OpenShift Container Platform 4.8

Serverless

OpenShift Serverless installation, usage, and release notes
OpenShift Container Platform 4.8 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 ........................................ 10
   1.1. RELEASE NOTES FOR RED HAT OPENSHIFT SERVERLESS 1.16.0 10
      1.1.1. New features 10
      1.1.2. Known issues 10

CHAPTER 2. OPENSHIFT SERVERLESS SUPPORT ............................................. 13
   2.1. GETTING SUPPORT 13
   2.2. GATHERING DIAGNOSTIC INFORMATION FOR SUPPORT 13
      2.2.1. About the must-gather tool 13
      2.2.2. About collecting OpenShift Serverless data 13

CHAPTER 3. GETTING STARTED WITH OPENSHIFT SERVERLESS ........................ 15
   3.1. HOW OPENSHIFT SERVERLESS WORKS 15
   3.2. SUPPORTED CONFIGURATIONS 15
   3.3. NEXT STEPS 15

CHAPTER 4. ADMINISTRATION GUIDE ............................................................. 16
   4.1. INSTALLING THE OPENSHIFT SERVERLESS OPERATOR 16
      4.1.1. Defining cluster size requirements for an OpenShift Serverless installation 16
      4.1.2. Additional requirements for advanced use cases 16
      4.1.3. Scaling your cluster using machine sets 16
      4.1.4. Installing the OpenShift Serverless Operator 17
      4.1.5. Next steps 18
   4.2. INSTALLING KNATIVE SERVING 18
      4.2.1. Prerequisites 18
      4.2.2. Installing Knative Serving using the web console 18
      4.2.3. Installing Knative Serving using YAML 21
      4.2.4. Knative Serving advanced configuration options 23
         4.2.4.1. Controller Custom Certs 23
         4.2.4.2. High availability 24
      4.2.5. Next steps 24
   4.3. INSTALLING KNATIVE EVENTING 24
      4.3.1. Prerequisites 24
      4.3.2. Installing Knative Eventing using the web console 24
      4.3.3. Installing Knative Eventing using YAML 28
   4.4. UPGRADING OPENSHIFT SERVERLESS ............................................ 29
      4.4.1. Upgrading the Subscription Channel 29
   4.5. REMOVING OPENSHIFT SERVERLESS .................................................. 30
      4.5.1. Uninstalling Knative Serving 30
      4.5.2. Uninstalling Knative Eventing 30
      4.5.3. Removing the OpenShift Serverless Operator 31
      4.5.4. Deleting OpenShift Serverless custom resource definitions 31
   4.6. INTEGRATING SERVICE MESH WITH OPENSHIFT SERVERLESS ............... 31
      4.6.1. Integrating Service Mesh with OpenShift Serverless natively 31
         4.6.1.1. Creating a certificate to encrypt incoming external traffic 32
      4.6.1.2. Integrating Service Mesh with OpenShift Serverless 33
      4.6.2. Integrating Service Mesh with OpenShift Serverless when Kourier is enabled 37
   4.7. CREATING EVENTING COMPONENTS IN THE ADMINISTRATOR PERSPECTIVE 38
      4.7.1. Creating an event source by using the Administrator perspective 38
      4.7.2. Creating a broker by using the Administrator perspective 39
      4.7.3. Creating a trigger by using the Administrator perspective 39
      4.7.4. Creating a channel by using the Administrator perspective 40
5.2.6. Verifying your serverless application deployment
5.2.7. Interacting with a serverless application using HTTP2 and gRPC
5.2.8. Enabling communication with Knative applications on a cluster with restrictive network policies
5.2.9. Using kn CLI in offline mode
  5.2.9.1. About offline mode
  5.2.9.2. Creating a service using offline mode
5.3. CONFIGURING KNATIVE SERVING AUTOSCALING
  5.3.1. Autoscaling workflows by using the Knative CLI
  5.3.2. Configuring concurrent requests for Knative Serving autoscaling
    5.3.2.1. Configuring concurrent requests using the target annotation
    5.3.2.2. Configuring concurrent requests using the containerConcurrency field
  5.3.3. Configuring scale bounds Knative Serving autoscaling
5.4. USING OPENSIFT LOGGING
  5.4.1. About deploying OpenShift Logging
  5.4.2. About deploying and configuring OpenShift Logging
    5.4.2.1. Configuring and Tuning OpenShift Logging
    5.4.2.2. Sample modified ClusterLogging custom resource
  5.4.3. Using OpenShift Logging to find logs for Knative Serving components
  5.4.4. Using OpenShift Logging to find logs for services deployed with Knative Serving
5.5. MAPPING AND SPLITTING TRAFFIC FOR DIFFERENT REVISIONS OF A SERVICE
  5.5.1. Splitting traffic between revisions using the Developer perspective
  5.5.2. Managing and splitting traffic by using the Knative CLI
    5.5.2.1. Assigning tag revisions
    5.5.2.2. Unassigning tag revisions
    5.5.2.3. Traffic flag operation precedence
    5.5.2.4. Traffic splitting flags by using the Knative CLI
5.6. TRACING REQUESTS USING JAEGER
  5.6.1. Configuring Jaeger for use with OpenShift Serverless
5.7. CONFIGURING JSON WEB TOKEN AUTHENTICATION FOR KNATIVE SERVICES
  5.7.1. Enabling sidecar injection for a Knative service
  5.7.2. Using JSON Web Token authentication with Service Mesh 2.x and OpenShift Serverless
  5.7.3. Using JSON Web Token authentication with Service Mesh 1.x and OpenShift Serverless
5.8. CONFIGURING A CUSTOM DOMAIN FOR A KNATIVE SERVICE
  5.8.1. Creating a custom domain mapping
  5.8.2. Creating a custom domain mapping by using the Knative CLI
  5.8.3. Configuring custom domains for private Knative services
    5.8.3.1. Setting cluster availability to cluster-local
    5.8.3.2. Creating necessary Service Mesh resources
    5.8.3.3. Accessing a service using your custom domain
  5.8.4. Additional resources
5.9. CONFIGURING TRANSPORT LAYER SECURITY FOR A CUSTOM DOMAIN USING RED HAT OPENSIFT SERVICE MESH AND KOURIER
  5.9.1. Prerequisites
  5.9.2. Configuring Transport Layer Security for a custom domain using Red Hat OpenShift Service Mesh 2.x
  5.9.3. Configuring Transport Layer Security for a custom domain using Red Hat OpenShift Service Mesh 1.x
5.10. CONFIGURING ROUTES FOR KNATIVE SERVICES
  5.10.1. Customizing labels and annotations for OpenShift Container Platform routes
  5.10.2. Configuring OpenShift Container Platform routes for Knative services
  5.10.3. Setting cluster availability to cluster-local
  5.10.4. Additional resources
5.11. METRICS
7.2.3.1. Listing event sources of a specific type only

7.3. USING THE API SERVER SOURCE

7.3.1. Prerequisites

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

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

7.3.4. Deleting an API server source using the Developer perspective

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

7.3.5.1. Knative CLI --sink flag

7.3.6. Deleting the API server source by using the Knative CLI

7.3.7. Using the API server source with the YAML method

7.3.8. Deleting the API server source

7.4. USING A PING SOURCE

7.4.1. Creating a ping source using the Developer perspective

7.4.2. Creating a ping source by using the Knative CLI

7.4.2.1. Knative CLI --sink flag

7.4.3. Deleting a ping source by using the Knative CLI

7.4.4. Using a ping source with YAML

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

7.5. USING SINK BINDING

7.5.1. Creating a sink binding by using the Knative CLI

7.5.1.1. Knative CLI --sink flag

7.5.2. Using sink binding with the YAML method

7.6. USING CONTAINER SOURCES

7.6.1. Creating custom event sources by using a container source

7.6.1.1. Guidelines for creating a container image

7.6.1.2. Example container images

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

7.6.3. Creating a container source by using the web console

7.7. USING A KAFKA SOURCE

7.7.1. Prerequisites

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

7.7.3. Creating a Kafka event source by using the Knative CLI

7.7.3.1. Knative CLI --sink flag

7.7.4. Creating a Kafka event source by using YAML

7.7.5. Additional resources

CHAPTER 8. CHANNELS

8.1. UNDERSTANDING CHANNELS

8.1.1. Next steps

8.2. CREATING AND DELETING CHANNELS

8.2.1. Creating a channel using the Developer perspective

8.2.2. Creating a channel by using the Knative CLI

8.2.3. Creating a default implementation channel by using YAML

8.2.4. Creating a Kafka channel by using YAML

8.2.5. Deleting a channel by using the Knative CLI

8.2.6. Next steps

8.3. SUBSCRIPTIONS

8.3.1. Creating subscriptions

8.3.1.1. Creating subscriptions in the Developer perspective

8.3.1.2. Creating subscriptions by using the Knative CLI

8.3.1.3. Creating subscriptions by using YAML

8.3.2. Configuring event delivery failure parameters using subscriptions

8.3.3. Describing subscriptions by using the Knative CLI
8.3.4. Listing subscriptions by using the Knative CLI
8.3.5. Updating subscriptions by using the Knative CLI
8.3.6. Deleting subscriptions by using the Knative CLI
8.4. CONFIGURING CHANNEL DEFAULTS
  8.4.1. Configuring the default channel implementation

CHAPTER 9. FUNCTIONS

9.1. ABOUT OPENSHIFT SERVERLESS FUNCTIONS
  9.1.1. Supported runtimes
  9.1.2. Next steps

9.2. SETTING UP OPENSHIFT SERVERLESS FUNCTIONS
  9.2.1. Prerequisites
  9.2.2. Using podman
  9.2.3. Next steps

9.3. GETTING STARTED WITH FUNCTIONS
  9.3.1. Prerequisites
  9.3.2. Creating functions
  9.3.3. Building functions
  9.3.4. Deploying functions
  9.3.5. Building and deploying functions with OpenShift Container Registry
  9.3.6. Emitting a test event to a deployed function
  9.3.7. Next steps

9.4. DEVELOPING NODE.JS FUNCTIONS
  9.4.1. Prerequisites
  9.4.2. Node.js function template structure
  9.4.3. About invoking Node.js functions
    9.4.3.1. Node.js context objects
      9.4.3.1.1. Context object methods
      9.4.3.1.2. CloudEvent data
    9.4.3.2. Node.js function return values
      9.4.3.2.1. Returning status codes
      9.4.3.2.2. Returning headers
  9.4.5. Testing Node.js functions
  9.4.6. Next steps

9.5. DEVELOPING GOLANG FUNCTIONS
  9.5.1. Prerequisites
  9.5.2. Golang function template structure
  9.5.3. About invoking Golang functions
    9.5.3.1. Functions triggered by an HTTP request
    9.5.3.2. Functions triggered by a cloud event
      9.5.3.2.1. CloudEvent trigger example
  9.5.4. Golang function return values
  9.5.5. Testing Golang functions
  9.5.6. Next steps

9.6. DEVELOPING PYTHON FUNCTIONS
  9.6.1. Prerequisites
  9.6.2. Python function template structure
  9.6.3. About invoking Python functions
  9.6.4. Python function return values
    9.6.4.1. Returning CloudEvents
  9.6.5. Testing Python functions
  9.6.6. Next steps

9.7. DEVELOPING QUARKUS FUNCTIONS
11.1.2. Installing the Knative CLI for Linux using an RPM
11.1.3. Installing the Knative CLI for Linux
11.1.4. Installing the Knative CLI for Linux on IBM Power Systems using an RPM
11.1.5. Installing the Knative CLI for Linux on IBM Power Systems
11.1.6. Installing the Knative CLI for Linux on IBM Z and LinuxONE using an RPM
11.1.7. Installing the Knative CLI for Linux on IBM Z and LinuxONE
11.1.8. Installing the Knative CLI for macOS
11.1.9. Installing the Knative CLI for Windows
11.1.10. Customizing the Knative CLI
11.1.11. Knative CLI plug-ins
11.2. KNATIVE CLI ADVANCED CONFIGURATION
   11.2.1. Customizing the Knative CLI
   11.2.2. Knative CLI plug-ins
11.3. KN FLAGS REFERENCE
   11.3.1. Knative CLI --sink flag
11.4. KNATIVE SERVING CLI COMMANDS
   11.4.1. kn service commands
   11.4.1.1. Creating serverless applications by using the Knative CLI
   11.4.1.2. Updating serverless applications by using the Knative CLI
   11.4.1.3. Applying service declarations
   11.4.1.4. Describing serverless applications by using the Knative CLI
   11.4.2. kn domain commands
   11.4.2.1. Creating a custom domain mapping by using the Knative CLI
   11.4.2.2. Managing custom domain mappings by using the Knative CLI
11.5. KNATIVE EVENTING CLI COMMANDS
   11.5.1. kn source commands
   11.5.1.1. Listing available event source types by using the Knative CLI
   11.5.1.2. Creating and managing container sources by using the Knative CLI
   11.5.1.3. Creating an API server source by using the Knative CLI
   11.5.1.4. Deleting the API server source by using the Knative CLI
   11.5.1.5. Creating a ping source by using the Knative CLI
   11.5.1.6. Deleting a ping source by using the Knative CLI
   11.5.1.7. Creating a Kafka event source by using the Knative CLI
11.6. KN FUNC
   11.6.1. Creating functions
   11.6.2. Building functions
   11.6.3. Deploying functions
   11.6.4. Listing existing functions
   11.6.5. Describing a function
   11.6.6. Emitting a test event to a deployed function
   11.6.6.1. kn func emit optional parameters
   11.6.7. Deleting a function
CHAPTER 1. OPENSIBLE SERVERLESS RELEASE NOTES

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

NOTE
OpenShift Serverless is based on the open source Knative project.
For details about the latest Knative component releases, see the Knative releases blog.

1.1. RELEASE NOTES FOR RED HAT OPENSIBLE SERVERLESS 1.16.0

1.1.1. New features

- OpenShift Serverless now uses Knative Serving 0.22.0.
- OpenShift Serverless now uses Knative Eventing 0.22.0.
- OpenShift Serverless now uses Kourier 0.22.0.
- OpenShift Serverless now uses Knative `kn` CLI 0.22.0.
- OpenShift Serverless now uses Knative Kafka 0.22.0.
- The `kn func` CLI plug-in now uses `func` 0.16.0.
- The `kn func emit` command has been added to the functions `kn` plug-in. You can use this command to send events to test locally deployed functions.

1.1.2. Known issues

- The AMQ Streams Operator might prevent the installation or upgrade of the OpenShift Serverless Operator. If this happens, the following error is thrown by Operator Lifecycle Manager (OLM):

  WARNING: found multiple channel heads: [amqstreams.v1.7.2 amqstreams.v1.6.2], please check the `replaces`/`skipRange` fields of the operator bundles.

  You can fix this issue by uninstalling the AMQ Streams Operator before installing or upgrading the OpenShift Serverless Operator. You can then reinstall the AMQ Streams Operator.

- If Service Mesh is enabled with mTLS, metrics for Knative Serving are disabled by default because Service Mesh prevents Prometheus from scraping metrics.
  If you want to enable Knative Serving metrics for use with Service Mesh and mTLS, you must complete the following steps:

  a. Specify `prometheus` as the `metrics.backend-destination` in the `observability` spec of the Knative Serving custom resource (CR):

      ```yaml
      apiVersion: operator.knative.dev/v1alpha1
      kind: KnativeServing
      metadata:
        name: knative-serving
      spec:
        ```
This step prevents metrics from being disabled by default.

b. Apply the following network policy to allow traffic from the Prometheus namespace:

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-from-openshift-monitoring-ns
  namespace: knative-serving
spec:
  ingress:
    - from:
      - namespaceSelector:
        matchLabels:
          name: "openshift-monitoring"
      podSelector: {}
  policyTypes:
  - Ingress
```

c. Modify and reapply the default Service Mesh control plane in the `istio-system` namespace, so that it includes the following spec:

```yaml
spec:
  proxy:
    networking:
      trafficControl:
        inbound:
          excludedPorts:
            - 8444
```

- If you deploy Service Mesh CRs with the Istio ingress enabled, you might see the following warning in the `istio-ingressgateway` pod:

```
2021-05-02T12:56:17.700398Z warning envoy config
[external/envoy/source/common/config/grpc_subscription_impl.cc:101] gRPC config for
type.googleapis.com/envoy.api.v2.Listener rejected: Error adding/updating listener(s)
0.0.0.0_8081: duplicate listener 0.0.0.0_8081 found
```

Your Knative services might also not be accessible.

You can use the following workaround to fix this issue by recreating the `knative-local-gateway` service:

a. Delete the existing `knative-local-gateway` service in the `istio-system` namespace:

```
$ oc delete services -n istio-system knative-local-gateway
```

b. Create and apply a `knative-local-gateway` service that contains the following YAML:

```yaml
apiVersion: v1
kind: Service
```
If you have 1000 Knative services on a cluster, and then perform a reinstall or upgrade of
Knative Serving, there is a delay when you create the first new service after the KnativeServing
custom resource definition (CRD) becomes Ready.
The 3scale-kourier-control service reconciles all previously existing Knative services before
processing the creation of a new service, which causes the new service to spend approximately
800 seconds in an IngressNotConfigured or Unknown state before the state updates to
Ready.

```yaml
metadata:
  name: knative-local-gateway
  namespace: istio-system
  labels:
    experimental.istio.io/disable-gateway-port-translation: "true"
spec:
  type: ClusterIP
  selector:
    istio: ingressgateway
  ports:
    - name: http2
      port: 80
      targetPort: 8081
```

OpenShift Container Platform 4.8 Serverless
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. 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.

Serverless is a cloud computing model where application developers do not 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 more quickly than in traditional models.

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. ADMINISTRATION GUIDE

4.1. INSTALLING THE OPENSHIFT SERVERLESS OPERATOR

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.

4.1.1. Defining cluster size requirements for an OpenShift Serverless installation

To install and use OpenShift Serverless, the OpenShift Container Platform cluster must be sized correctly.

NOTE

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

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 following limitations apply to all OpenShift Serverless deployments:

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

4.1.2. 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.

4.1.3. 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 Manual scaling a machine set.

### 4.1.4. 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.

**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 prevents you from being able to install the latest version of Knative Eventing using the OpenShift Serverless Operator.

**Procedure**

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

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

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.

### 4.1.5. 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**.

### 4.2. INSTALLING KNATIVE SERVING

After you have installed 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 **Knative Serving advanced configuration options**.

#### 4.2.1. Prerequisites

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

- You have installed the OpenShift Serverless Operator.

#### 4.2.2. 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.

4. Click the Create Knative Serving button.

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

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.

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.

   ```
   Conditions
   Type                  Status  Updated       Reason    Message
   DependenciesPresent  True    3 minutes ago -         -         -
   DeploymentsAvailable True    3 minutes ago -         -         -
   InstallationCompleted 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.

4.2.3. Installing Knative Serving using YAML
Procedure

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

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

2. Apply the `serving.yaml` file:

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

Verification

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

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

   **Example output**

   ```plaintext
   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:

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

   **Example output**

   ```plaintext
   NAME                               READY   STATUS    RESTARTS   AGE
   activator-5c596cf8d6-5i86c         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-tqwvvv   1/1     Running   0          9m26s
   controller-7fd5655f49-9gx2z5       1/1     Running   0          9m32s
   controller-7fd5655f49-pncv5        1/1     Running   0          9m14s
   kn-cli-downloads-8c65d4c9f-mt417   1/1     Running   0          9m42s
   webhook-5c7d678c7c-n267j           1/1     Running   0          9m35s
   ```
4.2.4. Knative Serving advanced 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.

### 4.2.4.1. 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**

```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 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).

### 4.2.4.2. 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**

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

### 4.2.5. Next steps

For cloud events functionality on OpenShift Serverless, you can install the Knative Eventing component. See the documentation on [Installing Knative Eventing](#).

### 4.3. 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.

#### 4.3.1. Prerequisites

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

#### 4.3.2. 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.

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

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**.

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

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.
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.

```
<table>
<thead>
<tr>
<th>Type</th>
<th>Status</th>
<th>Updated</th>
<th>Reason</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>InstallsSucceeded</td>
<td>True</td>
<td>2 minutes ago</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Ready</td>
<td>True</td>
<td>1 minute ago</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
```

**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.
4.3.3. Installing Knative Eventing using YAML

Procedure

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

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

   ```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:

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

Verification

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

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

   **Example output**
   
   ```text
   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:

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

   **Example output**
   
   ```text
   NAME                                   READY   STATUS    RESTARTS   AGE
   broker-controller-58765d9d49-g9zp6     1/1     Running   0          7m21s
   eventing-controller-65fdd66b54-jw7bh   1/1     Running   0          7m31s
   ```
4.4. 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 prevents you from being able to install the latest version of Knative Eventing using the OpenShift Serverless Operator.

4.4.1. 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:
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.

