Red Hat OpenShift Serverless 1.32

Serving

Getting started with Knative Serving and configuring services

Last Updated: 2024-03-18

Red Hat OpenShift Serverless 1.32 Serving

Getting started with Knative Serving and configuring services

Legal Notice

Abstract

This document provides information on getting started with Knative Serving. It shows how to configure applications and covers features such as autoscaling, traffic splitting, and external and ingress routing.

Table of Contents

CHAPTER 1. GETTING STARTED WITH KNATIVE SERVING

1.1. SERVERLESS APPLICATIONS

Serverless applications are created and deployed as Kubernetes services, defined by a route and a configuration, and contained in a YAML file. To deploy a serverless application using OpenShift Serverless, you must create a Knative **Service** object.

Example Knative Service object YAML file

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: showcase 1
  namespace: default 2
spec:
  template:
    spec:
      containers:
        - image: quay.io/openshift-knative/showcase 3
          env:
            - name: GREET 4
              value: Ciao
```

1. The name of the application.
2. The namespace the application uses.
3. The image of the application.
4. The environment variable printed out by the sample application.

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

- Create a Knative service from the OpenShift Container Platform web console. For OpenShift Container Platform, see Creating applications using the Developer perspective for more information.
- Create a Knative service by using the Knative (**kn**) CLI.
- Create and apply a Knative **Service** object as a YAML file, by using the **oc** CLI.

1.1.1. Creating serverless applications by using the Knative CLI

Using the Knative (**kn**) CLI to create serverless applications provides a more streamlined and intuitive user interface over modifying YAML files directly. You can use the **kn service create** command to create a basic serverless application.

Prerequisites

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

- You have created a project or have access to a project with the appropriate roles and permissions to create applications and other workloads in OpenShift Container Platform.

Procedure

- Create a Knative service:

```
$ kn service create <service-name> --image <image> --tag <tag-value>
```

  Where:

  - **--image** is the URI of the image for the application.
  - **--tag** is an optional flag that can be used to add a tag to the initial revision that is created with the service.

Example command

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

Example output

```
Creating service 'showcase' in namespace 'default':

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

Service 'showcase' created with latest revision 'showcase-00001' and URL:
http://showcase-default.apps-crc.testing
```

1.1.2. Creating serverless applications using YAML

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

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.

Prerequisites

- OpenShift Serverless Operator and Knative Serving 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.
- Install the OpenShift CLI (**oc**).

Procedure

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

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: showcase
  namespace: default
spec:
  template:
    spec:
      containers:
        - image: quay.io/openshift-knative/showcase
          env:
            - name: GREET
              value: Bonjour
```

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

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

If you do not want to switch to the **Developer** perspective in the OpenShift Container Platform web console or use the Knative (**kn**) CLI or YAML files, you can create Knative components by using the **Administator** perspective of the OpenShift Container Platform web console.

1.1.3. Creating serverless applications using the Administrator perspective

Serverless applications are created and deployed as Kubernetes services, defined by a route and a configuration, and contained in a YAML file. To deploy a serverless application using OpenShift Serverless, you must create a Knative **Service** object.

Example Knative Service object YAML file

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: showcase (1)
  namespace: default (2)
spec:
  template:
    spec:
      containers:
        - image: quay.io/openshift-knative/showcase (3)
          env:
            - name: GREET (4)
              value: Ciao
```

(1) The name of the application.

(2) The namespace the application uses.

(3) The image of the application.

4 The environment variable printed out by the sample application.

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.

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**.

1.1.4. Creating a service using offline mode

You can execute **kn service** commands in offline mode, so that no changes happen on the cluster, and instead the service descriptor file is created on your local machine. After the descriptor file is created, you can modify the file before propagating changes to the cluster.

IMPORTANT

The offline mode of the Knative 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 Technology Preview Features Support Scope .

Prerequisites

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

Procedure

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

```
$ kn service create showcase \
```

```
  --image quay.io/openshift-knative/showcase \
  --target ./ \
  --namespace test
```

Example output

```
Service 'showcase' 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 ./
```

Example output

