ReportQuery
metadata:
  name: knative-service-cpu-usage
spec:
  inputs:
  - name: ReportingStart
    type: time
  - name: ReportingEnd
    type: time
  - default: knative-service-cpu-usage
    name: KnativeServiceCpuUsageDataSource
    type: ReportDataSource
  columns:
  - name: period_start
    type: timestamp
    unit: date
  - name: period_end
    type: timestamp
    unit: date
  - name: namespace
    type: varchar
    unit: kubernetes_namespace
  - name: service
    type: varchar
  - name: data_start
    type: timestamp
    unit: date
  - name: data_end
    type: timestamp
    unit: date
  - name: service_cpu_seconds
    type: double
    unit: cpu_core_seconds
  query: |
    SELECT
      timestamp '{| default .Report.ReportingStart .Report.Inputs.ReportingStart| prestoTimestamp |}'
AS period_start,
      timestamp '{| default .Report.ReportingEnd .Report.Inputs.ReportingEnd | prestoTimestamp |}' AS
period_end,
      labels['namespace'] as project,
      labels['label_serving_knative_dev_service'] as service,
      min("timestamp") as data_start,
      max("timestamp") as data_end,
      sum(amount * "timeprecision") AS service_cpu_seconds
    FROM {| dataSourceTableName .Report.Inputs.KnativeServiceCpuUsageDataSource |}
    WHERE "timestamp" >= timestamp '{| default .Report.ReportingStart .Report.Inputs.ReportingStart
| prestoTimestamp |}'
    AND "timestamp" < timestamp '{| default .Report.ReportingEnd .Report.Inputs.ReportingEnd |
prestoTimestamp |}'
    GROUP BY labels['namespace'],labels['label_serving_knative_dev_service']
```

5.9.3.2. Query for memory usage in Knative Serving

YAML file

```
apiVersion: metering.openshift.io/v1
kind: ReportQuery
metadata:
  name: knative-service-memory-usage
spec:
  inputs:
  - name: ReportingStart
    type: time
  - name: ReportingEnd
    type: time
  - default: knative-service-memory-usage
    name: KnativeServiceMemoryUsageDataSource
    type: ReportDataSource
  columns:
  - name: period_start
    type: timestamp
    unit: date
  - name: period_end
    type: timestamp
    unit: date
  - name: namespace
    type: varchar
    unit: kubernetes_namespace
  - name: service
    type: varchar
  - name: data_start
    type: timestamp
    unit: date
  - name: data_end
    type: timestamp
    unit: date
  - name: service_usage_memory_byte_seconds
    type: double
    unit: byte_seconds
  query: |
    SELECT
      timestamp '{| default .Report.ReportingStart .Report.Inputs.ReportingStart| prestoTimestamp |}'
AS period_start,
      timestamp '{| default .Report.ReportingEnd .Report.Inputs.ReportingEnd | prestoTimestamp |}' AS
period_end,
      labels['namespace'] as project,
      labels['label_serving_knative_dev_service'] as service,
      min("timestamp") as data_start,
      max("timestamp") as data_end,
      sum(amount * "timeprecision") AS service_usage_memory_byte_seconds
    FROM {| dataSourceTableName .Report.Inputs.KnativeServiceMemoryUsageDataSource |}
    WHERE "timestamp" >= timestamp '{| default .Report.ReportingStart .Report.Inputs.ReportingStart
| prestoTimestamp |}'
    AND "timestamp" < timestamp '{| default .Report.ReportingEnd .Report.Inputs.ReportingEnd |
prestoTimestamp |}'
    GROUP BY labels['namespace'],labels['label_serving_knative_dev_service']
```

5.9.3.3. Applying Queries for Knative Serving metering

1. Apply the **ReportQuery** by entering the following command:

```
$ oc apply -f <query-name>.yaml
```

Example command

```
$ oc apply -f knative-service-memory-usage.yaml
```

5.9.4. Metering reports for Knative Serving

You can run metering reports against Knative Serving by creating **Report** resources. Before you run a report, you must modify the input parameter within the **Report** resource to specify the start and end dates of the reporting period.

YAML file

```
apiVersion: metering.openshift.io/v1
kind: Report
metadata:
  name: knative-service-cpu-usage
spec:
  reportingStart: '2019-06-01T00:00:00Z' (1)
  reportingEnd: '2019-06-30T23:59:59Z' (2)
  query: knative-service-cpu-usage (3)
runImmediately: true
```

(1) Start date of the report, in ISO 8601 format.

(2) End date of the report, in ISO 8601 format.

(3) Either **knative-service-cpu-usage** for CPU usage report or **knative-service-memory-usage** for a memory usage report.

5.9.4.1. Running a metering report

1. Run the report by entering the following command:

```
$ oc apply -f <report-name>.yml
```

2. You can then check the report by entering the following command:

```
$ oc get report
```

Example output

```
NAME                      QUERY                      SCHEDULE   RUNNING    FAILED   LAST
REPORT TIME      AGE
knative-service-cpu-usage   knative-service-cpu-usage            Finished            2019-06-
30T23:59:59Z   10h
```

CHAPTER 6. ARCHITECTURE

6.1. KNATIVE SERVING ARCHITECTURE

Knative Serving on OpenShift Container Platform enables developers to write cloud-native applications using serverless architecture. Serverless is a cloud computing model where application developers don't need to provision servers or manage scaling for their applications. These routine tasks are abstracted away by the platform, allowing developers to push code to production much faster than in traditional models.

Knative Serving supports deploying and managing cloud-native applications by providing a set of objects as Kubernetes custom resource definitions (CRDs) that define and control the behavior of serverless workloads on an OpenShift Container Platform cluster. For more information about CRDs, see Extending the Kubernetes API with custom resource definitions .

Developers use these CRDs to create custom resource (CR) instances that can be used as building blocks to address complex use cases. For example:

- Rapidly deploying serverless containers.
- Automatically scaling pods.

For more information about CRs, see Managing resources from Custom Resource Definitions .

6.1.1. Knative Serving custom resource definitions

Service

The **service.serving.knative.dev** CRD automatically manages the life cycle of your workload to ensure that the application is deployed and reachable through the network. It creates a route, a configuration, and a new revision for each change to a user created service, or custom resource. Most developer interactions in Knative are carried out by modifying services.

Revision

The **revision.serving.knative.dev** CRD is a point-in-time snapshot of the code and configuration for each modification made to the workload. Revisions are immutable objects and can be retained for as long as necessary.

Route

The **route.serving.knative.dev** CRD maps a network endpoint to one or more revisions. You can manage the traffic in several ways, including fractional traffic and named routes.

Configuration

The **configuration.serving.knative.dev** CRD maintains the desired state for your deployment. It provides a clean separation between code and configuration. Modifying a configuration creates a new revision.

6.2. KNATIVE EVENTING ARCHITECTURE

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 that create events, and event *sinks*, or consumers, that receive them.

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

You can propagate an event from an event source to multiple event sinks by using:

- channels and subscriptions, or
- brokers and triggers.

Events are buffered if the destination sink is unavailable. Knative Eventing supports the following scenarios:

Publish an event without creating a consumer

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

Consume an event without creating a publisher

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

6.2.1. Event sinks

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

Addressable objects

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

Callable objects

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 7. HIGH AVAILABILITY ON OPENSHIFT SERVERLESS

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 available to take 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 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 referred to as the leader.

7.1. CONFIGURING HIGH AVAILABILITY REPLICAS ON OPENSHIFT SERVERLESS

High availability (HA) functionality is available by default on OpenShift Serverless for the **autoscaler-hpa**, **controller**, **activator**, **kourier-control**, and **kourier-gateway** controllers. These components are configured with two replicas by default.

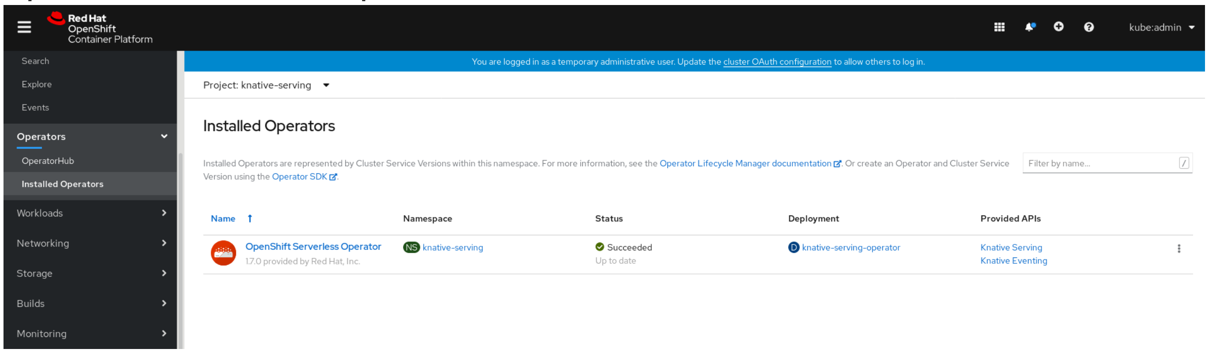
You modify the number of replicas that are created per controller by changing the configuration of **KnativeServing.spec.highAvailability** in the KnativeServing custom resource definition.

Prerequisites

- An OpenShift Container Platform account with cluster administrator access.
- Installed the OpenShift Serverless Operator and Knative Serving.

Procedure

1. In the OpenShift Container Platform web console **Administrator** perspective, navigate to **OperatorHub → Installed Operators**.



You are logged in as a temporary administrative user. Update the cluster OAuth configuration to allow others to log in.

Project: knative-serving

Installed Operators

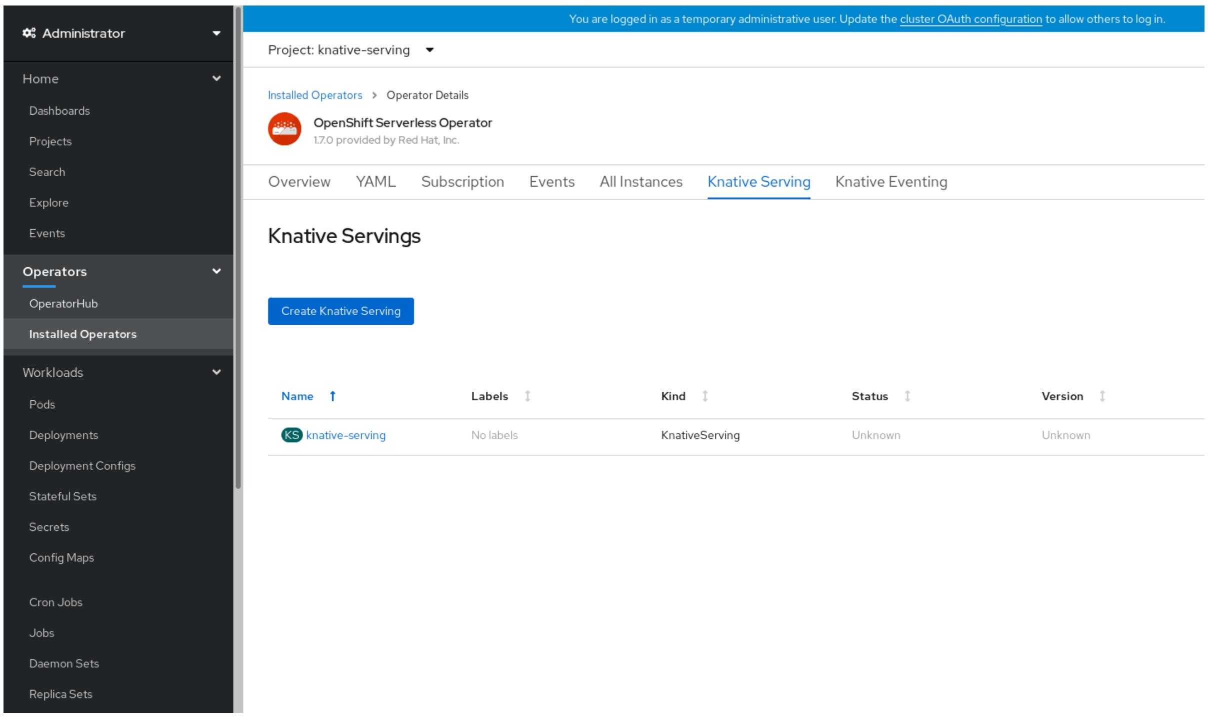
Installed Operators are represented by Cluster Service Versions within this namespace. For more information, see the Operator Lifecycle Manager documentation. Or create an Operator and Cluster Service Version using the Operator SDK.

Filter by name...

Name	Namespace	Status	Deployment	Provided APIs
OpenShift Serverless Operator 1.7.0 provided by Red Hat, Inc.	NS knative-serving	Succeeded Up to date	D knative-serving-operator	Knative Serving Knative Eventing

2. Select the **knative-serving** namespace.

3. Click **Knative Serving** in the list of **Provided APIs** for the OpenShift Serverless Operator to go to the **Knative Serving** tab.



You are logged in as a temporary administrative user. Update the cluster OAuth configuration to allow others to log in.

Project: knative-serving

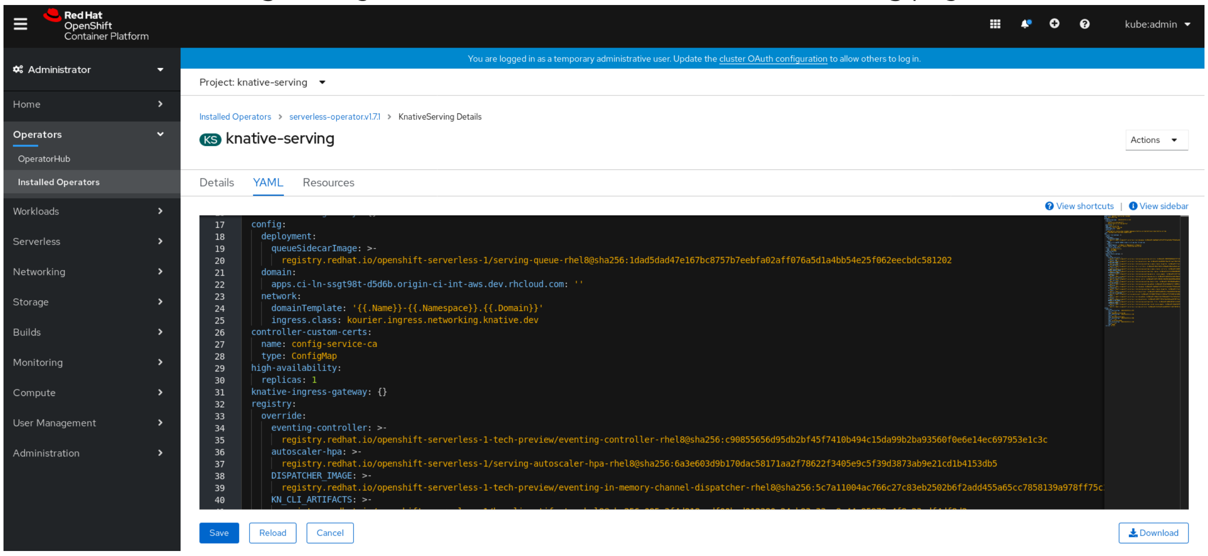
Installed Operators > Operator Details

OpenShift Serverless Operator
1.7.0 provided by Red Hat, Inc.

Knative Servings

Name	Labels	Kind	Status	Version
KS knative-serving	No labels	KnativeServing	Unknown	Unknown

4. Click **knative-serving**, then go to the **YAML** tab in the **knative-serving** page.



You are logged in as a temporary administrative user. Update the cluster OAuth configuration to allow others to log in.

Project: knative-serving

Installed Operators > serverless-operator.v1.7.1 > KnativeServing Details

KS knative-serving

```
config:
  deployment:
    queueSidecarImage: >-
      registry.redhat.io/openshift-serverless-1/serving-queue-rhel8@sha256:1dad5dad47e167bc8757b7eebfa02aff076a5d1a4bb54e25f062eecbdc581202
  domain:
    apps.ci-ln-ssgt98t-d5d6b.origin-ci-int-aws.dev.rhcloud.com: ''
  network:
    domainTemplate: '{{.Name}}-{{.Namespace}}.{{.Domain}}'
    ingress.class: kourier.ingress.networking.knative.dev
controller-custom-certs:
  name: config-service-ca
  type: ConfigMap
high-availability:
  replicas: 1
knative-ingress-gateway: {}
registry:
  override:
    eventing-controller: >-
      registry.redhat.io/openshift-serverless-1-tech-preview/eventing-controller-rhel8@sha256:c90855656d95db2bf45f7410b494c15da99b2ba93560f0e6e14ec697953e1c3c
    autoscaler-hpa: >-
      registry.redhat.io/openshift-serverless-1/serving-autoscaler-hpa-rhel8@sha256:6a3e603d9b170dac58171aa2f78622f3405e9c5f39d3873ab9e21cd1b4153db5
    DISPATCHER_IMAGE: >-
      registry.redhat.io/openshift-serverless-1-tech-preview/eventing-in-memory-channel-dispatcher-rhel8@sha256:5c7a11004ac766c27c83eb2502b6f2add455a65cc7858139a978ff75c
    KN_CLI_ARTIFACTS: >-
```

5. Edit the custom resource definition YAML:

Example YAML

```
apiVersion: operator.knative.dev/v1alpha1
kind: KnativeServing
metadata:
  name: knative-serving
  namespace: knative-serving
spec:
  high-availability:
    replicas: 3
```

IMPORTANT

Do not modify any YAML contained inside the **config** field. Some of the configuration values in this field are injected by the OpenShift Serverless Operator, and modifying them will cause your deployment to become unsupported.

- The default **replicas** value is **2**.
- Changing the value to **1** will disable HA, or you can increase the number of replicas as required. The example configuration shown specifies a replica count of **3** for all HA controllers.

CHAPTER 8. TRACING REQUESTS USING JAEGER

Using Jaeger with OpenShift Serverless allows you to enable *distributed tracing* for your serverless applications on OpenShift Container Platform.

Distributed tracing records the path of a request through the various services that make up an application.

It is used to tie information about different units of work together, to understand a whole chain of events in a distributed transaction. The units of work might be executed in different processes or hosts.

Developers can visualize call flows in large architectures with distributed tracing. which is useful for understanding serialization, parallelism, and sources of latency.

For more information about Jaeger, see Jaeger architecture and Installing Jaeger.

8.1. CONFIGURING JAEGER FOR USE WITH OPENSHIFT SERVERLESS

Prerequisites

To configure Jaeger for use with OpenShift Serverless, you will need:

- Cluster administrator permissions on an OpenShift Container Platform cluster.
- A current installation of OpenShift Serverless Operator and Knative Serving.
- A current installation of the Jaeger Operator.

Procedure

1. Create and apply a Jaeger custom resource YAML file that contains the following sample YAML:

   Jaeger custom resource YAML

   ```
   apiVersion: jaegertracing.io/v1
   kind: Jaeger
   metadata:
     name: jaeger
     namespace: default
   ```

2. Enable tracing for Knative Serving, by editing the **KnativeServing** resource and adding a YAML configuration for tracing.

   Tracing YAML example

   ```
   apiVersion: operator.knative.dev/v1alpha1
   kind: KnativeServing
   metadata:
     name: knative-serving
     namespace: knative-serving
   spec:
     config:
       tracing:
         sample-rate: "0.1" 1
   ```

```
backend: zipkin (2)
zipkin-endpoint: http://jaeger-collector.default.svc.cluster.local:9411/api/v2/spans (3)
debug: "false" (4)
```

1. The **sample-rate** defines sampling probability. Using **sample-rate: "0.1"** means that 1 in 10 traces will be sampled.
2. **backend** must be set to **zipkin**.
3. The **zipkin-endpoint** must point to your **jaeger-collector** service endpoint. To get this endpoint, substitute the namespace where the Jaeger custom resource is applied.
4. Debugging should be set to **false**. Enabling debug mode by setting **debug: "true"** allows all spans to be sent to the server, bypassing sampling.

Verification

Access the Jaeger web console to see tracing data. You can access the Jaeger web console by using the **jaeger** route.

1. Get the **jaeger** route's hostname by entering the following command:

```
$ oc get route jaeger
```

Example output

```
NAME     HOST/PORT                         PATH   SERVICES       PORT    TERMINATION
WILDCARD
jaeger   jaeger-default.apps.example.com          jaeger-query   <all>   reencrypt     None
```

2. Open the endpoint address in your browser to view the console.

CHAPTER 9. KNATIVE SERVING

9.1. SERVERLESS APPLICATIONS

9.1.1. Serverless applications using Knative services

To deploy a serverless application using OpenShift Serverless, you must create a *Knative service*. Knative services are Kubernetes services, defined by a route and a configuration, and contained in a YAML file.

Example Knative service YAML

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: hello (1)
  namespace: default (2)
spec:
  template:
    spec:
      containers:
        - image: docker.io/openshift/hello-openshift (3)
          env:
            - name: RESPONSE (4)
              value: "Hello Serverless!"
```

(1) The name of the application.

(2) The namespace the application will use.

(3) The image of the application.

(4) The environment variable printed out by the sample application.

9.1.2. Creating serverless applications

You can create a serverless application by using one of the following methods:

- Create a Knative service from the OpenShift Container Platform web console.
- Create a Knative service using the **kn** CLI.
- Create and apply a YAML file.

9.1.2.1. Creating serverless applications using the Developer perspective

For more information about creating applications using the **Developer** perspective in OpenShift Container Platform, see the documentation on Creating applications using the Developer perspective .

9.1.2.2. Creating serverless applications using the kn CLI

The following procedure describes how you can create a basic serverless application using the **kn** CLI.