### 4.5. 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.

#### 4.5.1. 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
```

#### 4.5.2. 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:
4.5.3. 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.

4.5.4. 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.

**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

4.6. INTEGRATING SERVICE MESH WITH OPENSHIFT SERVERLESS

Using Service Mesh with OpenShift Serverless enables developers to configure additional networking and routing options.

The OpenShift Serverless Operator provides Kourier as the default ingress for Knative. However, you can use Service Mesh with OpenShift Serverless whether Kourier is enabled or not. Integrating with Kourier disabled allows you to configure additional networking and routing options that the Kourier ingress does not support.

**IMPORTANT**

OpenShift Serverless only supports the use of Red Hat OpenShift Service Mesh functionality that is explicitly documented in this guide, and does not support other undocumented features.

4.6.1. Integrating Service Mesh with OpenShift Serverless natively

Integrating Service Mesh with OpenShift Serverless natively, without Kourier, allows you to use additional networking and routing options that are not supported by the default Kourier ingress, such as mTLS functionality.
IMPORTANT

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

For more information about the support scope of Red Hat Technology Preview features, see https://access.redhat.com/support/offerings/techpreview/.

The examples in the following procedures use the domain example.com. The example certificate for this domain is used as a certificate authority (CA) that signs the subdomain certificate.

To complete and verify these procedures 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.

You must configure the wildcard certificate to match the domain of your OpenShift Container Platform cluster. For example, if your OpenShift Container Platform console address is https://console-openshift-console.apps.openshift.example.com, you must configure the wildcard certificate so that the domain is *.apps.openshift.example.com. For more information about configuring wildcard certificates, see the following topic about Creating a certificate to encrypt incoming external traffic.

If you want to use any domain name, including those which are not subdomains of the default OpenShift Container Platform cluster domain, you must set up domain mapping for those domains. For more information, see the OpenShift Serverless documentation on Creating a custom domain mapping.

4.6.1.1. Creating a certificate to encrypt incoming external traffic

By default, the Service Mesh mTLS feature only secures traffic inside of the Service Mesh itself, between the ingress gateway and individual pods that have sidecars. To encrypt traffic as it flows into the OpenShift Container Platform cluster, you must generate a certificate before you enable the OpenShift Serverless and Service Mesh integration.

Procedure

1. Create a root certificate and private key that signs the certificates for your Knative services:

   ```
   $ openssl req -x509 -sha256 -nodes -days 365 -newkey rsa:2048 \
   -subj '/O=Example Inc./CN=example.com' \
   -keyout root.key \
   -out root.crt
   ```

2. Create a wildcard certificate:

   ```
   $ openssl req -nodes -newkey rsa:2048 \
   -subj "/CN=*.apps.openshift.example.com/O=Example Inc." \
   -keyout wildcard.key \
   -out wildcard.csr
   ```

3. Sign the wildcard certificate:

   ```
   $ openssl x509 -req -days 365 -set_serial 0 \
   ```
4. Create a secret by using the wildcard certificate:

```
$ oc create -n istio-system secret tls wildcard-certs \
    --key=wildcard.key \
    --cert=wildcard.crt
```

This certificate is picked up by the gateways created when you integrate OpenShift Serverless with Service Mesh, so that the ingress gateway serves traffic with this certificate.

### 4.6.1.2. Integrating Service Mesh with OpenShift Serverless

You can integrate Service Mesh with OpenShift Serverless without using Kourier by completing the following procedure.

**Prerequisites**

- You have installed the OpenShift Serverless Operator on your OpenShift Container Platform cluster.
- You have installed Red Hat OpenShift Service Mesh. OpenShift Serverless with Service Mesh only is supported for use with Red Hat OpenShift Service Mesh version 2.0.5 or higher.

**IMPORTANT**

Do not install the Knative Serving component before completing the following procedures. There are additional steps required when creating the `KnativeServing` custom resource definition (CRD) to integrate Knative Serving with Service Mesh, which are not covered in the general Knative Serving installation procedure of the *Administration guide*.

**Procedure**

1. Create a `ServiceMeshControlPlane` object in the `istio-system` namespace. If you want to use the mTLS functionality, this must be enabled for the `istio-system` namespace.

2. Add the namespaces that you would like to integrate with Service Mesh to the `ServiceMeshMemberRoll` object as members:

   ```yaml
   apiVersion: maistra.io/v1
   kind: ServiceMeshMemberRoll
   metadata:
     name: default
     namespace: istio-system
   spec:
     members: 1
     - knative-serving
     - <namespace>
   ```

   1 A list of namespaces to be integrated with Service Mesh.
IMPORTANT

This list of namespaces must include the **knative-serving** namespace.

3. Apply the **ServiceMeshMemberRoll** resource:

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

4. Create the necessary gateways so that Service Mesh can accept traffic:

**Example knative-local-gateway object using HTTP**

```yaml
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: knative-ingress-gateway
  namespace: knative-serving
spec:
  selector:
    istio: ingressgateway
  servers:
    - port:
        number: 443
          name: https
          protocol: HTTPS
        hosts:
          - "*"
        tls:
          mode: SIMPLE
          credentialName: <wildcard_certs>

---

apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: knative-local-gateway
  namespace: knative-serving
spec:
  selector:
    istio: ingressgateway
  servers:
    - port:
        number: 8081
          name: http
          protocol: HTTP
        hosts:
          - "*"

---

apiVersion: v1
kind: Service
metadata:
  name: knative-local-gateway
  namespace: istio-system
labels:
  experimental.istio.io/disable-gateway-port-translation: "true"
spec:
Add the name of your wildcard certificate.

The knative-local-gateway serves HTTP traffic. Using HTTP means that traffic coming from outside of Service Mesh, but using an internal hostname, such as example.default.svc.cluster.local, is not encrypted. You can set up encryption for this path by creating another wildcard certificate and an additional gateway that uses a different protocol spec.

**Example knative-local-gateway object using HTTPS**

```yaml
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: knative-local-gateway
  namespace: knative-serving
spec:
  selector:
    istio: ingressgateway
  servers:
    - port:
        number: 443
        name: https
        protocol: HTTPS
        hosts: ["*"]
        tls: SIMPLE
        credentialName: <wildcard_certs>

5. Apply the **Gateway** resources:

   $ oc apply -f <filename>

6. Install Knative Serving by creating the following **KnativeServing** custom resource definition (CRD), which also enables the Istio integration:

```yaml
apiVersion: operator.knative.dev/v1alpha1
kind: KnativeServing
metadata:
  name: knative-serving
  namespace: knative-serving
spec:
  ingress:
    istio:
      enabled: true
  deployments:
```
1. Enables Istio integration.

2. Enables sidecar injection for Knative Serving data plane pods.

7. Apply the **KnativeServing** resource:

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

8. Create a Knative Service that has sidecar injection enabled and uses a pass-through route:

   ```
   apiVersion: serving.knative.dev/v1
   kind: Service
   metadata:
     name: <service_name>
     namespace: <namespace>  
   annotations:
     serving.knative.openshift.io/enablePassthrough: "true"  
   spec:
     template:
       metadata:
         annotations:
           sidecar.istio.io/inject: "true"  
           sidecar.istio.io/rewriteAppHTTPProbers: "true"
   ```

1. A namespace that is part of the Service Mesh member roll.

2. Instructs Knative Serving to generate an OpenShift Container Platform pass-through enabled route, so that the certificates you have generated are served through the ingress gateway directly.

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

9. Apply the **Service** resource:

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

**Verification**

- Access your serverless application by using a secure connection that is now trusted by the CA:

  ```
  $ curl --cacert root.crt <service_url>
  ```
Example command

```
$ curl --cacert root.crt https://hello-default.apps.openshift.example.com
```

Example output

Hello Openshift!

### 4.6.2. Integrating Service Mesh with OpenShift Serverless when Kourier is enabled

#### Prerequisites

- You have installed the OpenShift Serverless Operator on your OpenShift Container Platform cluster.
- You have installed Red Hat OpenShift Service Mesh. OpenShift Serverless with Service Mesh and Kourier is supported for use with both Red Hat OpenShift Service Mesh versions 1.x and 2.x.
- You have installed Knative Serving.

#### Procedure

1. Add the namespaces that you would like to integrate with Service Mesh to the `ServiceMeshMemberRoll` object as members:

   ```yaml
   apiVersion: maistra.io/v1
   kind: ServiceMeshMemberRoll
   metadata:
     name: default
     namespace: istio-system
   spec:
     members:
       - <namespace>  
   ```

   A list of namespaces to be integrated with Service Mesh.

2. Apply the `ServiceMeshMemberRoll` resource:

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

3. 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:

   ```bash
   $ 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:
c. For each namespace that you want to integrate with Service Mesh, create a **NetworkPolicy** resource:

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-from-serving-system-namespace
  namespace: <namespace>  # 1
spec:
  ingress:
    - from:
      - namespaceSelector:
        matchLabels:
          serving.knative.openshift.io/system-namespace: "true"
      podSelector: {}
  policyTypes:
    - Ingress
```

1. Add the namespace that you want to integrate with Service Mesh.

d. Apply the **NetworkPolicy** resource:

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

### 4.7. CREATING EVENTING COMPONENTS IN THE ADMINISTRATOR PERSPECTIVE

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

#### 4.7.1. 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 Understanding event sources for more information on which event source types are supported and can be created using by OpenShift Serverless.

### 4.7.2. 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**.

### 4.7.3. 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**.

### 4.7.4. 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**.

### 4.7.5. 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**.

### 4.7.6. Additional resources

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

### 4.8. 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.

#### 4.8.1. 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**.

### 4.9. MONITORING SERVERLESS COMPONENTS
You can use OpenShift Container Platform monitoring dashboards to view health checks and metrics for OpenShift Serverless components.

### 4.9.1. Monitoring the overall health status of Knative components

You can use the OpenShift Container Platform monitoring dashboards to view the overall health status of Knative.

**Prerequisites**

- You have cluster administrator permissions, and access to the Administrator perspective in the OpenShift Container Platform web console.
- You have installed the OpenShift Serverless Operator, as well as the Knative Serving or Knative Eventing components.
- The OpenShift Container Platform monitoring stack is enabled on your cluster. You can enable monitoring for OpenShift Serverless during installation by checking the box to **Enable operator recommended cluster monitoring on this namespace** when installing the OpenShift Serverless Operator.

**Procedure**

1. In the Administrator perspective, navigate to Monitoring → Dashboards.
2. Select the Knative Health Status dashboard in the Dashboard drop-down to view the overall health status of Knative. If your Knative deployment is running as expected, the dashboard shows a status of Ready.

   ![Dashboard Screenshot](image)

   If you have Knative Serving or Knative Eventing installed, you can also scroll down to see the health status for each of these components.

### 4.9.2. 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 cluster administrator permissions, and access to the Administrator perspective in the OpenShift Container Platform web console.
- You have installed the OpenShift Serverless Operator, as well as the Knative Serving component.
- The OpenShift Container Platform monitoring stack is enabled on your cluster. You can enable monitoring for OpenShift Serverless during installation by checking the box to **Enable operator recommended cluster monitoring on this namespace** when installing the OpenShift Serverless Operator.
**Procedure**

1. In the Administrator perspective, navigate to **Monitoring → Dashboards**.

2. Select the **Knative Serving - Source CPU and Memory Usage** dashboard in the Dashboard drop-down list to view the following metrics:
   - Total CPU Usage (rate per minute)
   - Total Memory Usage (bytes)
   - Total Network I/O (rate per minute)
   - Total Network Errors (rate per minute)

3. Optional: You can filter this dashboard by **Namespace**, **Configuration**, or **Revision**, by selecting an option from the drop-down list.

### 4.9.3. Monitoring Knative Eventing source CPU and memory usage

You can use the OpenShift Container Platform monitoring dashboards to view source CPU and memory usage metrics for Knative Eventing components.

**Prerequisites**

- You have cluster administrator permissions, and access to the Administrator perspective in the OpenShift Container Platform web console.
- You have installed the OpenShift Serverless Operator, as well as the Knative Eventing component.
- The OpenShift Container Platform monitoring stack is enabled on your cluster. You can enable monitoring for OpenShift Serverless during installation by checking the box to **Enable operator recommended cluster monitoring on this namespace** when installing the OpenShift Serverless Operator.

**Procedure**

1. In the Administrator perspective, navigate to **Monitoring → Dashboards**.

2. Select the **Knative Eventing - Source CPU and Memory Usage** dashboard in the Dashboard drop-down list to view the following metrics:
   - Total CPU Usage (rate per minute)
   - Total Memory Usage (bytes)
   - Total Network I/O (rate per minute)
   - Total Network Errors (rate per minute)

### 4.9.4. Monitoring event sources
You can use the OpenShift Container Platform monitoring dashboards to view metrics for event sources in your cluster.

Prerequisites

- You have cluster administrator permissions, and access to the Administrator perspective in the OpenShift Container Platform web console.
- You have installed the OpenShift Serverless Operator, as well as the Knative Eventing component.
- The OpenShift Container Platform monitoring stack is enabled on your cluster. You can enable monitoring for OpenShift Serverless during installation by checking the box to Enable operator recommended cluster monitoring on this namespace when installing the OpenShift Serverless Operator.

Procedure

1. In the Administrator perspective, navigate to Monitoring → Dashboards.
2. Select the Knative Eventing - Sources dashboard in the Dashboard drop-down list.
3. You can now view the following metrics:
   a. For API server sources:
      - Event Count (rate per minute)
      - Success Rate (2xx Event, fraction rate per minute)
      - Event Count by Response Code Class (rate per minute)
      - Failure Rate (non-2xx Event, fraction rate per minute)
   b. For ping sources:
      - Event Count (rate per minute)
      - Success Rate (2xx Event, fraction rate per minute)
      - Event Count by Response Code Class (rate per minute)
      - Failure Rate (non-2xx Event, fraction rate per minute)
   c. For Kafka sources:
      - Event Count (rate per minute)
      - Success Rate (2xx Event, fraction rate per minute)
      - Event Count by Response Code Class (rate per minute)
      - Failure Rate (non-2xx Event, fraction rate per minute)

4.9.5. Monitoring Knative Eventing brokers and triggers
You can use the OpenShift Container Platform monitoring dashboards to view metrics for brokers and triggers in your cluster.

Prerequisites

- You have cluster administrator permissions, and access to the Administrator perspective in the OpenShift Container Platform web console.
- You have installed the OpenShift Serverless Operator, as well as the Knative Eventing component.
- The OpenShift Container Platform monitoring stack is enabled on your cluster. You can enable monitoring for OpenShift Serverless during installation by checking the box to Enable operator recommended cluster monitoring on this namespace when installing the OpenShift Serverless Operator.

Procedure

1. In the Administrator perspective, navigate to Monitoring → Dashboards.
2. Select the Knative Eventing - Broker/Trigger dashboard in the Dashboard drop-down list.
3. You can now view the following metrics:
   a. For brokers:
      - Event Count (rate per minute)
      - Success Rate (2xx Event, rate per minute)
      - Event Count by Event Type (rate per minute)
      - Failure Rate (non-2xx Event, fraction of rate per minute)
      - Event Count by Response Code Class (rate per minute)
      - Event Dispatch Latency (ms)
   b. For triggers:
      - Event Count (rate per minute)
      - Success Rate (2xx Event, fraction rate per minute)
      - Event Count by Response Code Class (rate per minute)
      - Failure Rate (non-2xx Event, fraction rate per minute)
      - Event Dispatch Latency (ms)
      - Event Processing Latency (ms)

4.10. METRICS

Metrics enable cluster administrators to monitor how OpenShift Serverless cluster components and workloads are performing.
4.10.1. Prerequisites

- See the OpenShift Container Platform documentation on *Managing metrics* for information about enabling metrics for your cluster.

- To view metrics for Knative components on OpenShift Container Platform, you need cluster administrator permissions, and access to the web console *Administrator* perspective.

**WARNING**

If Service Mesh is enabled with mTLS, metrics for Knative Serving are disabled by default because Service Mesh prevents Prometheus from scraping metrics.

For information about resolving this issue, see the *Serverless 1.16.0 release notes*.

4.10.2. Controller metrics

The following metrics are emitted by any component that implements a controller logic. These metrics show details about reconciliation operations and the work queue behavior upon which reconciliation requests are added to the work queue.

<table>
<thead>
<tr>
<th>Metric name</th>
<th>Description</th>
<th>Type</th>
<th>Tags</th>
<th>Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>work_queue_depth</td>
<td>The depth of the work queue.</td>
<td>Gauge</td>
<td>reconciler</td>
<td>Integer (no units)</td>
</tr>
<tr>
<td>reconcile_count</td>
<td>The number of reconcile operations.</td>
<td>Counter</td>
<td>reconciler, success</td>
<td>Integer (no units)</td>
</tr>
<tr>
<td>reconcile_latency</td>
<td>The latency of reconcile operations.</td>
<td>Histogram</td>
<td>reconciler, success</td>
<td>Milliseconds</td>
</tr>
<tr>
<td>workqueue_adds_total</td>
<td>The total number of add actions handled by the work queue.</td>
<td>Counter</td>
<td>name</td>
<td>Integer (no units)</td>
</tr>
<tr>
<td>workqueue_queue_latency_seconds</td>
<td>The length of time an item stays in the work queue before being requested.</td>
<td>Histogram</td>
<td>name</td>
<td>Seconds</td>
</tr>
<tr>
<td>workqueue_retries_total</td>
<td>The total number of retries that have been handled by the work queue.</td>
<td>Counter</td>
<td>name</td>
<td>Integer (no units)</td>
</tr>
</tbody>
</table>
### 4.10.3. Webhook metrics

Webhook metrics report useful information about operations. For example, if a large number of operations fail, this might indicate an issue with a user-created resource.
4.10.4. Knative Eventing metrics

Cluster administrators can view the following metrics for Knative Eventing components.

By aggregating the metrics from HTTP code, events can be separated into two categories; successful events (2xx) and failed events (5xx).

4.10.4.1. Broker ingress metrics

You can use the following metrics to debug the broker ingress, see how it is performing, and see which events are being dispatched by the ingress component.

<table>
<thead>
<tr>
<th>Metric name</th>
<th>Description</th>
<th>Type</th>
<th>Tags</th>
<th>Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>request_latencies</td>
<td>The response time for a webhook request.</td>
<td>Histogram</td>
<td>admission_allowed, kind_group, kind_kind, kind_version, request_operation, resource_group, resource_namespace, resource_resource, resource_version</td>
<td>Milliseconds</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Metric name</th>
<th>Description</th>
<th>Type</th>
<th>Tags</th>
<th>Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>event_count</td>
<td>Number of events received by a broker.</td>
<td>Counter</td>
<td>broker_name, event_type, namespace_name, response_code, response_code_class, unique_name</td>
<td>Integer (no units)</td>
</tr>
<tr>
<td>event_dispatch_latencies</td>
<td>The time taken to dispatch an event to a channel.</td>
<td>Histogram</td>
<td>broker_name, event_type, namespace_name, response_code, response_code_class, unique_name</td>
<td>Milliseconds</td>
</tr>
</tbody>
</table>
You can use the following metrics to debug broker filters, see how they are performing, and see which events are being dispatched by the filters. You can also measure the latency of the filtering action on an event.

<table>
<thead>
<tr>
<th>Metric name</th>
<th>Description</th>
<th>Type</th>
<th>Tags</th>
<th>Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>event_count</td>
<td>Number of events received by a broker.</td>
<td>Counter</td>
<td>broker_name, container_name, filter_type, namespace_name, response_code, response_code_class, trigger_name, unique_name</td>
<td>Integer (no units)</td>
</tr>
<tr>
<td>event_dispatch_latencies</td>
<td>The time taken to dispatch an event to a channel.</td>
<td>Histogram</td>
<td>broker_name, container_name, filter_type, namespace_name, response_code, response_code_class, trigger_name, unique_name</td>
<td>Milliseconds</td>
</tr>
<tr>
<td>event_processing_latencies</td>
<td>The time it takes to process an event before it is dispatched to a trigger subscriber.</td>
<td>Histogram</td>
<td>broker_name, container_name, filter_type, namespace_name, trigger_name, unique_name</td>
<td>Milliseconds</td>
</tr>
</tbody>
</table>

4.10.4.3. InMemoryChannel dispatcher metrics

You can use the following metrics to debug InMemoryChannel channels, see how they are performing, and see which events are being dispatched by the channels.

<table>
<thead>
<tr>
<th>Metric name</th>
<th>Description</th>
<th>Type</th>
<th>Tags</th>
<th>Unit</th>
</tr>
</thead>
</table>
### 4.10.4.4. Event source metrics

You can use the following metrics to verify that events have been delivered from the event source to the connected event sink.

<table>
<thead>
<tr>
<th>Metric name</th>
<th>Description</th>
<th>Type</th>
<th>Tags</th>
<th>Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>event_count</td>
<td>Number of events sent by the event source.</td>
<td>Counter</td>
<td>broker_name, container_name, filter_type, namespace_name, response_code, response_code_class, trigger_name, unique_name</td>
<td>Integer (no units)</td>
</tr>
<tr>
<td>event_dispatch_latencies</td>
<td>The time taken to dispatch an event from an InMemoryChannel channel.</td>
<td>Histogram</td>
<td>broker_name, container_name, filter_type, namespace_name, response_code, response_code_class, trigger_name, unique_name</td>
<td>Milliseconds</td>
</tr>
</tbody>
</table>
4.10.5. Knative Serving metrics

Cluster administrators can view the following metrics for Knative Serving components.

4.10.5.1. Activator metrics

You can use the following metrics to understand how applications respond when traffic passes through the activator.

<table>
<thead>
<tr>
<th>Metric name</th>
<th>Description</th>
<th>Type</th>
<th>Tags</th>
<th>Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>retry_event_count</td>
<td>Number of retried events sent by the event source after initially failing to be delivered.</td>
<td>Counter</td>
<td>event_source, event_type, name, namespace_name, resource_group, response_code, response_code_class, response_error, response_timeout</td>
<td>Integer (no units)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Metric name</th>
<th>Description</th>
<th>Type</th>
<th>Tags</th>
<th>Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>request_concurrency</td>
<td>The number of concurrent requests that are routed to the activator, or average concurrency over a reporting period.</td>
<td>Gauge</td>
<td>configuration_name, container_name, namespace_name, pod_name, revision_name, service_name</td>
<td>Integer (no units)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Metric name</th>
<th>Description</th>
<th>Type</th>
<th>Tags</th>
<th>Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>request_count</td>
<td>The number of requests that are routed to activator. These are requests that have been fulfilled from the activator handler.</td>
<td>Counter</td>
<td>configuration_name, container_name, namespace_name, pod_name, response_code, response_code_class, revision_name, service_name</td>
<td>Integer (no units)</td>
</tr>
</tbody>
</table>
4.10.5.2. Autoscaler metrics

The autoscaler component exposes a number of metrics related to autoscaler behavior for each revision. For example, at any given time, you can monitor the targeted number of pods the autoscaler tries to allocate for a service, the average number of requests per second during the stable window, or whether the autoscaler is in panic mode if you are using the Knative pod autoscaler (KPA).

<table>
<thead>
<tr>
<th>Metric name</th>
<th>Description</th>
<th>Type</th>
<th>Tags</th>
<th>Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>desired_pods</td>
<td>The number of pods the autoscaler tries to allocate for a service.</td>
<td>Gauge</td>
<td>configuration_name, namespace_name, revision_name, service_name</td>
<td>Integer (no units)</td>
</tr>
<tr>
<td>excess_burst_capacity</td>
<td>The excess burst capacity served over the stable window.</td>
<td>Gauge</td>
<td>configuration_name, namespace_name, revision_name, service_name</td>
<td>Integer (no units)</td>
</tr>
<tr>
<td>stable_request_concurrency</td>
<td>The average number of requests for each observed pod over the stable window.</td>
<td>Gauge</td>
<td>configuration_name, namespace_name, revision_name, service_name</td>
<td>Integer (no units)</td>
</tr>
<tr>
<td>panic_request_concurrency</td>
<td>The average number of requests for each observed pod over the panic window.</td>
<td>Gauge</td>
<td>configuration_name, namespace_name, revision_name, service_name</td>
<td>Integer (no units)</td>
</tr>
<tr>
<td>Metric name</td>
<td>Description</td>
<td>Type</td>
<td>Tags</td>
<td>Unit</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>------------------------------------------------------------------------------</td>
<td>--------</td>
<td>----------------------------------------------------------------------</td>
<td>------------------</td>
</tr>
<tr>
<td>target_concurrency_per_pod</td>
<td>The number of concurrent requests that the autoscaler tries to send to each pod.</td>
<td>Gauge</td>
<td>configuration_name, namespace_name, revision_name, service_name</td>
<td>Integer (no units)</td>
</tr>
<tr>
<td>stable_requests_per_second</td>
<td>The average number of requests-per-second for each observed pod over the stable window.</td>
<td>Gauge</td>
<td>configuration_name, namespace_name, revision_name, service_name</td>
<td>Integer (no units)</td>
</tr>
<tr>
<td>panic_requests_per_second</td>
<td>The average number of requests-per-second for each observed pod over the panic window.</td>
<td>Gauge</td>
<td>configuration_name, namespace_name, revision_name, service_name</td>
<td>Integer (no units)</td>
</tr>
<tr>
<td>target_requests_per_second</td>
<td>The number of requests-per-second that the autoscaler targets for each pod.</td>
<td>Gauge</td>
<td>configuration_name, namespace_name, revision_name, service_name</td>
<td>Integer (no units)</td>
</tr>
<tr>
<td>panic_mode</td>
<td>This value is 1 if the autoscaler is in panic mode, or 0 if the autoscaler is not in panic mode.</td>
<td>Gauge</td>
<td>configuration_name, namespace_name, revision_name, service_name</td>
<td>Integer (no units)</td>
</tr>
<tr>
<td>requested_pods</td>
<td>The number of pods that the autoscaler has requested from the Kubernetes cluster.</td>
<td>Gauge</td>
<td>configuration_name, namespace_name, revision_name, service_name</td>
<td>Integer (no units)</td>
</tr>
<tr>
<td>actual_pods</td>
<td>The number of pods that are allocated and currently have a ready state.</td>
<td>Gauge</td>
<td>configuration_name, namespace_name, revision_name, service_name</td>
<td>Integer (no units)</td>
</tr>
<tr>
<td>Metric name</td>
<td>Description</td>
<td>Type</td>
<td>Tags</td>
<td>Unit</td>
</tr>
<tr>
<td>--------------------</td>
<td>-----------------------------------------------------------</td>
<td>----------</td>
<td>-----------------------------------</td>
<td>------------------</td>
</tr>
<tr>
<td>not_ready_pods</td>
<td>The number of pods that have a not ready state.</td>
<td>Gauge</td>
<td>configuration_name, namespace_name, revision_name, service_name</td>
<td>Integer (no units)</td>
</tr>
<tr>
<td>pending_pods</td>
<td>The number of pods that are currently pending.</td>
<td>Gauge</td>
<td>configuration_name, namespace_name, revision_name, service_name</td>
<td>Integer (no units)</td>
</tr>
<tr>
<td>terminating_pods</td>
<td>The number of pods that are currently terminating.</td>
<td>Gauge</td>
<td>configuration_name, namespace_name, revision_name, service_name</td>
<td>Integer (no units)</td>
</tr>
</tbody>
</table>

**4.10.5.3. Go runtime metrics**

Each Knative Serving control plane process emits a number of Go runtime memory statistics (MemStats).

**NOTE**

The `name` tag for each metric is an empty tag.

<table>
<thead>
<tr>
<th>Metric name</th>
<th>Description</th>
<th>Type</th>
<th>Tags</th>
<th>Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>go_alloc</td>
<td>The number of bytes of allocated heap objects. This metric is the same as heap_alloc.</td>
<td>Gauge</td>
<td>name</td>
<td>Integer (no units)</td>
</tr>
<tr>
<td>go_total_alloc</td>
<td>The cumulative bytes allocated for heap objects.</td>
<td>Gauge</td>
<td>name</td>
<td>Integer (no units)</td>
</tr>
<tr>
<td>go_sys</td>
<td>The total bytes of memory obtained from the operating system.</td>
<td>Gauge</td>
<td>name</td>
<td>Integer (no units)</td>
</tr>
<tr>
<td>Metric name</td>
<td>Description</td>
<td>Type</td>
<td>Tags</td>
<td>Unit</td>
</tr>
<tr>
<td>--------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>-------</td>
<td>------</td>
<td>------------------------</td>
</tr>
<tr>
<td>go_lookups</td>
<td>The number of pointer lookups performed by the runtime.</td>
<td>Gauge</td>
<td>name</td>
<td>Integer (no units)</td>
</tr>
<tr>
<td>go_mallocs</td>
<td>The cumulative count of heap objects allocated.</td>
<td>Gauge</td>
<td>name</td>
<td>Integer (no units)</td>
</tr>
<tr>
<td>go_frees</td>
<td>The cumulative count of heap objects that have been freed.</td>
<td>Gauge</td>
<td>name</td>
<td>Integer (no units)</td>
</tr>
<tr>
<td>go_heap_alloc</td>
<td>The number of bytes of allocated heap objects.</td>
<td>Gauge</td>
<td>name</td>
<td>Integer (no units)</td>
</tr>
<tr>
<td>go_heap_sys</td>
<td>The number of bytes of heap memory obtained from the operating system.</td>
<td>Gauge</td>
<td>name</td>
<td>Integer (no units)</td>
</tr>
<tr>
<td>go_heap_idle</td>
<td>The number of bytes in idle, unused spans.</td>
<td>Gauge</td>
<td>name</td>
<td>Integer (no units)</td>
</tr>
<tr>
<td>go_heap_in_use</td>
<td>The number of bytes in spans that are currently in use.</td>
<td>Gauge</td>
<td>name</td>
<td>Integer (no units)</td>
</tr>
<tr>
<td>go_heap_released</td>
<td>The number of bytes of physical memory returned to the operating system.</td>
<td>Gauge</td>
<td>name</td>
<td>Integer (no units)</td>
</tr>
<tr>
<td>go_heap_objects</td>
<td>The number of allocated heap objects.</td>
<td>Gauge</td>
<td>name</td>
<td>Integer (no units)</td>
</tr>
<tr>
<td>go_stack_in_use</td>
<td>The number of bytes in stack spans that are currently in use.</td>
<td>Gauge</td>
<td>name</td>
<td>Integer (no units)</td>
</tr>
<tr>
<td>Metric name</td>
<td>Description</td>
<td>Type</td>
<td>Tags</td>
<td>Unit</td>
</tr>
<tr>
<td>---------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>--------</td>
<td>------</td>
<td>-----------------</td>
</tr>
<tr>
<td>go_stack_sys</td>
<td>The number of bytes of stack memory obtained from the operating system.</td>
<td>Gauge</td>
<td>name</td>
<td>Integer (no units)</td>
</tr>
<tr>
<td>go_mspan_in_use</td>
<td>The number of bytes of allocated mspan structures.</td>
<td>Gauge</td>
<td>name</td>
<td>Integer (no units)</td>
</tr>
<tr>
<td>go_mspan_sys</td>
<td>The number of bytes of memory obtained from the operating system for mspan structures.</td>
<td>Gauge</td>
<td>name</td>
<td>Integer (no units)</td>
</tr>
<tr>
<td>go_mcache_in_use</td>
<td>The number of bytes of allocated mcache structures.</td>
<td>Gauge</td>
<td>name</td>
<td>Integer (no units)</td>
</tr>
<tr>
<td>go_mcache_sys</td>
<td>The number of bytes of memory obtained from the operating system for mcache structures.</td>
<td>Gauge</td>
<td>name</td>
<td>Integer (no units)</td>
</tr>
<tr>
<td>go_bucket_has_sys</td>
<td>The number of bytes of memory in profiling bucket hash tables.</td>
<td>Gauge</td>
<td>name</td>
<td>Integer (no units)</td>
</tr>
<tr>
<td>go_gc_sys</td>
<td>The number of bytes of memory in garbage collection metadata.</td>
<td>Gauge</td>
<td>name</td>
<td>Integer (no units)</td>
</tr>
<tr>
<td>go_other_sys</td>
<td>The number of bytes of memory in miscellaneous, off-heap runtime allocations.</td>
<td>Gauge</td>
<td>name</td>
<td>Integer (no units)</td>
</tr>
<tr>
<td>go_next_gc</td>
<td>The target heap size of the next garbage collection cycle.</td>
<td>Gauge</td>
<td>name</td>
<td>Integer (no units)</td>
</tr>
<tr>
<td>Metric name</td>
<td>Description</td>
<td>Type</td>
<td>Tags</td>
<td>Unit</td>
</tr>
<tr>
<td>-----------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>----------</td>
<td>------</td>
<td>------------</td>
</tr>
<tr>
<td>go_last_gc</td>
<td>The time that the last garbage collection was completed in Epoch or Unix time.</td>
<td>Gauge</td>
<td>name</td>
<td>Nanoseconds</td>
</tr>
<tr>
<td>go_total_gc_pause_ns</td>
<td>The cumulative time in garbage collection stop-the-world pauses since the program started.</td>
<td>Gauge</td>
<td>name</td>
<td>Nanoseconds</td>
</tr>
<tr>
<td>go_num_gc</td>
<td>The number of completed garbage collection cycles.</td>
<td>Gauge</td>
<td>name</td>
<td>Integer (no units)</td>
</tr>
<tr>
<td>go_num_forced_gc</td>
<td>The number of garbage collection cycles that were forced due to an application calling the garbage collection function.</td>
<td>Gauge</td>
<td>name</td>
<td>Integer (no units)</td>
</tr>
<tr>
<td>go_gc_cpu_fraction</td>
<td>The fraction of the available CPU time of the program that has been used by the garbage collector since the program started.</td>
<td>Gauge</td>
<td>name</td>
<td>Integer (no units)</td>
</tr>
</tbody>
</table>

### 4.11. USING METERING WITH OPENSHIFT SERVERLESS

**IMPORTANT**

Metering is a deprecated feature. Deprecated functionality is still included in OpenShift Container Platform and continues to be supported; however, it will be removed in a future release of this product and is not recommended for new deployments.

For the most recent list of major functionality that has been deprecated or removed within OpenShift Container Platform, refer to the *Deprecated and removed features* section of the OpenShift Container Platform release notes.
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.

### 4.11.1. Installing metering

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

### 4.11.2. Datasources for Knative Serving metering

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

#### 4.11.2.1. 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**