```
./
└── test
    └── ksvc
        └── showcase.yaml

2 directories, 1 file
```

- 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 **showcase.yaml**, named according to the specified service name.

3. Examine the generated service descriptor file:

```
$ cat test/ksvc/showcase.yaml
```

Example output

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  creationTimestamp: null
  name: showcase
```

```
  namespace: test
spec:
  template:
    metadata:
      annotations:
        client.knative.dev/user-image: quay.io/openshift-knative/showcase
      creationTimestamp: null
    spec:
      containers:
      - image: quay.io/openshift-knative/showcase
        name: ""
        resources: {}
status: {}
```

4. List information about the new service:

```
$ kn service describe showcase --target ./ --namespace test
```

Example output

```
Name:       showcase
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/showcase.yaml
```

Example output

```
Creating service 'showcase' in namespace 'test':

  0.058s The Route is still working to reflect the latest desired specification.
  0.098s ...
  0.168s Configuration "showcase" 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 'showcase' created to latest revision 'showcase-00001' is available at URL:
http://showcase-test.apps.example.com
```

1.1.5. Additional resources

- Knative Serving CLI commands
- Configuring JSON Web Token authentication for Knative services

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

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

Prerequisites

- OpenShift Serverless Operator and Knative Serving are installed on your cluster.
- You have installed the **oc** CLI.
- You have created a Knative service.

Prerequisites

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

Procedure

1. Find the application URL:

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

Example output

```
NAME      URL                                   LATESTCREATED   LATESTREADY     READY
REASON
showcase  http://showcase-default.example.com   showcase-00001  showcase-00001  True
```

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

Example HTTP request (using HTTPie tool)

```
$ http showcase-default.example.com
```

Example HTTPS request

```
$ https showcase-default.example.com
```

Example output

```
HTTP/1.1 200 OK
Content-Type: application/json
Server: Quarkus/2.13.7.Final-redhat-00003 Java/17.0.7
X-Config: {"sink":"http://localhost:31111","greet":"Ciao","delay":0}
X-Version: v0.7.0-4-g23d460f
content-length: 49

{
    "artifact": "knative-showcase",
    "greeting": "Ciao"
}
```

3. Optional. If you don't have the HTTPie tool installed on your system, you can likely use curl tool instead:

Example HTTPS request

```
$ curl http://showcase-default.example.com
```

Example output

```
{"artifact":"knative-showcase","greeting":"Ciao"}
```

4. Optional. If you receive an error relating to a self-signed certificate in the certificate chain, you can add the **--verify=no** flag to the HTTPie command to ignore the error:

```
$ https --verify=no showcase-default.example.com
```

Example output

```
HTTP/1.1 200 OK
Content-Type: application/json
Server: Quarkus/2.13.7.Final-redhat-00003 Java/17.0.7
X-Config: {"sink":"http://localhost:31111","greet":"Ciao","delay":0}
X-Version: v0.7.0-4-g23d460f
content-length: 49

{
    "artifact": "knative-showcase",
    "greeting": "Ciao"
}
```

IMPORTANT

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

5. 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://showcase-default.example.com --cacert <file>
```

Example output

```
{"artifact":"knative-showcase","greeting":"Ciao"}
```

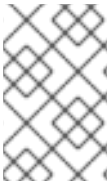
CHAPTER 2. AUTOSCALING

2.1. AUTOSCALING

Knative Serving provides automatic scaling, or *autoscaling*, for applications to match incoming demand. For example, if an application is receiving no traffic, and scale-to-zero is enabled, Knative Serving scales the application down to zero replicas. If scale-to-zero is disabled, the application is scaled down to the minimum number of replicas configured for applications on the cluster. Replicas can also be scaled up to meet demand if traffic to the application increases.

Autoscaling settings for Knative services can be global settings that are configured by cluster administrators (or dedicated administrators for Red Hat OpenShift Service on AWS and OpenShift Dedicated), or per-revision settings that are configured for individual services.

You can modify per-revision settings for your services by using the OpenShift Container Platform web console, by modifying the YAML file for your service, or by using the Knative (**kn**) CLI.



NOTE

Any limits or targets that you set for a service are measured against a single instance of your application. For example, setting the **target** annotation to **50** configures the autoscaler to scale the application so that each revision handles 50 requests at a time.

2.2. SCALE BOUNDS

Scale bounds determine the minimum and maximum numbers of replicas that can serve an application at any given time. You can set scale bounds for an application to help prevent cold starts or control computing costs.

2.2.1. Minimum scale bounds

The minimum number of replicas that can serve an application is determined by the **min-scale** annotation. If scale to zero is not enabled, the **min-scale** value defaults to **1**.

The **min-scale** value defaults to **0** replicas if the following conditions are met:

- The **min-scale** annotation is not set
- Scaling to zero is enabled
- The class **KPA** is used

Example service spec with min-scale annotation

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