Prerequisites

- OpenShift Serverless Operator and Knative Serving are installed on your cluster.
- You have installed **kn** CLI.

Procedure

- Create a Knative service:

```
$ kn service create <service-name> --image <image> --env <key=value>
```

Example command

```
$ kn service create event-display \
    --image quay.io/openshift-knative/knative-eventing-sources-event-display:latest
```

Example output

```
Creating service 'event-display' in namespace 'default':

  0.271s The Route is still working to reflect the latest desired specification.
  0.580s Configuration "event-display" is waiting for a Revision to become ready.
  3.857s ...
  3.861s Ingress has not yet been reconciled.
  4.270s Ready to serve.

Service 'event-display' created with latest revision 'event-display-bxshg-1' and URL:
http://event-display-default.apps-crc.testing
```

9.1.2.3. Creating serverless applications using YAML

To create a serverless application by using YAML, you must create a YAML file that defines a **Service** object, then apply it by using **oc apply**.

Procedure

1. Create a YAML file containing the following sample code:

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: event-delivery
  namespace: default
spec:
  template:
    spec:
      containers:
        - image: quay.io/openshift-knative/knative-eventing-sources-event-display:latest
          env:
            - name: RESPONSE
              value: "Hello Serverless!"
```

2. Navigate to the directory where the YAML file is contained, and deploy the application by applying the YAML file:

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

After the service is created and the application is deployed, Knative creates an immutable revision for this version of the application. Knative also performs network programming to create a route, ingress, service, and load balancer for your application and automatically scales your pods up and down based on traffic, including inactive pods.

9.1.3. Updating serverless applications

You can use the **kn service update** command for interactive sessions on the command line as you build up a service incrementally. In contrast to the **kn service apply** command, when using the **kn service update** command you only have to specify the changes that you want to update, rather than the full configuration for the Knative service.

Example commands

- Update a service by adding a new environment variable:

  ```
  $ kn service update <service_name> --env <key>=<value>
  ```

- Update a service by adding a new port:

  ```
  $ kn service update <service_name> --port 80
  ```

- Update a service by adding new request and limit parameters:

  ```
  $ kn service update <service_name> --request cpu=500m --limit memory=1024Mi --limit
  cpu=1000m
  ```

- Assign the **latest** tag to a revision:

  ```
  $ kn service update <service_name> --tag <revision_name>=latest
  ```

- Update a tag from **testing** to **staging** for the latest **READY** revision of a service:

  ```
  $ kn service update <service_name> --untag testing --tag @latest=staging
  ```

- Add the **test** tag to a revision that receives 10% of traffic, and send the rest of the traffic to the latest **READY** revision of a service:

  ```
  $ kn service update <service_name> --tag <revision_name>=test --traffic test=10,@latest=90
  ```

9.1.4. Applying service declarations

You can declaratively configure a Knative service by using the **kn service apply** command. If the service does not exist it is created, otherwise the existing service is updated with the options that have been changed.

The **kn service apply** command is especially useful for shell scripts or in a continuous integration pipeline, where users typically want to fully specify the state of the service in a single command to declare the target state.

When using **kn service apply** you must provide the full configuration for the Knative service. This is different from the **kn service update** command, which only requires you to specify in the command the options that you want to update.

Example commands

- Create a service:

```
$ kn service apply <service_name> --image <image>
```

- Add an environment variable to a service:

```
$ kn service apply <service_name> --image <image> --env <key>=<value>
```

- Read the service declaration from a JSON or YAML file:

```
$ kn service apply <service_name> -f <filename>
```

9.1.5. Describing serverless applications

You can describe a Knative service by using the **kn service describe** command.

Example commands

- Describe a service:

```
$ kn service describe --verbose <service_name>
```

The **--verbose** flag is optional but can be included to provide a more detailed description. The difference between a regular and verbose output is shown in the following examples:

Example output without --verbose flag

```
Name:       hello
Namespace:  default
Age:        2m
URL:        http://hello-default.apps.ocp.example.com

Revisions:
  100%  @latest (hello-00001) [1] (2m)
        Image:  docker.io/openshift/hello-openshift (pinned to aaea76)

Conditions:
  OK TYPE                   AGE REASON
  ++ Ready                   1m
  ++ ConfigurationsReady     1m
  ++ RoutesReady             1m
```

Example output with --verbose flag

```
Name:         hello
Namespace:    default
Annotations:  serving.knative.dev/creator=system:admin
              serving.knative.dev/lastModifier=system:admin
Age:          3m
URL:          http://hello-default.apps.ocp.example.com
Cluster:      http://hello.default.svc.cluster.local

Revisions:
  100%  @latest (hello-00001) [1] (3m)
        Image:  docker.io/openshift/hello-openshift (pinned to aaea76)
        Env:    RESPONSE=Hello Serverless!

Conditions:
  OK TYPE                   AGE REASON
  ++ Ready                  3m
  ++ ConfigurationsReady    3m
  ++ RoutesReady            3m
```

- Describe a service in YAML format:

```
$ kn service describe <service_name> -o yaml
```

- Describe a service in JSON format:

```
$ kn service describe <service_name> -o json
```

- Print the service URL only:

```
$ kn service describe <service_name> -o url
```

9.1.6. Verifying your serverless application deployment

To verify that your serverless application has been deployed successfully, you must get the application URL created by Knative, and then send a request to that URL and observe the output.

NOTE

OpenShift Serverless supports the use of both HTTP and HTTPS URLs, however the output from **oc get ksvc** will always print URLs using the **http://** format.

Procedure

1. Find the application URL:

```
$ oc get ksvc <service_name>
```

Example output

```
NAME            URL                                        LATESTCREATED         LATESTREADY
READY   REASON
event-delivery   http://event-delivery-default.example.com   event-delivery-4wsd2   event-
```

```
delivery-4wsd2   True
```

2. Make a request to your cluster and observe the output.

   Example HTTP request

   ```
   $ curl http://event-delivery-default.example.com
   ```

   Example HTTPS request

   ```
   $ curl https://event-delivery-default.example.com
   ```

   Example output

   ```
   Hello Serverless!
   ```

3. Optional. If you receive an error relating to a self-signed certificate in the certificate chain, you can add the **--insecure** flag to the curl command to ignore the error:

   ```
   $ curl https://event-delivery-default.example.com --insecure
   ```

   Example output

   ```
   Hello Serverless!
   ```

   IMPORTANT

   Self-signed certificates must not be used in a production deployment. This method is only for testing purposes.

4. Optional. If your OpenShift Container Platform cluster is configured with a certificate that is signed by a certificate authority (CA) but not yet globally configured for your system, you can specify this with the **curl** command. The path to the certificate can be passed to the curl command by using the **--cacert** flag:

   ```
   $ curl https://event-delivery-default.example.com --cacert <file>
   ```

   Example output

   ```
   Hello Serverless!
   ```

9.1.7. Interacting with a serverless application using HTTP2 and gRPC

OpenShift Serverless supports only insecure or edge-terminated routes.

Insecure or edge-terminated routes do not support HTTP2 on OpenShift Container Platform. These routes also do not support gRPC because gRPC is transported by HTTP2.

If you use these protocols in your application, you must call the application using the ingress gateway directly. To do this you must find the ingress gateway's public address and the application's specific host.

Procedure

1. Find the application host. See the instructions in *Verifying your serverless application deployment*.

2. Find the ingress gateway's public address:

```
$ oc -n knative-serving-ingress get svc kourier
```

Example output

```
NAME                TYPE           CLUSTER-IP      EXTERNAL-IP
PORT(S)
AGE
kourier   LoadBalancer   172.30.51.103   a83e86291bcdd11e993af02b7a65e514-
33544245.us-east-1.elb.amazonaws.com   80:31380/TCP,443:31390/TCP   67m
```

The public address is surfaced in the **EXTERNAL-IP** field, and in this case is **a83e86291bcdd11e993af02b7a65e514-33544245.us-east-1.elb.amazonaws.com**.

3. Manually set the host header of your HTTP request to the application's host, but direct the request itself against the public address of the ingress gateway.

```
$ curl -H "Host: hello-default.example.com" a83e86291bcdd11e993af02b7a65e514-
33544245.us-east-1.elb.amazonaws.com
```

Example output

```
Hello Serverless!
```

You can also make a gRPC request by setting the authority to the application's host, while directing the request against the ingress gateway directly:

```
grpc.Dial(
   "a83e86291bcdd11e993af02b7a65e514-33544245.us-east-1.elb.amazonaws.com:80",
   grpc.WithAuthority("hello-default.example.com:80"),
   grpc.WithInsecure(),
)
```

NOTE

Ensure that you append the respective port, 80 by default, to both hosts as shown in the previous example.

9.2. CONFIGURING KNATIVE SERVING AUTOSCALING

OpenShift Serverless provides capabilities for automatic pod scaling, including scaling inactive pods to zero. To enable autoscaling for Knative Serving, you must configure concurrency and scale bounds in the revision template.

NOTE

Any limits or targets set in the revision template are measured against a single instance of your application. For example, setting the **target** annotation to **50** will configure the autoscaler to scale the application so that each revision will handle 50 requests at a time.

9.2.1. Autoscaling workflow using the Knative kn CLI

You can edit autoscaling capabilities for your cluster by using **kn** to modify Knative services without editing YAML files directly.

You can use the **kn service create** and **kn service update** commands with the appropriate flags as described below to configure autoscaling behavior.

Flag	Description
--concurrency-limit int	Sets a hard limit of concurrent requests to be processed by a single revision.
--concurrency-target int	Provides a recommendation for when to scale up revisions, based on the concurrent number of incoming requests. Defaults to **--concurrency-limit**.
--max-scale int	Maximum number of revisions.
--min-scale int	Minimum number of revisions.

9.2.2. Configuring concurrent requests for Knative Serving autoscaling

You can specify the number of concurrent requests that should be handled by each instance of a revision container, or application, by adding the **target** annotation or the **containerConcurrency** field in the revision template.

Example revision template YAML using target annotation

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: myapp
spec:
  template:
    metadata:
      annotations:
        autoscaling.knative.dev/target: 50
    spec:
      containers:
      - image: myimage
```

Example revision template YAML using containerConcurrency annotation

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: myapp
spec:
  template:
    metadata:
      annotations:
    spec:
      containerConcurrency: 100
      containers:
      - image: myimage
```

Adding a value for both **target** and **containerConcurrency** will target the **target** number of concurrent requests, but impose a hard limit of the **containerConcurrency** number of requests.

For example, if the **target** value is 50 and the **containerConcurrency** value is 100, the targeted number of requests will be 50, but the hard limit will be 100.

If the **containerConcurrency** value is less than the **target** value, the **target** value will be tuned down, since there is no need to target more requests than the number that can actually be handled.

NOTE

containerConcurrency should only be used if there is a clear need to limit how many requests reach the application at a given time. Using **containerConcurrency** is only advised if the application needs to have an enforced constraint of concurrency.

9.2.2.1. Configuring concurrent requests using the target annotation

The default target for the number of concurrent requests is **100**, but you can override this value by adding or modifying the **autoscaling.knative.dev/target** annotation value in the revision template.

Here is an example of how this annotation is used in the revision template to set the target to **50**:

```
autoscaling.knative.dev/target: 50
```

9.2.2.2. Configuring concurrent requests using the containerConcurrency field

containerConcurrency sets a hard limit on the number of concurrent requests handled.

```
containerConcurrency: 0 | 1 | 2-N
```

0

allows unlimited concurrent requests.

1

guarantees that only one request is handled at a time by a given instance of the revision container.

2 or more

will limit request concurrency to that value.

NOTE

If there is no **target** annotation, autoscaling is configured as if **target** is equal to the value of **containerConcurrency**.

9.2.3. Configuring scale bounds Knative Serving autoscaling

The **minScale** and **maxScale** annotations can be used to configure the minimum and maximum number of pods that can serve applications. These annotations can be used to prevent cold starts or to help control computing costs.

minScale
: If the **minScale** annotation is not set, pods will scale to zero (or to 1 if enable-scale-to-zero is false per the **ConfigMap**).

maxScale
: If the **maxScale** annotation is not set, there will be no upper limit for the number of pods created.

minScale and **maxScale** can be configured as follows in the revision template:

```
spec:
  template:
    metadata:
      annotations:
        autoscaling.knative.dev/minScale: "2"
        autoscaling.knative.dev/maxScale: "10"
```

Using these annotations in the revision template will propagate this confguration to **PodAutoscaler** objects.

NOTE

These annotations apply for the full lifetime of a revision. Even when a revision is not referenced by any route, the minimal Pod count specified by **minScale** will still be provided. Keep in mind that non-routeable revisions may be garbage collected, which enables Knative to reclaim the resources.

9.3. USING OPENSHIFT LOGGING

9.3.1. About deploying OpenShift Logging

OpenShift Container Platform cluster administrators can deploy OpenShift Logging using the OpenShift Container Platform web console or CLI to install the OpenShift Elasticsearch Operator and Cluster Logging Operator. When the operators are installed, you create a **ClusterLogging** custom resource (CR) to schedule OpenShift Logging pods and other resources necessary to support OpenShift Logging. The operators are responsible for deploying, upgrading, and maintaining OpenShift Logging.

The **ClusterLogging** CR defines a complete OpenShift Logging environment that includes all the components of the logging stack to collect, store and visualize logs. The Cluster Logging Operator watches the OpenShift Logging CR and adjusts the logging deployment accordingly.

Administrators and application developers can view the logs of the projects for which they have view access.

9.3.2. About deploying and configuring OpenShift Logging

OpenShift Logging is designed to be used with the default configuration, which is tuned for small to medium sized OpenShift Container Platform clusters.

The installation instructions that follow include a sample **ClusterLogging** custom resource (CR), which you can use to create a OpenShift Logging instance and configure your OpenShift Logging environment.

If you want to use the default OpenShift Logging install, you can use the sample CR directly.

If you want to customize your deployment, make changes to the sample CR as needed. The following describes the configurations you can make when installing your OpenShift Logging instance or modify after installation. See the Configuring sections for more information on working with each component, including modifications you can make outside of the **ClusterLogging** custom resource.

9.3.2.1. Configuring and Tuning OpenShift Logging

You can configure your OpenShift Logging environment by modifying the **ClusterLogging** custom resource deployed in the **openshift-logging** project.

You can modify any of the following components upon install or after install:

Memory and CPU

You can adjust both the CPU and memory limits for each component by modifying the **resources** block with valid memory and CPU values:

```
spec:
  logStore:
    elasticsearch:
      resources:
        limits:
          cpu:
          memory: 16Gi
        requests:
          cpu: 500m
          memory: 16Gi
      type: "elasticsearch"
  collection:
    logs:
      fluentd:
        resources:
          limits:
            cpu:
            memory:
          requests:
            cpu:
            memory:
        type: "fluentd"
  visualization:
    kibana:
      resources:
        limits:
          cpu:
          memory:
        requests:
          cpu:
          memory:
     type: kibana
  curation:
    curator:
      resources:
        limits:
          memory: 200Mi
        requests:
          cpu: 200m
          memory: 200Mi
      type: "curator"
```

Elasticsearch storage

You can configure a persistent storage class and size for the Elasticsearch cluster using the **storageClass name** and **size** parameters. The Cluster Logging Operator creates a persistent volume claim (PVC) for each data node in the Elasticsearch cluster based on these parameters.

```
spec:
  logStore:
    type: "elasticsearch"
    elasticsearch:
      nodeCount: 3
      storage:
        storageClassName: "gp2"
        size: "200G"
```

This example specifies each data node in the cluster will be bound to a PVC that requests "200G" of "gp2" storage. Each primary shard will be backed by a single replica.

NOTE

Omitting the **storage** block results in a deployment that includes ephemeral storage only.

```
spec:
  logStore:
    type: "elasticsearch"
    elasticsearch:
      nodeCount: 3
      storage: {}
```

Elasticsearch replication policy

You can set the policy that defines how Elasticsearch shards are replicated across data nodes in the cluster:

- **FullRedundancy**. The shards for each index are fully replicated to every data node.
- **MultipleRedundancy**. The shards for each index are spread over half of the data nodes.
- **SingleRedundancy**. A single copy of each shard. Logs are always available and recoverable as long as at least two data nodes exist.
- **ZeroRedundancy**. No copies of any shards. Logs may be unavailable (or lost) in the event a node is down or fails.

Curator schedule

You specify the schedule for Curator in the cron format.

```
  spec:
    curation:
    type: "curator"
    resources:
    curator:
      schedule: "30 3 * * *"
```

9.3.2.2. Sample modified ClusterLogging custom resource

The following is an example of a **ClusterLogging** custom resource modified using the options previously described.

Sample modified ClusterLogging custom resource

```
apiVersion: "logging.openshift.io/v1"
kind: "ClusterLogging"
metadata:
  name: "instance"
  namespace: "openshift-logging"
spec:
  managementState: "Managed"
  logStore:
    type: "elasticsearch"
    retentionPolicy:
      application:
        maxAge: 1d
      infra:
        maxAge: 7d
      audit:
        maxAge: 7d
    elasticsearch:
      nodeCount: 3
      resources:
        limits:
          memory: 32Gi
        requests:
          cpu: 3
          memory: 32Gi
        storage:
          storageClassName: "gp2"
          size: "200G"
      redundancyPolicy: "SingleRedundancy"
  visualization:
    type: "kibana"
    kibana:
      resources:
        limits:
          memory: 1Gi
        requests:
          cpu: 500m
          memory: 1Gi
      replicas: 1
  curation:
    type: "curator"
    curator:
      resources:
        limits:
          memory: 200Mi
        requests:
          cpu: 200m
          memory: 200Mi
      schedule: "*/5 * * * *"
  collection:
    logs:
      type: "fluentd"
      fluentd:
        resources:
          limits:
            memory: 1Gi
          requests:
            cpu: 200m
            memory: 1Gi
```

9.3.3. Using OpenShift Logging to find logs for Knative Serving components

Procedure

1. Get the Kibana route:

   ```
   $ oc -n openshift-logging get route kibana
   ```

2. Use the route's URL to navigate to the Kibana dashboard and log in.

3. Check that the index is set to **.all**. If the index is not set to **.all**, only the OpenShift system logs will be listed.

4. Filter the logs by using the **knative-serving** namespace. Enter **kubernetes.namespace_name:knative-serving** in the search box to filter results.

> **NOTE**
>
> Knative Serving uses structured logging by default. You can enable the parsing of these logs by customizing the OpenShift Logging Fluentd settings. This makes the logs more searchable and enables filtering on the log level to quickly identify issues.

9.3.4. Using OpenShift Logging to find logs for services deployed with Knative Serving

With OpenShift Logging, the logs that your applications write to the console are collected in Elasticsearch. The following procedure outlines how to apply these capabilities to applications deployed by using Knative Serving.

Procedure

1. Get the Kibana route:

   ```
   $ oc -n openshift-logging get route kibana
   ```

2. Use the route's URL to navigate to the Kibana dashboard and log in.

3. Check that the index is set to **.all**. If the index is not set to **.all**, only the OpenShift system logs will be listed.

4. Filter the logs by using the **knative-serving** namespace. Enter a filter for the service in the search box to filter results.

   Example filter

   ```
   kubernetes.namespace_name:default AND kubernetes.labels.serving_knative_dev\/service:
   {service_name}
   ```

   You can also filter by using **/configuration** or **/revision**.

5. Narrow your search by using **kubernetes.container_name:<user_container>** to only display the logs generated by your application. Otherwise, you will see logs from the queue-proxy.

NOTE

Use JSON-based structured logging in your application to allow for the quick filtering of these logs in production environments.

9.4. MAPPING AND SPLITTING TRAFFIC FOR DIFFERENT REVISIONS OF A SERVICE

With each update to the configuration of a service, a new revision for the service is created. The service route points all traffic to the latest ready revision by default. You can change this behavior by defining which revisions gets a portion of the traffic.

Knative services allow for traffic mapping, which means that revisions of a service can be mapped to an allocated portion of traffic. Traffic mapping also provides an option to create unique URLs for particular revisions.

9.4.1. Splitting traffic between revisions using the Developer perspective

After you create a serverless application, the serverless application is displayed in the **Topology** view of the **Developer** perspective. The application revision is represented by the node and the serverless resource service is indicated by a quadrilateral around the node.

Any new change in the code or the service configuration triggers a revision, a snapshot of the code at a given time. For a service, you can manage the traffic between the revisions of the service by splitting and routing it to the different revisions as required.

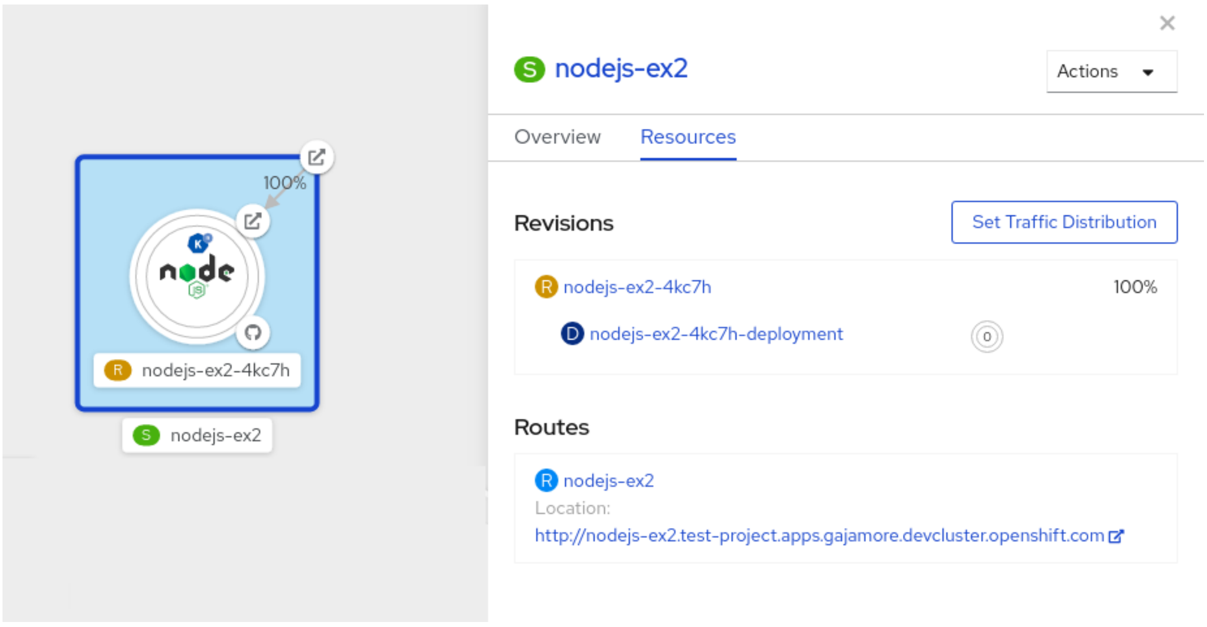
Procedure

To split traffic between multiple revisions of an application in the **Topology** view:

1. Click the serverless resource service, indicated by the quadrilateral, to see its overview in the side panel.

2. Click the **Resources** tab, to see a list of **Revisions** and **Routes** for the service.

Figure 9.1. Serverless application



3. Click the service, indicated by the **S** icon at the top of the side panel, to see an overview of the service details.

4. Click the **YAML** tab and modify the service configuration in the YAML editor, and click **Save**. For example, change the **timeoutseconds** from 300 to 301 . This change in the configuration triggers a new revision. In the **Topology** view, the latest revision is displayed and the **Resources** tab for the service now displays the two revisions.

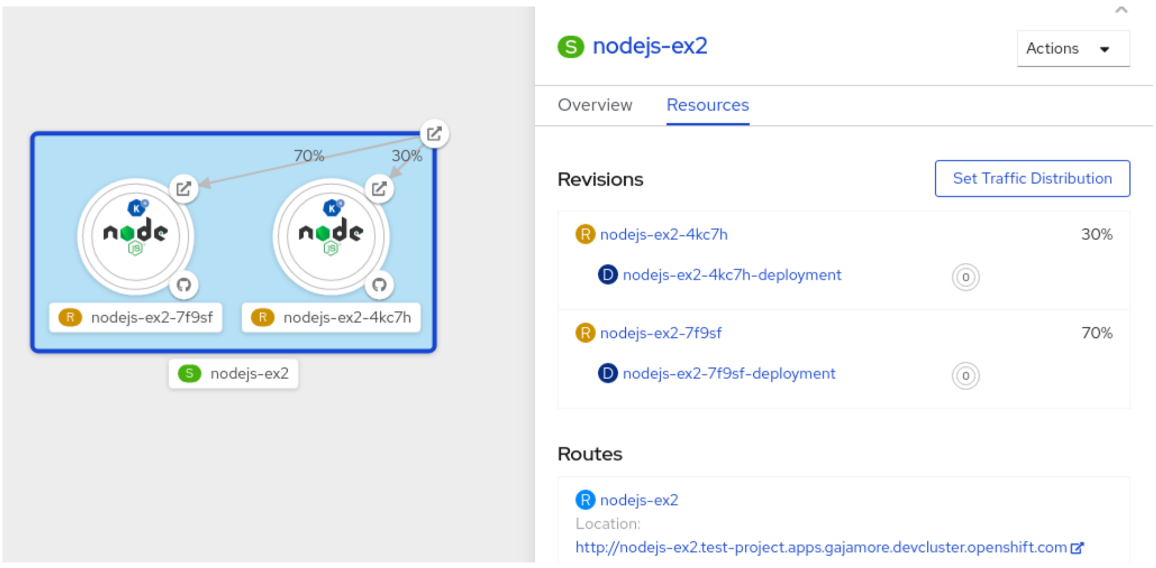
5. In the **Resources** tab, click the **Set Traffic Distribution** button to see the traffic distribution dialog box:

   a. Add the split traffic percentage portion for the two revisions in the **Splits** field.

   b. Add tags to create custom URLs for the two revisions.

   c. Click **Save** to see two nodes representing the two revisions in the Topology view.

Figure 9.2. Serverless application revisions



9.4.2. Traffic mapping and splitting using kn

kn provides commands that help you to control traffic mapping and how traffic is split between revisions.

You can use the **kn service update** command with the **--traffic** flag to update the traffic. This flag uses the following syntax:

```
--traffic RevisionName=Percent
```

where * The **--traffic** flag requires two values separated by separated by an equals sign (**=**). * The **RevisionName** string refers to the name of the revision. * **Percent** integer denotes the traffic portion assigned to the revision.

IMPORTANT

The **--traffic** flag can be specified multiple times in one command, and is valid only if the sum of the **Percent** values in all flags totals 100.

Procedure

- Update the percentage of traffic to be routed to a revision:

```
$ kn service update --traffic <@revision_name>=<percent_integer>
```

NOTE

You can use the identifier **@latest** for the revision name, to refer to the latest ready revision of the service. You can use this identifier only once per command with the **--traffic** flag.

9.4.2.1. Assigning tag revisions

A tag in a traffic block of service creates a custom URL, which points to a referenced revision. A user can define a unique tag for an available revision of a service which creates a custom URL by using the format **http(s)://TAG-SERVICE.DOMAIN**.

A given tag must be unique to its traffic block of the service. **kn** supports assigning and unassigning custom tags for revisions of services as part of the **kn service update** command.

NOTE

If you have assigned a tag to a particular revision, a user can reference the revision by its tag in the **--traffic** flag as **--traffic Tag=Percent**.

Procedure

- Use the following command:

```
$ kn service update svc --tag @latest=candidate --tag svc-vwxyz=current
```

NOTE

--tag RevisionName=Tag uses the following syntax:

- **--tag** flag requires two values separated by a **=**.
- **RevisionName** string refers to name of the **Revision**.
- **Tag** string denotes the custom tag to be given for this Revision.
- Use the identifier **@latest** for the RevisionName to refer to the latest ready revision of the service. You can use this identifier only once with the **--tag** flag.
- If the **service update** command is updating the configuration values for the Service (along with tag flags), **@latest** reference will be pointed to the created Revision after applying the update.
- **--tag** flag can be specified multiple times.
- **--tag** flag may assign different tags to the same revision.

9.4.2.2. Unassigning tag revisions

Tags assigned to revisions in a traffic block can be unassigned. Unassigning tags removes the custom URLs.

NOTE

If a revision is untagged and it is assigned 0% of the traffic, it is removed from the traffic block entirely.

Procedure

- A user can unassign the tags for revisions using the **kn service update** command:

  ```
  $ kn service update svc --untag candidate
  ```

NOTE

--untag Tag uses the following syntax:

- The **--untag** flag requires one value.
- The **tag** string denotes the unique tag in the traffic block of the service which needs to be unassigned. This also removes the respective custom URL.
- The **--untag** flag can be specified multiple times.

9.4.2.3. Traffic flag operation precedence

All traffic-related flags can be specified using a single **kn service update** command. **kn** defines the precedence of these flags. The order of the flags specified when using the command is not taken into account.

The precedence of the flags as they are evaluated by **kn** are:

1. **--untag**: All the referenced revisions with this flag are removed from the traffic block.
2. **--tag**: Revisions are tagged as specified in the traffic block.
3. **--traffic**: The referenced revisions are assigned a portion of the traffic split.

9.4.2.4. Traffic splitting flags

kn supports traffic operations on the traffic block of a service as part of the **kn service update** command.

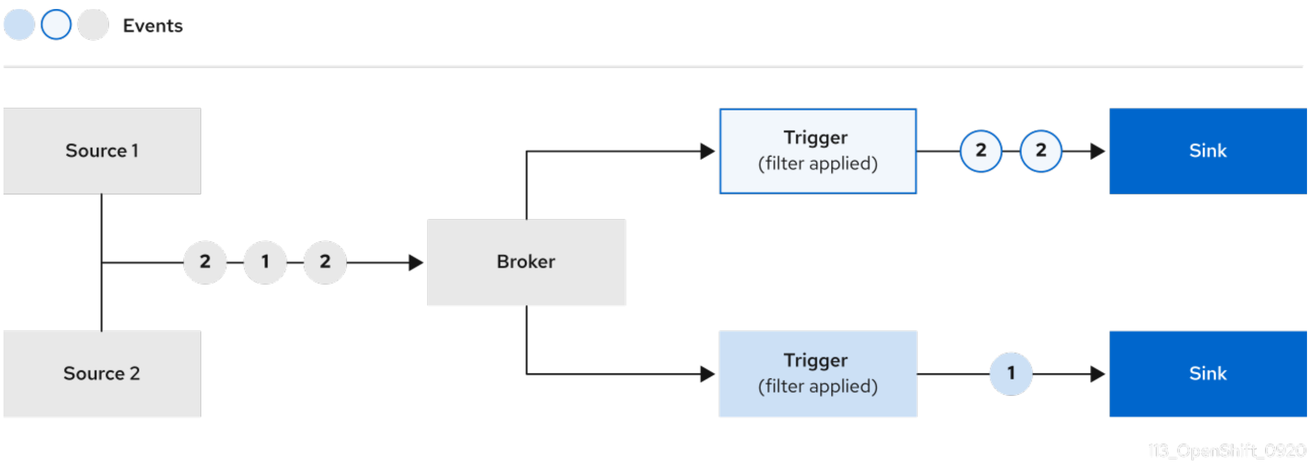
The following table displays a summary of traffic splitting flags, value formats, and the operation the flag performs. The **Repetition** column denotes whether repeating the particular value of flag is allowed in a **kn service update** command.

Flag	Value(s)	Operation	Repetition
--traffic	**RevisionName=Percent**	Gives **Percent** traffic to **RevisionName**	Yes
--traffic	**Tag=Percent**	Gives **Percent** traffic to the revision having **Tag**	Yes
--traffic	**@latest=Percent**	Gives **Percent** traffic to the latest ready revision	No
--tag	**RevisionName=Tag**	Gives **Tag** to **RevisionName**	Yes
--tag	**@latest=Tag**	Gives **Tag** to the latest ready revision	No
--untag	**Tag**	Removes **Tag** from revision	Yes

CHAPTER 10. KNATIVE EVENTING

10.1. BROKERS

Brokers can be used in combination with triggers to deliver events from an event source to an event sink.



Events can be 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.

10.1.1. Creating a broker

OpenShift Serverless provides a **default** Knative broker that you can create by using the **kn** CLI. You can also create the **default** broker by adding the **eventing.knative.dev/injection: enabled** annotation to a trigger, or by adding the **eventing.knative.dev/injection=enabled** label to a namespace.

10.1.1.1. Creating a broker using the Knative CLI

Prerequisites

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

Procedure

- Create the **default** broker:

```
$ kn broker create default
```

Verification

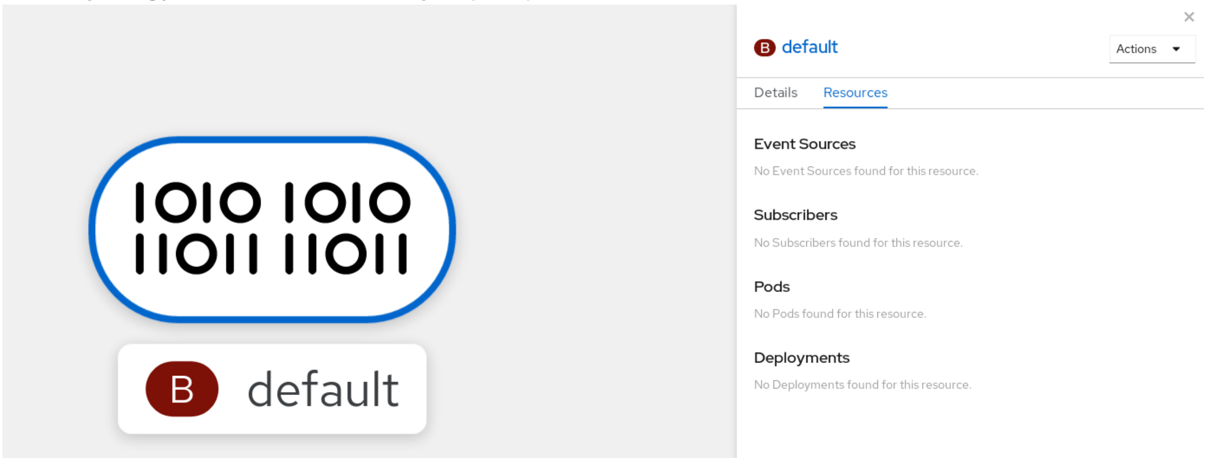
1. Use the **kn** command to list all existing brokers:

```
$ 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:



Event Sources

No Event Sources found for this resource.

Subscribers

No Subscribers found for this resource.

Pods

No Pods found for this resource.

Deployments

No Deployments found for this resource.

10.1.1.2. Creating a broker by annotating a trigger

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.

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 **.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.

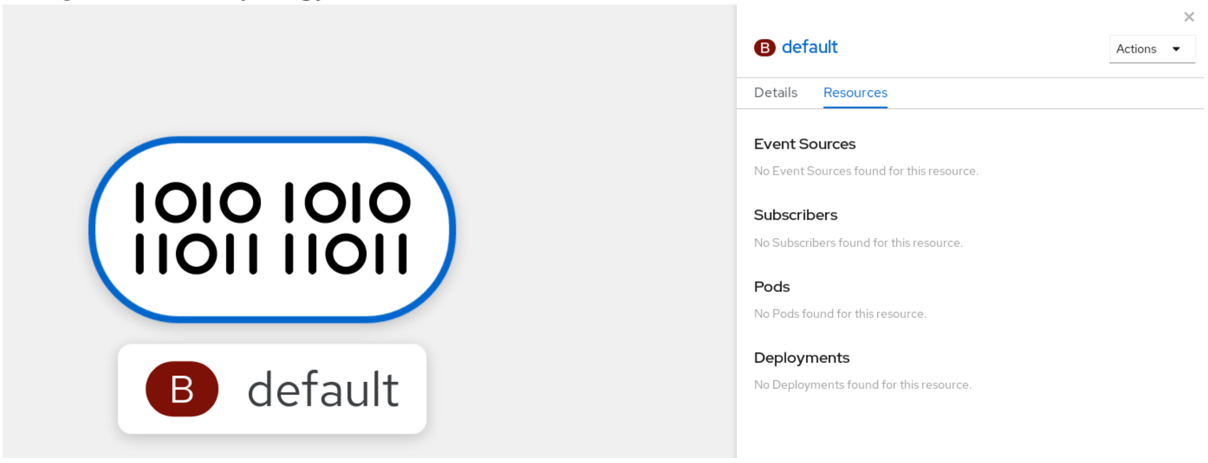
1. Use the **oc** command to get the broker:

```
$ 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. Navigate to the **Topology** view in the web console, and observe that the broker exists:



Event Sources
No Event Sources found for this resource.
Subscribers
No Subscribers found for this resource.
Pods
No Pods found for this resource.
Deployments
No Deployments found for this resource.

10.1.1.3. Creating a broker by labeling a namespace

You can create the **default** broker automatically by labeling a namespace that you own or have write permissions for.

NOTE

Brokers created using this method will not be 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.

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>
```

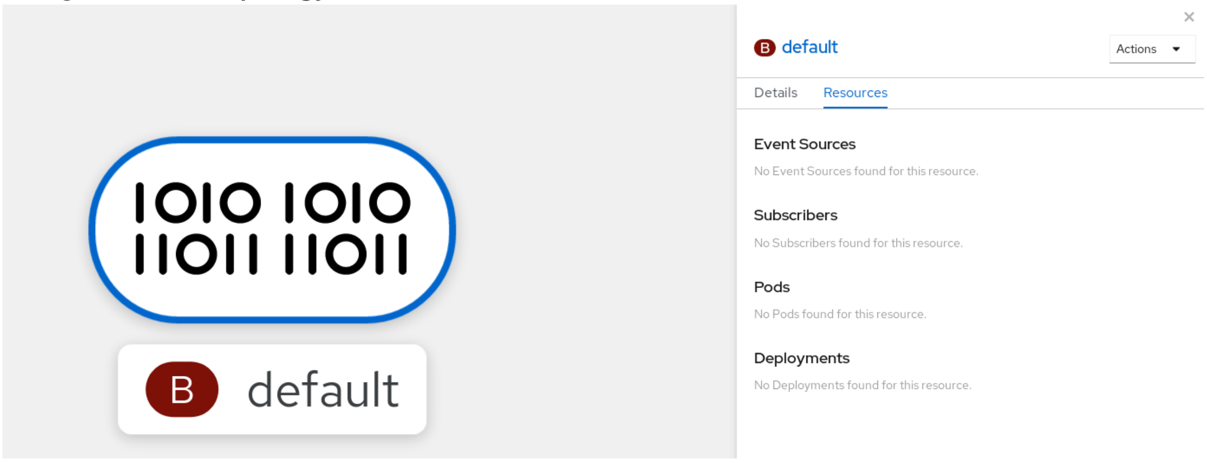
Example command

```
$ 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. Navigate to the **Topology** view in the web console, and observe that the broker exists:



Event Sources
No Event Sources found for this resource.
Subscribers
No Subscribers found for this resource.
Pods
No Pods found for this resource.
Deployments
No Deployments found for this resource.

10.1.1.4. Deleting a broker that was created by injection

Brokers created by injection, by using a namespace label or trigger annotation, are not deleted permanently if a developer removes the label or annotation. You must manually delete these brokers.

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
```

10.1.2. Managing brokers

The **kn** CLI provides commands that can be used to list, describe, update, and delete brokers.

10.1.2.1. Listing existing brokers using the Knative CLI

Prerequisites

- The OpenShift Serverless Operator, Knative Serving and Knative Eventing are installed on your OpenShift Container Platform cluster.
- You have installed the **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
```

10.1.2.2. Describing an existing broker using the Knative CLI

Prerequisites

- The OpenShift Serverless Operator, Knative Serving and Knative Eventing are installed on your OpenShift Container Platform cluster.
- You have installed the **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
```

10.2. FILTERING EVENTS USING TRIGGERS

Using triggers enables you to filter events from the broker for delivery to event sinks.

10.2.1. Prerequisites

- You have installed Knative Eventing and the **kn** CLI.
- You have access to an available broker.
- You have access to an available event consumer, such as a Knative service.

10.2.2. Creating a trigger using the Developer perspective

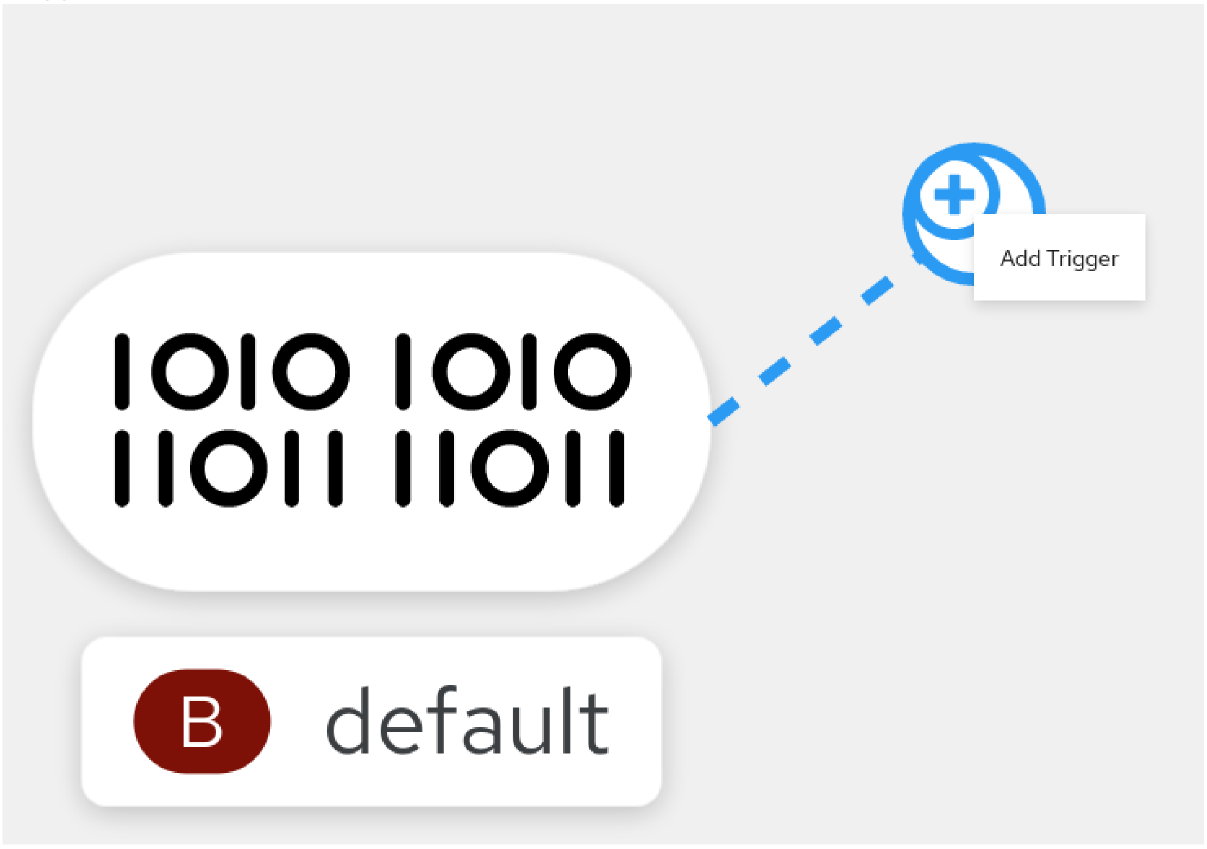
After you have created a broker, you can create a trigger in the web console **Developer** perspective.