```yaml
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!=""}
      )

apiVersion: metering.openshift.io/v1
```

#### 4.11.2.2. Datasource for memory usage in Knative Serving

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

**YAML file**

```yaml
apiVersion: metering.openshift.io/v1
```
4.11.2.3. Applying Datasources for Knative Serving metering

You can apply the ReportDataSources by using the following command:

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

Example

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

4.11.3. Queries for Knative Serving metering

The following ReportQuery resources reference the example DataSources provided.

4.11.3.1. Query for CPU usage in Knative Serving

**YAML file**

```yaml
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
```
4.11.3.2. 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

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
  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 |}
GROUP BY labels['namespace'],labels['label_serving_knative_dev_service']
4.11.3.3. Applying Queries for Knative Serving metering

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

   ```bash
   $ oc apply -f <query-name>.yaml
   
   Example command
   
   $ oc apply -f knative-service-memory-usage.yaml
   ```

4.11.4. 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
4.11.4.1. Running a metering report

1. Run the report by entering the following command:

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

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

   ```bash
   $ oc get report
   ```

   **Example output**

<table>
<thead>
<tr>
<th>NAME</th>
<th>QUERY</th>
<th>SCHEDULE</th>
<th>RUNNING</th>
<th>FAILED</th>
<th>LAST REPORT TIME</th>
<th>AGE</th>
</tr>
</thead>
</table>

4.12. HIGH AVAILABILITY ON OPENSFiGT 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 or Eventing control plane is installed.

When using a leader election HA pattern, instances of controllers are already scheduled and running inside the cluster before they are required. These controller instances compete to use a shared resource, known as the leader election lock. The instance of the controller that has access to the leader election lock resource at any given time is referred to as the leader.

4.12.1. Configuring high availability replicas on OpenShift Serverless
High availability (HA) functionality is available by default on OpenShift Serverless for Knative Serving, Knative Eventing, and Knative Kafka. These are the components scaled for each of them:

- Knative Serving: activator, autoscaler, autoscaler-hpa, controller, webhook, kourier-control, kourier-gateway.
- Knative Eventing: eventing-controller, eventing-webhook, imc-controller, imc-dispatcher, mt-broker-controller, sugar-controller.

These components are configured with two replicas by default.

For Knative Eventing, the mt-broker-filter and mt-broker-ingress deployments are not scaled by HA. If multiple deployments are needed, scale these components manually.

You modify the number of replicas that are created per component by changing the configuration of spec.high-availability.replicas in the KnativeServing custom resource (CR), the KnativeEventing CR, or the KnativeKafka CR.

4.12.1.1. Configuring high availability replicas for Serving

You can scale Knative Serving components by modifying the spec.high-availability.replicas value in the KnativeServing custom resource.

Prerequisites

- You have access to an OpenShift Container Platform cluster with cluster administrator permissions.
- The OpenShift Serverless Operator and Knative Serving are installed on your cluster.

Procedure

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

2. Select the knative-serving namespace.

3. Click Knative Serving in the list of Provided APIs for the OpenShift Serverless Operator to go to the Knative Serving tab.
4. Click knative-serving, then go to the YAML tab in the knative-serving page.

5. Modify the number of replicas in the KnativeServing CRD:

Example YAML

```yaml
apiVersion: operator.knative.dev/v1alpha1
kind: KnativeServing
metadata:
  name: knative-serving
  namespace: knative-serving
```
Sets the number of replicas to 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 **replicas** value sets the replica count for all HA controllers.
- The default **replicas** value is 2.
- You can increase the number of replicas by changing the value to 3 or more.

### 4.12.1.2. Configuring high availability replicas for Eventing

You can scale Knative Eventing components by modifying the `spec.high-availability.replicas` value in the KnativeEventing custom resource.

**Prerequisites**

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

**Procedure**

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

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

3. Click **Knative Eventing** in the list of **Provided APIs** for the OpenShift Serverless Operator to go to the **Knative Eventing** tab.
4. Click `knative-eventing`, then go to the **YAML** tab in the **knative-eventing** page.

5. Modify the number of replicas in the **KnativeEventing** CRD:

   **Example YAML**

   ```yaml
   apiVersion: operator.knative.dev/v1alpha1
   kind: KnativeEventing
   metadata:
     name: knative-eventing
     namespace: knative-eventing
   ```
Sets the number of replicas to 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 `replicas` value sets the replica count for all HA controllers.
- The default `replicas` value is 2.
- You can increase the number of replicas by changing the value to 3 or more.

### 4.12.1.3. Configuring high availability replicas for Kafka

You can scale Knative Kafka components by modifying the `spec.high-availability.replicas` value in the `KnativeKafka` custom resource.

**Prerequisites**

- You have access to an OpenShift Container Platform cluster with cluster administrator permissions.
- The OpenShift Serverless Operator and Knative Kafka are installed on your cluster.

**Procedure**

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

```yaml
spec:
  high-availability:
    replicas: 3
```

1. Sets the number of replicas to 3.
2. Select the **knative-eventing** namespace.

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

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

5. Modify the number of replicas in the **KnativeKafka** CRD:

   **Example YAML**

   ```yaml
   apiVersion: operator.serverless.openshift.io/v1alpha1
   kind: KnativeKafka
   ```
Sets the number of replicas to 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 `replicas` value sets the replica count for all HA controllers.
- The default `replicas` value is 2.
- You can increase the number of replicas by changing the value to 3 or more.
CHAPTER 5. KNATIVE SERVING

5.1. UNDERSTANDING KNATIVE SERVING

Knative Serving on OpenShift Container Platform enables developers to write cloud-native applications using serverless architecture.

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.

5.1.1. 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.

5.2. SERVERLESS APPLICATIONS

5.2.1. 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

```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
event:
  - 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.

5.2.2. 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.

5.2.2.1. 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.

5.2.2.2. Creating serverless applications by using the Knative 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 the `kn` CLI.

**Procedure**

- Create a Knative service:

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

  **Example command**
5.2.2.3. 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:

   ```yaml
   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:

   ```bash
   $ 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.

5.2.3. Updating serverless applications by using the Knative CLI

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

- Update a service by adding a new port:
  ```bash
  $ kn service update <service_name> --port 80
  ```

- Update a service by adding new request and limit parameters:
  ```bash
  $ kn service update <service_name> --request cpu=500m --limit memory=1024Mi --limit cpu=1000m
  ```

- Assign the **latest** tag to a revision:
  ```bash
  $ kn service update <service_name> --tag <revision_name>=latest
  ```

- Update a tag from **testing** to **staging** for the latest READY revision of a service:
  ```bash
  $ 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:
  ```bash
  $ kn service update <service_name> --tag <revision_name>=test --traffic test=10,@latest=90
  ```

5.2.4. 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:
  ```bash
  $ kn service apply <service_name> --image <image>
  ```

- Add an environment variable to a service:
  ```none
  ```
$ 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>

5.2.5. Describing serverless applications by using the Knative CLI

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**

<table>
<thead>
<tr>
<th>Name:</th>
<th>hello</th>
</tr>
</thead>
<tbody>
<tr>
<td>Namespace:</td>
<td>default</td>
</tr>
<tr>
<td>Age:</td>
<td>2m</td>
</tr>
<tr>
<td>URL:</td>
<td><a href="http://hello-default.apps.ocp.example.com">http://hello-default.apps.ocp.example.com</a></td>
</tr>
</tbody>
</table>

Revisions:

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

Conditions:

<table>
<thead>
<tr>
<th>OK</th>
<th>TYPE</th>
<th>AGE</th>
<th>REASON</th>
</tr>
</thead>
<tbody>
<tr>
<td>++</td>
<td>Ready</td>
<td>1m</td>
<td></td>
</tr>
<tr>
<td>++</td>
<td>ConfigurationsReady</td>
<td>1m</td>
<td></td>
</tr>
<tr>
<td>++</td>
<td>RoutesReady</td>
<td>1m</td>
<td></td>
</tr>
</tbody>
</table>

**Example output with `--verbose` flag**

<table>
<thead>
<tr>
<th>Name:</th>
<th>hello</th>
</tr>
</thead>
<tbody>
<tr>
<td>Namespace:</td>
<td>default</td>
</tr>
<tr>
<td>Annotations:</td>
<td>serving.knative.dev/creator=system:admin</td>
</tr>
<tr>
<td></td>
<td>serving.knative.dev/lastModifier=system:admin</td>
</tr>
<tr>
<td>Age:</td>
<td>3m</td>
</tr>
<tr>
<td>URL:</td>
<td><a href="http://hello-default.apps.ocp.example.com">http://hello-default.apps.ocp.example.com</a></td>
</tr>
<tr>
<td>Cluster:</td>
<td><a href="http://hello.default.svc.cluster.local">http://hello.default.svc.cluster.local</a></td>
</tr>
</tbody>
</table>

Revisions:

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

Conditions:

<table>
<thead>
<tr>
<th>OK</th>
<th>TYPE</th>
<th>AGE</th>
<th>REASON</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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

5.2.6. 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**

<table>
<thead>
<tr>
<th>NAME</th>
<th>URL</th>
<th>LATESTCREATED</th>
<th>LATESTREADY</th>
</tr>
</thead>
<tbody>
<tr>
<td>READY REASON</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>event-delivery</td>
<td><a href="http://event-delivery-default.example.com">http://event-delivery-default.example.com</a></td>
<td>event-delivery-4wsd2</td>
<td>event-delivery-4wsd2</td>
</tr>
</tbody>
</table>

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**
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!

### 5.2.7. 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**

<table>
<thead>
<tr>
<th>NAME</th>
<th>TYPE</th>
<th>CLUSTER-IP</th>
<th>EXTERNAL-IP</th>
</tr>
</thead>
<tbody>
<tr>
<td>PORT(S)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The public address is surfaced in the `EXTERNAL-IP` field, and in this case is `a83e86291bcd11e993af02b7a65e514-33544245.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.

   ```bash
   $ curl -H "Host: hello-default.example.com" a83e86291bcd11e993af02b7a65e514-33544245.us-east-1.elb.amazonaws.com
   Hello Serverless!
   
   Example output
   
   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:
   
   ```golang
   grpc.Dial(
      "a83e86291bcd11e993af02b7a65e514-33544245.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.

5.2.8. Enabling communication with Knative applications on a cluster with restrictive network policies

If you are using a cluster that multiple users have access to, your cluster might use network policies to control which pods, services, and namespaces can communicate with each other over the network.

If your cluster uses restrictive network policies, it is possible that Knative system pods are not able to access your Knative application. For example, if your namespace has the following network policy, which denies all requests, Knative system pods cannot access your Knative application:

**Example NetworkPolicy object that denies all requests to the namespace**

```yaml
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: deny-by-default
  namespace: example-namespace
spec:
  podSelector: {}
  ingress: []
```
To allow access to your applications from Knative system pods, you must add a label to each of the Knative system namespaces, and then create a **NetworkPolicy** object in your application namespace that allows access to the namespace for other namespaces that have this label.

**IMPORTANT**

A network policy that denies requests to non-Knative services on your cluster still prevents access to these services. However, by allowing access from Knative system namespaces to your Knative application, you are allowing access to your Knative application from all namespaces in the cluster.

If you do not want to allow access to your Knative application from all namespaces on the cluster, you might want to use **JSON Web Token authentication for Knative services** instead (see the *Knative Serving* documentation). JSON Web Token authentication for Knative services requires Service Mesh.

**Procedure**

1. Add the `knative.openshift.io/system-namespace=true` label to each Knative system namespace that requires access to your application:
   
   a. Label the `knative-serving` namespace:
      
      $ oc label namespace knative-serving knative.openshift.io/system-namespace=true
   b. Label the `knative-serving-ingress` namespace:
      
      $ oc label namespace knative-serving-ingress knative.openshift.io/system-namespace=true
   c. Label the `knative-eventing` namespace:
      
      $ oc label namespace knative-eventing knative.openshift.io/system-namespace=true
   d. Label the `knative-kafka` namespace:
      
      $ oc label namespace knative-kafka knative.openshift.io/system-namespace=true

2. Create a **NetworkPolicy** object in your application namespace to allow access from namespaces with the `knative.openshift.io/system-namespace` label:

**Example NetworkPolicy object**

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: <network_policy_name>  
  namespace: <namespace>  
spec:
  ingress:
    - from:
      - namespaceSelector:
        matchLabels:
          knative.openshift.io/system-namespace: "true"
```

---

OpenShift Container Platform 4.8 Serverless

78
5.2.9. Using kn CLI in offline mode

**IMPORTANT**

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

For more information about the support scope of Red Hat Technology Preview features, see [https://access.redhat.com/support/offerings/techpreview/](https://access.redhat.com/support/offerings/techpreview/).

5.2.9.1. About offline mode

Normally, when you execute `kn service` commands, the changes immediately propagate to the cluster. However, as an alternative, you can execute `kn service` commands in offline mode:

1. When you create a service in offline mode, no changes happen on the cluster. Instead, the only thing that happens is the creation of the service descriptor file on your local machine.

2. After the descriptor file is created, you can manually modify it and track it in a version control system.

3. Finally, you can propagate changes to the cluster by using the `kn service create -f`, `kn service apply -f`, or `oc apply -f` commands on the descriptor files.

The offline mode has several uses:

- You can manually modify the descriptor file before using it to make changes on the cluster.
- You can locally track the descriptor file of a service in a version control system. This enables you to reuse the descriptor file in places other than the target cluster, for example in continuous integration (CI) pipelines, development environments, or demos.
- You can examine the created descriptor files to learn about Knative services. In particular, you can see how the resulting service is influenced by the different arguments passed to the `kn` command.

The offline mode has its advantages: it is fast, and does not require a connection to the cluster. However, offline mode lacks server-side validation. Consequently, you cannot, for example, verify that the service name is unique or that the specified image can be pulled.

5.2.9.2. Creating a service using offline mode

**Prerequisites**
Procedure

1. In offline mode, create a local Knative service descriptor file:

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

   Example output

   Service 'event-display' created in namespace 'test'.

   - The `--target ./` flag enables offline mode and specifies `./` as the directory for storing the new directory tree.
     If you do not specify an existing directory, but use a filename, such as `--target my-service.yaml`, then no directory tree is created. Instead, only the service descriptor file `my-service.yaml` is created in the current directory.

     The filename can have the `.yaml`, `.yml`, or `.json` extension. Choosing `.json` creates the service descriptor file in the JSON format.

   - The `--namespace test` option places the new service in the `test` namespace.
     If you do not use `--namespace`, and you are logged in to an OpenShift Container Platform cluster, the descriptor file is created in the current namespace. Otherwise, the descriptor file is created in the `default` namespace.

2. Examine the created directory structure:

   ```
   $ tree ./
   ./
   └── test
       └── ksvc
           └── event-display.yaml
   ```

   Example output

   - The current `./` directory specified with `--target` contains the new `test/` directory that is named after the specified namespace.

   - The `test/` directory contains the `ksvc` directory, named after the resource type.

   - The `ksvc` directory contains the descriptor file `event-display.yaml`, named according to the specified service name.