```
      annotations:
        autoscaling.knative.dev/min-scale: "0"
...
```

2.2.1.1. Setting the min-scale annotation by using the Knative CLI

Using the Knative (**kn**) CLI to set the **min-scale** annotation provides a more streamlined and intuitive user interface over modifying YAML files directly. You can use the **kn service** command with the **--scale-min** flag to create or modify the **min-scale** value for a service.

Prerequisites

- Knative Serving is installed on the cluster.
- You have installed the Knative (**kn**) CLI.

Procedure

- Set the minimum number of replicas for the service by using the **--scale-min** flag:

```
$ kn service create <service_name> --image <image_uri> --scale-min <integer>
```

Example command

```
$ kn service create showcase --image quay.io/openshift-knative/showcase --scale-min 2
```

2.2.2. Maximum scale bounds

The maximum number of replicas that can serve an application is determined by the **max-scale** annotation. If the **max-scale** annotation is not set, there is no upper limit for the number of replicas created.

Example service spec with max-scale annotation

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: showcase
  namespace: default
spec:
  template:
    metadata:
      annotations:
        autoscaling.knative.dev/max-scale: "10"
...
```

2.2.2.1. Setting the max-scale annotation by using the Knative CLI

Using the Knative (**kn**) CLI to set the **max-scale** annotation provides a more streamlined and intuitive user interface over modifying YAML files directly. You can use the **kn service** command with the **--scale-max** flag to create or modify the **max-scale** value for a service.

Prerequisites

- Knative Serving is installed on the cluster.
- You have installed the Knative (**kn**) CLI.

Procedure

- Set the maximum number of replicas for the service by using the **--scale-max** flag:

```
$ kn service create <service_name> --image <image_uri> --scale-max <integer>
```

Example command

```
$ kn service create showcase --image quay.io/openshift-knative/showcase --scale-max 10
```

2.3. CONCURRENCY

Concurrency determines the number of simultaneous requests that can be processed by each replica of an application at any given time. Concurrency can be configured as a *soft limit* or a *hard limit*:

- A soft limit is a targeted requests limit, rather than a strictly enforced bound. For example, if there is a sudden burst of traffic, the soft limit target can be exceeded.
- A hard limit is a strictly enforced upper bound requests limit. If concurrency reaches the hard limit, surplus requests are buffered and must wait until there is enough free capacity to execute the requests.

IMPORTANT

Using a hard limit configuration is only recommended if there is a clear use case for it with your application. Having a low, hard limit specified may have a negative impact on the throughput and latency of an application, and might cause cold starts.

Adding a soft target and a hard limit means that the autoscaler targets the soft target number of concurrent requests, but imposes a hard limit of the hard limit value for the maximum number of requests.

If the hard limit value is less than the soft limit value, the soft limit value is tuned down, because there is no need to target more requests than the number that can actually be handled.

2.3.1. Configuring a soft concurrency target

A soft limit is a targeted requests limit, rather than a strictly enforced bound. For example, if there is a sudden burst of traffic, the soft limit target can be exceeded. You can specify a soft concurrency target for your Knative service by setting the **autoscaling.knative.dev/target** annotation in the spec, or by using the **kn service** command with the correct flags.