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.
- You 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 broker and a Knative service or other event sink to connect to the trigger.

Procedure

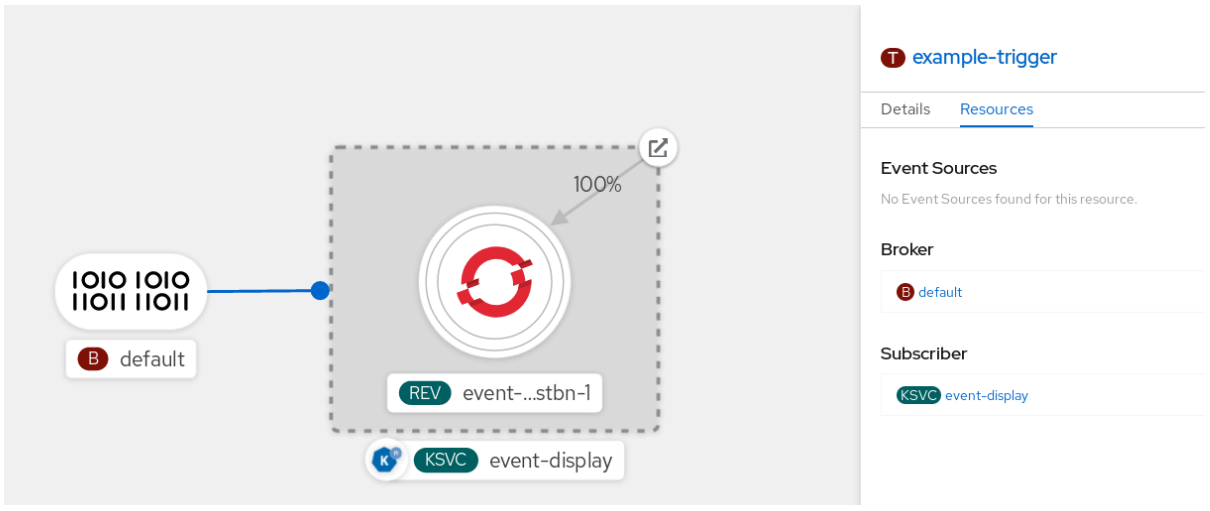
1. In the **Developer** perspective, navigate to the **Topology** page.
2. Hover over the broker that you want to create a trigger for, and drag the arrow. The **Add Trigger** option is displayed.



3. Click **Add Trigger**.
4. Select your sink as a **Subscriber** from the drop-down list.
5. Click **Add**.

Verification

- After the subscription has been created, it is represented as a line that connects the broker to the service in the **Topology** view:



10.2.3. Deleting a trigger using the Developer perspective

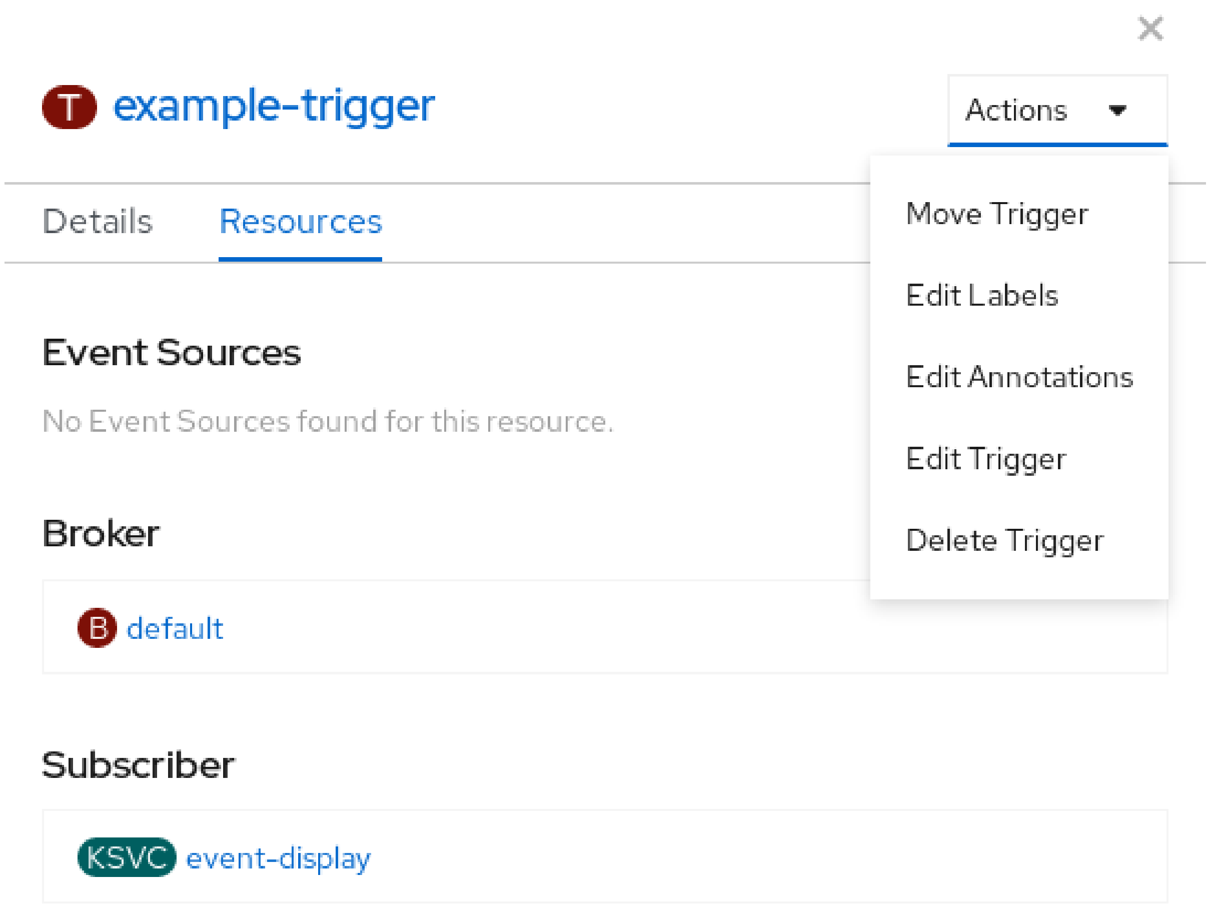
You can delete triggers in the web console **Developer** perspective.

Prerequisites

- To delete a trigger using the **Developer** perspective, ensure that you have logged in to the web console.

Procedure

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**.



10.2.4. Creating a trigger using kn

You can create a trigger by using the **kn trigger create** command.

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.

10.2.5. Listing triggers using kn

The **kn trigger list** command prints a list of available triggers.

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
```

10.2.6. Describing a trigger using kn

You can use the **kn trigger describe** command to print information about 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
```

10.2.7. Filtering events using triggers

In the following trigger example, only events with the attribute **type: dev.knative.samples.helloworld** will reach 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 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
```

10.2.8. Updating a trigger using kn

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

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
    ```

10.2.9. Deleting a trigger using kn

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.
```

10.3. EVENT DELIVERY

You can configure event delivery parameters for Knative Eventing that are applied in cases where an event fails to be delivered by a subscription. Event delivery parameters are configured individually per subscription.

10.3.1. Event delivery behavior for Knative Eventing channels

Different Knative Eventing channel types have their own behavior patterns that are followed for event delivery. Developers can set event delivery parameters in the subscription configuration to ensure that any events that fail to be delivered from channels to an event sink are retried. You must also configure a dead letter sink for subscriptions if you want to provide a sink where events that are not eventually delivered can be stored, otherwise undelivered events are dropped.

10.3.1.1. Event delivery behavior for Knative Kafka channels

If an event is successfully delivered to a Kafka channel or broker receiver, 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 this issue.

10.3.1.2. Delivery failure status codes

The channel or broker receiver can respond with the following status codes if an event fails to be delivered:

500

This is a generic status code which means that the event was not delivered successfully.

404

This status code means that the channel or broker the event is being delivered to does not exist, or that the **Host** header is incorrect.

400

This status code means that the event being sent to the receiver is invalid.

10.3.2. Configurable 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 sent to the specified event sink.

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.

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 the time interval specified between retries. When using the **exponential** backoff policy, the back off delay is equal to **backoffDelay*2^<numberOfRetries>**.

10.3.3. Configuring event delivery failure parameters using subscriptions

Developers can configure event delivery parameters for individual subscriptions by modifying the **delivery** settings for a **Subscription** object.

Example subscription YAML

```
apiVersion: messaging.knative.dev/v1
kind: Subscription
metadata:
  name: <subscription_name>
  namespace: <subscription_namespace>
spec:
  delivery:
    deadLetterSink: 1
      ref:
        apiVersion: serving.knative.dev/v1
        kind: Service
        name: <sink_name>
    backoffDelay: <duration> 2
    backoffPolicy: <policy_type> 3
    retry: <integer> 4
```

1 Configuration settings to enable using a dead letter sink. This tells the subscription what happens to events that cannot be delivered to the subscriber.

When this is configured, events that fail to be delivered are sent to the dead letter sink destination. The destination can be a Knative service or a URI.

2 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.

3 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 the time interval specified between retries. When using the **exponential** back off policy, the back off delay is equal to **backoffDelay*2^<numberOfRetries>**.

4 The number of times that event delivery is retried before the event is sent to the dead letter sink.

10.3.4. Additional resources

- See Knative Eventing workflows using channels for more information about subscriptions.

- See Creating subscriptions.

10.4. KNATIVE KAFKA

You can use the **KafkaChannel** channel type and **KafkaSource** event source with OpenShift Serverless. To do this, you must install the Knative Kafka components, and configure the integration between OpenShift Serverless and a supported Red Hat AMQ Streams cluster.

NOTE

Knative Kafka is not currently supported for IBM Z and IBM Power Systems.

The OpenShift Serverless Operator provides the Knative Kafka API that can be used to create a **KnativeKafka** custom resource:

Example KnativeKafka custom resource

```
apiVersion: operator.serverless.openshift.io/v1alpha1
kind: KnativeKafka
metadata:
  name: knative-kafka
  namespace: knative-eventing
spec:
  channel:
    enabled: true 1
    bootstrapServers: <bootstrap_server> 2
  source:
    enabled: true 3
```

1 Enables developers to use the **KafkaChannel** channel type in the cluster.

2 A comma-separated list of bootstrap servers from your AMQ Streams cluster.

3 Enables developers to use the **KafkaSource** event source type in the cluster.

10.4.1. Installing Knative Kafka components by using the web console

Cluster administrators can enable the use of Knative Kafka functionality in an OpenShift Serverless deployment by instantiating the **KnativeKafka** custom resource definition provided by the **Knative Kafka** OpenShift Serverless Operator API.

Prerequisites

- You have installed OpenShift Serverless, including Knative Eventing, in your OpenShift Container Platform cluster.
- You have access to a Red Hat AMQ Streams cluster.
- You have cluster administrator permissions on OpenShift Container Platform.
- You are logged in to the web console.

Procedure

1. In the **Administrator** perspective, navigate to **Operators → Installed Operators**.
2. Check that the **Project** dropdown at the top of the page is set to **Project: knative-eventing**.
3. In the list of **Provided APIs** for the OpenShift Serverless Operator, find the **Knative Kafka** box and click **Create Instance**.
4. Configure the **KnativeKafka** object in the **Create Knative Kafka** page.

   IMPORTANT

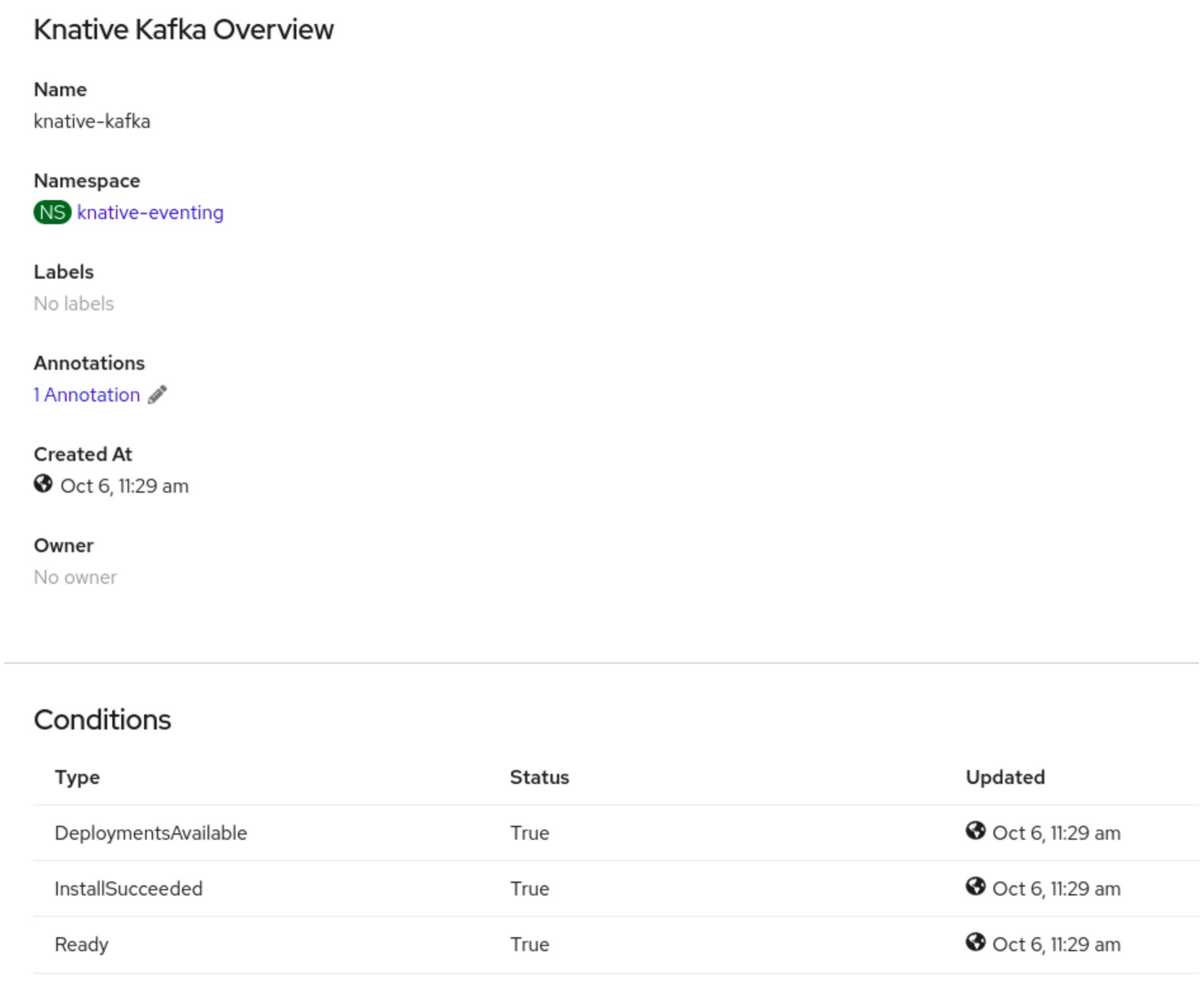
   To use the Kafka channel or Kafka source on your cluster, you must toggle the **Enable** switch for the options you want to use to **true**. These switches are set to **false** by default. Additionally, to use the Kafka channel, you must specify the Boostrap Servers.

   a. Using the form is recommended for simpler configurations that do not require full control of **KnativeKafka** object creation.

   b. Editing the YAML is recommended for more complex configurations that require full control of **KnativeKafka** object creation. You can access the YAML by clicking the **Edit YAML** link in the top right of the **Create Knative Kafka** page.
5. Click **Create** after you have completed any of the optional configurations for Kafka. You are automatically directed to the **Knative Kafka** tab where **knative-kafka** is in the list of resources.

Verification

1. Click on the **knative-kafka** resource in the **Knative Kafka** tab. You are automatically directed to the **Knative Kafka Overview** page.
2. View the list of **Conditions** for the resource and confirm that they have a status of **True**.


Knative Kafka Overview

Name
knative-kafka

Namespace
NS knative-eventing

Labels
No labels

Annotations
1 Annotation

Created At
Oct 6, 11:29 am

Owner
No owner

Conditions

Type	Status	Updated
DeploymentsAvailable	True	Oct 6, 11:29 am
InstallSucceeded	True	Oct 6, 11:29 am
Ready	True	Oct 6, 11:29 am

If the conditions have a status of **Unknown** or **False**, wait a few moments to refresh the page.

3. Check that the Knative Kafka resources have been created:

```
$ oc get pods -n knative-eventing
```

Example output

```
NAME                                          READY   STATUS    RESTARTS   AGE
kafka-ch-controller-85f879d577-xcbjh          1/1     Running   0          44s
kafka-ch-dispatcher-55d76d7db8-ggqjl          1/1     Running   0          44s
kafka-controller-manager-bc994c465-pt7qd      1/1     Running   0          40s
kafka-webhook-54646f474f-wr7bb                1/1     Running   0          42s
```

10.4.2. Using Kafka channels

Create a Kafka channel.

10.4.3. Using Kafka source

Create a Kafka event source.

10.4.4. Configuring authentication for Kafka

In production, Kafka clusters are often secured using the TLS or SASL authentication methods. This section shows how to configure the Kafka channel to work against a protected Red Hat AMQ Streams (Kafka) cluster using TLS or SASL.

NOTE

If you choose to enable SASL, Red Hat recommends to also enable TLS.

10.4.4.1. Configuring TLS authentication

Prerequisites

- A Kafka cluster CA certificate as a **.pem** file.
- A Kafka cluster client certificate and key as **.pem** files.

Procedure

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

```
$ kubectl create secret --namespace <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_server>
    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
```

Additional resources

- TLS and SASL on Kafka

10.4.4.2. Configuring SASL authentication

Prerequisites

- A username and password for the Kafka cluster.
- Choose 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.

NOTE

Red Hat recommends to enable TLS in addition to SASL.

Procedure

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

```
$ kubectl create secret --namespace <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"
```

IMPORTANT

Use the key names **ca.crt**, **password**, and **saslType**. 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_server>
    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
```

Additional resources

- TLS and SASL on Kafka

CHAPTER 11. EVENT SOURCES

11.1. GETTING STARTED WITH EVENT SOURCES

An *event source* is an object that links an event producer with an event *sink*, or consumer. A sink can be a Knative service, channel, or broker that receives events from an event source.

Currently, OpenShift Serverless supports the following event source types:

API server source
: Connects a sink to the Kubernetes API server.

Ping source
: Periodically sends ping events with a constant payload. It can be used as a timer.

Sink binding
: Allows you to connect core Kubernetes resource objects, such as **Deployment**, **Job**, or **StatefulSet** objects, with a sink.

Knative Kafka source
: Connect a Kafka cluster to a sink as an event source.

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

- Create an API server source.
- Create an ping source.
- Create a sink binding.
- Create a Kafka source.

11.1.1. Additional resources

- For more information about eventing workflows using OpenShift Serverless, see Knative Eventing architecture.

11.2. LISTING EVENT SOURCES AND EVENT SOURCE TYPES

You can use the **kn** CLI or the **Developer** perspective in the OpenShift Container Platform web console to list and manage available event sources or event source types.

Currently, OpenShift Serverless supports the following event source types:

API server source
: Connects a sink to the Kubernetes API server.

Ping source
: Periodically sends ping events with a constant payload. It can be used as a timer.

Sink binding
: Allows you to connect core Kubernetes resource objects, such as **Deployment**, **Job**, or **StatefulSet** objects, with a sink.

Knative Kafka source

Connect a Kafka cluster to a sink as an event source.

11.2.1. Listing available event source types by using the Knative 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: You can also list the available event source types in YAML format:

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

11.2.2. Viewing available event source types within the Developer perspective

You can use the web console to view available event source types.

NOTE

Additional event source types can be added by cluster administrators by installing Operators on OpenShift Container Platform.

Procedure

1. Access the **Developer** perspective.
2. Click **+Add**.
3. Click **Event source**.

11.2.3. Listing available event sources using the Knative CLI

- List the available event sources:

```
$ 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
```

11.2.3.1. Listing event sources of a specific type only

You can list event sources of a specific type only, by using the **--type** flag.

- List the available ping sources:

  ```
  $ kn source list --type PingSource
  ```

  Example output

  ```
  NAME   TYPE           RESOURCE                          SINK        READY
  p1    PingSource      pingsources.sources.knative.dev      ksvc:eshow1   True
  ```

11.3. USING THE 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.

11.3.1. Prerequisites

- You must have a current installation of OpenShift Serverless, including Knative Serving and Eventing, in your OpenShift Container Platform cluster. This can be installed by a cluster administrator.
- Event sources need a service to use as an event *sink*. The sink is the service or application that events are sent to from the event source.
- You must create or update a service account, role and role binding for the event source.

NOTE

Some of the following procedures require 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.

11.3.2. Creating a service account, role, and binding for event sources

Procedure

1. Create a service account, role, and role binding for the event source by creating a file named **authentication.yaml** and copying the following sample code into it:

   ```
   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.