3. Examine the generated service descriptor file:
$ cat test/ksvc/event-display.yaml

Example output

```yaml
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  creationTimestamp: null
name: event-display
namespace: test
spec:
template:
  metadata:
    annotations:
      client.knative.dev/user-image: quay.io/openshift-knative/knative-eventing-sources-event-display:latest
        creationTimestamp: null
    spec:
      containers:  
        - image: quay.io/openshift-knative/knative-eventing-sources-event-display:latest
          name: ""
          resources: {}
    status: {}  
```

4. List information about the new service:

$ kn service describe event-display --target ./ --namespace test

Example output

Name:     event-display
Namespace: test
Age:      
URL:      

Revisions: 

Conditions:  
    OK TYPE   AGE REASON

- The **--target** ./ option specifies the root directory for the directory structure containing namespace subdirectories. Alternatively, you can directly specify a YAML or JSON filename with the **--target** option. The accepted file extensions are `.yaml`, `.yml`, and `.json`.

- The **--namespace** option specifies the namespace, which communicates to **kn** the subdirectory that contains the necessary service descriptor file. If you do not use **--namespace**, and you are logged in to an OpenShift Container Platform cluster, **kn** searches for the service in the subdirectory that is named after the current namespace. Otherwise, **kn** searches in the default/ subdirectory.

5. Use the service descriptor file to create the service on the cluster:

$ kn service create -f test/ksvc/event-display.yaml
5.3. 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.

5.3.1. Autoscaling workflows by using the Knative 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.

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--concurrency-limit</td>
<td>Sets a hard limit of concurrent requests to be processed by a single revision.</td>
</tr>
<tr>
<td>--concurrency-target</td>
<td>Provides a recommendation for when to scale up revisions, based on the concurrent number of incoming requests. Defaults to <code>--concurrency-limit</code>.</td>
</tr>
<tr>
<td>--max-scale</td>
<td>Maximum number of revisions.</td>
</tr>
<tr>
<td>--min-scale</td>
<td>Minimum number of revisions.</td>
</tr>
</tbody>
</table>

5.3.2. Configuring concurrent requests for Knative Serving autoscaling

Example output

Creating service 'event-display' in namespace 'test':

0.058s The Route is still working to reflect the latest desired specification.
0.098s ...
0.168s Configuration "event-display" is waiting for a Revision to become ready.
23.377s ...
23.419s Ingress has not yet been reconciled.
23.534s Waiting for load balancer to be ready
23.723s Ready to serve.

Service 'event-display' created to latest revision 'event-display-00001' is available at URL: http://event-display-test.apps.example.com
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**

```yaml
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**

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

**5.3.2.1. 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
```

### 5.3.2.2. 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`.

### 5.3.3. 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 configuration 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.

5.4. USING OPENSOURCES LOGGING

5.4.1. 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 Red Hat OpenShift Logging Operator. When the operators are installed, you can 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 Red Hat OpenShift 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.

5.4.2. 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 an 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.

5.4.2.1. 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:

```yaml
spec:
  logStore:
```

**Note:**

This annotation applies 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.
Elasticsearch storage

You can configure a persistent storage class and size for the Elasticsearch cluster using the `storageClassName` and `size` parameters. The Red Hat OpenShift Logging Operator creates a persistent volume claim (PVC) for each data node in the Elasticsearch cluster based on these parameters.

```yaml
elasticsearch:
  resources:
    limits:
      cpu: 16
      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
```

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

```yaml
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.

### 5.4.2.2. 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**

```yaml
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:
```
5.4.3. Using OpenShift Logging to find logs for Knative Serving components

Procedure

1. Get the Kibana route:

   ```bash
   $ 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 Container Platform 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.

5.4.4. 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.

**5.5. 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 get 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.

**5.5.1. 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 5.1. 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.
5.5.2. Managing and splitting traffic by using the Knative CLI

**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 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.

5.5.2.1. 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 RevisionName=Tag
```

**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.

### 5.5.2.2. 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.

5.5.2.3. 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.

5.5.2.4. Traffic splitting flags by using the Knative CLI

The `kn` CLI 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.

<table>
<thead>
<tr>
<th>Flag</th>
<th>Value(s)</th>
<th>Operation</th>
<th>Repetition</th>
</tr>
</thead>
<tbody>
<tr>
<td>--traffic</td>
<td>RevisionName=Percent</td>
<td>Gives Percent traffic to RevisionName</td>
<td>Yes</td>
</tr>
<tr>
<td>--traffic</td>
<td>Tag=Percent</td>
<td>Gives Percent traffic to the revision having Tag</td>
<td>Yes</td>
</tr>
<tr>
<td>--traffic</td>
<td>@latest=Percent</td>
<td>Gives Percent traffic to the latest ready revision</td>
<td>No</td>
</tr>
<tr>
<td>--tag</td>
<td>RevisionName=Tag</td>
<td>Gives Tag to RevisionName</td>
<td>Yes</td>
</tr>
<tr>
<td>--tag</td>
<td>@latest=Tag</td>
<td>Gives Tag to the latest ready revision</td>
<td>No</td>
</tr>
<tr>
<td>--untag</td>
<td>Tag</td>
<td>Removes Tag from revision</td>
<td>Yes</td>
</tr>
</tbody>
</table>
5.6. 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.

5.6.1. 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 the Jaeger Operator.

Procedure

1. Create and apply a Jaeger custom resource (CR) that contains the following:

   **Jaeger CR**

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

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

   **Tracing YAML example**

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

You can access the Jaeger web console to see tracing data, 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-default.apps.example.com      jaeger-query <all> reencrypt None

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

5.7. CONFIGURING JSON WEB TOKEN AUTHENTICATION FOR KNATIVE SERVICES

After the Service Mesh integration with OpenShift Serverless and Kourier has been configured on your cluster, you can enable JSON Web Token (JWT) authentication for your Knative services.

IMPORTANT

You must set the annotation sidecar.istio.io/rewriteAppHTTPProbers: "true" in your Knative service as OpenShift Serverless versions 1.14.0 and higher use an HTTP probe as the readiness probe for Knative services by default.

5.7.1. Enabling sidecar injection for a Knative service

You can add the sidecar.istio.io/inject="true" annotation to a Knative service to enable sidecar injection for that service.
IMPORTANT

Adding sidecar injection to pods in system namespaces, such as knative-serving and knative-serving-ingress, is not supported when Kourier is enabled.

If you require sidecar injection for pods in these namespaces, see the OpenShift Serverless documentation on Integrating Service Mesh with OpenShift Serverless natively.

Procedure

1. Add the sidecar.istio.io/inject="true" annotation to your Service resource:

Example service

```yaml
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: <service_name>
spec:
template:
  metadata:
    annotations:
    sidecar.istio.io/inject: "true" 1
    sidecar.istio.io/rewriteAppHTTPProbers: "true" 2
...
```

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 your Service resource YAML file:

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

5.7.2. Using JSON Web Token authentication with Service Mesh 2.x and OpenShift Serverless

Procedure

1. Create a RequestAuthentication resource in each serverless application namespace that is a member in the ServiceMeshMemberRoll object:

```yaml
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 `RequestAuthentication` resource from system pods for each serverless application namespace that is a member in the `ServiceMeshMemberRoll` object, by creating the following `AuthorizationPolicy` resource:

   ```yaml
   apiVersion: security.istio.io/v1beta1
   kind: AuthorizationPolicy
   metadata:
     name: allowlist-by-paths
     namespace: <namespace>
   spec:
     action: ALLOW
     rules:
       - to:
         - operation:
           paths:
             - /metrics
             - /healthz

   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:

   $ oc apply -f <filename>

5. For each serverless application namespace that is a member in the `ServiceMeshMemberRoll` object, create the following `AuthorizationPolicy` resource:

   ```yaml
   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:

   $ oc apply -f <filename>

Verification

1. If you try to use a `curl` request to get the Knative service URL, it is denied:
Example command

$ 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:

      $ 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:


      The request is now allowed:

      Example output

      Hello OpenShift!

5.7.3. 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 `ServiceMeshMemberRole` object, that only allows requests with valid JSON Web Tokens (JWT):

   IMPORTANT

   The paths `/metrics` and `/healthz` must be included in `excludedPaths` because they are accessed from system pods in the `knative-serving` namespace.

   ```yaml
   apiVersion: authentication.istio.io/v1alpha1
   kind: Policy
   metadata:
     name: default
     namespace: <namespace>
   spec:
     origins:
       - jwt:
         issuer: testing@secure.istio.io
   ```
The path on your application to collect metrics by system pod.

The path on your application to probe by system pod.

2. Apply the **Policy** resource:

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

**Verification**

1. If you try to use a `curl` request to get the Knative service URL, it is denied:

```bash
$ 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:

```bash
$ 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:

```bash
```

**Example output**

```
Hello OpenShift!
```

## 5.8. CONFIGURING A CUSTOM DOMAIN FOR 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 customize the domain for your Knative service by using one of the following methods:

- Configure the service as a private service and create the required Service Mesh resources.
IMPORTANT

This method of configuring custom domains is only supported for clusters that have Kourier enabled. If you want to configure custom domains using only OpenShift Serverless with Service Mesh, without Kourier enabled, use the DomainMapping resources method instead.

- Map a custom domain name that you own to a Knative service by creating a DomainMapping resource for the service. You can also create multiple DomainMapping resources to map multiple domains and subdomains to a single service.

IMPORTANT

You can use DomainMapping resources to map custom domains either with or without Kourier enabled in your cluster, however TLS is not supported in clusters that have both Kourier and domain mapping enabled.

5.8.1. Creating a custom domain mapping

To map a custom domain name to a custom resource (CR), you must create a DomainMapping CR that maps to an Addressable target CR, such as a Knative service or a Knative route.

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 IP address of the OpenShift Container Platform cluster.

Procedure

1. Create a YAML file containing the DomainMapping CR in the same namespace as the target CR you want to map to:

   ```yaml
   apiVersion: serving.knative.dev/v1alpha1
   kind: DomainMapping
   metadata:
     name: <domain_name>  1
     namespace: <namespace> 2
   spec:
     ref:
       name: <target_name> 3
       kind: <target_type> 4
   apiVersion: serving.knative.dev/v1
   ```

1. The custom domain name that you want to map to the target CR.
2. The namespace of both the DomainMapping CR and the target CR.
3. The name of the target CR to map to the custom domain.

4. The type of CR being mapped to the custom domain.

**Example service domain mapping**

```yaml
apiVersion: serving.knative.dev/v1alpha1
kind: DomainMapping
metadata:
  name: example-domain
namespace: default
spec:
  ref:
    name: example-service
    kind: Service
    apiVersion: serving.knative.dev/v1
```

**Example route domain mapping**

```yaml
apiVersion: serving.knative.dev/v1alpha1
kind: DomainMapping
metadata:
  name: example-domain
namespace: default
spec:
  ref:
    name: example-route
    kind: Route
    apiVersion: serving.knative.dev/v1
```

2. Apply the `DomainMapping` CR as a YAML file:

   $ oc apply -f <filename>

### 5.8.2. Creating a custom domain mapping by using the Knative CLI

You can use the `kn` CLI to create a `DomainMapping` custom resource (CR) that maps to an Addressable target CR, such as a Knative service or a Knative route.

The `--ref` flag specifies an Addressable target CR for domain mapping.

If a prefix is not provided when using the `--ref` flag, it is assumed that the target is a Knative service in the current namespace. The examples in the following procedure show the prefixes for mapping to a Knative service or a Knative route.

**Prerequisites**

- The OpenShift Serverless Operator and Knative Serving are installed on your cluster.
- You have created a Knative service or route, and control a custom domain that you want to map to that CR.
NOTE

Your custom domain must point to the DNS of the OpenShift Container Platform cluster.

- You have installed the `kn` CLI tool.

Procedure

- Map a domain to a CR in the current namespace:
  
  ```
  $ kn domain create <domain_mapping_name> --ref <target_name>
  ```

  **Example command**
  
  ```
  $ kn domain create example-domain-map --ref example-service
  ```

- Map a domain to a Knative service in a specified namespace:
  
  ```
  $ kn domain create <domain_mapping_name> --ref <ksvc:service_name:service_namespace>
  ```

  **Example command**
  
  ```
  $ kn domain create example-domain-map --ref ksvc:example-service:example-namespace
  ```

- Map a domain to a Knative route:
  
  ```
  $ kn domain create <domain_mapping_name> --ref <kroute:route_name>
  ```

  **Example command**
  
  ```
  $ kn domain create example-domain-map --ref kroute:example-route
  ```

5.8.3. Configuring custom domains for private Knative services

You can configure a custom domain for an existing Knative service by completing the following procedures.

Prerequisites

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

- Red Hat OpenShift Service Mesh version 1.x or 2.x is installed on your cluster, and the integration between Red Hat OpenShift Service Mesh and OpenShift Serverless has been correctly configured.
Chapter 5: Knative Serving

5.8.3.1. 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 `networking.knative.dev/visibility=cluster-local` label to make them private.

**IMPORTANT**

For OpenShift Serverless 1.15.0 and newer versions, the `serving.knative.dev/visibility` label is no longer available. You must update existing services to use the `networking.knative.dev/visibility` label instead.

**Procedure**

- Set the visibility for your service by adding the `networking.knative.dev/visibility=cluster-local` label:

  ```bash
  $ oc label ksvc <service_name> networking.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:

  ```bash
  $ oc get ksvc
  ```

**Example output**

<table>
<thead>
<tr>
<th>NAME</th>
<th>URL</th>
<th>LATESTCREATED</th>
</tr>
</thead>
<tbody>
<tr>
<td>hello</td>
<td><a href="http://hello.default.svc.cluster.local">http://hello.default.svc.cluster.local</a></td>
<td>hello-659883f534</td>
</tr>
</tbody>
</table>

5.8.3.2. Creating necessary Service Mesh resources

To use custom domains with Service Mesh, you must first create some necessary Service Mesh resources.

**Procedure**

1. Create an Istio gateway to accept traffic:

   ```bash
   apiVersion: networking.istio.io/v1alpha3
   ```
The namespace where your Knative services have been created. This namespace must be a member of the Service Mesh member roll.

2. Apply the **Gateway** resource:

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

3. Create an Istio virtual service to rewrite the host header:

```yaml
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
route:
  - destination:
    host: hello.default.svc
    port:
      number: 80
```

Your Knative service in the format `<service_name>.<namespace>.svc`.

4. Apply the **VirtualService** resource:

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

5. Create an Istio service entry. This is required for OpenShift Serverless because Kourier is outside of the Service Mesh:

```yaml
apiVersion: networking.istio.io/v1alpha3
kind: ServiceEntry
metadata:
```
5.8.3.3. 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.
### 5.8.4. Additional resources

- For more information about Red Hat OpenShift Service Mesh, see [Understanding Red Hat OpenShift Service Mesh](#).

### 5.9. CONFIGURING TRANSPORT LAYER SECURITY FOR A CUSTOM DOMAIN USING RED HAT OPENSHIFT SERVICE MESH AND KOURIER

You can create a Transport Layer Security (TLS) key and certificates for a custom domain and subdomain using Red Hat OpenShift Service Mesh.

**IMPORTANT**

OpenShift Serverless only supports the use of Red Hat OpenShift Service Mesh functionality that is explicitly documented in this guide, and does not support other undocumented features.

#### 5.9.1. Prerequisites

- Install [OpenShift Serverless](#).

- Install Red Hat OpenShift Service Mesh 1.x or 2.x.

**IMPORTANT**

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.

- Complete the configuration steps in [Integrating Service Mesh and OpenShift Serverless with Kourier enabled](#).

- Configure a custom domain. See [Configuring a custom domain for a Knative service](#).

- 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.
5.9.2. 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
test.example.com.crt
test.example.com.key
```

5. Create a secret:

```
$ oc create -n istio-system secret tls custom.example.com \n--key=custom.example.com.key \n--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:

```yaml
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:

```yaml
- 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:

```yaml
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:

```bash
$ 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:

```bash
$ oc get svc -n openshift-ingress router-default
   -o jsonpath='{.status.loadBalancer.ingress[0].ip}
```

**Example output**

```
Hello OpenShift!
```

### 5.9.3. 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:

   ```bash
   $ 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:

   ```bash
   $ 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:

   ```bash
   $ 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
e.example.com.crt
e.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!
```

### 5.10. 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.

5.10.1. Customizing labels and annotations for OpenShift Container Platform routes

OpenShift Container Platform routes support the use of custom labels and annotations, which you can configure by modifying the metadata spec of a Knative service. Custom labels and annotations are propagated from the service to the Knative route, then to the Knative ingress, and finally to the OpenShift Container Platform route.

Prerequisites

- You must have the OpenShift Serverless Operator and Knative Serving installed on your OpenShift Container Platform cluster.

Procedure

1. Create a Knative service that contains the label or annotation that you want to propagate to the OpenShift Container Platform route:
   - To create a service by using YAML:

   **Example service created by using YAML**

   ```yaml
   apiVersion: serving.knative.dev/v1
   kind: Service
   metadata:
     name: <service_name>
     labels:
       <label_name>: <label_value>
     annotations:
       <annotation_name>: <annotation_value>
   ...
   ```

   - To create a service by using the `kn` CLI, enter:

   **Example service created by using a `kn` command**

   ```bash
   $ kn service create <service_name> \
   --image=<image> \
   --annotation <annotation_name>=<annotation_value> \
   --label <label_value>=<label_value>
   ```

2. Verify that the OpenShift Container Platform route has been created with the annotation or label that you added by inspecting the output from the following command:

   **Example command for verification**

   ```
5.10.2. 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 for the service.

**NOTE**

When you complete the following procedure, the default OpenShift Container Platform route in the `knative-serving-ingress` namespace is not created. However, the Knative route for the application is still created in this namespace.

**Prerequisites**

- The OpenShift Serverless Operator and Knative Serving component must be installed on your OpenShift Container Platform cluster.

**NOTE**

You must modify the replaceable values in the example commands for the following procedure.

**Procedure**

1. Create a Knative service that includes the `serving.knative.openshift.io/disableRoute=true` annotation:

   a. Optional. Create a Knative service resource:

   ```bash
   $ oc get routes.route.openshift.io \
   -l serving.knative.openshift.io/ingressName=<service_name> \
   -l serving.knative.openshift.io/ingressNamespace=<service_namespace> \
   -n knative-serving-ingress \
   | grep -e "<label_name>: "<label_value>" -e "<annotation_name>: <annotation_value>"
   ```

   1. Use the name of your service.
   2. Use the namespace where your service was created.
   3. Use your values for the label and annotation names and values.

   **Example resource**

   ```yaml
   apiVersion: serving.knative.dev/v1
   kind: Service
   metadata:
     name: <service_name>
   annotations:
     serving.knative.openshift.io/disableRoute: true
   spec:
     template:
   ```
b. Apply the `Service` resource:

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

c. Optional. Create a Knative service by using the `kn service create` command:

**Example kn command**

```bash
$ kn service create <service_name> \ 
   --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**

```bash
$ $ 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 will see the following output:

```
No resources found in knative-serving-ingress namespace.
```

3. Create a `Route` resource in the `knative-serving-ingress` namespace:

```yaml
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-----
```
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).

The name of the OpenShift Container Platform route.

The namespace for the OpenShift Container Platform route. This must be `knative-serving-ingress`.

The hostname for external access. You can set this to `<service_name>-<service_namespace>.<domain>`.

The certificates you want to use. Currently, only edge termination is supported.

4. Apply the **Route** resource:

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

5.10.3. 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 `networking.knative.dev/visibility=cluster-local` label to make them private.

**IMPORTANT**

For OpenShift Serverless 1.15.0 and newer versions, the `serving.knative.dev/visibility` label is no longer available. You must update existing services to use the `networking.knative.dev/visibility` label instead.

**Procedure**

- Set the visibility for your service by adding the `networking.knative.dev/visibility=cluster-local` label:

```
$ oc label ksvc <service_name> networking.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:

```bash
$ oc get ksvc
```

**Example output**

<table>
<thead>
<tr>
<th>NAME</th>
<th>URL</th>
<th>LATESTCREATED</th>
</tr>
</thead>
<tbody>
<tr>
<td>hello</td>
<td><a href="http://hello.default.svc.cluster.local">http://hello.default.svc.cluster.local</a></td>
<td>hello-tx2g7</td>
</tr>
<tr>
<td>tx2g7</td>
<td>True</td>
<td>hello-tx2g7</td>
</tr>
</tbody>
</table>

### 5.10.4. Additional resources

- For more information about supported OpenShift Container Platform route annotations, see [Route-specific annotations](#).

### 5.11. METRICS

Metrics enable developers to monitor how Knative services are performing.

#### 5.11.1. Prerequisites

- To view metrics for Knative components on OpenShift Container Platform, you need access to the web console **Developer** perspective.

**WARNING**

If Service Mesh is enabled with mTLS, metrics for Knative Serving are disabled by default because Service Mesh prevents Prometheus from scraping metrics.

For information about resolving this issue, see the [Serverless 1.16.0 release notes](#).

#### 5.11.2. Queue proxy metrics

Each Knative service has a proxy container that proxies the connections to the application container. A number of metrics are reported for the queue proxy performance.

You can use the following metrics to measure if requests are queued at the proxy side and the actual delay in serving requests at the application side.

<table>
<thead>
<tr>
<th>Metric name</th>
<th>Description</th>
<th>Type</th>
<th>Tags</th>
<th>Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Metric name</td>
<td>Description</td>
<td>Type</td>
<td>Tags</td>
<td>Unit</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>------------------------------------------------------------------</td>
<td>--------------</td>
<td>----------------------------------------------------------------------</td>
<td>-----------------</td>
</tr>
<tr>
<td>revision_request_count</td>
<td>The number of requests that are routed to <code>queue-proxy</code> pod.</td>
<td>Counter</td>
<td>configuration_name, container_name, namespace_name, pod_name, response_code, response_code_class, revision_name, service_name</td>
<td>Integer (no units)</td>
</tr>
<tr>
<td>revision_request_latencies</td>
<td>The response time of revision requests.</td>
<td>Histogram</td>
<td>configuration_name, container_name, namespace_name, pod_name, response_code, response_code_class, revision_name, service_name</td>
<td>Milliseconds</td>
</tr>
<tr>
<td>revision_app_request_count</td>
<td>The number of requests that are routed to the <code>user-container</code> pod.</td>
<td>Counter</td>
<td>configuration_name, container_name, namespace_name, pod_name, response_code, response_code_class, revision_name, service_name</td>
<td>Integer (no units)</td>
</tr>
<tr>
<td>revision_app_request_latencies</td>
<td>The response time of revision app requests.</td>
<td>Histogram</td>
<td>configuration_name, namespace_name, pod_name, response_code, response_code_class, revision_name, service_name</td>
<td>Milliseconds</td>
</tr>
<tr>
<td>Metric name</td>
<td>Description</td>
<td>Type</td>
<td>Tags</td>
<td>Unit</td>
</tr>
<tr>
<td>--------------------</td>
<td>------------------------------------------------------------------------------</td>
<td>--------</td>
<td>----------------------------------------------------------------------</td>
<td>---------------------------</td>
</tr>
<tr>
<td>revision_queue_depth</td>
<td>The current number of items in the serving and waiting queues. This metric is not reported if unlimited concurrency is configured.</td>
<td>Gauge</td>
<td>configuration_name, event-display, container_name, namespace_name, pod_name, response_code_class, revision_name, service_name</td>
<td>Integer (no units)</td>
</tr>
</tbody>
</table>
CHAPTER 6. KNATIVE EVENTING

6.1. UNDERSTANDING KNATIVE EVENTING

Knative Eventing on OpenShift Container Platform enables developers to use an event-driven architecture with serverless applications.

6.1.1. Event-driven architecture

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 events.

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

6.1.2. Knative Eventing use cases

Knative Eventing supports the following use cases:

**Publish an event without creating a consumer**
You can send events to a broker as an HTTP POST, and use binding to decouple the destination configuration from your application that produces events.

**Consume an event without creating a publisher**
You can use a trigger to consume events from a broker based on event attributes. The application receives events as an HTTP POST.

6.1.3. Knative Eventing custom resources (CRs)

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

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

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

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

- channels and subscriptions, or
- brokers and triggers.

6.2. EVENT SINKS

A sink is an Addressable custom resource (CR) that can receive incoming events from other resources. Knative services, channels, and brokers are all examples of sinks.
TIP

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

6.2.1. Knative CLI --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

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

6.2.2. Connect an event source to a sink using the Developer perspective

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

Prerequisites

To connect an event source to a sink 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 created a sink.
- You have logged in to the web console and are in the Developer perspective.
- You have created a project or have access to a project with the appropriate roles and permissions to create applications and other workloads in OpenShift Container Platform.

Procedure

1. Create an event source of any type, by navigating to Add Event Sources and then selecting the event source type that you want to create.

2. In the Sink section of the Event Sources form view, select Resource. Then use the drop-down list to select your sink.

3. Click Create.

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

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

### 6.2.3. Connecting a trigger to a sink

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

**Example of a trigger connected to a Kafka sink**

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

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

### 6.3. BROKERS

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

![Diagram of brokers and triggers](image)

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.
6.3.1. 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.

6.3.1.1. Creating a broker by using the Knative CLI

**Prerequisites**

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

<table>
<thead>
<tr>
<th>NAME</th>
<th>URL</th>
<th>AGE</th>
<th>CONDITIONS</th>
<th>READY</th>
</tr>
</thead>
<tbody>
<tr>
<td>default</td>
<td><a href="http://broker-ingress.knative-eventing.svc.cluster.local/test/default">http://broker-ingress.knative-eventing.svc.cluster.local/test/default</a></td>
<td>45s</td>
<td>5 OK / 5</td>
<td>True</td>
</tr>
</tbody>
</table>

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:

6.3.1.2. Creating a broker by annotating a trigger
You can create a broker by adding the \texttt{eventing.knative.dev/injection: enabled} annotation to a \texttt{Trigger} object.

**IMPORTANT**

If you create a broker by using the \texttt{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.
- You have installed the \texttt{oc} CLI.

**Procedure**

1. Create a \texttt{Trigger} object as a YAML file that has the \texttt{eventing.knative.dev/injection: enabled} annotation:

   ```yaml
   apiVersion: eventing.knative.dev/v1
   kind: Trigger
   metadata:
     annotations:
       eventing.knative.dev/injection: enabled
       name: <trigger_name>
   spec:
     broker: default
     subscriber:
       ref:
         apiVersion: serving.knative.dev/v1
         kind: Service
         name: <service_name>
   ```

   Specify details about the event sink, or \texttt{subscriber}, that the trigger sends events to.

2. Apply the \texttt{Trigger} YAML file:

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

**Verification**

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

1. Enter the following \texttt{oc} command to get the broker:

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

**Example output**
Navigate to the Topology view in the web console, and observe that the broker exists:

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

![Topology view](image)

### 6.3.1.3. 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`:

  ```bash
  $ oc label namespace <namespace> eventing.knative.dev/injection=enabled
  $ oc -n <namespace> get broker <broker_name>
  $ oc -n default get broker default
  ```

**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:

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

**Example command**

```bash
$ oc -n default get broker default
```
Example output

<table>
<thead>
<tr>
<th>NAME</th>
<th>READY</th>
<th>REASON</th>
<th>URL</th>
</tr>
</thead>
<tbody>
<tr>
<td>default</td>
<td>True</td>
<td></td>
<td><a href="http://broker-ingress.knative-eventing.svc.cluster.local/test/default">http://broker-ingress.knative-eventing.svc.cluster.local/test/default</a></td>
</tr>
<tr>
<td>3m56s</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

![Topology View](image)

6.3.1.4. 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:

   ```bash
   $ 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:

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

**Verification**

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

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

**Example command**

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

**Example output**

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

The \texttt{kn} CLI provides commands that can be used to list, describe, update, and delete brokers.

6.3.2.1. Listing existing brokers by 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 \texttt{kn} CLI.

Procedure

- List all existing brokers:

  $ \texttt{kn broker list}

Example output

<table>
<thead>
<tr>
<th>NAME</th>
<th>URL</th>
<th>AGE</th>
<th>CONDITIONS</th>
<th>READY</th>
</tr>
</thead>
<tbody>
<tr>
<td>default</td>
<td><a href="http://broker-ingress.knative-eventing.svc.cluster.local/test/default">http://broker-ingress.knative-eventing.svc.cluster.local/test/default</a></td>
<td>45s</td>
<td>5 OK / 5</td>
<td>True</td>
</tr>
</tbody>
</table>

6.3.2.2. Describing an existing broker by 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 \texttt{kn} CLI.

Procedure

- Describe an existing broker:

  $ \texttt{kn broker describe <broker_name>}

Example command using default broker

$ \texttt{kn broker describe default}

Example output

Name:     default
Namespace: default
Annotations: eventing.knative.dev/broker.class=MTChannelBasedBroker, eventing.knative.dev/created ...
Age:     22s
6.4. FILTERING EVENTS FROM A BROKER BY USING TRIGGERS

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

6.4.1. 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.

6.4.2. 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:

6.4.3. 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.

6.4.4. Creating a trigger by using the Knative CLI

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

Prerequisites

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

Procedure

- Create a trigger:
  
  ```bash
  $ 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.

### 6.4.5. Listing triggers by using the Knative CLI

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

**Prerequisites**

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

**Procedure**

1. Print a list of available triggers:

   ```
   $ kn trigger list
   ```

   **Example output**

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

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

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

### 6.4.6. Describing a trigger by using the Knative CLI

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

**Prerequisites**

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

**Procedure**

- Enter the command:

  ```
  $ kn trigger describe <trigger_name>
  ```
### 6.4.7. Filtering events with triggers by using the Knative CLI

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

```bash
$ 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.

```bash
$ 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
```

### 6.4.8. Updating a trigger by using the Knative CLI

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

**Prerequisites**

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

**Procedure**

---

**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:  
<table>
<thead>
<tr>
<th>OK TYPE</th>
<th>AGE</th>
<th>REASON</th>
</tr>
</thead>
<tbody>
<tr>
<td>++ Ready</td>
<td>2m</td>
<td></td>
</tr>
<tr>
<td>++ BrokerReady</td>
<td>2m</td>
<td></td>
</tr>
<tr>
<td>++ DependencyReady</td>
<td>2m</td>
<td></td>
</tr>
<tr>
<td>++ Subscribed</td>
<td>2m</td>
<td></td>
</tr>
<tr>
<td>++ SubscriberResolved</td>
<td>2m</td>
<td></td>
</tr>
</tbody>
</table>
Update a trigger:

- 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
  ```

### 6.4.9. Deleting a trigger by using the Knative CLI

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

**Prerequisites**

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

**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.
  ```

### 6.5. 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.

### 6.5.1. 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.

#### 6.5.1.1. 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.

#### 6.5.1.2. 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.

### 6.5.2. 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>`. 
6.5.3. 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

```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.

6.5.4. Additional resources

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

6.6. KNATIVE KAFKA
IMPORTANT

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

For more information about the support scope of Red Hat Technology Preview features, see https://access.redhat.com/support/offerings/techpreview/.

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**

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

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.

### 6.6.1. 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 Bootstrap 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.
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**

<table>
<thead>
<tr>
<th>NAME</th>
<th>READY</th>
<th>STATUS</th>
<th>RESTARTS</th>
<th>AGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>kafka-ch-controller-85f879d577-xcbjh</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>44s</td>
</tr>
<tr>
<td>kafka-ch-dispatcher-55d76d7db8-ggqjl</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>44s</td>
</tr>
<tr>
<td>kafka-controller-manager-bc994c465-pt7qd</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>40s</td>
</tr>
<tr>
<td>kafka-webhook-54646f474f-wr7bb</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>42s</td>
</tr>
</tbody>
</table>

### 6.6.2. Using Kafka channels

Create a [Kafka channel](#).

### 6.6.3. Using Kafka source

Create a [Kafka event source](#).

### 6.6.4. 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.

### 6.6.4.1. 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:

   ```bash
   $ kubectl create secret -n <namespace> generic <kafka_auth_secret> \   --from-file=ca.crt=caroot.pem \   --from-file=user.crt=certificate.pem \   --from-file=user.key=key.pem
   ```

2. Start editing the `KnativeKafka` custom resource:

   ```bash
   $ oc edit knativekafka
   ```

3. Reference your secret and the namespace of the secret:

   ```yaml
   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:

```yaml
apiVersion: operator.serverless.openshift.io/v1alpha1
class: KnativeKafka
metadata:
  name: knative-eventing
spec:
  authSecretName: tls-user
  authSecretNamespace: kafka
  bootstrapServers: eventing-kafka-bootstrap.kafka.svc:9094
  enabled: true
  source:
    enabled: true
```

Additional resources

- TLS and SASL on Kafka

### 6.6.4.2. 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:

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

   ```bash
   $ oc edit knativekafka
   ```
3. Reference your secret and the namespace of the secret:

```yaml
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:

```yaml
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

**6.6.4.3. Configuring SASL authentication using public CA certificates**

If you want to use SASL with public CA certificates, you must use the `tls.enabled=true` flag, rather than the `ca.crt` argument, when creating the secret. For example:

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

**Additional resources**
• TLS and SASL on Kafka
CHAPTER 7. EVENT SOURCES

7.1. UNDERSTANDING EVENT SOURCES

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

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

Currently, OpenShift Serverless supports the following event source types:

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

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

**Sink binding**
- Connects core Kubernetes resource objects, such as Deployment, Job, or StatefulSet objects, with a sink.

**Container source**
- Starts a container image that generates cloud events and sends them to a sink. Container sources can also be used to support your own custom event sources in Knative.

**Kafka source**
- Connects a Kafka cluster to a sink as an event source.

7.2. 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**
- Connects core Kubernetes resource objects, such as Deployment, Job, or StatefulSet objects, with a sink.

**Container source**
- Creates a custom event source by using an image.

**Knative Kafka source**
- Connects a Kafka cluster to a sink as an event source.

7.2.1. Listing available event source types by using the Knative CLI
CHAPTER 7. EVENT SOURCES

Procedure

1. List the available event source types in the terminal:

   $ kn source list-types

Example output

<table>
<thead>
<tr>
<th>TYPE</th>
<th>NAME</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>ApiServerSource</td>
<td>apiserversources.sources.knative.dev</td>
<td>Watch and send Kubernetes API events to a sink</td>
</tr>
<tr>
<td>PingSource</td>
<td>pingsources.sources.knative.dev</td>
<td>Periodically send ping events to a sink</td>
</tr>
<tr>
<td>SinkBinding</td>
<td>sinkbindings.sources.knative.dev</td>
<td>Binding for connecting a PodSpecable to a sink</td>
</tr>
</tbody>
</table>

2. Optional: You can also list the available event source types in YAML format:

   $ kn source list-types -o yaml

7.2.2. 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.

7.2.3. Listing available event sources by using the Knative CLI

- List the available event sources:

  $ kn source list

Example output

<table>
<thead>
<tr>
<th>NAME</th>
<th>TYPE</th>
<th>RESOURCE</th>
<th>SINK</th>
<th>READY</th>
</tr>
</thead>
<tbody>
<tr>
<td>a1</td>
<td>ApiServerSource</td>
<td>apiserversources.sources.knative.dev</td>
<td>ksvc:eshow2</td>
<td>True</td>
</tr>
<tr>
<td>b1</td>
<td>SinkBinding</td>
<td>sinkbindings.sources.knative.dev</td>
<td>ksvc:eshow3</td>
<td>False</td>
</tr>
<tr>
<td>p1</td>
<td>PingSource</td>
<td>pingsources.sources.knative.dev</td>
<td>ksvc:eshow1</td>
<td>True</td>
</tr>
</tbody>
</table>

7.2.3.1. 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:

  ```bash
  $ kn source list --type PingSource
  
  Example output
  
  NAME TYPE RESOURCE SINK READY
  p1 PingSource pingsources.sources.knative.dev ksvc:eshow1 True
  
  7.3. 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.

  7.3.1. 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.

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

  **Procedure**

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

  ```yaml
  apiVersion: v1
  kind: ServiceAccount
  metadata:
    name: events-sa
  namespace: default
  ```
Change this namespace to the namespace that you have selected for installing the event source.

2. Apply the YAML file:

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

### 7.3.3. 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.

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

7.3.4. 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.
7.3.5. Creating an API server source by using 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**

<table>
<thead>
<tr>
<th>Name:</th>
<th>mysource</th>
</tr>
</thead>
<tbody>
<tr>
<td>Namespace:</td>
<td>default</td>
</tr>
<tr>
<td>Annotations:</td>
<td>sources.knative.dev/creator=developer, sources.knative.dev/lastModifier=developer</td>
</tr>
<tr>
<td>Age:</td>
<td>3m</td>
</tr>
<tr>
<td>ServiceAccountName:</td>
<td>events-sa</td>
</tr>
<tr>
<td>Mode:</td>
<td>Resource</td>
</tr>
<tr>
<td>Sink:</td>
<td></td>
</tr>
<tr>
<td>Name:</td>
<td>default</td>
</tr>
<tr>
<td>Namespace:</td>
<td>default</td>
</tr>
<tr>
<td>Kind:</td>
<td>Broker (eventing.knative.dev/v1)</td>
</tr>
<tr>
<td>Resources:</td>
<td></td>
</tr>
<tr>
<td>Kind:</td>
<td>event (v1)</td>
</tr>
<tr>
<td>Controller:</td>
<td>false</td>
</tr>
<tr>
<td>Conditions:</td>
<td></td>
</tr>
<tr>
<td>OK TYPE</td>
<td>AGE REASON</td>
</tr>
<tr>
<td>++ Ready</td>
<td>3m</td>
</tr>
<tr>
<td>++ Deployed</td>
<td>3m</td>
</tr>
<tr>
<td>++ SinkProvided</td>
<td>3m</td>
</tr>
<tr>
<td>++ SufficientPermissions</td>
<td>3m</td>
</tr>
<tr>
<td>++ EventTypesProvided</td>
<td>3m</td>
</tr>
</tbody>
</table>

**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**

```yaml
  cloudevents.Event
  Validation: valid
  Context Attributes,
  specversion: 1.0
  type: dev.knative.apiserver.resource.update
datacontenttype: application/json
...
Data,
{
  "apiVersion": "v1",
```
7.3.5.1. Knative CLI --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 \
   --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.
```

7.3.6. Deleting the API server source by 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 sourceapiserver delete <source_name>

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

   $ oc delete -f authentication.yaml

### 7.3.7. 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. Create a service account, role, and role binding for the API server source as a YAML file:

   ```yaml
   apiVersion: v1
   kind: ServiceAccount
   metadata:
     name: events-sa
     namespace: default
   ---
   apiVersion: rbac.authorization.k8s.io/v1
   kind: Role
   metadata:
     name: event-watcher
     namespace: default
   rules:
     - apiGroups:
       - ""
       resources:
       - events
       verbs:
       - get
       - list
       - watch
   ---
   apiVersion: rbac.authorization.k8s.io/v1
   ```

   **NOTE**

   If you want to re-use an existing service account, you can modify your existing `ServiceAccount` resource to include the required permissions instead of creating a new resource.
1. Change this namespace to the namespace that you have selected for installing the API server source.

2. Apply the YAML file:

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

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

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

4. Apply the `ApiServerSource` YAML file:

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

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

   ```yaml
   apiVersion: serving.knative.dev/v1
   kind: Service
   metadata:
     name: event-display
     namespace: default
   spec:
     template:
   ```
Apply the **Service** YAML file:

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

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

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

Apply the **Trigger** YAML file:

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

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
```

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

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

**Example output**

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

To verify that the Kubernetes events were sent to Knative, you can look at the message dumper function logs.

1. Get the pods by entering the following command:

```bash
$ oc get pods
```

2. View the message dumper function logs for the pods by entering the following command:

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

**Example output**

```json
[...
  "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",
      ...}
  }]
```
7.3.8. 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

7.4. 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

```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**.
7.4.1. 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:

   ```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.

![PingSource and sink](image)

### 7.4.2. Creating a ping source by using 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:

   ```bash
   $ 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:

   ```bash
   $ 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:**

<table>
<thead>
<tr>
<th>OK TYPE</th>
<th>AGE</th>
<th>REASON</th>
</tr>
</thead>
<tbody>
<tr>
<td>++ Ready</td>
<td>8s</td>
<td></td>
</tr>
<tr>
<td>++ Deployed</td>
<td>8s</td>
<td></td>
</tr>
<tr>
<td>++ SinkProvided</td>
<td>15s</td>
<td></td>
</tr>
<tr>
<td>++ ValidSchedule</td>
<td>15s</td>
<td></td>
</tr>
<tr>
<td>++ EventTypeProvided</td>
<td>15s</td>
<td></td>
</tr>
<tr>
<td>++ ResourcesCorrect</td>
<td>15s</td>
<td></td>
</tr>
</tbody>
</table>

**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**

```yaml
clodevents.Event
Validation: valid
Context Attributes,
specversion: 1.0
type: dev.knative.sources.ping
source: /apis/v1/namespaces/default/pingsources/test-ping-source
```
7.4.2.1. Knative CLI --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

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

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

7.4.3. 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:

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

7.4.4. 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
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`:

```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`:

```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**

```yaml
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"
```
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!"
```

7.4.5. 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_yml_filename>

Example command

  $ oc delete -f ping-source.yaml

7.5. 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

7.5.1. Creating a sink binding by using 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:
2. Create a sink binding instance that directs events to the service:

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

3. Create a *CronJob* custom resource (CR).
   a. Create a file named `heartbeats-cronjob.yaml` and copy the following sample code into it:

```
apiVersion: batch/v1
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
```
To use sink binding, you must manually add a `bindings.knative.dev/include=true` label to your Knative CRs.

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

```yaml
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=minikube ...
- Age: 2m
- Subject: job (batch/v1)
- Selector:
  - app: heartbeat-cron
- Sink:
  - Name: event-display
  - Resource: Service (serving.knative.dev/v1)

**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-a433-608fbcddd2596
contenttype: application/json
Extensions,
  beats: true
  heart: yes
the: 42
Data,
{
  "id": 1,
  "label": ""
}
```

7.5.1.1. Knative CLI --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 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 \
   --ce-override "sink=bound"
```

`sink=bound` determines that the sink is a Knative service. Other default sink prefixes include `channel`, and `broker`.

7.5.2. 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`:

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

   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:

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

```yaml
apiVersion: batch/v1
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:

```yaml
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**

```yaml
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
        bindings.knative.dev/include: "true"
```

**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
```
7.6. USING CONTAINER SOURCES

Container sources create a container image that generates events and sends events to a sink. You can use a container source to create a custom event source.

7.6.1. Creating custom event sources by using a container source

You can use a container source to create and manage a container for your custom event source image.

To implement a custom event source by using a container source, you must first create a container image of your event source, and then create a container source that specifies the correct configuration, including the container image URI.

7.6.1.1. Guidelines for creating a container image

- A container image can be developed using any language, and can be built and published with any tool that you prefer.

- The main process of the container image must accept parameters from arguments and environment variables.

- Two environment variables are injected by the container source controller: K_SINK and K_CE_OVERRIDES. These variables are resolved from the sink and ceOverrides spec, respectively.

- Event messages are sent to the sink URI specified in the K_SINK environment variable. The event message can be in any format; however, using the CloudEvent spec is recommended.