Procedure

- Optional: Set the **autoscaling.knative.dev/target** annotation for your Knative service in the spec of the **Service** custom resource:

Example service spec

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: showcase
  namespace: default
spec:
  template:
    metadata:
      annotations:
        autoscaling.knative.dev/target: "200"
```

- Optional: Use the **kn service** command to specify the **--concurrency-target** flag:

```
$ kn service create <service_name> --image <image_uri> --concurrency-target <integer>
```

Example command to create a service with a concurrency target of 50 requests

```
$ kn service create showcase --image quay.io/openshift-knative/showcase --concurrency-target 50
```

2.3.2. Configuring a hard concurrency limit

A hard concurrency limit is a strictly enforced upper bound requests limit. If concurrency reaches the hard limit, surplus requests are buffered and must wait until there is enough free capacity to execute the requests. You can specify a hard concurrency limit for your Knative service by modifying the **containerConcurrency** spec, or by using the **kn service** command with the correct flags.

Procedure

- Optional: Set the **containerConcurrency** spec for your Knative service in the spec of the **Service** custom resource:

Example service spec

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: showcase
  namespace: default
spec:
  template:
    spec:
      containerConcurrency: 50
```

The default value is **0**, which means that there is no limit on the number of simultaneous requests that are permitted to flow into one replica of the service at a time.

A value greater than **0** specifies the exact number of requests that are permitted to flow into one replica of the service at a time. This example would enable a hard concurrency limit of 50 requests.

- Optional: Use the **kn service** command to specify the **--concurrency-limit** flag:

```
$ kn service create <service_name> --image <image_uri> --concurrency-limit <integer>
```

Example command to create a service with a concurrency limit of 50 requests

```
$ kn service create showcase --image quay.io/openshift-knative/showcase --concurrency-limit 50
```

2.3.3. Concurrency target utilization

This value specifies the percentage of the concurrency limit that is actually targeted by the autoscaler. This is also known as specifying the *hotness* at which a replica runs, which enables the autoscaler to scale up before the defined hard limit is reached.

For example, if the **containerConcurrency** value is set to 10, and the **target-utilization-percentage** value is set to 70 percent, the autoscaler creates a new replica when the average number of concurrent requests across all existing replicas reaches 7. Requests numbered 7 to 10 are still sent to the existing replicas, but additional replicas are started in anticipation of being required after the **containerConcurrency** value is reached.

Example service configured using the target-utilization-percentage annotation

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: showcase
  namespace: default
spec:
  template:
    metadata:
      annotations:
        autoscaling.knative.dev/target-utilization-percentage: "70"
...
```

2.4. SCALE-TO-ZERO

Knative Serving provides automatic scaling, or *autoscaling*, for applications to match incoming demand.

2.4.1. Enabling scale-to-zero

You can use the **enable-scale-to-zero** spec to enable or disable scale-to-zero globally for applications on the cluster.

Prerequisites

- You have installed OpenShift Serverless Operator and Knative Serving on your cluster.
- You have cluster administrator permissions on OpenShift Container Platform, or you have cluster or dedicated administrator permissions on Red Hat OpenShift Service on AWS or OpenShift Dedicated.

- You are using the default Knative Pod Autoscaler. The scale to zero feature is not available if you are using the Kubernetes Horizontal Pod Autoscaler.

Procedure

- Modify the **enable-scale-to-zero** spec in the **KnativeServing** custom resource (CR):

  Example KnativeServing CR

```
apiVersion: operator.knative.dev/v1beta1
kind: KnativeServing
metadata:
  name: knative-serving
spec:
  config:
    autoscaler:
      enable-scale-to-zero: "false" 1
```

1 The **enable-scale-to-zero** spec can be either **"true"** or **"false"**. If set to true, scale-to-zero is enabled. If set to false, applications are scaled down to the configured *minimum scale bound*. The default value is **"true"**.

2.4.2. Configuring the scale-to-zero grace period

Knative Serving provides automatic scaling down to zero pods for applications. You can use the **scale-to-zero-grace-period** spec to define an upper bound time limit that Knative waits for scale-to-zero machinery to be in place before the last replica of an application is removed.

Prerequisites

- You have installed OpenShift Serverless Operator and Knative Serving on your cluster.
- You have cluster administrator permissions on OpenShift Container Platform, or you have cluster or dedicated administrator permissions on Red Hat OpenShift Service on AWS or OpenShift Dedicated.
- You are using the default Knative Pod Autoscaler. The scale-to-zero feature is not available if you are using the Kubernetes Horizontal Pod Autoscaler.

Procedure

- Modify the **scale-to-zero-grace-period** spec in the **KnativeServing** custom resource (CR):

  Example KnativeServing CR