NOTE

If you want to re-use an existing service account with the appropriate permissions, you must modify the **authentication.yaml** for that service account.

2. Create the service account, role binding, and cluster binding by entering the following command:

```
$ oc apply --filename authentication.yaml
```

11.3.3. Creating an API server source event source using the Developer perspective

Procedure

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

2. 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.

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

4. 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**.

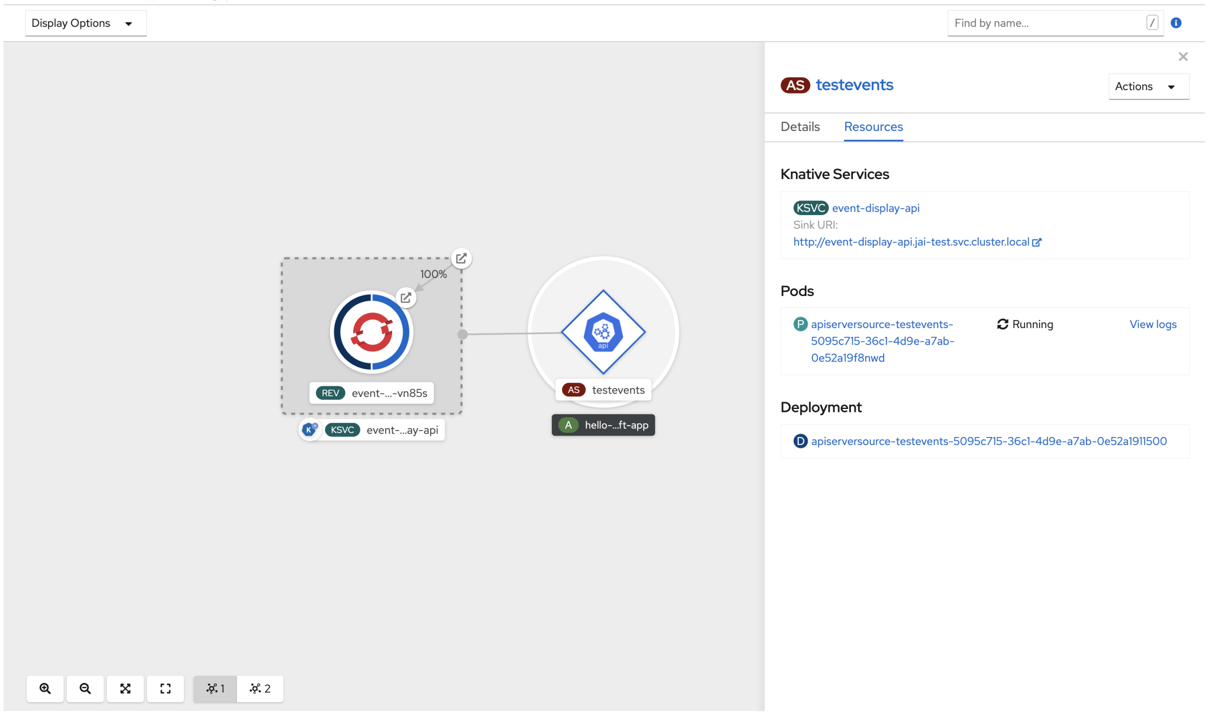
   b. Select the **Service Account Name** for the service account that you created.

   c. Select the **Sink** for the event source. A **Sink** can be either a **Resource**, such as a channel, broker, or service, or a **URI**.

5. Click **Create**.

Verification

- After you have created the API server source, you will see it connected to the service it is sinked to in the **Topology** view.



AS testevents

Actions

Details Resources

Knative Services

KSVC event-display-api
Sink URI:
http://event-display-api.jai-test.svc.cluster.local

Pods

P apiserversource-testevents-5095c715-36c1-4d9e-a7ab-0e52a19f8nwd Running View logs

Deployment

D apiserversource-testevents-5095c715-36c1-4d9e-a7ab-0e52a1911500

NOTE

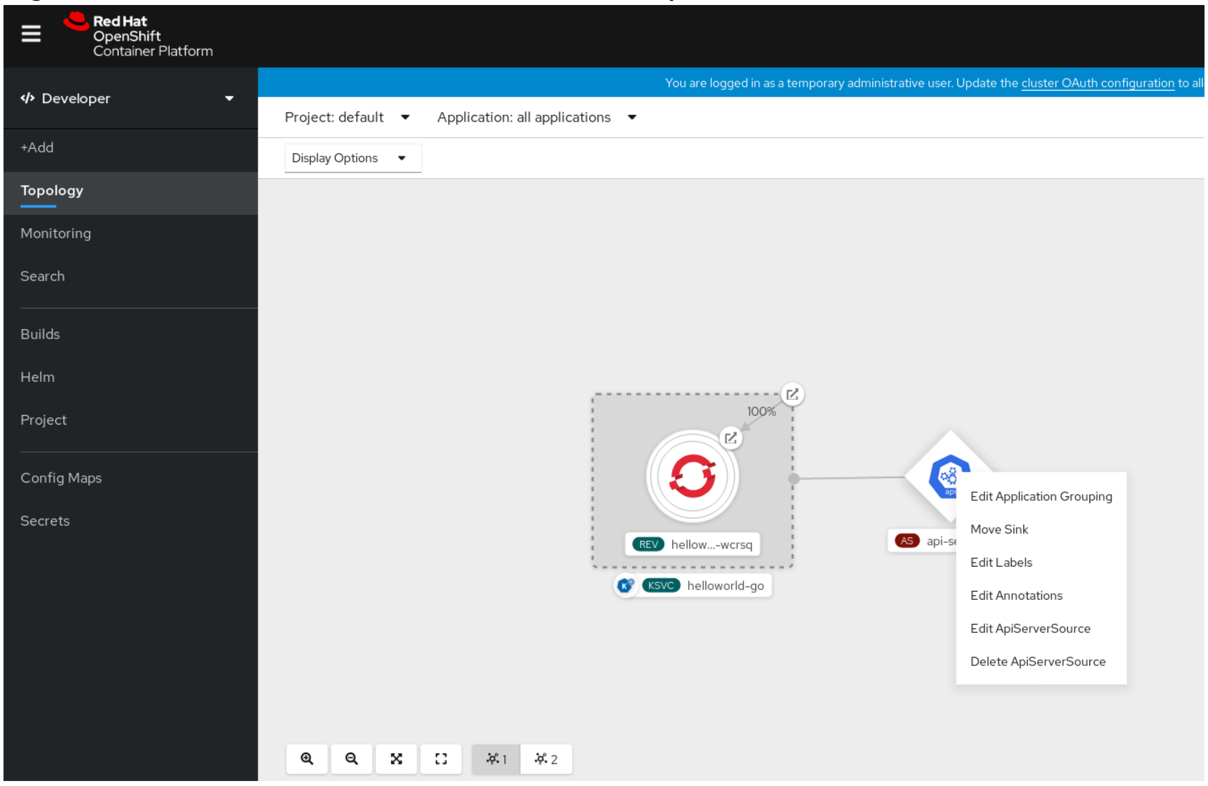
If a URI sink is used, modify the URI by right-clicking on **URI sink** → **Edit URI**.

11.3.4. Deleting an API server source using the Developer perspective

Procedure

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

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



11.3.5. Using the API server source with the Knative CLI

This section describes the steps required to create an API server source using **kn** commands.

Prerequisites

- You must have OpenShift Serverless, the Knative Serving and Eventing components, and the **kn** CLI installed.

Procedure

1. Create an API server source that uses a broker as a sink:

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

2. 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 <service_name> --image quay.io/openshift-knative/knative-eventing-
sources-event-display:latest
```

3. Create a trigger to filter events from the **default** broker to the service:

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

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

```
$ oc create deployment hello-node --image quay.io/openshift-knative/knative-eventing-
sources-event-display:latest
```

5. 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

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

1. Get the pods:

```
$ oc get pods
```

2. View the message dumper function logs for the 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.apiserver.resource.update
  datacontenttype: application/json
  ...
Data,
  {
    "apiVersion": "v1",
  "involvedObject": {
    "apiVersion": "v1",
    "fieldPath": "spec.containers{hello-node}",
    "kind": "Pod",
    "name": "hello-node",
    "namespace": "default",
     .....
  },
  "kind": "Event",
  "message": "Started container",
  "metadata": {
    "name": "hello-node.159d7608e3a3572c",
    "namespace": "default",
    ....
  },
  "reason": "Started",
  ...
}
```

11.3.5.1. Knative CLI (kn) --sink flag

When you create an event-producing custom resource 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 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**.

11.3.6. Deleting the API server source using the Knative CLI

This section describes the steps used to delete the API server source, trigger, service account, cluster role, and cluster role binding using **kn** and **oc** commands.

Prerequisites

- You must have the **kn** CLI installed.

Procedure

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
```

11.3.7. Using the API server source with the YAML method

This guide describes the steps required to create an API server source using YAML files.

Prerequisites

- You will need to have a Knative Serving and Eventing installation.
- You will need to have created the **default** broker in the same namespace as the one defined in the API server source YAML file.

Procedure

1. To create a service account, role, and role binding for the API server source, create a file named **authentication.yaml** and copy the following sample code into it:

```
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 API server source.

NOTE

If you want to re-use an existing service account with the appropriate permissions, you must modify the **authentication.yaml** for that service account.

After you have created the **authentication.yaml** file, apply it:

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

2. To create an API server source, create a file named **k8s-events.yaml** and copy the following sample code into it:

```
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
```

After you have created the **k8s-events.yaml** file, apply it:

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

3. To check that the API server source is set up correctly, create a Knative service that dumps incoming messages to its log.
Copy the following sample YAML into a file named **service.yaml**:

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: event-display
  namespace: default
spec:
    template:
      spec:
        containers:
          - image: quay.io/openshift-knative/knative-eventing-sources-event-display:latest
```

After you have created the **service.yaml** file, apply it:

```
$ oc apply -f service.yaml
```

4. To create a trigger from the **default** broker that filters events to the service created in the previous step, create a file named **trigger.yaml** and copy the following sample code into it:

```
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
```

After you have created the **trigger.yaml** file, apply it:

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

5. To create events, launch a Pod in the default namespace:

```
$ oc create deployment hello-node --image=quay.io/openshift-knative/knative-eventing-sources-event-display
```

6. To check that the controller is mapped correctly, enter the following command and inspect 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 message dumper function logs.

1. Get the pods:

   ```
   $ oc get pods
   ```

2. View the message dumper function logs for the 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.apiserver.resource.update
     datacontenttype: application/json
     ...
   Data,
     {
       "apiVersion": "v1",
       "involvedObject": {
         "apiVersion": "v1",
         "fieldPath": "spec.containers{hello-node}",
         "kind": "Pod",
         "name": "hello-node",
         "namespace": "default",
          .....
       },
       "kind": "Event",
       "message": "Started container",
       "metadata": {
   ```

```
      "name": "hello-node.159d7608e3a3572c",
      "namespace": "default",
      ....
    },
    "reason": "Started",
    ...
  }
```

11.3.8. Deleting the API server source

This section describes how to delete the API server source, trigger, service account, cluster role, and cluster role binding by deleting their YAML files.

Procedure

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
```

11.4. USING A PING SOURCE

A ping source is 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.

Example ping source YAML

```
apiVersion: sources.knative.dev/v1alpha2
kind: PingSource
metadata:
  name: test-ping-source
spec:
  schedule: "*/2 * * * *" (1)
  jsonData: '{"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**.

11.4.1. Creating a ping source using the Developer perspective

You can create and verify a basic ping source from the OpenShift Container Platform web console.

Prerequisites

To create a ping source using the **Developer** perspective, ensure that:

- 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 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.

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/knative-eventing-sources-event-display:latest
```

   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. Select a **Sink**. This can be either a **Resource** or a **URI**. In this example, the **event-display** service created in the previous step is used as the **Resource** sink.

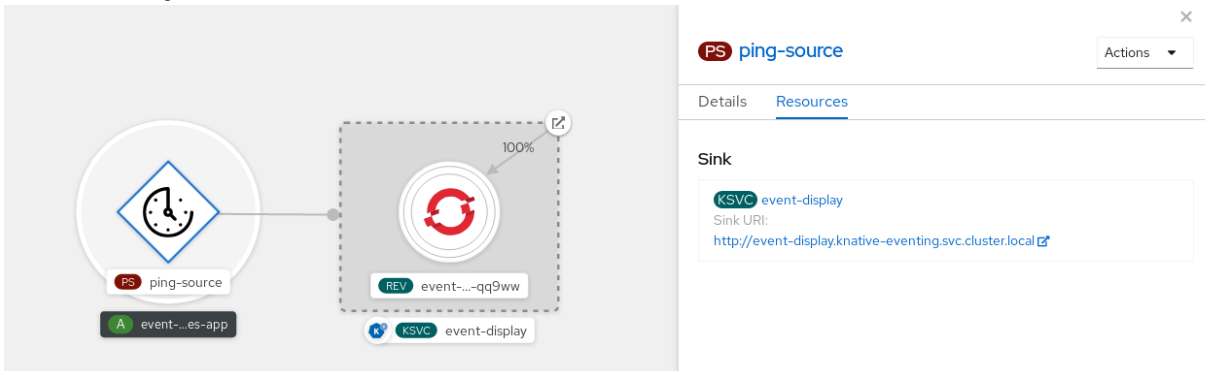
g. Click **Create**.

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 PingSource and sink.



Sink

KSVC event-display

Sink URI:

http://event-display.knative-eventing.svc.cluster.local

11.4.2. Using a ping source with the Knative CLI

The following procedure describes how to create a basic ping source by using the **kn** CLI.

Prerequisites

- You have Knative Serving and Eventing installed.
- You have the **kn** CLI installed.

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/knative-eventing-sources-event-display:latest
```

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!"
  }
```

11.4.2.1. Knative CLI (kn) --sink flag

When you create an event-producing custom resource 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 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**.

11.4.3. Deleting a ping source by using the Knative CLI

The following procedure describes how to delete a ping source using the **kn** CLI.

- Delete the ping source:

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

11.4.4. Using a ping source with YAML

The following sections describe how to create a basic ping source using YAML files.

Prerequisites

- You have Knative Serving and Eventing installed.

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 verify that the ping source is working, create a simple Knative service that dumps incoming messages to the service's logs.

   a. Copy the example YAML into a file named **service.yaml**:

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

   b. Create the service:

   ```
   $ oc apply --filename service.yaml
   ```

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. Copy the example YAML into a file named **ping-source.yaml**:

   ```
   apiVersion: sources.knative.dev/v1alpha2
   kind: PingSource
   metadata:
     name: test-ping-source
   spec:
     schedule: "*/2 * * * *"
     jsonData: '{"message": "Hello world!"}'
     sink:
       ref:
         apiVersion: serving.knative.dev/v1
         kind: Service
         name: event-display
   ```

   b. Create the ping source:

   ```
   $ oc apply --filename ping-source.yaml
   ```

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

   ```
   $ oc get pingsource.sources.knative.dev test-ping-source -oyaml
   ```

Example output

```
apiVersion: sources.knative.dev/v1alpha2
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/v1alpha2/namespaces/default/pingsources/test-ping-
source
  uid: 3d80d50b-f8c7-4c1b-99f7-3ec00e0a8164
spec:
  jsonData: '{ 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!"
}
```

11.4.5. Deleting a ping source that was created by using YAML

The following procedure describes how to delete a ping source that was created by using YAML.

Procedure

- Delete the ping source:

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

  Example command

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

11.5. USING SINK BINDING

Sink binding is used to connect event producers, or *event sources*, to an event consumer, or *event sink*, for example, a Knative service or application.

IMPORTANT

Before developers can use sink binding, cluster administrators must label the namespace that will be configured for sink binding with **bindings.knative.dev/include:"true"**:

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

11.5.1. Using sink binding with the Knative CLI

This guide describes the steps required to create a sink binding instance using **kn** commands.

Prerequisites

- You have Knative Serving and Eventing installed.
- You have the **kn** CLI installed.

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.

IMPORTANT

Before developers can use sink binding, cluster administrators must label the namespace that will be configured for sink binding with **bindings.knative.dev/include:"true"**:

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

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/knative-eventing-
sources-event-display:latest
```

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** resource.

   a. Create a file named **heartbeats-cronjob.yaml** and copy the following sample code into it:

```
apiVersion: batch/v1beta1
kind: CronJob
metadata:
  name: heartbeat-cron
spec:
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/knative-eventing-sources-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. After you have created the **heartbeats-cronjob.yaml** file, apply it by entering:

```
$ oc apply -f heartbeats-cronjob.yaml
```

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": ""
  }
```

11.5.1.1. Knative CLI (kn) --sink flag

When you create an event-producing custom resource 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 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**.

11.5.2. Using sink binding with the YAML method

This guide describes the steps required to create a sink binding instance using YAML files.

Prerequisites

- You have Knative Serving and Eventing installed.

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.

IMPORTANT

Before developers can use a SinkBinding, cluster administrators must label the namespace that will be configured in the SinkBinding with **bindings.knative.dev/include:"true"**:

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

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. Copy the following sample YAML into a file named **service.yaml**:

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

   b. After you have created the **service.yaml** file, apply it by entering:

   ```
   $ oc apply -f service.yaml
   ```

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

   a. Create a file named **sinkbinding.yaml** and copy the following sample code into it:

   ```
   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. After you have created the **sinkbinding.yaml** file, apply it by entering:

```
$ oc apply -f sinkbinding.yaml
```

3. Create a **CronJob** resource.

a. Create a file named **heartbeats-cronjob.yaml** and copy the following sample code into it:

```
apiVersion: batch/v1beta1
kind: CronJob
metadata:
  name: heartbeat-cron
spec:
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/knative-eventing-sources-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. After you have created the **heartbeats-cronjob.yaml** file, apply it by entering:

```
$ oc apply -f heartbeats-cronjob.yaml
```

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": ""
  }
```

11.6. USING A KAFKA SOURCE

You can create a Knative Kafka event source that reads events from an Apache Kafka cluster and passes these events to a sink.

11.6.1. Prerequisites

You can use the **KafkaSource** event source with OpenShift Serverless after you have Knative Eventing and Knative Kafka installed on your cluster.

11.6.2. Creating a Kafka event source by using the web console

You can create and verify a Kafka event source from the OpenShift Container Platform web console.

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 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.

Procedure

1. 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. Select the **Sink** for the event source. A **Sink** can be either a **Resource**, such as a channel, broker, or service, or a **URI**.

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

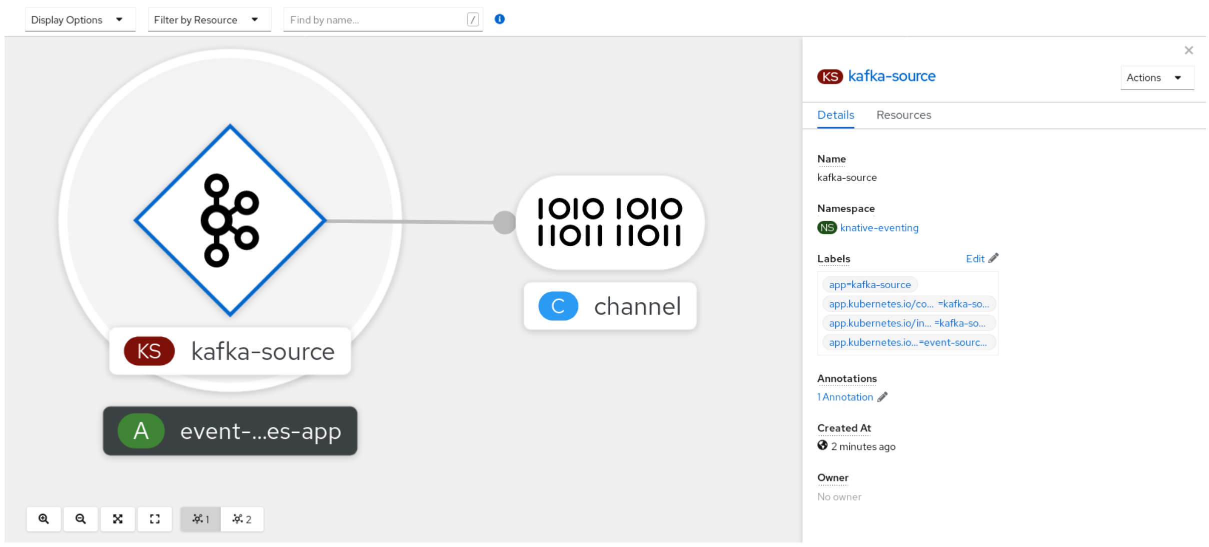
4. Click **Create**.

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.



Name
kafka-source

Namespace
NS knative-eventing

Labels Edit
app=kafka-source
app.kubernetes.io/co... =kafka-so...
app.kubernetes.io/in... =kafka-so...
app.kubernetes.io...=event-sourc...

Annotations
1 Annotation

Created At
2 minutes ago

Owner
No owner

11.6.3. Creating a Kafka event source by using the kn CLI

This section describes how to create a Kafka event source by using the **kn** command.

Prerequisites

- The OpenShift Serverless Operator, Knative Eventing, Knative Serving, 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.

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/knative-eventing-sources-event-display
   ```

2. Create a **KafkaSource** resource:

   ```
   $ kn source kafka create mykafkasrc \
       --servers my-cluster-kafka-bootstrap.kafka.svc:9092 \
       --topics my-topic --consumergroup my-consumer-group \
       --sink event-display
   ```

   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** resource you created:

   ```
   $ kn source kafka describe mykafkasrc
   ```

   Example output

   ```
   Name:              mykafkasrc
   Namespace:         kafka
   Age:               1h
   BootstrapServers:  my-cluster-kafka-bootstrap.kafka.svc:9092
   Topics:            my-topic
   ConsumerGroup:     my-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 my-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/mykafkasrc#my-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!
```

11.6.3.1. Knative CLI (kn) --sink flag

When you create an event-producing custom resource 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 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**.