7.6.1.2. Example container images

The following is an example of a heartbeats container image:

```go
package main

import "cloudevents"

func main() {
    // Create a CloudEvent
    event := cloudevents.Event{
        Validation: "valid",
        ContextAttributes: {
            SpecVersion: "1.0",
            Type: "dev.knative.eventing.samples.heartbeat",
            Source: "https://knative.dev/eventing-contrib/cmd/heartbeats/#event-test/mypod",
            Id: "2b72d7bf-c38f-4a98-a433-608fbcd2596",
            ContentType: "application/json",
        },
        Extensions: {
            "beats": true,
            "heart": true,
            "the": 42,
        },
        Data: {
            "id": 1,
            "label": "",
        },
    }
    // Send the event to the sink
    // ...
}
```
import (  "context"  "encoding/json"  "flag"  "fmt"  "log"  "os"  "strconv"  "time"
)

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

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


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

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

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

type envConfig struct {
  // Sink URL where to send heartbeat cloud events
  Sink string `envconfig:"K_SINK"

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

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

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

  // Whether to run continuously or exit.
  OneShot bool `envconfig:"ONE_SHOT" default:"false"
}
func main() {
    flag.Parse()

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

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

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

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

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

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

    if eventSource == "" {
        log.Printf("Heartbeats Source: %s", eventSource)
    }

    if len(label) > 0 && label[0] == "" {
        label, _ = strconv.Unquote(label)
    }

    hb := &Heartbeat{
        Sequence: 0,
        Label: label,
    }

    ticker := time.NewTicker(period)
    for {
The following is an example of a container source that references the previous heartbeats container image:

```go
hb.Sequence++

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

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

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

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

if env.OneShot {
    return
}

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

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

```yaml
apiVersion: sources.knative.dev/v1
type: ContainerSource
metadata:
    name: test-heartbeats
spec:
    template:
        spec:
            containers:
                # This corresponds to a heartbeats image URI that you have built and published
                - image: gcr.io/knative-releases/knative.dev/eventing/cmd/heartbeats
                  name: heartbeats
                    args:
                        - --period=1
                  env:
                    - name: POD_NAME
                      value: "example-pod"
                    - name: POD_NAMESPACE
                      value: "event-test"
            sink:
                ref:
```
7.6.2. Creating and managing container sources by using the Knative CLI

You can use the following `kn` commands to create and manage container sources:

Create a container source

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

Delete a container source

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

Describe a container source

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

List existing container sources

```
$ kn source container list
```

List existing container sources in YAML format

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

Update a container source

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

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

7.6.3. Creating a container source by using the web console

You can create a container source by using the Developer perspective of the OpenShift Container Platform web console.

Prerequisites

To create a container source 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 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. In the **Developer** perspective, navigate to **Add → Event Source**. The **Event Sources** page is displayed.

2. Select **Container Source** and then click **Create Event Source** The **Create Event Source** page is displayed.

3. Configure the **Container Source** settings by using the **Form view** or **YAML view**:

   **NOTE**

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

   a. In the **Image** field, enter the URI of the image that you want to run in the container created by the container source.

   b. In the **Name** field, enter the name of the image.

   c. Optional: In the **Arguments** field, enter any arguments to be passed to the container.

   d. Optional: In the **Environment variables** field, add any environment variables to set in the container.

   e. In the **Sink** section, add a sink where events from the container source are routed to. If you are using the **Form view**, you can choose from the following options:

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

      ii. Select **URI** to specify where the events from the container source are routed to.

4. After you have finished configuring the container source, click **Create**.

**7.7. 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.

**7.7.1. Prerequisites**

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

**7.7.2. 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.

**7.7.3. Creating a Kafka event source by using the Knative CLI**

This section describes how to create a Kafka event source by using the kn command.
Creating a Kafka event source by using the `kn` CLI is a Technology Preview feature only. Technology Preview features are not supported with Red Hat production service level agreements (SLAs) and might not be functionally complete. Red Hat does not recommend using them in production. These features provide early access to upcoming product features, enabling customers to test functionality and provide feedback during the development process.

For more information about the support scope of Red Hat Technology Preview features, see [https://access.redhat.com/support/offerings/techpreview/](https://access.redhat.com/support/offerings/techpreview/).

**Prerequisites**

- The OpenShift Serverless Operator, Knative Eventing, Knative Serving, and the `KnativeKafka` custom resource (CR) are installed on your cluster.
- You have created a project or have access to a project with the appropriate roles and permissions to create applications and other workloads in OpenShift Container Platform.
- You have access to a Red Hat AMQ Streams (Kafka) cluster that produces the Kafka messages you want to import.

**Procedure**

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

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

2. Create a `KafkaSource` CR:

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

   **NOTE**

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

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

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

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

**Example output**

```
Name:              example-kafka-source
Namespace:         kafka
```
Verification steps

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

   ```
   $ oc -n kafka run kafka-producer \
     -ti --image=quay.io/strimzi/kafka:latest-kafka-2.7.0 --rm=true \
     --restart=Never -- bin/kafka-console-producer.sh \
     --broker-list <cluster_kafka_bootstrap>:9092 --topic my-topic
   ```

   Enter the message in the prompt. This command assumes that:
   
   - The Kafka cluster is installed in the `kafka` namespace.
   - The `KafkaSource` object has been configured to use the `my-topic` topic.

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

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

Example output

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

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

### 7.7.4. 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 **KafkaSource** object as a YAML file:

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

1. A consumer group is a group of consumers that use the same group ID, and consume data from a topic.
2. A topic provides a destination for the storage of data. Each topic is split into one or more partitions.
3. A sink specifies where events are sent to from a source.
IMPORTANT

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

Example KafkaSource object

```yaml
apiVersion: sources.knative.dev/v1beta1
kind: KafkaSource
metadata:
  name: kafka-source
spec:
  consumerGroup: knative-group
  bootstrapServers:
    - my-cluster-kafka-bootstrap.kafka:9092
  topics:
    - knative-demo-topic
  sink:
    ref:
      apiVersion: serving.knative.dev/v1
      kind: Service
      name: event-display
```

2. Apply the KafkaSource YAML file:

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

Verification

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

  ```bash
  $ oc get pods
  ```

Example output

<table>
<thead>
<tr>
<th>NAME</th>
<th>READY</th>
<th>STATUS</th>
<th>RESTARTS</th>
<th>AGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>kafkasource-kafka-source-5ca0248f-...</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>13m</td>
</tr>
</tbody>
</table>

7.7.5. Additional resources

- See Understanding event sources.
- See Knative Kafka.
- See the Red Hat AMQ Streams documentation for more information about Kafka concepts.
8.1. 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.

8.1.1. Next steps

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

8.2. 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:

```yaml
apiVersion: messaging.knative.dev/v1
kind: Channel
metadata:
```
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.

### 8.2.1. 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**
8.2.2. Creating a channel by 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**

<table>
<thead>
<tr>
<th>NAME</th>
<th>TYPE</th>
<th>URL</th>
<th>AGE</th>
<th>READY</th>
<th>REASON</th>
</tr>
</thead>
<tbody>
<tr>
<td>mychannel</td>
<td>InMemoryChannel</td>
<td><a href="http://mychannel-kn-channel.default.svc.cluster.local">http://mychannel-kn-channel.default.svc.cluster.local</a></td>
<td>93s</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
8.2.3. 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:

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

2. Apply the YAML file:

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

8.2.4. 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 KafkaChannel object as a YAML file:

   ```yaml
   apiVersion: messaging.knative.dev/v1beta1
   kind: KafkaChannel
   metadata:
     name: example-channel
     namespace: default
   spec:
     numPartitions: 3
     replicationFactor: 1
   ```
2. Apply the KafkaChannel YAML file:

   $ oc apply -f <filename>

8.2.5. Deleting a channel by 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>

8.2.6. Next steps

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

8.3. 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.

8.3.1. Creating subscriptions

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

8.3.1.1. 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.

8.3.1.2. Creating subscriptions by 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:

  ```bash
  $ kn subscription create <subscription_name> \
  --channel <group:version:kind>:<channel_name> \
  --sink <sink_prefix>:<sink_name> \
  --sink-dead-letter <sink_prefix>:<sink_name>
  ```

  1. `--channel` specifies the source for cloud events that should be processed. You must provide the channel name. If you are not using the default `InMemoryChannel` channel that is backed by the `Channel` custom resource, you must prefix the channel name with the `<group:version:kind>` for the specified channel type. For example, this will be `messaging.knative.dev:v1beta1:KafkaChannel` for 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.

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

```bash
$ 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:

```bash
$ kn subscription list
```

Example output

```
NAME            CHANNEL             SUBSCRIBER           REPLY   DEAD LETTER SINK
READY   REASON
mysubscription   Channel:mychannel   ksvc:event-display                              True
```

8.3.1.3. 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:

```yaml
apiVersion: messaging.knative.dev/v1beta1
kind: Subscription
metadata:
  name: my-subscription
namespace: default
spec:
  channel:
    apiVersion: messaging.knative.dev/v1beta1
    kind: Channel
    name: example-channel
delivery:
```
Name of the subscription.

Configuration settings for the channel that the subscription connects to.

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.

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>
```

### 8.3.2. 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**

```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.

### 8.3.3. Describing subscriptions by 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:

  ```sh
  $ 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
```
### 8.3.4. Listing subscriptions by 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**

1. 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
```

### 8.3.5. Updating subscriptions by using the Knative CLI

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**

1. Update a subscription:

   ```
   $ kn subscription update <subscription_name> \ 
   --sink <sink_prefix>:<sink_name> \ 1
   --sink-dead-letter <sink_prefix>:<sink_name> 2
   ```

   - `--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.

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
```

### 8.3.6. Deleting subscriptions by using the Knative CLI

You can delete a subscription by using the **kn** CLI.

**Procedure**

- Delete a subscription:

  ```
  $ kn subscription delete <subscription_name>
  ```

### 8.4. 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.

#### 8.4.1. 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
  ```
In `spec.config`, you can specify the config maps that you want to add modified configurations for.

The `default-ch-webhook` config map can be used to specify the default channel implementation for the cluster or for one or more namespaces.

The cluster-wide default channel type configuration. In this example, the default channel implementation for the cluster is `InMemoryChannel`.

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 9. FUNCTIONS

9.1. ABOUT OPENSHIFT SERVERLESS FUNCTIONS

IMPORTANT

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

For more information about the support scope of Red Hat Technology Preview features, see https://access.redhat.com/support/offerings/techpreview/.

OpenShift Serverless Functions enables developers to create and deploy stateless, event-driven functions as a Knative service on OpenShift Container Platform.

The \texttt{kn func} CLI is provided as a plug-in for the Knative \texttt{kn} CLI. OpenShift Serverless Functions uses the \textit{CNCF Buildpack API} to create container images. After a container image has been created, you can use the \texttt{kn func} CLI to deploy the container image as a Knative service on the cluster.

9.1.1. Supported runtimes

OpenShift Serverless Functions provides templates that can be used to create basic functions for the following runtimes:

- Node.js
- Python
- Golang
- Quarkus

9.1.2. Next steps

- See \textit{Getting started with functions}.

9.2. SETTING UP OPENSHIFT SERVERLESS FUNCTIONS

IMPORTANT

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

For more information about the support scope of Red Hat Technology Preview features, see https://access.redhat.com/support/offerings/techpreview/.
Before you can develop functions on OpenShift Serverless, you must complete the set up steps.

### 9.2.1. Prerequisites

To enable the use of OpenShift Serverless Functions on your cluster, you must complete the following steps:

- OpenShift Serverless is installed on your cluster.
- The `oc` CLI is installed on your cluster.
- The Knative (kn) CLI is installed on your cluster. Installing the `kn` CLI enables the use of `kn func` commands which you can use to create and manage functions.
- You have installed Docker Container Engine or podman, and have access to an available image registry.
- If you are using Quay.io as the image registry, you must ensure that either the repository is not private, or that you have followed the OpenShift Container Platform documentation on **Allowing pods to reference images from other secured registries**.
- If you are using the OpenShift Container Registry, a cluster administrator must **expose the registry**.

### 9.2.2. Using podman

If you are using podman, you must run the following commands before getting started with OpenShift Serverless Functions:

1. Start the podman service that listens on port **1234**:

   ```bash
   $ podman system service --time=0 tcp:0.0.0.0:1234 & # let run in background or another terminal
   ```

2. Establish the environment variable that is used to build a function:

   ```bash
   $ export DOCKER_HOST=tcp://127.0.0.1:1234
   $ func build -v
   ```

### 9.2.3. Next steps

- For more information about Docker Container Engine or podman, see **Container build tool options**.
- See **Getting started with functions**.

### 9.3. GETTING STARTED WITH FUNCTIONS
IMPORTANT

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

For more information about the support scope of Red Hat Technology Preview features, see https://access.redhat.com/support/offerings/techpreview/.

This guide explains how you can get started with creating, building, and deploying a function on an OpenShift Serverless installation.

9.3.1. Prerequisites

Before you can complete the following procedures, you must ensure that you have completed all of the prerequisite tasks in Setting up OpenShift Serverless Functions.

9.3.2. Creating functions

You can create a basic serverless function using the `kn` CLI.

You can specify the runtime, trigger, image, and namespace as flags on the command line, or use the `-c` flag to start the interactive experience using the CLI prompt.

The values provided for image and registry are persisted to the `func.yaml` file, so that subsequent invocations do not require the user to specify these again.

Example `func.yaml`

```
name: example-function
namespace: default
runtime: node
image: <image_from_registry>
imageDigest: ""
trigger: http
builder: default
builderMap:
  default: quay.io/boson/faas-nodejs-builder
envs: {}
```

Procedure

- Create a function project:

  ```
  $ kn func create <path> -r <registry> -l <runtime> -t <trigger> -i <image> -n <namespace>
  ```

  - Supported runtimes include `node`, `go`, `python`, and `quarkus`.
  - If the image is unspecified, you are prompted for a registry name. The image name is derived from this registry and the function name.
### Example command

$ kn func create functions/example-function

### Example output

Project path: /home/user/functions/example-function
Function name: example-function
Runtime: node
Trigger: http

### 9.3.3. Building functions

Before you can run a function, you must build the function project by using the `kn func build` command. The build command reads the `func.yaml` file from the function project directory to determine the image name and registry.

**Example func.yaml**

```yaml
name: example-function
namespace: default
runtime: node
image: <image_from_registry>
imageDigest: ""
trigger: http
builder: default
builderMap:
  default: quay.io/boson/faas-nodejs-builder
envs: {}
```

If the image name and registry are not set in the `func.yaml` file, you must either specify the registry flag, `-r` when using the `kn func build` command, or you are prompted to provide a registry value in the terminal when building a function. An image name is then derived from the registry value that you have provided.

**Example command using the -r registry flag**

```
$ kn func build [-i <image> -r <registry> -p <path>]
```

**Example output**

Building function image
Function image has been built, image: quay.io/username/example-function:latest

This command creates an OCI container image that can be run locally on your computer, or on a Kubernetes cluster.

**Example using the registry prompt**

```
$ kn func build
A registry for function images is required (e.g. 'quay.io/boson').
```
Registries for function images: quay.io/username
Building function image
Function image has been built, image: quay.io/username/example-function:latest

The values for image and registry are persisted to the `func.yaml` file, so that subsequent invocations do not require the user to specify these again.

### 9.3.4. Deploying functions

You can deploy a function to your cluster as a Knative service by using the `kn func deploy` command.

If the targeted function is already deployed, it is updated with a new container image that is pushed to a container image registry, and the Knative service is updated.

**Prerequisites**

- You must have already initialized the function that you want to deploy.

**Procedure**

- Deploy a function:
  
  ```
  $ kn func deploy [-n <namespace> -p <path> -i <image> -r <registry>]
  
  Example output
  
  Function deployed at: http://func.example.com
  
  - If no `namespace` is specified, the function is deployed in the current namespace.
  - The function is deployed from the current directory, unless a `path` is specified.
  - The Knative service name is derived from the project name, and cannot be changed using this command.
  ```

### 9.3.5. Building and deploying functions with OpenShift Container Registry

When building and deploying functions, the resulting container image is stored in an image registry. Usually this will be a public registry, such as Quay. However, you can use the integrated OpenShift Container Registry instead if it has been exposed by a cluster administrator.

**Procedure**

- Run the `kn func build` command, or the `kn func deploy` command, with the OpenShift Container Registry specified for the `-r` parameter:
  
  **Example build command**
  
  ```
  $ kn func build -r $(oc get route -n openshift-image-registry)
  
  Example deploy command
  
  $ kn func deploy -r $(oc get route -n openshift-image-registry)
  ```
You can verify that the function deployed successfully by emitting a test event to it.

### 9.3.6. Emitting a test event to a deployed function

You can use the `kn func emit` CLI command to emit a CloudEvent to a function that is either deployed locally or deployed to your OpenShift Container Platform cluster. This command can be used to test that a function is working and able to receive events correctly.

**Example command**

```
$ kn func emit
```

The `kn func emit` command executes on the local directory by default, and assumes that this directory is a function project.

### 9.3.7. Next steps

- See [Using functions with Knative Eventing](#).

### 9.4. DEVELOPING NODE.JS FUNCTIONS

**IMPORTANT**

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

For more information about the support scope of Red Hat Technology Preview features, see [https://access.redhat.com/support/offerings/techpreview/](https://access.redhat.com/support/offerings/techpreview/).

After you have [created a Node.js function project](#), you can modify the template files provided to add business logic to your function.

### 9.4.1. Prerequisites

- Before you can develop functions, you must complete the steps in [Setting up OpenShift Serverless Functions](#).

### 9.4.2. Node.js function template structure

When you create a Node.js function using the `kn` CLI, the project directory looks like a typical Node.js project, with the exception of an additional `func.yaml` configuration file.

Both `http` and `event` trigger functions have the same template structure:

**Template structure**

```
The `func.yaml` configuration file is used to determine the image name and registry.

Your project must contain an `index.js` file which exports a single function.

You are not restricted to the dependencies provided in the template `package.json` file. You can add additional dependencies as you would in any other Node.js project.

**Example of adding npm dependencies**

```bash
npm install --save opossum
```

When the project is built for deployment, these dependencies are included in the created runtime container image.

Integration and unit test scripts are provided as part of the function template.

### 9.4.3. About invoking Node.js functions

When using the `kn` CLI to create a function project, you can generate a project that responds to CloudEvents, or one that responds to simple HTTP requests. CloudEvents in Knative are transported over HTTP as a POST request, so both function types listen for and respond to incoming HTTP events.

Node.js functions can be invoked with a simple HTTP request. When an incoming request is received, functions are invoked with a `context` object as the first parameter.

### 9.4.3.1. Node.js context objects

Functions are invoked by providing a `context` object as the first parameter.

**Example context object**

```javascript
function handle(context, data)
```

This object provides access to the incoming HTTP request information, including the HTTP request method, any query strings or headers sent with the request, the HTTP version, and the request body. Incoming requests that contain a CloudEvent attach the incoming instance of the CloudEvent to the context object so that it can be accessed by using `context.cloudevent`.

### 9.4.3.1.1. Context object methods

The `context` object has a single method, `cloudEventResponse()`, that accepts a data value and returns a CloudEvent.
In a Knative system, if a function deployed as a service is invoked by an event broker sending a CloudEvent, the broker examines the response. If the response is a CloudEvent, this event is handled by the broker.

**Example context object method**

```javascript
// Expects to receive a CloudEvent with customer data
function handle(context, customer) {
    // process the customer
    const processed = handle(customer);
    return context.cloudEventResponse(customer)
        .source('/handle')
        .type('fn.process.customer')
        .response();
}
```

**9.4.3.1.2. CloudEvent data**

If the incoming request is a CloudEvent, any data associated with the CloudEvent is extracted from the event and provided as a second parameter. For example, if a CloudEvent is received that contains a JSON string in its data property that is similar to the following:

```json
{
    "customerId": "0123456",
    "productId": "6543210"
}
```

When invoked, the second parameter to the function, after the context object, will be a JavaScript object that has `customerId` and `productId` properties.

**Example signature**

```javascript
function handle(context, data)
```

The `data` parameter in this example is a JavaScript object that contains the `customerId` and `productId` properties.

**9.4.4. Node.js function return values**

Functions can return any valid JavaScript type or can have no return value. When a function has no return value specified, and no failure is indicated, the caller receives a 204 No Content response.

Functions can also return a CloudEvent or a Message object in order to push events into the Knative Eventing system. In this case, the developer is not required to understand or implement the CloudEvent messaging specification. Headers and other relevant information from the returned values are extracted and sent with the response.

**Example**

```javascript
function handle(context, customer) {
    // process customer and return a new CloudEvent
    return new CloudEvent({
        source: 'customer.processor',
    });
}
```
9.4.4.1. Returning headers

You can set a response header by adding a `headers` property to the `return` object. These headers are extracted and sent with the response to the caller.

Example response header

```javascript
function handle(context, customer) {
  // process customer and return custom headers
  // the response will be '204 No content'
  return { headers: { customerid: customer.id } };
}
```

9.4.4.2. Returning status codes

You can set a status code that is returned to the caller by adding a `statusCode` property to the `return` object:

Example status code

```javascript
function handle(context, customer) {
  // process customer
  if (customer.restricted) {
    return { statusCode: 451 }
  }
}
```

Status codes can also be set for errors that are created and thrown by the function:

Example error status code

```javascript
function handle(context, customer) {
  // process customer
  if (customer.restricted) {
    const err = new Error('Unavailable for legal reasons');
    err.statusCode = 451;
    throw err;
  }
}
```

9.4.5. Testing Node.js functions

Node.js functions can be tested locally on your computer. In the default project that is created when you create a function using `kn func create`, there is a `test` folder that contains some simple unit and integration tests.

Procedure

- Run the tests:
9.4.6. Next steps

- See the Node.js context object reference documentation.
- Build and deploy a function.

9.5. DEVELOPING GOLANG FUNCTIONS

**IMPORTANT**

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

For more information about the support scope of Red Hat Technology Preview features, see [https://access.redhat.com/support/offerings/techpreview/](https://access.redhat.com/support/offerings/techpreview/).

After you have created a Golang function project, you can modify the template files provided to add business logic to your function.

9.5.1. Prerequisites

- Before you can develop functions, you must complete the steps in Setting up OpenShift Serverless Functions.

9.5.2. Golang function template structure

When you create a Golang function using the `kn` CLI, the project directory looks like a typical Go project, with the exception of an additional `func.yaml` configuration file.

Golang functions have few restrictions. The only requirements are that your project must be defined in a `function` module, and must export the function `Handle()`.

Both http and event trigger functions have the same template structure:

**Template structure**

```
├── README.md
├── func.yaml 1
├── go.mod 2
├── go.sum
├── handle.go
└── handle_test.go
```

1 The `func.yaml` configuration file is used to determine the image name and registry.
You can add any required dependencies to the `go.mod` file, which can include additional local Golang files. When the project is built for deployment, these dependencies are included in the resulting runtime container image.

**Example of adding dependencies**

```
$ go get gopkg.in/yaml.v2@v2.4.0
```

### 9.5.3. About invoking Golang functions

Golang functions are invoked by using different methods, depending on whether they are triggered by an HTTP request or a CloudEvent.

#### 9.5.3.1. Functions triggered by an HTTP request

When an incoming HTTP request is received, your function is invoked with a standard Golang `Context` as the first parameter, followed by two more parameters:

- `http.ResponseWriter`
- `http.Request`

You can use standard Golang techniques to access the request, and set a proper HTTP response of your function.

**Example HTTP response**

```go
func Handle(ctx context.Context, res http.ResponseWriter, req *http.Request) {
    // Read body
    body, err := ioutil.ReadAll(req.Body)
    defer req.Body.Close()
    if err != nil {
        http.Error(res, err.Error(), 500)
        return
    }
    // Process body and function logic
    // ...
}
```

#### 9.5.3.2. Functions triggered by a cloud event

When an incoming cloud event is received, the event is invoked by the CloudEvents Golang SDK and the `Event` type as a parameter.

You can leverage the Golang `Context` as an optional parameter in the function contract, as shown in the list of supported function signatures:

**Supported function signatures**

- `Handle() error`
- `Handle(context.Context) error`
9.5.3.2.1 CloudEvent trigger example

A cloud event is received which contains a JSON string in the data property:

```json
{
  "customerId": "0123456",
  "productId": "6543210"
}
```

To access this data, a structure must be defined which maps properties in the cloud event data, and retrieves the data from the incoming event. The following example uses the `Purchase` structure:

```go
type Purchase struct {
  CustomerId string `json:"customerId"
  ProductId string `json:"productId"
}

func Handle(ctx context.Context, event cloudevents.Event) (err error) {
  purchase := &Purchase{}
  if err = event.DataAs(purchase); err != nil {
    fmt.Fprintf(os.Stderr, "failed to parse incoming CloudEvent %s\n", err)
    return
  }
  // ...
}
```

Alternatively, a GoLang `encoding/json` package could be used to access the cloud event directly as JSON in the form of a bytes array:

```go
func Handle(ctx context.Context, event cloudevents.Event) {
  bytes, err := json.Marshal(event)
  // ...
}
```

9.5.4. GoLang function return values

HTTP triggered functions can set the response directly by using the GoLang `http.ResponseWriter`.

Example HTTP response

```go
func Handle(ctx context.Context, res http.ResponseWriter, req *http.Request) {
  // Set response
  res.Header().Add("Content-Type", "text/plain")
  res.Header().Add("Content-Length", "3")
```

Functions triggered by a cloud event might return nothing, error, or `CloudEvent` in order to push events into the Knative Eventing system. In this case, you must set a unique ID, proper Source, and a Type for the cloud event. The data can be populated from a defined structure, or from a map.

**Example CloudEvent response**

```go
func Handle(ctx context.Context, event cloudevents.Event) (resp *cloudevents.Event, err error) {
    // ...
    response := cloudevents.NewEvent()
    response.SetID("example-uuid-32943bac6fea")
    response.SetSource("purchase/getter")
    response.SetType("purchase")
    // Set the data from Purchase type
    response.SetData(cloudevents.ApplicationJSON, Purchase{
        CustomerId: custId,
        ProductId: prodId,
    })
    // OR set the data directly from map
    response.SetData(cloudevents.ApplicationJSON, map[string]string{"customerId": custId, "productId": prodId})
    // Validate the response
    resp = &response
    if err = resp.Validate(); err != nil {
        fmt.Printf("invalid event created. %v", err)
    }
    return
}
```

**9.5.5. Testing Golang functions**

Golang functions can be tested locally on your computer. In the default project that is created when you create a function using `kn func create`, there is a `handle_test.go` file which contains some basic tests. These tests can be extended as needed.

**Procedure**

- Run the tests:
  
  ```bash
  $ go test
  ```

**9.5.6. Next steps**

- Build and deploy a function.