```
apiVersion: operator.knative.dev/v1beta1
kind: KnativeServing
metadata:
  name: knative-serving
spec:
  config:
    autoscaler:
      scale-to-zero-grace-period: "30s" 1
```

1 The grace period time in seconds. The default value is 30 seconds.

CHAPTER 3. CONFIGURING SERVERLESS APPLICATIONS

3.1. MULTI-CONTAINER SUPPORT FOR SERVING

You can deploy a multi-container pod by using a single Knative service. This method is useful for separating application responsibilities into smaller, specialized parts.

3.1.1. Configuring a multi-container service

Multi-container support is enabled by default. You can create a multi-container pod by specifiying multiple containers in the service.

Procedure

1. Modify your service to include additional containers. Only one container can handle requests, so specify **ports** for exactly one container. Here is an example configuration with two containers:

   Multiple containers configuration

```
apiVersion: serving.knative.dev/v1
kind: Service
...
spec:
  template:
    spec:
      containers:
        - name: first-container 1
          image: gcr.io/knative-samples/helloworld-go
          ports:
            - containerPort: 8080 2
        - name: second-container 3
          image: gcr.io/knative-samples/helloworld-java
```

1 First container configuration.

2 Port specification for the first container.

3 Second container configuration.

3.2. EMPTYDIR VOLUMES

emptyDir volumes are empty volumes that are created when a pod is created, and are used to provide temporary working disk space. **emptyDir** volumes are deleted when the pod they were created for is deleted.

3.2.1. Configuring the EmptyDir extension

The **kubernetes.podspec-volumes-emptydir** extension controls whether **emptyDir** volumes can be used with Knative Serving. To enable using **emptyDir** volumes, you must modify the **KnativeServing** custom resource (CR) to include the following YAML:

Example KnativeServing CR

```
apiVersion: operator.knative.dev/v1beta1
kind: KnativeServing
metadata:
  name: knative-serving
spec:
  config:
    features:
      kubernetes.podspec-volumes-emptydir: enabled
...
```

3.3. PERSISTENT VOLUME CLAIMS FOR SERVING

Some serverless applications require permanent data storage. By configuring different volume types, you can provide data storage for Knative services. Serving supports mounting of the volume types such as **secret**, **configMap**, **projected**, and **emptyDir**.

You can configure persistent volume claims (PVCs) for your Knative services. The Persistent volume types are implemented as plugins. To determine if there are any persistent volume types available, you can check the available or installed storage classes in your cluster. Persistent volumes are supported, but require a feature flag to be enabled.

WARNING

The mounting of large volumes can lead to a considerable delay in the start time of the application.

3.3.1. Enabling PVC support

Procedure

1. To enable Knative Serving to use PVCs and write to them, modify the **KnativeServing** custom resource (CR) to include the following YAML:

   Enabling PVCs with write access

   ```
   ...
   spec:
     config:
       features:
         "kubernetes.podspec-persistent-volume-claim": enabled
         "kubernetes.podspec-persistent-volume-write": enabled
   ...
   ```

   - The **kubernetes.podspec-persistent-volume-claim** extension controls whether persistent volumes (PVs) can be used with Knative Serving.
   - The **kubernetes.podspec-persistent-volume-write** extension controls whether PVs are available to Knative Serving with the write access.

2. To claim a PV, modify your service to include the PV configuration. For example, you might have a persistent volume claim with the following configuration:

NOTE

Use the storage class that supports the access mode you are requesting. For example, you can use the **ocs-storagecluster-cephfs** storage class for the **ReadWriteMany** access mode.

The **ocs-storagecluster-cephfs** storage class is supported and comes from Red Hat OpenShift Data Foundation.

PersistentVolumeClaim configuration

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: example-pv-claim
  namespace: my-ns
spec:
  accessModes:
    - ReadWriteMany
  storageClassName: ocs-storagecluster-cephfs
  resources:
    requests:
      storage: 1Gi
```