11.6.4. Creating a Kafka event source by using YAML

You can create a Kafka event source by using YAML.

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.

Procedure

1. Create a YAML file containing the following:

```
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:
```

(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.

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 YAML file:

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

Verification

- Verify that the Kafka event source was created:

```
$ oc get pods
```

Example output

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

11.6.5. Additional resources

- See Getting started with event sources.
- See Knative Kafka.
- See the Red Hat AMQ Streams documentation for more information about Kafka concepts.

11.7. CONNECTING EVENT SOURCES TO SINKS

When you create an event source, you can specify a sink where events are sent to from the source. A sink is an addressable resource that can receive incoming events from other Knative Eventing resources. Knative services, channels, and brokers are all examples of sinks.

11.7.1. Knative CLI (kn) --sink flag

When you create an event-producing custom resource 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 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**.

11.7.2. Connect an event source to a channel using the Developer perspective

You can create multiple event source types in OpenShift Container Platform that can be connected to channels.

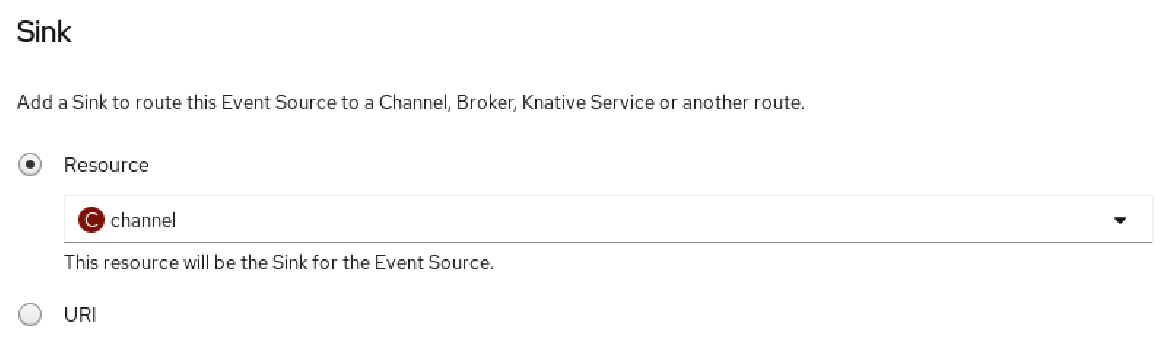
Prerequisites

To connect an event source to a channel using the **Developer** perspective, ensure that:

- 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 channel by following the documentation on *Creating a channel using OpenShift Container Platform web console*.

Procedure

1. Create an event source of any type, by following the documentation on *Getting started with event sources*.
2. In the **Developer** perspective, navigate to **Event Sources**.
3. In the **Sink** section of the **Event Sources** form view, select **Resource**. Then use the drop-down to select your channel.



Sink

Add a Sink to route this Event Source to a Channel, Broker, Knative Service or another route.

[✓ Resource]

C channel

This resource will be the Sink for the Event Source.

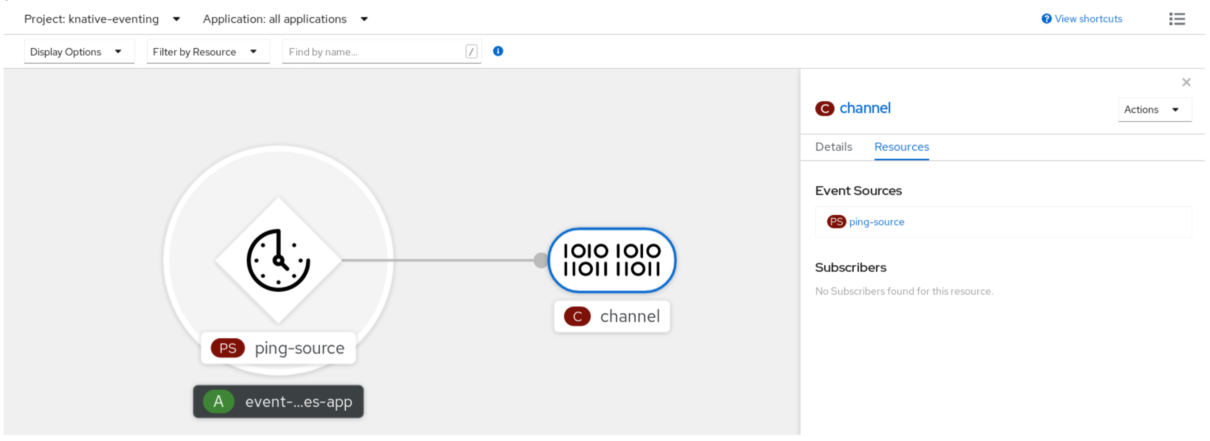
URI

4. 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 on the connected channel to see the channel details in the side panel.



Project: knative-eventing

Application: all applications

Display Options

Filter by Resource

Find by name...

View shortcuts

C channel

Actions

Details

Resources

Event Sources

PS ping-source

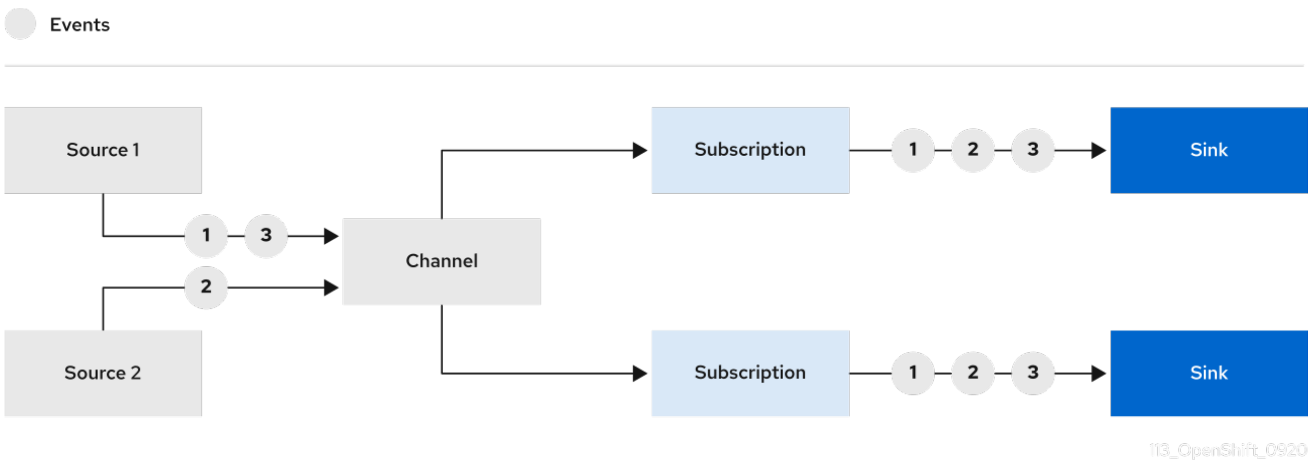
Subscribers

No Subscribers found for this resource.

CHAPTER 12. CHANNELS

12.1. UNDERSTANDING 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.

InMemoryChannel and **KafkaChannel** channel implementations can be used with OpenShift Serverless for development use.

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.

12.1.1. Next steps

- If you are a cluster administrator, you can configure default settings for channels. See Configuring channel defaults.
- See Creating and deleting channels.

12.2. CREATING AND DELETING CHANNELS

Developers can create channels by instantiating a supported **Channel** 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
```

NOTE

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

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

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 Knative 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.

12.2.1. Creating a channel using the Developer perspective

You can create a channel with the cluster default configuration by using the OpenShift Container Platform web console.

Prerequisites

To create channels using the **Developer** perspective ensure that:

- The OpenShift Serverless Operator 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.

Procedure

1. In the **Developer** perspective, navigate to **+Add → Channel**.
2. Select the type of **Channel** object that you want to create from the **Type** drop-down.

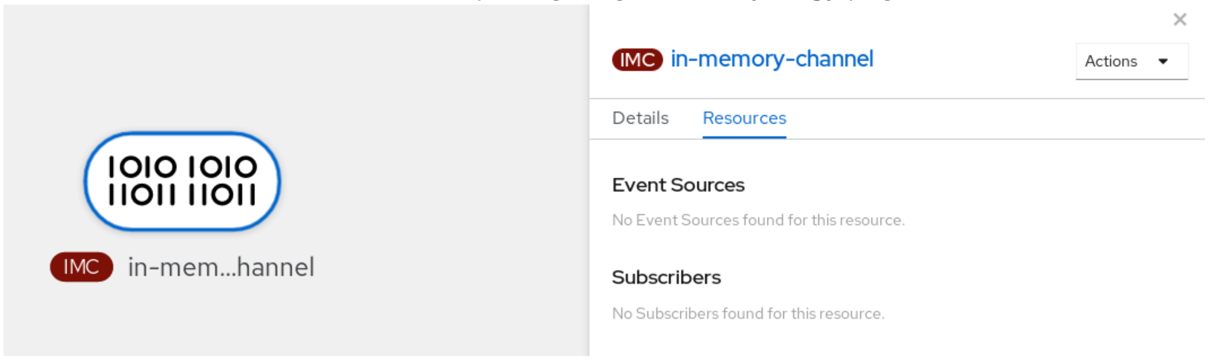
   NOTE

   Currently only InMemoryChannel type **Channel** objects are supported.

3. Click **Create**.

Verification

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



12.2.2. Creating a channel using the Knative CLI

You can create a channel with the cluster default configuration by using the **kn** CLI.

Prerequisites

To create channels using the **kn** CLI, ensure that:

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

12.2.3. Creating a default implementation channel by using YAML

You can create a channel by using YAML with the cluster default configuration.

Prerequisites

- 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

To create a **Channel** object:

1. Create a YAML file and copy the following sample code into it:

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

2. Apply the YAML file:

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

12.2.4. Creating a Kafka channel by using YAML

You can create a Kafka channel by using YAML to create the **KafkaChannel** object.

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.

Procedure

1. Create a YAML file and copy the following sample code into it:

```
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 YAML file:

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

12.2.5. Deleting a channel using the Knative CLI

You can delete a channel with the cluster default configuration by using the **kn** CLI.

Procedure

- Delete a channel:

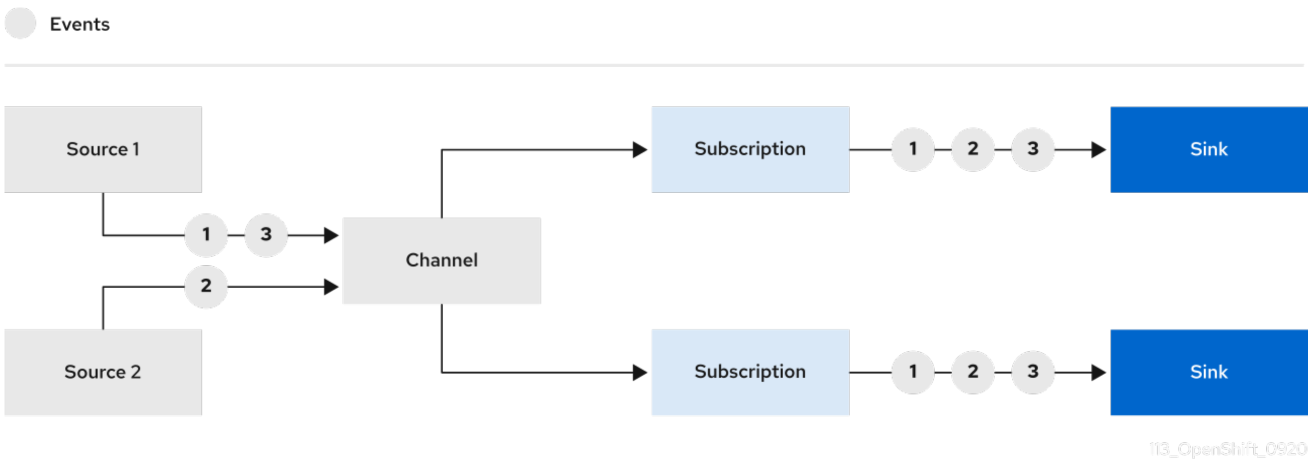
```
$ kn channel delete <channel_name>
```

12.2.6. Next steps

- See Connecting a channel to an event source .
- After you have created a channel, see Using subscriptions for information about creating and using subscriptions for event delivery.

12.3. SUBSCRIPTIONS

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.



If a subscriber rejects an event, there are no re-delivery attempts by default. Developers can configure re-delivery attempts by modifying the **delivery** spec in a **Subscription** object.

12.3.1. Creating subscriptions

Developers can create subscriptions that allow event sinks to subscribe to channels and receive events.

12.3.1.1. Creating subscriptions in the Developer perspective

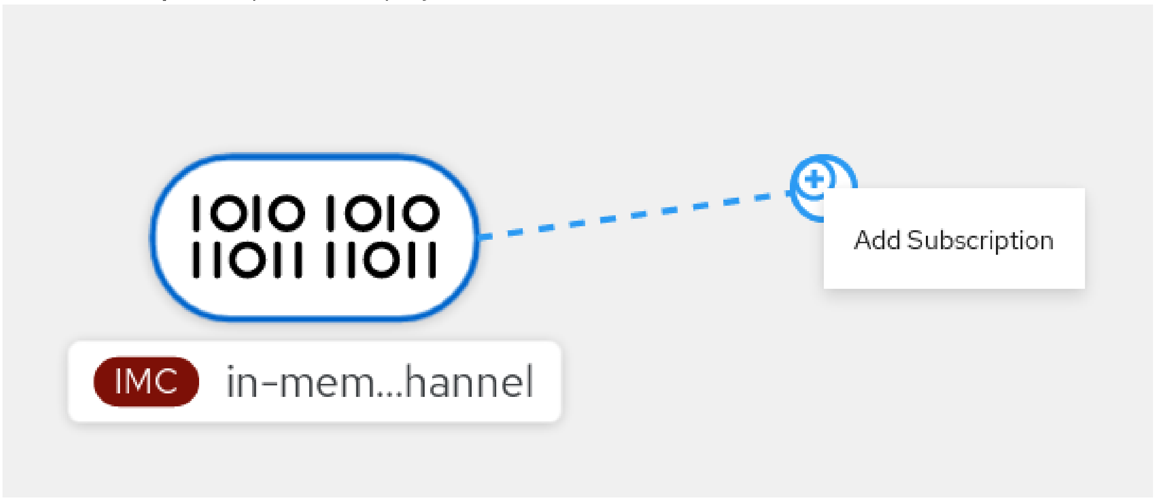
Prerequisites

To create subscriptions using the **Developer** perspective, ensure that:

- 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 an event sink, such as a Knative service, and a channel.

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.

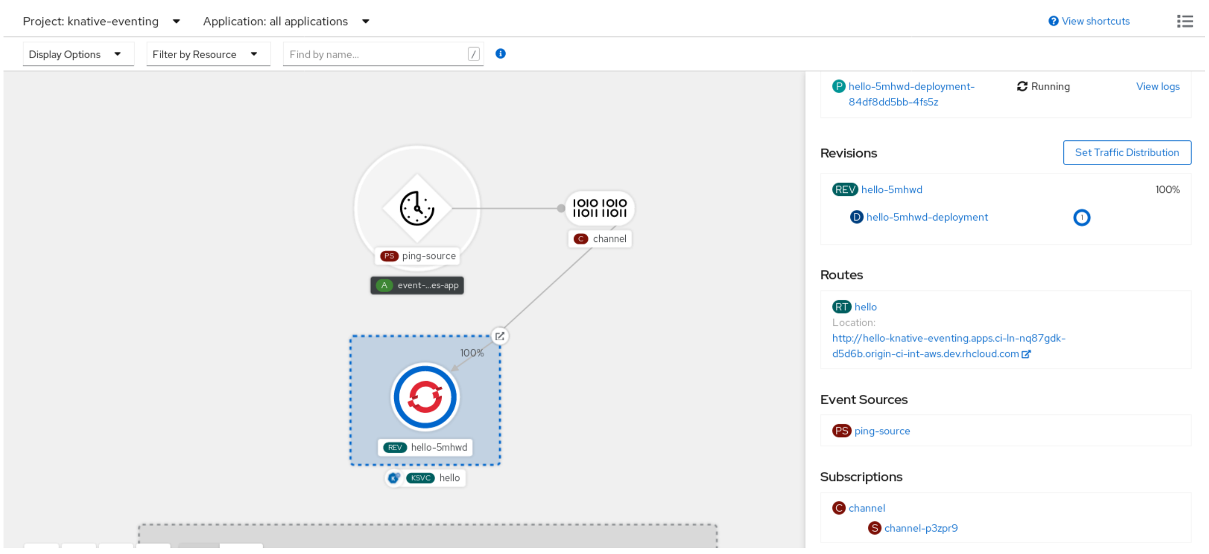


      i. Select your sink as a subscriber from the drop-down list.
      ii. Click **Add**.

   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:



You can view the event source, channel, and subscriptions for the sink by clicking on the service.

12.3.1.2. Creating subscriptions using the Knative CLI

You can create a subscription to connect a channel to a sink by using the **kn** CLI.

Prerequisites

To create subscriptions using the **kn** CLI, ensure that:

- The OpenShift Serverless Operator and Knative Eventing are installed on your OpenShift Container Platform cluster.
- You have installed the **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** 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 a 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
  ```

12.3.1.3. Creating subscriptions by using YAML

You can create a subscription to connect a channel to a sink by using YAML.

Procedure

- Create a **Subscription** object.
  - Create a YAML file and copy the following sample code into it:

    ```
    apiVersion: messaging.knative.dev/v1beta1
    kind: Subscription
    metadata:
      name: my-subscription (1)
      namespace: default
    spec:
      channel: (2)
        apiVersion: messaging.knative.dev/v1beta1
        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>
```

12.3.2. Configuring event delivery failure parameters using subscriptions

Developers can configure event delivery parameters for individual subscriptions by modifying the **delivery** settings for a **Subscription** object.

Example subscription YAML

```
apiVersion: messaging.knative.dev/v1
kind: Subscription
metadata:
  name: <subscription_name>
  namespace: <subscription_namespace>
spec:
  delivery:
    deadLetterSink: 1
      ref:
        apiVersion: serving.knative.dev/v1
        kind: Service
        name: <sink_name>
    backoffDelay: <duration> 2
    backoffPolicy: <policy_type> 3
    retry: <integer> 4
```

1. Configuration settings to enable using a dead letter sink. This tells the subscription what happens to events that cannot be delivered to the subscriber.

When this is configured, events that fail to be delivered are sent to the dead letter sink destination. The destination can be a Knative service or a URI.

2 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.

3 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 the time interval specified between retries. When using the **exponential** back off policy, the back off delay is equal to **backoffDelay*2^<numberOfRetries>**.

4 The number of times that event delivery is retried before the event is sent to the dead letter sink.

12.3.3. Describing subscriptions using the Knative CLI

You can print information about a subscription in the terminal by using the **kn** CLI.

Prerequisites

To describe subscriptions using the **kn** CLI, ensure that:

- You have installed the **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
```

12.3.4. Listing subscriptions using the Knative CLI

You can list existing subscriptions on your cluster by using the **kn** CLI.

Prerequisites

- You have installed the **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
```

12.3.5. Updating subscriptions

You can update a subscription by using the **kn** CLI.

Prerequisites

To update subscriptions using the **kn** CLI, ensure that:

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

12.3.6. Deleting subscriptions using the Knative CLI

You can delete a subscription by using the **kn** CLI.

Procedure

- Delete a subscription:

```
$ kn subscription delete <subscription_name>
```

12.4. CONNECTING A CHANNEL TO AN EVENT SOURCE

Connecting a channel to an event source allows the channel to receive events from that source. These events can then be forwarded to an event sink by using subscriptions.

12.4.1. Connect an event source to a channel using the Developer perspective

You can create multiple event source types in OpenShift Container Platform that can be connected to channels.

Prerequisites

To connect an event source to a channel using the **Developer** perspective, ensure that:

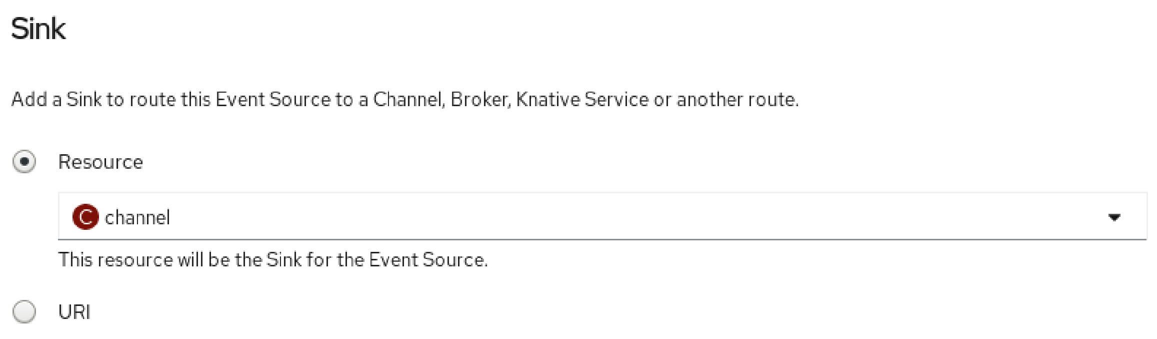
- The OpenShift Serverless Operator, Knative Serving, and Knative Eventing are installed on your OpenShift Container Platform cluster.
- You have logged in to the web console.
- You have created a project or have access to a project with the appropriate roles and permissions to create applications and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

Procedure

1. Create an event source of any type, by following the documentation on *Getting started with event sources*.

2. In the **Developer** perspective, navigate to **Event Sources**.

3. In the **Sink** section of the **Event Sources** form view, select **Resource**. Then use the drop-down to select your channel.