**9.6. DEVELOPING PYTHON FUNCTIONS**
IMPORTANT

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

For more information about the support scope of Red Hat Technology Preview features, see https://access.redhat.com/support/offerings/techpreview/.

After you have created a Python function project, you can modify the template files provided to add business logic to your function.

9.6.1. Prerequisites

- Before you can develop functions, you must complete the steps in Setting up OpenShift Serverless Functions.

9.6.2. Python function template structure

When you create a Python function by using the `kn` CLI, the project directory looks similar to a typical Python project.

Python functions have very few restrictions. The only requirements are that your project contains a `func.py` file that contains a `main()` function, and a `func.yaml` configuration file.

Developers are not restricted to the dependencies provided in the template `requirements.txt` file. Additional dependencies can be added as they would be in any other Python project. When the project is built for deployment, these dependencies will be included in the created runtime container image.

Both `http` and `event` trigger functions have the same template structure:

**Template structure**

```
fn
├── func.py 1
├── func.yaml 2
├── requirements.txt 3
└── test_func.py 4
```

1. Contains a `main()` function.
2. Used to determine the image name and registry.
3. Additional dependencies can be added to the `requirements.txt` file as they are in any other Python project.
4. Contains a simple unit test that can be used to test your function locally.

9.6.3. About invoking Python functions
Python functions can be invoked with a simple HTTP request. When an incoming request is received, functions are invoked with a context object as the first parameter. The context object is a Python class with two attributes:

- The request attribute is always present, and contains the Flask request object.
- The second attribute, cloud_event, is populated if the incoming request is a CloudEvent object.

Developers can access any CloudEvent data from the context object.

**Example context object**

```python
def main(context: Context):
    """
    The context parameter contains the Flask request object and any CloudEvent received with the request.
    """
    print(f"Method: {context.request.method}")
    print(f"Event data {context.cloud_event.data}")
    # ... business logic here
```

### 9.6.4. Python function return values

Functions can return any value supported by Flask because the invocation framework proxies these values directly to the Flask server.

**Example**

```python
def main(context: Context):
    body = { "message": "Howdy!" }
    headers = { "content-type": "application/json" }
    return body, 200, headers
```

Functions can set both headers and response codes as secondary and tertiary response values from function invocation.

#### 9.6.4.1. Returning CloudEvents

Developers can use the @event decorator to tell the invoker that the function return value must be converted to a CloudEvent before sending the response.

**Example**

```python
@event("event_source":"/my/function", "event_type":"my.type")
def main(context):
    # business logic here
    data = do_something()
    # more data processing
    return data
```

This example sends a CloudEvent as the response value, with a type of "my.type" and a source of "my/function". The CloudEvent data property is set to the returned data variable. The event_source and event_type decorator attributes are both optional.
9.6.5. Testing Python functions

You can test Python functions locally on your computer. The default project contains a `test_func.py` file, which provides a simple unit test for functions.

**NOTE**

The default test framework for Python functions is `unittest`. You can use a different test framework if you prefer.

**Prerequisites**

- To run Python functions tests locally, you must install the required dependencies:

```
$ pip install -r requirements.txt
```

**Procedure**

- After you have installed the dependencies, run the tests:

```
$ python3 test_func.py
```

9.6.6. Next steps

- Build and deploy a function.

9.7. DEVELOPING QUARKUS FUNCTIONS

**IMPORTANT**

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

For more information about the support scope of Red Hat Technology Preview features, see [https://access.redhat.com/support/offerings/techpreview/](https://access.redhat.com/support/offerings/techpreview/).

After you have created a Quarkus function project, you can modify the template files provided to add business logic to your function.

9.7.1. Prerequisites

- Before you can develop functions, you must complete the setup steps in Setting up OpenShift Serverless Functions.

9.7.2. Quarkus function template structure

When you create a Quarkus function by using the `kn` CLI, the project directory looks similar to a typical Maven project.
Both http and event trigger functions have the same template structure:

**Template structure**

```
 func.yaml
 mvnw
 mvnw.cmd
 pom.xml
 README.md
 src
   main
     java
       functions
         Function.java
         Input.java
         Output.java
     resources
     test
       application.properties
       test
         java
           functions
             FunctionTest.java
             NativeFunctionIT.java
```

1. Used to determine the image name and registry.
2. The Project Object Model (POM) file contains project configuration, such as information about dependencies. You can add additional dependencies by modifying this file.

**Example of additional dependencies**

```
...  
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.11</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>org.assertj</groupId>
      <artifactId>assertj-core</artifactId>
      <version>3.8.0</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
...  
```

Dependencies are downloaded during the first compilation.

3. The function project must contain a Java method annotated with `@Funq`. You can place this method in the `Function.java` class.
Contains simple test cases that can be used to test your function locally.

9.7.3. About invoking Quarkus functions

You can create a Quarkus project that responds to cloud events, or one that responds to simple HTTP requests. Cloud events in Knative are transported over HTTP as a POST request, so either function type can listen and respond to incoming HTTP requests.

When an incoming request is received, Quarkus functions are invoked with an instance of a permitted type.

Table 9.1. Function invocation options

<table>
<thead>
<tr>
<th>Invocation method</th>
<th>Data type contained in the instance</th>
<th>Example of data</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTTP POST request</td>
<td>JSON object in the body of the request</td>
<td>{ &quot;customerId&quot;: &quot;0123456&quot;, &quot;productId&quot;: &quot;6543210&quot; }</td>
</tr>
<tr>
<td>HTTP GET request</td>
<td>Data in the query string</td>
<td>? customerId=0123456&amp;productId=6543210</td>
</tr>
<tr>
<td>CloudEvent</td>
<td>JSON object in the data property</td>
<td>{ &quot;customerId&quot;: &quot;0123456&quot;, &quot;productId&quot;: &quot;6543210&quot; }</td>
</tr>
</tbody>
</table>

The following example shows a function that receives and processes the customerId and productId purchase data that is listed in the previous table:

Example Quarkus function

```java
public class Functions {
    @Funq
    public void processPurchase(Purchase purchase) {
        // process the purchase
    }
}
```

The corresponding Purchase JavaBean class that contains the purchase data looks as follows:

Example class

```java
public class Purchase {
    private long customerId;
    private long productId;
    // getters and setters
}
```

9.7.3.1. Invocation examples
The following example code defines three functions named `withBeans`, `withCloudEvent`, and `withBinary`:

```java
import io.quarkus.funqy.Funq;
import io.quarkus.funqy.knative.events.CloudEvent;

public class Input {
    private String message;
    // getters and setters
}

public class Output {
    private String message;
    // getters and setters
}

public class Functions {
    @Funq
    public Output withBeans(Input in) {
        // function body
    }

    @Funq
    public CloudEvent<Output> withCloudEvent(CloudEvent<Input> in) {
        // function body
    }

    @Funq
    public void withBinary(byte[] in) {
        // function body
    }
}
```

The `withBeans` function of the `Functions` class can be invoked by:

- An HTTP POST request with a JSON body:
  ```bash
  $ curl "http://localhost:8080/withBeans" -X POST \
  -H "Content-Type: application/json" \
  -d '["message": "Hello there."]'
  ```

- An HTTP GET request with query parameters:
  ```bash
  ```

- A `CloudEvent` object in binary encoding:
  ```bash
  $ curl "http://localhost:8080/" -X POST \
  -H "Content-Type: application/json" \
  -H "Ce-SpecVersion: 1.0" \
  -H "Ce-Type: withBeans"
  ```
A CloudEvent object in structured encoding:

```
$ curl http://localhost:8080/ 
   -H "Content-Type: application/cloudevents+json" 
   -d '{ "data": { "message": "Hello there." }, 
      "datacontenttype": "application/json", 
      "id": "42", 
      "source": "curl", 
      "type": "withBeans", 
      "specversion": "1.0"}'
```

The withCloudEvent function of the Functions class can be invoked by using a CloudEvent object, similarly to the withBeans function. However, unlike withBeans, withCloudEvent cannot be invoked with a plain HTTP request.

The withBinary function of the Functions class can be invoked by:

- A CloudEvent object in binary encoding:

```
$ curl "http://localhost:8080/" -X POST 
   -H "Content-Type: application/octet-stream" 
   -H "Ce-SpecVersion: 1.0"
   -H "Ce-Type: withBinary"
   -H "Ce-Source: cURL"
   -H "Ce-Id: 42" 
   --data-binary @img.jpg'
```

- A CloudEvent object in structured encoding:

```
$ curl http://localhost:8080/ 
   -H "Content-Type: application/cloudevents+json" 
   -d "{"data_base64": "$(base64 --wrap=0 img.jpg)", 
       "datacontenttype": "application/octet-stream", 
       "id": "42", 
       "source": "curl", 
       "type": "withBinary", 
       "specversion": "1.0"}"
```

9.7.4. CloudEvent attributes

If you need to read or write the attributes of a CloudEvent, such as type or subject, you can use the CloudEvent<T> generic interface and the CloudEventBuilder builder. The <T> type parameter must be one of the permitted types.

In the following example, CloudEventBuilder is used to return success or failure of processing the purchase:

```java
public class Functions {

    private boolean _processPurchase(Purchase purchase) {
```
9.7.5. Quarkus function return values

Functions can return an instance of:

- Any type from the list of permitted types.
- The `Uni<T>` type, where the `<T>` type parameter can be of any type from the permitted types.

The `Uni<T>` type is useful if a function calls asynchronous APIs, because the returned object is serialized in the same format as the received object. For example:

- If a function receives an HTTP request, then the returned object is sent in the body of an HTTP response.
- If a function receives a `CloudEvent` object in binary encoding, then the returned object is sent in the data property of a binary-encoded `CloudEvent` object.

The following example shows a function that fetches a list of purchases:

**Example command**

```java
// do stuff

public CloudEvent<Void> processPurchase(CloudEvent<Purchase> purchaseEvent) {
    System.out.println("subject is: " + purchaseEvent.subject());

    if (!_processPurchase(purchaseEvent.data())) {
        return CloudEventBuilder.create()
            .type("purchase.error")
            .build();
    }
    return CloudEventBuilder.create()
        .type("purchase.success")
        .build();
}
```

```java
public class Functions {
    @Funq
    public List<Purchase> getPurchasesByName(String name) {
        // logic to retrieve purchases
    }
}
```

- Invoking this function through an HTTP request produces an HTTP response that contains a list of purchases in the body of the response.
- Invoking this function through an incoming `CloudEvent` object produces a `CloudEvent` response with a list of purchases in the `data` property.

9.7.5.1. Permitted types

The input and output types of a function can be any of the following:
• void

• String

• byte[]

• Primitive types and their wrappers (for example, int and Integer).

• A JavaBean, if its attributes are of types listed here.

• A map, list, or array of the types in this list.

• The special `CloudEvents<T>` type, where the `<T>` type parameter is of a type in this list.

Example

```java
public class Functions {
    public List<Integer> getIds();
    public Purchase[] getPurchasesByName(String name);
    public String getNameById(int id);
    public Map<String,Integer> getNameIdMapping();
    public void processImage(byte[] img);
}
```

9.7.6. Testing Quarkus functions

You can test Quarkus functions locally on your computer by running the Maven tests that are included in the project template.

Procedure

• Run the Maven tests:

  ```bash
  $ ./mvnw test
  ```

9.7.7. Next steps

• Build and deploy a function.

9.8. USING FUNCTIONS WITH KNATIVE EVENTING

**IMPORTANT**

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

For more information about the support scope of Red Hat Technology Preview features, see [https://access.redhat.com/support/offerings/techpreview/](https://access.redhat.com/support/offerings/techpreview/).
Functions are deployed as a Knative service on a OpenShift Container Platform cluster, and can be connected as a sink to Knative Eventing components.

9.8.1. Connect an event source to a sink using the Developer perspective

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

Prerequisites

To connect an event source to a sink 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 created a sink.
- You have logged in to the web console and are in the Developer perspective.
- You have created a project or have access to a project with the appropriate roles and permissions to create applications and other workloads in OpenShift Container Platform.

Procedure

1. Create an event source of any type, by navigating to +Add → Event Sources and then selecting the event source type that you want to create.

2. In the Sink section of the Event Sources form view, select Resource. Then use the drop-down list to select your sink.

3. Click Create.

Verification

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

1. In the Developer perspective, navigate to Topology.

2. View the event source and click on the connected sink to see the sink details in the side panel.

9.9. FUNCTION PROJECT CONFIGURATION IN FUNC.YAML

The func.yaml file contains the configuration for your function project.

Generally, these values are used when you execute a kn func command. For example, when you run the kn func build command, the value in the builder field is used.

NOTE

In many cases, you can override these values with command line flags or environment variables.

9.9.1. Configurable fields in func.yaml
Many of the fields in `func.yaml` are generated automatically when you create, build, and deploy your function. However, there are also fields that you modify manually to change things, such as the function name or the image name.

### 9.9.1.1. builder

The `builder` field specifies the Buildpack builder image to use when building the function. In most cases, this value should not be changed. When you do change it, use a value that is listed in the `builderMap` field.

### 9.9.1.2. builderMap

Some function runtimes can be built in multiple ways. For example, a Quarkus function can be built for the JVM or as a native binary. The `builderMap` field contains all of the builders available for a given runtime.

### 9.9.1.3. envs

The `envs` field enables you to set environment variables to be available to your function at runtime. You can set an environment variable in several different ways:

1. Directly from a value.
2. From a value assigned to a local environment variable. See the section "Referencing local environment variables from func.yaml fields" for more information.
3. From a key-value pair stored in a secret or config map.
4. You can also import all key-value pairs stored in a secret or config map, with keys used as names of the created environment variables.

This examples demonstrates the different ways to set an environment variable:

```yaml
name: test
namespace: ""
runtime: go
...
envs:
  - name: EXAMPLE1
    value: value
  - name: EXAMPLE2
    value: '{{ env:LOCAL_ENV_VALUE }}'
  - name: EXAMPLE3
    value: '{{ secret:mysecret:key }}'
  - name: EXAMPLE4
    value: '{{ configMap:myconfigmap:key }}'
  - value: '{{ secret:mysecret2 }}'
  - value: '{{ configMap:myconfigmap2 }}'
```

1. An environment variable set directly from a value.
2. An environment variable set from a value assigned to a local environment variable.
3. An environment variable assigned from a key-value pair stored in a secret.
An environment variable assigned from a key-value pair stored in a config map.

A set of environment variables imported from key-value pairs of a secret.

A set of environment variables imported from key-value pairs of a config map.

9.9.1.4. volumes

The *volumes* field enables you to mount secrets and config maps as a volume accessible to the function at the specified path, as shown in the following example:

```yaml
name: test
namespace: ""
runtime: go
...
volumes:
  - secret: mysecret
    path: /workspace/secret
  - configMap: myconfigmap
    path: /workspace/configmap
```

1. The *mysecret* secret is mounted as a volume residing at `/workspace/secret`.
2. The *myconfigmap* config map is mounted as a volume residing at `/workspace/configmap`.

9.9.1.5. options

The *options* field enables you to modify Knative Service properties for the deployed function, such as autoscaling. If these options are not set, the default ones are used.

These options are available:

- **scale**
  - **min**: The minimum number of replicas. Must be a non-negative integer. The default is 0.
  - **max**: The maximum number of replicas. Must be a non-negative integer. The default is 0, which means no limit.
  - **metric**: Defines which metric type is watched by the Autoscaler. It can be set to `concurrency`, which is the default, or `rps`.
  - **target**: Recommendation for when to scale up based on the number of concurrently incoming requests. The `target` option can be a float value greater than 0.01. The default is 100, unless the `options.resources.limits.concurrency` is set, in which case `target` defaults to its value.
  - **utilization**: Percentage of concurrent requests utilization allowed before scaling up. It can be a float value between 1 and 100. The default is 70.

- **resources**
  - **requests**
- **cpu**: A CPU resource request for the container with deployed function.
- **memory**: A memory resource request for the container with deployed function.

  - **limits**
    - **cpu**: A CPU resource limit for the container with deployed function.
    - **memory**: A memory resource limit for the container with deployed function.
    - **concurrency**: Hard Limit of concurrent requests to be processed by a single replica. It can be integer value greater than or equal to 0, default is 0 - meaning no limit.

This is an example configuration of the **scale** options:

```
name: test
namespace: ""
runtime: go
...
options:
scale:
  min: 0
  max: 10
  metric: concurrency
  target: 75
  utilization: 75
resources:
  requests:
    cpu: 100m
    memory: 128Mi
  limits:
    cpu: 1000m
    memory: 256Mi
  concurrency: 100
```

**9.9.1.6. image**

The **image** field sets the image name for your function after it has been built. You can modify this field. If you do, the next time you run `kn func build` or `kn func deploy`, the function image will be created with the new name.

**9.9.1.7. imageDigest**

The **imageDigest** field contains the SHA256 hash of the image manifest when the function is deployed. Do not modify this value.

**9.9.1.8. name**

The **name** field defines the name of your function. This value is used as the name of your Knative service when it is deployed. You can change this field to rename the function on subsequent deployments.

**9.9.1.9. namespace**

The **namespace** field specifies the namespace in which your function is deployed.
9.9.1.10. runtime

The **runtime** field specifies the language runtime for your function, for example, **python**.

9.9.1.11. template

The **template** field specifies the type of the invocation event that triggers your function. You can set it to **http** for triggering with plain HTTP requests or to **events** for triggering with cloud events.

9.9.2. Referencing local environment variables from func.yaml fields

In the **envs** field in the **func.yaml**, you can put a reference to an environment variable available in the local environment. This can be useful for avoiding storing sensitive information, such as an API key in the function configuration.

**Procedure**

- To refer to a local environment variable, use the following syntax:

  ```yaml
  {{ env:ENV_VAR }}
  ```

  Substitute **ENV_VAR** with the name of the variable in the local environment that you want to use.

  For example, you might have the **API_KEY** variable available in the local environment. You can assign its value to the **MY_API_KEY** variable, which you can then directly use within your function:

  ```yaml
  name: test
  namespace: ""
  runtime: go
  ...
  envs:
  - name: MY_API_KEY
    value: '{{ env:API_KEY }}'
  ```

**Additional resources**

- For information on overriding values in the **func.yaml** file, see [Getting started with functions](#).

- For more information on accessing data in secrets and config maps, see Accessing secrets and config maps from Serverless functions.

- For more information on the **scale** set of options, see the Knative documentation on Autoscaling.

- For more information on the **resources** set of options, see the Kubernetes documentation on managing resources for containers and the Knative documentation on configuring concurrency.

## 9.10. Accessing Secrets and Config Maps from Serverless Functions

Your functions, after deployed to the cluster, can access data stored in secrets and config maps. This data can be mounted as volumes, or assigned to environment variables. You can configure this access...
interactively by using the Knative CLI `kn func` commands or manually by editing the function configuration file.

**IMPORTANT**

To access secrets and config maps, the function needs to be deployed on the cluster. This functionality is not available to a function running locally.

If a secret or config map value cannot be accessed, the deployment fails with an error message specifying the inaccessible values.

### 9.10.1. Modifying function access to secrets and config maps interactively

You can manage the secrets and config maps accessed by your function by using the `kn func config` interactive utility.

**Procedure**

1. Run the following command in the function project directory:

   ```
   $ kn func config
   ```

   Alternatively, you can specify the function project directory using the `--path` or `-p` option.

2. Use the interactive interface to perform the necessary operation. For example, using the utility to list configured volumes produces an output similar to this:

   ```
   $ kn func config
   ? What do you want to configure? Volumes
   ? What operation do you want to perform? List
   Configured Volumes mounts:
   - Secret "mysecret" mounted at path: "/workspace/secret"
   - Secret "mysecret2" mounted at path: "/workspace/secret2"
   ```

   This scheme shows all operations available in the interactive utility and how to navigate to them:

   ![Knative CLI Operation Diagram]

   3. Optional. Deploy the function to make the changes take effect:
9.10.2. Modifying function access to secrets and config maps interactively with specialized commands

Every time you run the `kn func config` utility, you need to navigate the entire dialogue to select the operation you need, as shown in the previous section. To save steps, you can directly execute a specific operation by running a more specific form of the `kn func config` command:

- To list configured environment variables:
  ```
  $ kn func config envs [-p <function-project-path>]
  ```

- To add environment variables to the function configuration:
  ```
  $ kn func config envs add [-p <function-project-path>]
  ```

- To remove environment variables from the function configuration:
  ```
  $ kn func config envs remove [-p <function-project-path>]
  ```

- To list configured volumes:
  ```
  $ kn func config volumes [-p <function-project-path>]
  ```

- To add a volume to the function configuration:
  ```
  $ kn func config volumes add [-p <function-project-path>]
  ```

- To remove a volume from the function configuration:
  ```
  $ kn func config volumes remove [-p <function-project-path>]
  ```

9.10.3. Adding function access to secrets and config maps manually

You can manually add configuration for accessing secrets and config maps to your function.

9.10.3.1. Mounting a secret as a volume

1. Open the `func.yaml` file for your function.

2. For each secret you want to mount as a volume, add the following YAML to the `volumes` section:

   ```yaml
   name: test
   namespace: ""
   runtime: go
   ...
   volumes:
   - secret: mysecret
     path: /workspace/secret
   ```
• Substitute `mysecret` with the name of the target secret.

• Substitute `/workspace/secret` with the path where you want to mount the secret.

3. Save the configuration.

9.10.3.2. Mounting a config map as a volume

1. Open the `func.yaml` file for your function.

2. For each config map you want to mount as a volume, add the following YAML to the `volumes` section:

```yaml
name: test
namespace: ""
runtime: go
...
volumes:
  - configMap: myconfigmap
    path: /workspace/configmap
```

• Substitute `myconfigmap` with the name of the target config map.

• Substitute `/workspace/configmap` with the path where you want to mount the config map.

3. Save the configuration.

9.10.3.3. Setting environment variable from a key value defined in a secret

1. Open the `func.yaml` file for your function.

2. For each value from a secret key-value pair that you want to assign to an environment variable, add the following YAML to the `envs` section:

```yaml
name: test
namespace: ""
runtime: go
...
envs:
  - name: EXAMPLE
    value: '${{ secret:mysecret:key }}'
```

• Substitute `EXAMPLE` with the name of the environment variable.

• Substitute `mysecret` with the name of the target secret.

• Substitute `key` with the key mapped to the target value.

3. Save the configuration.

9.10.3.4. Setting environment variable from a key value defined in a config map

1. Open the `func.yaml` file for your function.
2. For each value from a config map key-value pair that you want to assign to an environment variable, add the following YAML to the `envs` section:

```yaml
name: test
namespace: ""
runtime: go
...
envs:
  - name: EXAMPLE
    value: '{{$ configMap:myconfigmap:key }}'
```

- Substitute `EXAMPLE` with the name of the environment variable.
- Substitute `myconfigmap` with the name of the target config map.
- Substitute `key` with the key mapped to the target value.

3. Save the configuration.

9.10.3.5. Setting environment variables from all values defined in a secret

1. Open the `func.yaml` file for your function.

2. For every secret for which you want to import all key-value pairs as environment variables, add the following YAML to the `envs` section:

```yaml
name: test
namespace: ""
runtime: go
...
envs:
  - value: '{{$ secret:mysecret }}'
```

1 Substitute `mysecret` with the name of the target secret.

3. Save the configuration.

9.10.3.6. Setting environment variables from all values defined in a config map

1. Open the `func.yaml` file for your function.

2. For every config map for which you want to import all key-value pairs as environment variables, add the following YAML to the `envs` section:

```yaml
name: test
namespace: ""
runtime: go
...
envs:
  - value: '{{$ configMap:myconfigmap }}'
```

1 Substitute `myconfigmap` with the name of the target config map.
3. Save the file.

9.11. ADDING ANNOTATIONS TO FUNCTIONS

You can add Kubernetes annotations to a deployed Serverless function by adding them to the annotations section in the `func.yaml` configuration file.