In this case, to claim a PV with write access, modify your service as follows:

Knative service PVC configuration

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  namespace: my-ns
...
spec:
 template:
   spec:
     containers:
         ...
         volumeMounts: (1)
           - mountPath: /data
             name: mydata
             readOnly: false
     volumes:
       - name: mydata
         persistentVolumeClaim: (2)
           claimName: example-pv-claim
           readOnly: false (3)
```

(1) Volume mount specification.

(2) Persistent volume claim specification.

3 Flag that enables read-only access.

NOTE

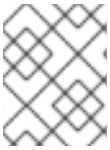
To successfully use persistent storage in Knative services, you need additional configuration, such as the user permissions for the Knative container user.

3.3.2. Additional resources for OpenShift Container Platform

- Understanding persistent storage

3.4. INIT CONTAINERS

Init containers are specialized containers that are run before application containers in a pod. They are generally used to implement initialization logic for an application, which may include running setup scripts or downloading required configurations. You can enable the use of init containers for Knative services by modifying the **KnativeServing** custom resource (CR).



NOTE

Init containers may cause longer application start-up times and should be used with caution for serverless applications, which are expected to scale up and down frequently.

3.4.1. Enabling init containers

Prerequisites

- You have installed OpenShift Serverless Operator and Knative Serving on your cluster.
- You have cluster administrator permissions on OpenShift Container Platform, or you have cluster or dedicated administrator permissions on Red Hat OpenShift Service on AWS or OpenShift Dedicated.

Procedure

- Enable the use of init containers by adding the **kubernetes.podspec-init-containers** flag to the **KnativeServing** CR:

  Example KnativeServing CR

```
apiVersion: operator.knative.dev/v1beta1
kind: KnativeServing
metadata:
  name: knative-serving
spec:
  config:
    features:
      kubernetes.podspec-init-containers: enabled
...
```

3.5. RESOLVING IMAGE TAGS TO DIGESTS

If the Knative Serving controller has access to the container registry, Knative Serving resolves image tags to a digest when you create a revision of a service. This is known as *tag-to-digest resolution*, and helps to provide consistency for deployments.

3.5.1. Tag-to-digest resolution

To give the controller access to the container registry on OpenShift Container Platform, you must create a secret and then configure controller custom certificates. You can configure controller custom certificates by modifying the **controller-custom-certs** spec in the **KnativeServing** custom resource (CR). The secret must reside in the same namespace as the **KnativeServing** CR.

If a secret is not included in the **KnativeServing** CR, this setting defaults to using public key infrastructure (PKI). When using PKI, the cluster-wide certificates are automatically injected into the Knative Serving controller by using the **config-service-sa** config map. The OpenShift Serverless Operator populates the **config-service-sa** config map with cluster-wide certificates and mounts the config map as a volume to the controller.

3.5.1.1. Configuring tag-to-digest resolution by using a secret

If the **controller-custom-certs** spec uses the **Secret** type, the secret is mounted as a secret volume. Knative components consume the secret directly, assuming that the secret has the required certificates.

Prerequisites

- You have cluster administrator permissions on OpenShift Container Platform, or you have cluster or dedicated administrator permissions on Red Hat OpenShift Service on AWS or OpenShift Dedicated.
- You have installed the OpenShift Serverless Operator and Knative Serving on your cluster.

Procedure

1. Create a secret:

   Example command

   ```
   $ oc -n knative-serving create secret generic custom-secret --from-file=<secret_name>.crt=
   <path_to_certificate>
   ```

2. Configure the **controller-custom-certs** spec in the **KnativeServing** custom resource (CR) to use the **Secret** type:

   Example KnativeServing CR

   ```
   apiVersion: operator.knative.dev/v1beta1
   kind: KnativeServing
   metadata:
     name: knative-serving
     namespace: knative-serving
   spec:
     controller-custom-certs:
       name: custom-secret
       type: Secret
   ```

3.6. CONFIGURING TLS AUTHENTICATION

You can use *Transport Layer Security* (TLS) to encrypt Knative traffic and for authentication.

TLS is the only supported method of traffic encryption for Knative Kafka. Red Hat recommends using both SASL and TLS together for Knative broker for Apache Kafka resources.

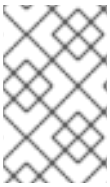
NOTE

If you want to enable internal TLS with a Red Hat OpenShift Service Mesh integration, you must enable Service Mesh with mTLS instead of the internal encryption explained in the following procedure.

For OpenShift Container Platform and Red Hat OpenShift Service on AWS, see the documentation for Enabling Knative Serving metrics when using Service Mesh with mTLS .

3.6.1. Enabling TLS authentication for internal traffic

OpenShift Serverless supports TLS edge termination by default, so that HTTPS traffic from end users is encrypted. However, internal traffic behind the OpenShift route is forwarded to applications by using plain data. By enabling TLS for internal traffic, the traffic sent between components is encrypted, which makes this traffic more secure.



NOTE

If you want to enable internal TLS with a Red Hat OpenShift Service Mesh integration, you must enable Service Mesh with mTLS instead of the internal encryption explained in the following procedure.

IMPORTANT

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

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

Prerequisites

- You have installed the OpenShift Serverless Operator and Knative Serving.
- You have installed the OpenShift (**oc**) CLI.

Procedure

1. Create or update your **KnativeServing** resource and make sure that it includes the **internal-encryption: "true"** field in the spec:

```
...
spec:
```

```
  config:
    network:
      internal-encryption: "true"
...
```

2. Restart the activator pods in the **knative-serving** namespace to load the certificates:

```
$ oc delete pod -n knative-serving --selector app=activator
```

Additional resources

- Configuring TLS authentication for the Knative broker for Apache Kafka
- Configuring TLS authentication for channels for Apache Kafka
- Enabling Knative Serving metrics when using Service Mesh with mTLS

3.7. CONFIGURING KOURIER

Kourier is a lightweight Kubernetes-native Ingress for Knative Serving. Kourier acts as a gateway for Knative, routing HTTP traffic to Knative services.

3.7.1. Customizing kourier-bootstrap for Kourier getaways

The Envoy proxy component in Kourier handles inbound and outbound HTTP traffic for the Knative services. By default, Kourier contains an Envoy bootstrap configuration in the **kourier-bootstrap** configuration map in the **knative-serving-ingress** namespace. You can change this configuration.

Prerequisites

- You have installed the OpenShift Serverless Operator and Knative Serving.
- You have cluster administrator permissions on OpenShift Container Platform, or you have cluster or dedicated administrator permissions on Red Hat OpenShift Service on AWS or OpenShift Dedicated.

Procedure

- Specify a custom bootstrapping configuration map by changing the **spec.ingress.kourier.bootstrap-configmap** field in the **KnativeServing** custom resource (CR):

  Example KnativeServing CR

```
apiVersion: operator.knative.dev/v1beta1
kind: KnativeServing
metadata:
  name: knative-serving
  namespace: knative-serving
spec:
  config:
    network:
      ingress-class: kourier.ingress.networking.knative.dev
  ingress:
```

```
    kourier:
      bootstrap-configmap: my-configmap
      enabled: true
# ...
```

3.8. RESTRICTIVE NETWORK POLICIES

3.8.1. Clusters 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

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: deny-by-default
  namespace: example-namespace
spec:
  podSelector:
  ingress: []
```

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

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. JSON Web Token authentication for Knative services requires Service Mesh.

Prerequisites

- Install the OpenShift CLI (**oc**).
- OpenShift Serverless Operator and Knative Serving are installed on your cluster.

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

   ```
   apiVersion: networking.k8s.io/v1
   kind: NetworkPolicy
   metadata:
     name: <network_policy_name> 1
     namespace: <namespace> 2
   spec:
     ingress:
     - from:
       - namespaceSelector:
           matchLabels:
             knative.openshift.io/system-namespace: "true"
     podSelector: {}
     policyTypes:
     - Ingress
   ```

   1 Provide a name for your network policy.

   2 The namespace where your application exists.

CHAPTER 4. DEBUGGING SERVERLESS APPLICATIONS

You can use a variety of methods to troubleshoot a Serverless application.