Sink

Add a Sink to route this Event Source to a Channel, Broker, Knative Service or another route.

(●) Resource

C channel

This resource will be the Sink for the Event Source.

( ) URI

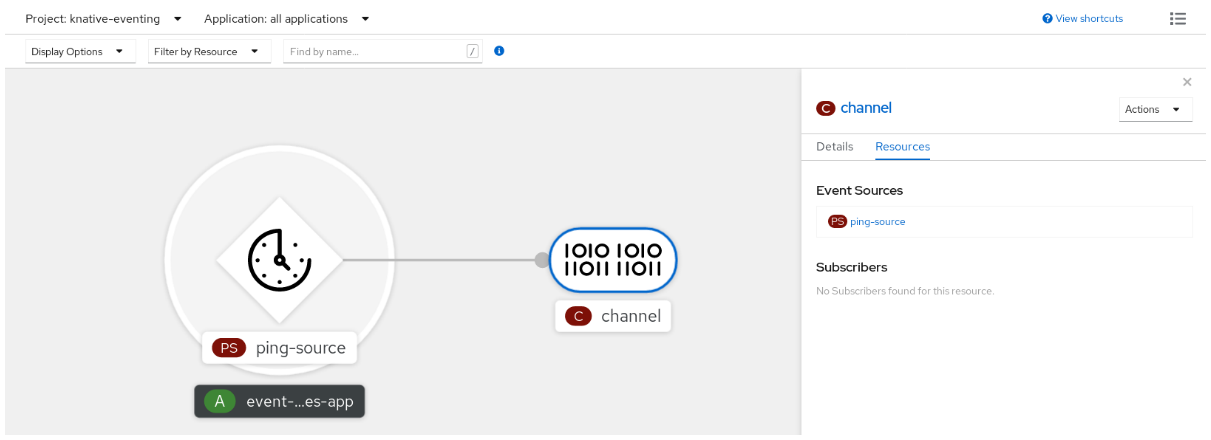
4. Click **Create**.

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 on the connected channel to see the channel details in the side panel.



12.5. CONFIGURING CHANNEL DEFAULTS

If you have cluster administrator permissions, you can set default options for channels, either for the whole cluster or for a specific namespace. These options are modified using config maps.

12.5.1. Configuring the default channel implementation

The **default-ch-webhook** config map can be used to specify the default channel implementation for the cluster or for one or more namespaces.

You can make changes to the **knative-eventing** namespace config maps, including the **default-ch-webhook** config map, by using the OpenShift Serverless Operator to propagate changes. To do this, you must modify the **KnativeEventing** custom resource.

Prerequisites

- You have cluster administrator permissions on OpenShift Container Platform.
- You have installed the OpenShift Serverless Operator and Knative Eventing on your cluster.

Procedure

- Modify the **KnativeEventing** custom resource to add configuration details for the **default-ch-webhook** config map:

```
apiVersion: operator.knative.dev/v1alpha1
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.

CHAPTER 13. NETWORKING

13.1. MAPPING A CUSTOM DOMAIN NAME TO A KNATIVE SERVICE

Knative services are automatically assigned a default domain name based on your cluster configuration. For example, **<service_name>.<namespace>.example.com**. You can map a custom domain name that you own to a Knative service by creating a **DomainMapping** custom resource (CR) for the service. You can also create multiple CRs to map multiple domains and subdomains to a single service.

13.1.1. Creating custom domain mapping for a service

To map a custom domain name to a service you must create a **DomainMapping** custom resource (CR).

Prerequisites

- The OpenShift Serverless Operator and Knative Serving are installed on your cluster.
- You have created a Knative service and control a custom domain that you want to map to that service.

NOTE

Your custom domain must point to the DNS of the OpenShift Container Platform cluster.

Procedure

1. Create a YAML file containing the **DomainMapping** CR in the same namespace as the Knative service you want to map to:

```
apiVersion: serving.knative.dev/v1alpha1
kind: DomainMapping
metadata:
 name: <domain_name> 1
 namespace: <namespace> 2
spec:
 ref:
   name: <service_name> 3
   kind: Service
   apiVersion: serving.knative.dev/v1
```

1 The custom domain name that you want to map to the service.

2 The namespace of both the **DomainMapping** CR and the Knative service.

3 The name of the service to map to the custom domain.

2. Apply the **DomainMapping** CR as a YAML file:

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

13.2. CONFIGURING ROUTES FOR KNATIVE SERVICES

Knative leverages OpenShift Container Platform TLS termination to provide routing for Knative services. When a Knative service is created, a OpenShift Container Platform route is automatically created for the service. This route is managed by the OpenShift Serverless Operator. The OpenShift Container Platform route exposes the Knative service through the same domain as the OpenShift Container Platform cluster.

You can disable Operator control of OpenShift Container Platform routing so that you can configure a Knative route to directly use your TLS certificates instead.

Knative routes can also be used alongside the OpenShift Container Platform route to provide additional fine-grained routing capabilities, such as traffic splitting.

13.2.1. Configuring OpenShift Container Platform routes for Knative services

If you want to configure a Knative service to use your TLS certificate on OpenShift Container Platform, you must disable the automatic creation of a route for the service by the OpenShift Serverless Operator, and instead manually create a **Route** resource for the service.

Prerequisites

- The OpenShift Serverless Operator and Knative Serving component must be installed on your OpenShift Container Platform cluster.

Procedure

1. Create a Knative service that includes the **serving.knative.openshift.io/disableRoute=true** annotation:

   Example YAML

   ```
   apiVersion: serving.knative.dev/v1
   kind: Service
   metadata:
     name: <service_name>
     annotations:
       serving.knative.openshift.io/disableRoute: true
   spec:
     template:
       spec:
         containers:
         - image: <image>
   ```

   Example kn command

   ```
   $ kn service create hello-example \
     --image=gcr.io/knative-samples/helloworld-go \
     --annotation serving.knative.openshift.io/disableRoute=true
   ```

2. Verify that no OpenShift Container Platform route has been created for the service:

   Example command

```
$ oc get routes.route.openshift.io -l
serving.knative.openshift.io/ingressName=$KSERVICE_NAME -l
serving.knative.openshift.io/ingressNamespace=$KSERVICE_NAMESPACE -n knative-
serving-ingress
```

You should see the following output:

```
No resources found in knative-serving-ingress namespace.
```

3. Create a **Route** object in the **knative-serving-ingress** namespace by copying the following sample YAML and modifying the replaceable values:

```
apiVersion: route.openshift.io/v1
kind: Route
metadata:
  annotations:
    haproxy.router.openshift.io/timeout: 600s (1)
  name: <route_name> (2)
  namespace: knative-serving-ingress (3)
spec:
  host: <service_host> (4)
  port:
    targetPort: http2
  to:
    kind: Service
    name: kourier
    weight: 100
  tls:
    insecureEdgeTerminationPolicy: Allow
    termination: edge (5)
    key: |-
      -----BEGIN PRIVATE KEY-----
      [...]
      -----END PRIVATE KEY-----
    certificate: |-
      -----BEGIN CERTIFICATE-----
      [...]
      -----END CERTIFICATE-----
    caCertificate: |-
      -----BEGIN CERTIFICATE-----
      [...]
      -----END CERTIFICATE----
  wildcardPolicy: None
```

(1) The timeout value for the OpenShift Container Platform route. You must set the same value as the **max-revision-timeout-seconds** setting (**600s** by default).

(2) The name of the OpenShift Container Platform route.

(3) The namespace for the OpenShift Container Platform route. This must be **knative-serving-ingress**.

(4) The hostname for external access. You can set this to **<service_name>-<service_namespace>.<domain>**.

5 The certificates you want to use. Currently, only **edge** termination is supported.

13.3. USING SERVICE MESH WITH OPENSHIFT SERVERLESS

Using Service Mesh with OpenShift Serverless enables developers to configure additional networking and routing options that are not supported when using OpenShift Serverless with the default Kourier implementation. These options include setting custom domains, using TLS certificates, and using JSON Web Token authentication.

13.3.1. Prerequisites

- Install the OpenShift Serverless Operator and Knative Serving.
- Install Red Hat OpenShift Service Mesh. OpenShift Serverless is supported for use with both Red Hat OpenShift Service Mesh version 1.x and version 2.x.

13.3.2. Using Service Mesh with OpenShift Serverless

Procedure

1. Add the namespaces that you would like to integrate with Service Mesh to the **ServiceMeshMemberRoll** object as members:

```
apiVersion: maistra.io/v1
kind: ServiceMeshMemberRoll
metadata:
  name: default
  namespace: istio-system
spec:
  members:
    - <namespace> 1
```

1 A list of namespaces to be integrated with Service Mesh.

IMPORTANT

Adding sidecar injection to Pods in system namespaces such as **knative-serving** and **knative-serving-ingress** is not supported.

2. Create a network policy that permits traffic flow from Knative system Pods to Knative services:

   a. Add the **serving.knative.openshift.io/system-namespace=true** label to the **knative-serving** namespace:

```
$ oc label namespace knative-serving serving.knative.openshift.io/system-
namespace=true
```

   b. Add the **serving.knative.openshift.io/system-namespace=true** label to the **knative-serving-ingress** namespace:

```
$ oc label namespace knative-serving-ingress serving.knative.openshift.io/system-
namespace=true
```

c. For each namespace that you would like to integrate with Service Mesh, copy the following **NetworkPolicy** resource into a YAML file:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-from-serving-system-namespace
  namespace: <namespace>
spec:
  ingress:
  - from:
    - namespaceSelector:
        matchLabels:
          serving.knative.openshift.io/system-namespace: "true"
  podSelector: {}
  policyTypes:
  - Ingress
```

d. Apply the **NetworkPolicy** resource:

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

13.3.3. Enabling sidecar injection for a Knative service

You can add an annotation to the Service resource YAML file to enable sidecar injection for a Knative service.

Procedure

1. Add the **sidecar.istio.io/inject="true"** annotation to the Service resource:

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: hello-example-1
spec:
  template:
    metadata:
      annotations:
        sidecar.istio.io/inject: "true" 1
        sidecar.istio.io/rewriteAppHTTPProbers: "true" 2
    spec:
      containers:
      - image: docker.io/openshift/hello-openshift
        name: container
```

1 Add the **sidecar.istio.io/inject="true"** annotation.

2 Optional: Add the **sidecar.istio.io/rewriteAppHTTPProbers="true"** annotation if you have enabled JSON Web Token (JWT) authentication.

2. Apply the Service resource YAML file:

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

13.4. USING JSON WEB TOKEN AUTHENTICATION WITH SERVICE MESH AND OPENSHIFT SERVERLESS

You can enable JSON Web Token (JWT) authentication for Knative services.

13.4.1. Prerequisites

- Install OpenShift Serverless.
- Install Red Hat OpenShift Service Mesh. OpenShift Serverless is supported for use with both Red Hat OpenShift Service Mesh version 1.x and version 2.x.
- Configure Service Mesh with OpenShift Serverless, including enabling sidecar injection for your Knative services.

IMPORTANT

Adding sidecar injection to pods in system namespaces such as **knative-serving** and **knative-serving-ingress** is not supported.

IMPORTANT

You must set the annotation **sidecar.istio.io/rewriteAppHTTPProbers: "true"** in your Knative service as OpenShift Serverless versions 1.14.0 and later use a HTTP probe as the readiness probe for Knative services by default.

13.4.2. Using JSON Web Token authentication with Service Mesh 2.x and OpenShift Serverless

Procedure

1. Create the **RequestAuthentication** resource in each serverless application namespace that is a member in the **ServiceMeshMemberRoll** object, by copying the following code into a YAML file:

```
apiVersion: security.istio.io/v1beta1
kind: RequestAuthentication
metadata:
  name: jwt-example
  namespace: <namespace>
spec:
  jwtRules:
  - issuer: testing@secure.istio.io
    jwksUri: https://raw.githubusercontent.com/istio/istio/release-
1.8/security/tools/jwt/samples/jwks.json
```

2. Apply the **RequestAuthentication** resource:

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

3. Allow access to the **RequestAuthenticaton** resource from system pods for each serverless application namespace that is a member in the **ServiceMeshMemberRoll** object, by copying the following **AuthorizationPolicy** resources into a YAML file:

```
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: allowlist-by-paths
  namespace: <namespace>
spec:
  action: ALLOW
  rules:
  - to:
    - operation:
        paths:
        - /metrics (1)
        - /healthz (2)
```

(1) The path on your application to collect metrics by system pod.

(2) The path on your application to probe by system pod.

4. Apply the **AuthorizationPolicy** resource YAML file:

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

5. For each serverless application namespace that is a member in the **ServiceMeshMemberRoll** object, copy the following **AuthorizationPolicy** resource, into a YAML file:

```
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: require-jwt
  namespace: <namespace>
spec:
  action: ALLOW
  rules:
  - from:
    - source:
       requestPrincipals: ["testing@secure.istio.io/testing@secure.istio.io"]
```

6. Apply the **AuthorizationPolicy** resource YAML file:

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

Verification

1. If you try to use a **curl** request to get the Knative service URL, it is denied.

```
$ curl http://hello-example-1-default.apps.mycluster.example.com/
```

Example output

```
RBAC: access denied
```

2. Verify the request with a valid JWT.

   a. Get the valid JWT token by entering the following command:

   ```
   $ TOKEN=$(curl https://raw.githubusercontent.com/istio/istio/release-1.8/security/tools/jwt/samples/demo.jwt -s) && echo "$TOKEN" | cut -d '.' -f2 - | base64 --decode -
   ```

   b. Access the service by using the valid token in the **curl** request header:

   ```
   $ curl -H "Authorization: Bearer $TOKEN"  http://hello-example-1-default.apps.example.com
   ```

   The request is now allowed.

   ### Example output

   ```
   Hello OpenShift!
   ```

13.4.3. Using JSON Web Token authentication with Service Mesh 1.x and OpenShift Serverless

Procedure

1. Create a policy in a serverless application namespace which is a member in the **ServiceMeshMemberRoll** object, that only allows requests with valid JSON Web Tokens (JWT):

   a. Copy the following **Policy** resource into a YAML file:

   IMPORTANT

   The paths /**metrics** and /**healthz** must be included in **excludedPaths** because they are accessed from system Pods in the **knative-serving** namespace.

   ```
   apiVersion: authentication.istio.io/v1alpha1
   kind: Policy
   metadata:
     name: default
     namespace: <namespace>
   spec:
     origins:
     - jwt:
         issuer: testing@secure.istio.io
         jwksUri: "https://raw.githubusercontent.com/istio/istio/release-1.6/security/tools/jwt/samples/jwks.json"
         triggerRules:
   ```

```
      - excludedPaths:
        - prefix: /metrics (1)
        - prefix: /healthz (2)
  principalBinding: USE_ORIGIN
```

(1) The path on your application to collect metrics by system pod.

(2) The path on your application to probe by system pod.

b. Apply the **Policy** resource:

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

Verification

1. If you try to use a **curl** request to get the Knative service URL, it is denied.

   ```
   $ curl http://hello-example-default.apps.mycluster.example.com/
   ```

   Example output

   ```
   Origin authentication failed.
   ```

2. Verify the request with a valid JWT.

   a. Get the valid JWT token by entering the following command:

   ```
   $ TOKEN=$(curl https://raw.githubusercontent.com/istio/istio/release-1.6/security/tools/jwt/samples/demo.jwt -s) && echo "$TOKEN" | cut -d '.' -f2 - | base64 --decode -
   ```

   b. Access the service by using the valid token in the **curl** request header:

   ```
   $ curl http://hello-example-default.apps.mycluster.example.com/ -H "Authorization: Bearer $TOKEN"
   ```

   The request is now allowed.

   Example output

   ```
   Hello OpenShift!
   ```

13.5. USING CUSTOM DOMAINS FOR KNATIVE SERVICES WITH SERVICE MESH

By default, Knative services have a fixed domain format:

```
<application_name>-<namespace>.<openshift_cluster_domain>
```

You can customize the domain for your Knative service by configuring the service as a private service and creating the required Service Mesh resources.

Prerequisites

- Install the OpenShift Serverless Operator and Knative Serving.
- Install Red Hat OpenShift Service Mesh .
- Complete the configuration steps in Using Service Mesh with OpenShift Serverless .
- You can configure a custom domain for an existing Knative service, or create a new sample service. To create a new service, see Creating and managing serverless applications .

13.5.1. Setting cluster availability to cluster-local

By default, Knative services are published to a public IP address. Being published to a public IP address means that Knative services are public applications, and have a publicly accessible URL.

Publicly accessible URLs are accessible from outside of the cluster. However, developers may need to build back-end services that are only be accessible from inside the cluster, known as *private services*. Developers can label individual services in the cluster with the **serving.knative.dev/visibility=cluster-local** label to make them private.

Procedure

- Set the visibility for your service by adding the **serving.knative.dev/visibility=cluster-local** label:

```
$ oc label ksvc <service_name> serving.knative.dev/visibility=cluster-local
```

Verification

- Check that the URL for your service is now in the format **http://<service_name>.<namespace>.svc.cluster.local**, by entering the following command and reviewing the output:

```
$ oc get ksvc
```

Example output

```
NAME          URL                                                    LATESTCREATED
LATESTREADY      READY   REASON
hello         http://hello.default.svc.cluster.local                      hello-tx2g7      hello-
tx2g7       True
```

13.5.2. Creating necessary Service Mesh resources

Procedure

1. Create an Istio gateway to accept traffic.
   a. Create a YAML file, and copy the following YAML into it:

```
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: default-gateway
spec:
  selector:
    istio: ingressgateway
  servers:
  - port:
      number: 80
      name: http
      protocol: HTTP
    hosts:
    - "*"
```

b. Apply the YAML file:

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

2. Create an Istio **VirtualService** object to rewrite the host header.

a. Create a YAML file, and copy the following YAML into it:

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: hello
spec:
  hosts:
  - custom-ksvc-domain.example.com
  gateways:
  - default-gateway
  http:
  - rewrite:
      authority: hello.default.svc 1
    route:
    - destination:
        host: hello.default.svc 2
        port:
          number: 80
```

1 2 Your Knative service in the format **<service_name>.<namespace>.svc**.

b. Apply the YAML file:

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

3. Create an Istio **ServiceEntry** object. This is required for OpenShift Serverless because Kourier is outside of the service mesh.

a. Create a YAML file, and copy the following YAML into it:

```
apiVersion: networking.istio.io/v1alpha3
kind: ServiceEntry
metadata:
  name: hello.default.svc
spec:
```

```
hosts:
- hello.default.svc (1)
location: MESH_EXTERNAL
endpoints:
- address: kourier-internal.knative-serving-ingress.svc
ports:
- number: 80
  name: http
  protocol: HTTP
resolution: DNS
```

(1) Your Knative service in the format **<service_name>.<namespace>.svc**.

b. Apply the YAML file:

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

4. Create an OpenShift Container Platform route that points to the **VirtualService** object.

   a. Create a YAML file, and copy the following YAML into it:

```
apiVersion: route.openshift.io/v1
kind: Route
metadata:
  name: hello
  namespace: istio-system (1)
spec:
  host: custom-ksvc-domain.example.com
  port:
    targetPort: 8080
  to:
    kind: Service
    name: istio-ingressgateway
```

(1) The OpenShift Container Platform route must be created in the same namespace as the ServiceMeshControlPlane. In this example, the ServiceMeshControlPlane is deployed in the **istio-system** namespace.

a. Apply the YAML file:

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

13.5.3. Accessing a service using your custom domain

Procedure

1. Access the custom domain by using the **Host** header in a **curl** request. For example:

```
$ curl -H "Host: custom-ksvc-domain.example.com" http://<ip_address>
```

where **<ip_address>** is the IP address that the OpenShift Container Platform ingress router is exposed to.

Example output

```
Hello OpenShift!
```

13.5.4. Additional resources

- For more information about Red Hat OpenShift Service Mesh, see Understanding Red Hat OpenShift Service Mesh.

13.6. CONFIGURING TRANSPORT LAYER SECURITY FOR A CUSTOM DOMAIN USING RED HAT OPENSHIFT SERVICE MESH

You can create a Transport Layer Security (TLS) key and certificates for a custom domain and subdomain using Red Hat OpenShift Service Mesh.

NOTE

OpenShift Serverless is compatible only with full implementations of either Red Hat OpenShift Service Mesh 1.x or 2.x. OpenShift Serverless does not support custom usage of some 1.x resources and some 2.x resources in the same deployment. For example, upgrading to 2.x while still using the control plane **maistra.io/v1** spec is not supported.

13.6.1. Prerequisites

- Install OpenShift Serverless.
- Install Red Hat OpenShift Service Mesh 1.x or 2.x.
- Complete the configuration steps in Using Service Mesh with OpenShift Serverless .
- Configure a custom domain. See Using custom domains for Knative services with Service Mesh .
- In this example, **openssl** is used to generate certificates, but you can use any certificate generation tool to create these.

IMPORTANT

This example uses the **example.com** domain. The example certificate for this domain is used as a certificate authority (CA) that signs the subdomain certificate.

To complete and verify this procedure in your deployment, you need either a certificate signed by a widely trusted public CA, or a CA provided by your organization.