**IMPORTANT**

There are two limitations of the function annotation feature:

- Once a function annotation propagates to the corresponding Knative service on the cluster, it cannot be removed from the service by deleting it from the `func.yaml` file. You can remove the annotation from the Knative service by modifying the YAML file of the service directly, or by using the Developer Console.

- You cannot set annotations that are set by Knative, for example, the `autoscaling` annotations.

9.11.1. Adding annotations to a function

**Procedure**

1. Open the `func.yaml` file for your function.

2. For every annotation that you want to add, add the following YAML to the annotations section:

```yaml
<annotation_name>: "<annotation_value>
```

Substitute `<annotation_name>: "<annotation_value>` with your annotation.

For example, to indicate that a function was authored by Alice, you might include the following annotation:

```yaml
name: test
namespace: ""
runtime: go
...
annotations:
  author: "alice@example.com"
```

3. Save the configuration.

The next time you deploy your function to the cluster, the annotations are added to the corresponding Knative service.
OpenShift Serverless Functions is a Technology Preview feature only. Technology Preview features are not supported with Red Hat production service level agreements (SLAs) and might not be functionally complete. Red Hat does not recommend using them in production. These features provide early access to upcoming product features, enabling customers to test functionality and provide feedback during the development process.

For more information about the support scope of Red Hat Technology Preview features, see https://access.redhat.com/support/offerings/techpreview/.

OpenShift Serverless Functions provides templates that can be used to create basic functions for the following runtimes:

- Node.js
- Python
- Golang
- Quarkus

This guide provides reference information that you can use to develop functions.

### 9.12.1. Node.js context object reference

The `context` object has several properties that can be accessed by the function developer.

#### 9.12.1.1. log

Provides a logging object that can be used to write output to the cluster logs. The log adheres to the Pino logging API.

**Example log**

```javascript
function handle(context) {
  context.log.info("Processing customer");
}
```

You can access the function by using the `curl` command to invoke it:

**Example command**

```
$ curl http://example.com
```

**Example output**

```
{"level":30,"time":1604511655265,"pid":3430203,"hostname":"localhost.localdomain","reqId":1,"msg":"Processing customer"}
```
CHAPTER 9. FUNCTIONS

9.12.1.2. query
Returns the query string for the request, if any, as key-value pairs. These attributes are also found on the
context object itself.

Example query
function handle(context) {
// Log the 'name' query parameter
context.log.info(context.query.name);
// Query parameters are also attached to the context
context.log.info(context.name);
}
You can access the function by using the curl command to invoke it:

Example command
$ curl http://example.com?name=tiger

Example output
{"level":30,"time":1604511655265,"pid":3430203,"hostname":"localhost.localdomain","reqId":1,"msg":"tig
er"}

9.12.1.3. body
Returns the request body if any. If the request body contains JSON code, this will be parsed so that the
attributes are directly available.

Example body
function handle(context) {
// log the incoming request body's 'hello' parameter
context.log.info(context.body.hello);
}
You can access the function by using the curl command to invoke it:

Example command
$ curl -X POST -d '{"hello": "world"}' -H 'Content-type: application/json' http://example.com

Example output
{"level":30,"time":1604511655265,"pid":3430203,"hostname":"localhost.localdomain","reqId":1,"msg":"w
orld"}

9.12.1.4. headers
Returns the HTTP request headers as an object.

225


Example header

```javascript
function handle(context) {
  context.log.info(context.headers["custom-header"]);
}
```

You can access the function by using the `curl` command to invoke it:

Example command

```
$ curl -H 'x-custom-header: some-value' http://example.com
```

Example output

```
{"level":30,"time":1604511655265,"pid":3430203,"hostname":"localhost.localdomain","reqId":1,"msg":"some-value"}
```

9.12.1.5. HTTP requests

**method**

Returns the HTTP request method as a string.

**httpVersion**

Returns the HTTP version as a string.

**httpVersionMajor**

Returns the HTTP major version number as a string.

**httpVersionMinor**

Returns the HTTP minor version number as a string.
10.1. 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.

10.1.1. Specifying GPU requirements for a service

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**

1. Create a Knative service and set the GPU resource requirement limit to 1 by using the `--limit nvidia.com/gpu=1` flag:

   ```
   $ kn service create hello --image <service-image> --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.

2. Optional. For an existing service, you can change the GPU resource requirement limit to 3 by using the `--limit nvidia.com/gpu=3` flag:

   ```
   $ kn service update hello --limit nvidia.com/gpu=3
   ```

10.1.2. Additional resources

- For more information about limits, see Setting resource quotas for extended resources.
CHAPTER 11. CLI TOOLS

11.1. 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.

11.1.1. 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
   ```

11.1.2. 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:
   
   ```bash
   # subscription-manager register
   ```

2. Enter the command:
   
   ```bash
   # subscription-manager refresh
   ```

3. Enter the command:
   
   ```bash
   # subscription-manager attach --pool=<pool_id>  
   ```

   1 Pool ID for an active OpenShift Container Platform subscription

4. Enter the command:
   
   ```bash
   # subscription-manager repos --enable="openshift-serverless-1-for-rhel-8-x86_64-rpms"
   ```

5. Enter the command:
   
   ```bash
   # yum install openshift-serverless-clients
   ```

11.1.3. 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:
   
   ```bash
   $ tar -xf <file>
   ```

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

4. To check your PATH, run:
   
   ```bash
   $ 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:

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

11.1.4. 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
   ```

11.1.5. 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

11.1.6. 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>

   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

11.1.7. 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
   ```

### 11.1.8. 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
   ```

### 11.1.9. 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
   ```

### 11.1.10. Customizing the Knative CLI

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**

```yaml
globals:
  plugins:
    path-lookup: true  # 1
    directory: ~/.config/kn/plugins  # 2
  eventing:
    sink-mappings:
      - 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`.

### 11.1.11. Knative CLI 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.

### 11.2. KNATIVE CLI ADVANCED 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.

### 11.2.1. Customizing the Knative CLI

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

```yaml
plugins:
  path-lookup: true
  directory: ~/.config/kn/plugins
  eventing:
    sink-mappings:
      - prefix: svc
        group: core
        version: v1
        resource: services
```

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.

### 11.2.2. Knative CLI plug-ins

The kn CLI supports the use of plug-ins, which enable you to extend the functionality of your kn plugins:
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.

### 11.3. KN FLAGS REFERENCE

#### 11.3.1. Knative CLI --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 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**

```bash
$ 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. KNATIVE SERVING CLI COMMANDS

You can use the following `kn` CLI commands to complete Knative Serving tasks on the cluster.

#### 11.4.1. kn service commands

You can use the following commands to create and manage Knative services.

##### 11.4.1.1. Creating serverless applications by using the Knative 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 the `kn` CLI.

**Procedure**

- Create a Knative service:

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

```bash
$ 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

11.4.1.2. Updating serverless applications by using the Knative CLI

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:
  ```bash
  $ kn service update <service_name> --env <key>=<value>
  ```
- Update a service by adding a new port:
  ```bash
  $ kn service update <service_name> --port 80
  ```
- Update a service by adding new request and limit parameters:
  ```bash
  $ kn service update <service_name> --request cpu=500m --limit memory=1024Mi --limit cpu=1000m
  ```
- Assign the `latest` tag to a revision:
  ```bash
  $ kn service update <service_name> --tag <revision_name>=latest
  ```
- Update a tag from `testing` to `staging` for the latest `READY` revision of a service:
  ```bash
  $ 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:
  ```bash
  $ kn service update <service_name> --tag <revision_name>=test --traffic test=10,@latest=90
  ```
11.4.1.3. 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>
  ```

11.4.1.4. Describing serverless applications by using the Knative CLI

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
```
### 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:
  
  ```bash  
  $ kn service describe <service_name> -o yaml
  ```

- Describe a service in JSON format:
  
  ```bash  
  $ kn service describe <service_name> -o json
  ```

- Print the service URL only:
  
  ```bash  
  $ kn service describe <service_name> -o url
  ```

### 11.4.2. kn domain commands

You can use the following commands to create and manage domain mappings.

### 11.4.2.1. Creating a custom domain mapping by using the Knative CLI

You can use the `kn` CLI to create a **DomainMapping** custom resource (CR) that maps to an Addressable target CR, such as a Knative service or a Knative route.

The `--ref` flag specifies an Addressable target CR for domain mapping.

If a prefix is not provided when using the `--ref` flag, it is assumed that the target is a Knative service in the current namespace. The examples in the following procedure show the prefixes for mapping to a Knative service or a Knative route.

**Prerequisites**
The OpenShift Serverless Operator and Knative Serving are installed on your cluster.

You have created a Knative service or route, and control a custom domain that you want to map to that CR.

**NOTE**

Your custom domain must point to the DNS of the OpenShift Container Platform cluster.

You have installed the kn CLI tool.

### Procedure

- Map a domain to a CR in the current namespace:
  ```
  $ kn domain create <domain_mapping_name> --ref <target_name>
  
  **Example command**
  
  $ kn domain create example-domain-map --ref example-service
  ```

- Map a domain to a Knative service in a specified namespace:
  ```
  $ kn domain create <domain_mapping_name> --ref <ksvc:service_name:service_namespace>
  
  **Example command**
  
  $ kn domain create example-domain-map --ref ksvc:example-service:example-namespace
  ```

- Map a domain to a Knative route:
  ```
  $ kn domain create <domain_mapping_name> --ref <kroute:route_name>
  
  **Example command**
  
  $ kn domain create example-domain-map --ref kroute:example-route
  ```

### 11.4.2.2. Managing custom domain mappings by using the Knative CLI

After you have created a **DomainMapping** custom resource (CR), you can list existing CRs, view information about an existing CR, update CRs, or delete CRs by using the kn CLI.

**Prerequisites**

- The OpenShift Serverless Operator and Knative Serving are installed on your cluster.
- You have created at least one **DomainMapping** CR.
- You have installed the kn CLI tool.
Procedure

- List existing **DomainMapping** CRs:
  
  ```
  $ kn domain list -n <domain_mapping_namespace>
  ```

- View details of an existing **DomainMapping** CR:
  
  ```
  $ kn domain describe <domain_mapping_name>
  ```

- Update a **DomainMapping** CR to point to a new target:
  
  ```
  $ kn domain update --ref <target>
  ```

- Delete a **DomainMapping** CR:
  
  ```
  $ kn domain delete <domain_mapping_name>
  ```

### 11.5. KNATIVE EVENTING CLI COMMANDS

You can use the following `kn` CLI commands to complete Knative Eventing tasks on the cluster.

#### 11.5.1. kn source commands

You can use the following commands to list, create, and manage Knative event sources.

##### 11.5.1.1. 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**

<table>
<thead>
<tr>
<th>TYPE</th>
<th>NAME</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>ApiServerSource</td>
<td>apiserversources.sources.knative.dev</td>
<td>Watch and send Kubernetes API events to a sink</td>
</tr>
<tr>
<td>PingSource</td>
<td>pingsources.sources.knative.dev</td>
<td>Periodically send ping events to a sink</td>
</tr>
<tr>
<td>SinkBinding</td>
<td>sinkbindings.sources.knative.dev</td>
<td>Binding for connecting a PodSpecable to a sink</td>
</tr>
</tbody>
</table>

2. Optional: You can also list the available event source types in YAML format:

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

##### 11.5.1.2. Creating and managing container sources by using the Knative CLI

You can use the following `kn` commands to create and manage container sources:
Create a container source

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

Delete a container source

$ kn source container delete <container_source_name>

Describe a container source

$ kn source container describe <container_source_name>

List existing container sources

$ kn source container list

List existing container sources in YAML format

$ kn source container list -o yaml

Update a container source

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

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

11.5.1.3. Creating an API server source by using 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
```
11.5.1.4. Deleting the API server source by 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:

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

2. Delete the event source:

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

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

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

11.5.1.5. Creating a ping source by using 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:

   ```bash
   $ watch oc get pods
   ```

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

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

**Example output**

```yaml
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!"
  }
```

### 11.5.1.6. 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:

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

### 11.5.1.7. Creating a Kafka event source by using the Knative CLI

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

**IMPORTANT**

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

For more information about the support scope of Red Hat Technology Preview features, see [https://access.redhat.com/support/offerings/techpreview/](https://access.redhat.com/support/offerings/techpreview/).

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

- You have created a project or have access to a project with the appropriate roles and permissions to create applications and other workloads in OpenShift Container Platform.
- You have access to a Red Hat AMQ Streams (Kafka) cluster that produces the Kafka messages you want to import.

**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** CR:

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

   **NOTE**

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

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

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

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

**Example output**

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

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

Conditions:
OK TYPE              AGE REASON
```
Verification steps

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

   ```bash
   $ oc -n kafka run kafka-producer \
   -ti --image=quay.io/strimzi/kafka:latest-kafka-2.7.0 --rm=true \
   --restart=Never -- bin/kafka-console-producer.sh \
   --broker-list <cluster_kafka_bootstrap>:9092 --topic my-topic
   ```

   Enter the message in the prompt. This command assumes that:
   - The Kafka cluster is installed in the `kafka` namespace.
   - The `KafkaSource` object has been configured to use the `my-topic` topic.

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

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

   Example output

   ```yaml
   $ cat func.yaml
   ```

   ```yaml
   ++ Ready            1h
   ++ Deployed         1h
   ++ SinkProvided     1h
   ```

11.6. KN FUNC

11.6.1. Creating functions

You can create a basic serverless function using the `kn` CLI.

You can specify the runtime, trigger, image, and namespace as flags on the command line, or use the `-c` flag to start the interactive experience using the CLI prompt.

The values provided for image and registry are persisted to the `func.yaml` file, so that subsequent invocations do not require the user to specify these again.

Example `func.yaml`
Procedure

- Create a function project:

  $ kn func create <path> -r <registry> -l <runtime> -t <trigger> -i <image> -n <namespace>

  - Supported runtimes include node, go, python, and quarkus.
  - If the image is unspecified, you are prompted for a registry name. The image name is derived from this registry and the function name.

Example command

  $ kn func create functions/example-function

Example output

  Project path: /home/user/functions/example-function
  Function name: example-function
  Runtime: node
  Trigger: http

11.6.2. Building functions

Before you can run a function, you must build the function project by using the `kn func build` command. The build command reads the `func.yaml` file from the function project directory to determine the image name and registry.

Example `func.yaml`

```yaml
name: example-function
namespace: default
runtime: node
image: <image_from_registry>
imageDigest: ""
trigger: http
builder: default
builderMap:
  default: quay.io/boson/faas-nodejs-builder
envs: {}
```

If the image name and registry are not set in the `func.yaml` file, you must either specify the registry flag,
-r when using the `kn func build` command, or you are prompted to provide a registry value in the terminal when building a function. An image name is then derived from the registry value that you have provided.

**Example command using the -r registry flag**

```bash
$ kn func build [-i <image> -r <registry> -p <path>]
```

**Example output**

```
Building function image
Function image has been built, image: quay.io/username/example-function:latest
```

This command creates an OCI container image that can be run locally on your computer, or on a Kubernetes cluster.

**Example using the registry prompt**

```bash
$ kn func build
A registry for function images is required (e.g. 'quay.io/boson').

Registry for function images: quay.io/username
Building function image
Function image has been built, image: quay.io/username/example-function:latest
```

The values for image and registry are persisted to the `func.yaml` file, so that subsequent invocations do not require the user to specify these again.

### 11.6.3. Deploying functions

You can deploy a function to your cluster as a Knative service by using the `kn func deploy` command.

If the targeted function is already deployed, it is updated with a new container image that is pushed to a container image registry, and the Knative service is updated.

**Prerequisites**

- You must have already initialized the function that you want to deploy.

**Procedure**

- Deploy a function:

  ```bash
  $ kn func deploy [-n <namespace> -p <path> -i <image> -r <registry>]
  ```

**Example output**

```
Function deployed at: http://func.example.com
```

- If no `namespace` is specified, the function is deployed in the current namespace.
- The function is deployed from the current directory, unless a `path` is specified.
- The Knative service name is derived from the project name, and cannot be changed using this command.

### 11.6.4. Listing existing functions

You can list existing functions by using `kn func list`. If you want to list functions that have been deployed as Knative services, you can also use `kn service list`.

#### Procedure

- List existing functions:

  ```
  $ kn func list [-n <namespace> -p <path>]
  ```

  **Example output**

<table>
<thead>
<tr>
<th>NAME</th>
<th>NAMESPACE</th>
<th>RUNTIME</th>
<th>URL</th>
</tr>
</thead>
<tbody>
<tr>
<td>example-function</td>
<td>default</td>
<td>node</td>
<td><a href="http://example-function.default.apps.ci-ln-g9f36hb-d5d6b.origin-ci-int-aws.dev.rhcloud.com">http://example-function.default.apps.ci-ln-g9f36hb-d5d6b.origin-ci-int-aws.dev.rhcloud.com</a></td>
</tr>
<tr>
<td>READY</td>
<td></td>
<td></td>
<td>True</td>
</tr>
</tbody>
</table>

- List functions deployed as Knative services:

  ```
  $ kn service list -n <namespace>
  ```

  **Example output**

<table>
<thead>
<tr>
<th>NAME</th>
<th>URL</th>
<th>LATEST</th>
</tr>
</thead>
<tbody>
<tr>
<td>example-function</td>
<td><a href="http://example-function.default.apps.ci-ln-g9f36hb-d5d6b.origin-ci-int-aws.dev.rhcloud.com">http://example-function.default.apps.ci-ln-g9f36hb-d5d6b.origin-ci-int-aws.dev.rhcloud.com</a></td>
<td></td>
</tr>
<tr>
<td>AGE   CONDITIONS</td>
<td>READY   REASON</td>
<td></td>
</tr>
<tr>
<td>example-function</td>
<td>example-function-gzl4c</td>
<td>16m</td>
</tr>
</tbody>
</table>

### 11.6.5. Describing a function

The `kn func describe` command prints information about a deployed function, such as the function name, image, namespace, information about the Knative service, route information, and event subscriptions.

#### Procedure

- Describe a function:

  ```
  $ kn func describe [-f <format> -n <namespace> -p <path>]
  ```

  **Example command**

  ```
  $ kn func describe -p function/example-function
  ```

  **Example output**

  Function name:
11.6.6. Emitting a test event to a deployed function

You can use the `kn func emit` CLI command to emit a CloudEvent to a function that is either deployed locally or deployed to your OpenShift Container Platform cluster. This command can be used to test that a function is working and able to receive events correctly.

**Example command**

```
$ kn func emit
```

The `kn func emit` command executes on the local directory by default, and assumes that this directory is a function project.

### 11.6.6.1. `kn func emit` optional parameters

You can specify optional parameters for the emitted CloudEvent by using the `kn func emit` CLI command flags.

**List of flags from --help command output**

<table>
<thead>
<tr>
<th>Flags</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-c, --content-type</td>
<td>The MIME Content-Type for the CloudEvent data (Env: $FUNC_CONTENT_TYPE)</td>
</tr>
<tr>
<td></td>
<td>(default &quot;application/json&quot;)</td>
</tr>
<tr>
<td>-d, --data</td>
<td>Any arbitrary string to be sent as the CloudEvent data. Ignored if --file is provided (Env: $FUNC_DATA)</td>
</tr>
<tr>
<td>-f, --file</td>
<td>Path to a local file containing CloudEvent data to be sent (Env: $FUNC_FILE)</td>
</tr>
<tr>
<td>-h, --help</td>
<td>help for emit</td>
</tr>
<tr>
<td>-i, --id</td>
<td>CloudEvent ID (Env: $FUNC_ID) (default &quot;306bd6a0-0b0a-48ba-b187-b633571d072a&quot;)</td>
</tr>
<tr>
<td>-p, --path</td>
<td>Path to the project directory. Ignored when --sink is provided (Env: $FUNC_PATH) (default &quot;/home/lanceball/src/github.com/nodeshift/opossum&quot;)</td>
</tr>
<tr>
<td>-k, --sink</td>
<td>Send the CloudEvent to the function running at [sink]. The special value &quot;local&quot; can be used to send the event to a function running on the local host. When provided, the --path flag is ignored (Env: $FUNC_SINK)</td>
</tr>
<tr>
<td>-s, --source</td>
<td>CloudEvent source (Env: $FUNC_SOURCE) (default &quot;/bosen/fn&quot;)</td>
</tr>
<tr>
<td>-t, --type</td>
<td>CloudEvent type (Env: $FUNC_TYPE) (default &quot;bosen.fn&quot;)</td>
</tr>
</tbody>
</table>

In particular, you might find it useful to specify the following parameters:

**Event type**

The type of event being emitted. You can find information about the `type` parameter that is set for events from a certain event producer in the documentation for that event producer. For example, the API server source may set the `type` parameter of produced events as `dev.knative.apiserver.resource.update`.
Event source
The unique event source that produced the event. This may be a URI for the event source, for example https://10.96.0.1/, or the name of the event source.

Event ID
A random, unique ID that is created by the event producer.

Event data
Allows you to specify a data value for the event sent by the kn func emit command. For example, you can specify a --data value such as "Hello world!" so that the event contains this data string. By default, no data is included in the events created by kn func emit.

NOTE
Functions that have been deployed to a cluster can respond to events from an existing event source that provides values for properties such as source and type. These events often have a data value in JSON form, which captures the domain specific context of the event. Using the CLI flags noted in this document, developers can simulate those events for local testing.

You can also send event data using the --file flag to provide a local file containing data for the event.

Data content type
If you are using the --data flag to add data for events, you can also specify what type of data is carried by the event, by using the --content-type flag. In the previous example, the data is plain text, so you might specify kn func emit --data "Hello world!" --content-type "text/plain".

Example commands specifying event parameters by using flags

```
$ kn func emit --type <event_type> --source <event_source> --data <event_data> --content-type <content_type> -i <event_ID>

$ kn func emit --type ping --source example-ping --data "Hello world!" --content-type "text/plain" -i example-ID
```

Example commands specifying a file on disk that contains the event parameters

```
$ kn func emit --file <path>

$ kn func emit --file ./test.json
```

Example commands specifying a path to the function
You can specify a path to the function project by using the --path flag, or specify an endpoint for the function by using the --sink flag:

```
$ kn func emit --path <path_to_function>

$ kn func emit --path ./example/example-function
```

Example commands specifying a function deployed as a Knative service (sink)
$ kn func emit --sink <service_URL>

$ kn func emit --sink "http://example.function.com"

The **--sink** flag also accepts the special value **local** to send an event to a function running locally:

$ kn func emit --sink local

### 11.6.7. Deleting a function

You can delete a function from your cluster by using the `kn func delete` command.

**Procedure**

- Delete a function:

  ```shell
  $ kn func delete [<function_name> -n <namespace> -p <path>]
  ```

  - If the name or path of the function to delete is not specified, the current directory is searched for a `func.yaml` file that is used to determine the function to delete.
  
  - If the namespace is not specified, it defaults to the `namespace` value in the `func.yaml` file.