Example commands must be adjusted according to your domain, subdomain and CA.

13.6.2. Configuring Transport Layer Security for a custom domain using Red Hat OpenShift Service Mesh 2.x

You can create a Transport Layer Security (TLS) key and certificates for a custom domain and subdomain using Red Hat OpenShift Service Mesh.

Procedure

1. Create a root certificate and private key to sign the certificates for your services:

```
$ openssl req -x509 -sha256 -nodes -days 365 -newkey rsa:2048 \
    -subj '/O=Example Inc./CN=example.com' \
    -keyout example.com.key \
    -out example.com.crt
```

2. Create a certificate signing request for your domain:

```
$ openssl req -out custom.example.com.csr -newkey rsa:2048 -nodes \
    -keyout custom.example.com.key \
    -subj "/CN=custom-ksvc-domain.example.com/O=Example Inc."
```

3. Sign the request with your CA:

```
$ openssl x509 -req -days 365 -set_serial 0 \
    -CA example.com.crt \
    -CAkey example.com.key \
    -in custom.example.com.csr \
    -out custom.example.com.crt
```

4. Check that the certificates appear in your directory:

```
$ ls -1
```

Example output

```
custom.example.com.crt
custom.example.com.csr
custom.example.com.key
example.com.crt
example.com.key
```

5. Create a secret:

```
$ oc create -n istio-system secret tls custom.example.com \
    --key=custom.example.com.key \
    --cert=custom.example.com.crt
```

6. Attach the secret to the Istio ingress gateway by editing the **ServiceMeshControlPlane** resource.

   a. Edit the **ServiceMeshControlPlane** resource:

   ```
   $ oc edit -n istio-system ServiceMeshControlPlane <control-plane-name>
   ```

   b. Check that the following lines exist in the resource, and if they do not, add them:

   ```
   spec:
     gateways:
       ingress:
         volumes:
         - volume:
   ```

```
        secret:
          secretName: custom.example.com
      volumeMount:
        name: custom-example-com
        mountPath: /custom.example.com
```

7. Update the Istio ingress gateway to use your secret.

    a. Edit the **default-gateway** resource:

    ```
    $ oc edit gateway default-gateway
    ```

    b. Check that the following lines exist in the resource, and if they do not, add them:

    ```
    - hosts:
      - custom-ksvc-domain.example.com
      port:
        name: https
        number: 443
        protocol: HTTPS
      tls:
        mode: SIMPLE
        privateKey: /custom.example.com/tls.key
        serverCertificate: /custom.example.com/tls.crt
    ```

8. Update the route to use pass-through TLS and **8443** as the **spec.port.targetPort**.

    a. Edit the route:

    ```
    $ oc edit route -n istio-system hello
    ```

    b. Add the following configuration to the route:

    ```
    spec:
      host: custom-ksvc-domain.example.com
      port:
        targetPort: 8443
      tls:
        insecureEdgeTerminationPolicy: None
        termination: passthrough
      to:
        kind: Service
        name: istio-ingressgateway
        weight: 100
      wildcardPolicy: None
    ```

Verification

- Access your serverless application by a secure connection that is now trusted by the CA:

```
$ curl --cacert example.com.crt \
    --header "Host: custom-ksvc-domain.example.com" \
    --resolve "custom-ksvc-domain.example.com:443:<ingress_router_IP>" \
     https://custom-ksvc-domain.example.com:443
```

NOTE

You must substitute your own value for **<ingress_router_IP>**. Steps for finding this IP or host name value vary depending on your OpenShift Container Platform provider platform.

Example command to find the ingress IP

This command is valid for GCP and Azure provider platforms:

```
$ oc get svc -n openshift-ingress router-default \
    -o jsonpath='{.status.loadBalancer.ingress[0].ip}'
```

Example output

```
Hello OpenShift!
```

13.6.3. Configuring Transport Layer Security for a custom domain using Red Hat OpenShift Service Mesh 1.x

You can create a Transport Layer Security (TLS) key and certificates for a custom domain and subdomain using Red Hat OpenShift Service Mesh.

Procedure

1. Create a root certificate and private key to sign the certificates for your services:

```
$ openssl req -x509 -sha256 -nodes -days 365 -newkey rsa:2048 \
    -subj '/O=Example Inc./CN=example.com' \
    -keyout example.com.key \
    -out example.com.crt
```

2. Create a certificate signing request for your domain:

```
$ openssl req -out custom.example.com.csr -newkey rsa:2048 -nodes \
    -keyout custom.example.com.key \
    -subj "/CN=custom-ksvc-domain.example.com/O=Example Inc."
```

3. Sign the request with your CA:

```
$ openssl x509 -req -days 365 -set_serial 0 \
    -CA example.com.crt \
    -CAkey example.com.key \
    -in custom.example.com.csr \
    -out custom.example.com.crt
```

4. Check that the certificates appear in your directory:

```
$ ls -1
```

Example output

```
custom.example.com.crt
custom.example.com.csr
custom.example.com.key
example.com.crt
example.com.key
```

5. Create a secret:

   ```
   $ oc create -n istio-system secret tls custom.example.com \
       --key=custom.example.com.key \
       --cert=custom.example.com.crt
   ```

6. Attach the secret to the Istio ingress gateway by editing the **ServiceMeshControlPlane** resource.

   a. Edit the **ServiceMeshControlPlane** resource:

   ```
   $ oc edit -n istio-system ServiceMeshControlPlane <control_plane_name>
   ```

   b. Check that the following lines exist in the resource, and if they do not, add them:

   ```
   spec:
     istio:
       gateways:
         istio-ingressgateway:
           secretVolumes:
           - mountPath: /custom.example.com
             name: custom-example-com
             secretName: custom.example.com
   ```

7. Update the Istio ingress gateway to use your secret.

   a. Edit the **default-gateway** resource:

   ```
   $ oc edit gateway default-gateway
   ```

   b. Check that the following lines exist in the resource, and if they do not, add them:

   ```
   - hosts:
     - custom-ksvc-domain.example.com
     port:
       name: https
       number: 443
       protocol: HTTPS
     tls:
       mode: SIMPLE
       privateKey: /custom.example.com/tls.key
       serverCertificate: /custom.example.com/tls.crt
   ```

8. Update the route to use pass-through TLS and **8443** as the **spec.port.targetPort**.

   a. Edit the route:

   ```
   $ oc edit route -n istio-system hello
   ```

b. Add the following configuration to the route:

```
spec:
  host: custom-ksvc-domain.example.com
  port:
    targetPort: 8443
  tls:
    insecureEdgeTerminationPolicy: None
    termination: passthrough
  to:
    kind: Service
    name: istio-ingressgateway
    weight: 100
  wildcardPolicy: None
```

Verification

- Access your serverless application by a secure connection that is now trusted by the CA:

```
$ curl --cacert example.com.crt \
    --header "Host: custom-ksvc-domain.example.com" \
    --resolve "custom-ksvc-domain.example.com:443:<ingress_router_IP>" \
     https://custom-ksvc-domain.example.com:443
```

NOTE

You must substitute your own value for **<ingress_router_IP>**. Steps for finding this IP or host name value vary depending on your OpenShift Container Platform provider platform.

Example command to find the ingress IP

This command is valid for GCP and Azure provider platforms:

```
$ oc get svc -n openshift-ingress router-default \
    -o jsonpath='{.status.loadBalancer.ingress[0].ip}'
```

Example output

```
Hello OpenShift!
```

CHAPTER 14. MONITORING

14.1. MONITORING SERVERLESS COMPONENTS

You can use OpenShift Container Platform monitoring dashboards to view health checks and metrics for OpenShift Serverless components.

14.1.1. Monitoring serverless components overall health status

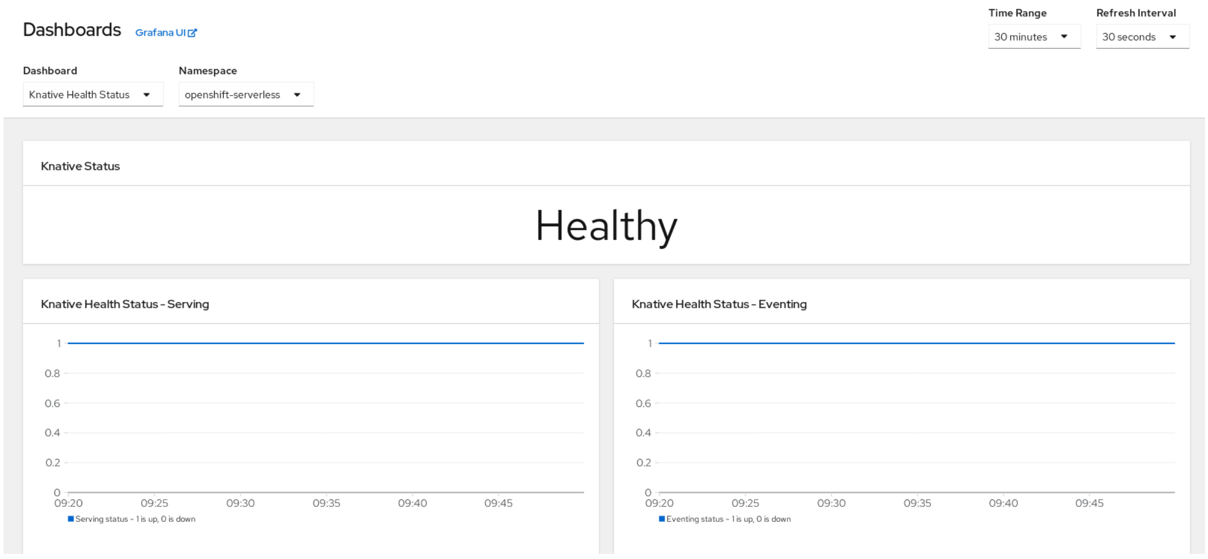
You can use the OpenShift Container Platform monitoring dashboards to view the overall health status of Knative Serving and Eventing.

Prerequisites

- You have installed OpenShift Serverless.
- The OpenShift Container Platform monitoring stack is enabled.
- You have cluster administrator permissions, and access to the **Administrator** perspective in the OpenShift Container Platform web console.

Procedure

1. In the **Administrator** perspective, navigate to **Monitoring** → **Dashboards**.
2. Select the **Knative Health Status** dashboard to view the overall health status of Knative Serving and Eventing.



14.1.2. Monitoring Knative Serving revision CPU and memory usage

You can use the OpenShift Container Platform monitoring dashboards to view revision CPU and memory usage metrics for Knative Serving components.

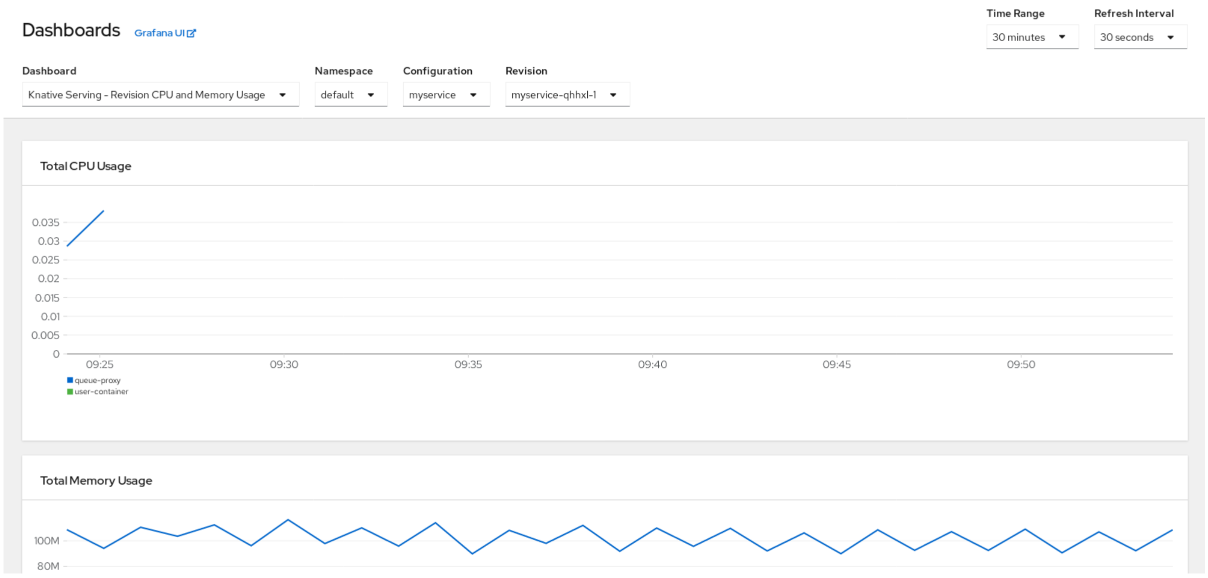
Prerequisites

- You have installed OpenShift Serverless.
- The OpenShift Container Platform monitoring stack is enabled.

- You have cluster administrator permissions, and access to the **Administrator** perspective in the OpenShift Container Platform web console.

Procedure

1. In the **Administrator** perspective, navigate to **Monitoring** → **Dashboards**.
2. Select the **Knative Serving - Revision CPU and Memory Usage** dashboard to view metrics for Knative service revisions.
3. Optional: You can filter this dashboard by **Namespace**, **Configuration**, or **Revision**, by using the drop-down selections.



CHAPTER 15. INTEGRATIONS

15.1. USING NVIDIA GPU RESOURCES WITH SERVERLESS APPLICATIONS

NVIDIA supports experimental use of GPU resources on OpenShift Container Platform. See OpenShift Container Platform on NVIDIA GPU accelerated clusters for more information about setting up GPU resources on OpenShift Container Platform.

After GPU resources are enabled for your OpenShift Container Platform cluster, you can specify GPU requirements for a Knative service using the **kn** CLI.

NOTE

Using NVIDIA GPU resources is not supported for IBM Z and IBM Power Systems.

Procedure

You can specify a GPU resource requirement when you create a Knative service using **kn**.

1. Create a service.
2. Set the GPU resource requirement limit to **1** by using **nvidia.com/gpu=1**:

```
$ kn service create hello --image docker.io/knativesamples/hellocuda-go --limit
nvidia.com/gpu=1
```

   A GPU resource requirement limit of **1** means that the service has 1 GPU resource dedicated. Services do not share GPU resources. Any other services that require GPU resources must wait until the GPU resource is no longer in use.

   A limit of 1 GPU also means that applications exceeding usage of 1 GPU resource are restricted. If a service requests more than 1 GPU resource, it is deployed on a node where the GPU resource requirements can be met.

Updating GPU requirements for a Knative service using kn

- Update the service. Change the GPU resource requirement limit to **3** by using **nvidia.com/gpu=3**:

```
$ kn service update hello --limit nvidia.com/gpu=3
```

15.1.1. Additional resources

- For more information about limits, see Setting resource quotas for extended resources .

CHAPTER 16. CLI REFERENCE

16.1. ADVANCED KN CONFIGURATION

You can customize and extend the **kn** CLI by using advanced features, such as configuring a **config.yaml** file for **kn** or using plug-ins.

16.1.1. Customizing kn

You can customize your **kn** CLI setup by creating a **config.yaml** configuration file. You can provide this configuration by using the **--config** flag, otherwise the configuration is picked up from a default location. The default configuration location conforms to the XDG Base Directory Specification, and is different for Unix systems and Windows systems.

For Unix systems:

- If the **XDG_CONFIG_HOME** environment variable is set, the default configuration location that the **kn** CLI looks for is **$XDG_CONFIG_HOME/kn**.
- If the **XDG_CONFIG_HOME** environment variable is not set, the **kn** CLI looks for the configuration in the home directory of the user at **$HOME/.config/kn/config.yaml**.

For Windows systems, the default **kn** CLI configuration location is **%APPDATA%\kn**.

Example configuration file

```
plugins:
  path-lookup: true 1
  directory: ~/.config/kn/plugins 2
eventing:
  sink-mappings: 3
  - prefix: svc 4
    group: core 5
    version: v1 6
    resource: services 7
```

1 Specifies whether the **kn** CLI should look for plug-ins in the **PATH** environment variable. This is a boolean configuration option. The default value is **false**.

2 Specifies the directory where the **kn** CLI will look for plug-ins. The default path depends on the operating system, as described above. This can be any directory that is visible to the user.

3 The **sink-mappings** spec defines the Kubernetes addressable resource that is used when you use the **--sink** flag with a **kn** CLI command.

4 The prefix you want to use to describe your sink. **svc** for a service, **channel**, and **broker** are predefined prefixes in **kn**.

5 The API group of the Kubernetes resource.

6 The version of the Kubernetes resource.

7 The plural name of the Kubernetes resource type. For example, **services** or **brokers**.

16.1.2. kn plug-ins

The **kn** CLI supports the use of plug-ins, which enable you to extend the functionality of your **kn** installation by adding custom commands and other shared commands that are not part of the core distribution. **kn** CLI plug-ins are used in the same way as the main **kn** functionality.

Currently, Red Hat supports the **kn-source-kafka** plug-in.

16.2. KN FLAGS REFERENCE

16.2.1. Knative CLI (kn) --sink flag

When you create an event-producing custom resource 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 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**.

16.3. KN SERVICE

16.3.1. Creating serverless applications using the kn CLI

The following procedure describes how you can create a basic serverless application using the **kn** CLI.

Prerequisites

- OpenShift Serverless Operator and Knative Serving are installed on your cluster.
- You have installed **kn** CLI.

Procedure

- Create a Knative service:

  ```
  $ kn service create <service-name> --image <image> --env <key=value>
  ```

  #### Example command

  ```
  $ kn service create event-display \
      --image quay.io/openshift-knative/knative-eventing-sources-event-display:latest
  ```

Example output

```
Creating service 'event-display' in namespace 'default':

  0.271s The Route is still working to reflect the latest desired specification.
  0.580s Configuration "event-display" is waiting for a Revision to become ready.
  3.857s ...
  3.861s Ingress has not yet been reconciled.
  4.270s Ready to serve.

Service 'event-display' created with latest revision 'event-display-bxshg-1' and URL:
http://event-display-default.apps-crc.testing
```

16.3.2. Updating serverless applications

You can use the **kn service update** command for interactive sessions on the command line as you build up a service incrementally. In contrast to the **kn service apply** command, when using the **kn service update** command you only have to specify the changes that you want to update, rather than the full configuration for the Knative service.

Example commands

- Update a service by adding a new environment variable:

```
$ kn service update <service_name> --env <key>=<value>
```

- Update a service by adding a new port:

```
$ kn service update <service_name> --port 80
```

- Update a service by adding new request and limit parameters:

```
$ kn service update <service_name> --request cpu=500m --limit memory=1024Mi --limit
cpu=1000m
```

- Assign the **latest** tag to a revision:

```
$ kn service update <service_name> --tag <revision_name>=latest
```

- Update a tag from **testing** to **staging** for the latest **READY** revision of a service:

```
$ kn service update <service_name> --untag testing --tag @latest=staging
```

- Add the **test** tag to a revision that receives 10% of traffic, and send the rest of the traffic to the latest **READY** revision of a service:

```
$ kn service update <service_name> --tag <revision_name>=test --traffic test=10,@latest=90
```

16.3.3. Applying service declarations

You can declaratively configure a Knative service by using the **kn service apply** command. If the service does not exist it is created, otherwise the existing service is updated with the options that have been changed.

The **kn service apply** command is especially useful for shell scripts or in a continuous integration pipeline, where users typically want to fully specify the state of the service in a single command to declare the target state.

When using **kn service apply** you must provide the full configuration for the Knative service. This is different from the **kn service update** command, which only requires you to specify in the command the options that you want to update.

Example commands

- Create a service:

  ```
  $ kn service apply <service_name> --image <image>
  ```

- Add an environment variable to a service:

  ```
  $ kn service apply <service_name> --image <image> --env <key>=<value>
  ```

- Read the service declaration from a JSON or YAML file:

  ```
  $ kn service apply <service_name> -f <filename>
  ```

16.3.4. Describing serverless applications

You can describe a Knative service by using the **kn service describe** command.

Example commands

- Describe a service:

  ```
  $ kn service describe --verbose <service_name>
  ```

  The **--verbose** flag is optional but can be included to provide a more detailed description. The difference between a regular and verbose output is shown in the following examples:

  Example output without --verbose flag

  ```
  Name:       hello
  Namespace:  default
  Age:        2m
  URL:        http://hello-default.apps.ocp.example.com

  Revisions:
    100%  @latest (hello-00001) [1] (2m)
          Image:  docker.io/openshift/hello-openshift (pinned to aaea76)

  Conditions:
    OK TYPE                   AGE REASON
  ```

```
  ++ Ready                  1m
  ++ ConfigurationsReady     1m
  ++ RoutesReady            1m
```

Example output with --verbose flag

```
Name:        hello
Namespace:    default
Annotations:  serving.knative.dev/creator=system:admin
              serving.knative.dev/lastModifier=system:admin
Age:         3m
URL:         http://hello-default.apps.ocp.example.com
Cluster:      http://hello.default.svc.cluster.local

Revisions:
  100%  @latest (hello-00001) [1] (3m)
        Image:  docker.io/openshift/hello-openshift (pinned to aaea76)
        Env:    RESPONSE=Hello Serverless!

Conditions:
  OK TYPE                  AGE REASON
  ++ Ready                  3m
  ++ ConfigurationsReady     3m
  ++ RoutesReady            3m
```

- Describe a service in YAML format:

```
$ kn service describe <service_name> -o yaml
```

- Describe a service in JSON format:

```
$ kn service describe <service_name> -o json
```

- Print the service URL only:

```
$ kn service describe <service_name> -o url
```