4.1. CHECKING TERMINAL OUTPUT

You can check your deploy command output to see whether deployment succeeded or not. If your deployment process was terminated, you should see an error message in the output that describes the reason why the deployment failed. This kind of failure is most likely due to either a misconfigured manifest or an invalid command.

Procedure

- Open the command output on the client where you deploy and manage your application. The following example is an error that you might see after a failed **oc apply** command:

```
Error from server (InternalError): error when applying patch:
{"metadata":{"annotations":{"kubectl.kubernetes.io/last-applied-configuration":"
{\"apiVersion\":\"serving.knative.dev/v1\",\"kind\":\"Route\",\"metadata\":{\"annotations\":
{},\"name\":\"route-example\",\"namespace\":\"default\"},\"spec\":{\"traffic\":
[{\"configurationName\":\"configuration-example\",\"percent\":50}]}}\n"}},"spec":{"traffic":
[{"configurationName":"configuration-example","percent":50}]}}
to:
&{0xc421d98240 0xc421e77490 default route-example STDIN 0xc421db0488 264682 false}
for: "STDIN": Internal error occurred: admission webhook "webhook.knative.dev" denied the
request: mutation failed: The route must have traffic percent sum equal to 100.
ERROR: Non-zero return code '1' from command: Process exited with status 1
```

This output indicates that you must configure the route traffic percent to be equal to 100.

4.2. CHECKING POD STATUS

You might need to check the status of your **Pod** object to identify the issue with your Serverless application.

Procedure

1. List all pods for your deployment by running the following command:

```
$ oc get pods
```

Example output

```
NAME                                             READY     STATUS             RESTARTS   AGE
configuration-example-00001-deployment-659747ff99-9bvr4   2/2       Running            0
3h
configuration-example-00002-deployment-5f475b7849-gxcht   1/2       CrashLoopBackOff   2
36s
```

In the output, you can see all pods with selected data about their status.

2. View the detailed information on the status of a pod by running the following command:

Example output

```
$ oc get pod <pod_name> --output yaml
```

In the output, the **conditions** and **containerStatuses** fields might be particularly useful for debugging.

4.3. CHECKING REVISION STATUS

You might need to check the status of your revision to identify the issue with your Serverless application.

Procedure

1. If you configure your route with a **Configuration** object, get the name of the **Revision** object created for your deployment by running the following command:

```
$ oc get configuration <configuration_name> --output jsonpath="
{.status.latestCreatedRevisionName}"
```

You can find the configuration name in the **Route.yaml** file, which specifies routing settings by defining an OpenShift **Route** resource.

If you configure your route with revision directly, look up the revision name in the **Route.yaml** file.

2. Query for the status of the revision by running the following command:

```
$ oc get revision <revision-name> --output yaml
```

A ready revision should have the **reason: ServiceReady**, **status: "True"**, and **type: Ready** conditions in its status. If these conditions are present, you might want to check pod status or Istio routing. Otherwise, the resource status contains the error message.

4.3.1. Additional resources

- Route configuration

4.4. CHECKING INGRESS STATUS

You might need to check the status of your Ingress to identify the issue with your Serverless application.

Procedure

- Check the IP address of your Ingress by running the following command:

```
$ oc get svc -n istio-system istio-ingressgateway
```

The **istio-ingressgateway** service is the **LoadBalancer** service used by Knative.

If there is no external IP address, run the following command:

```
$ oc describe svc istio-ingressgateway -n istio-system
```

This command prints the reason why IP addresses were not provisioned. Most likely, it is due to a quota issue.

4.5. CHECKING ROUTE STATUS

In some cases, the **Route** object has issues. You can check its status by using the OpenShift CLI (**oc**).

Procedure

- View the status of the **Route** object with which you deployed your application by running the following command:

```
$ oc get route <route_name> --output yaml
```

Substitute **<route_name>** with the name of your **Route** object.

The **conditions** object in the **status** object states the reason in case of a failure.

4.6. CHECKING INGRESS AND ISTIO ROUTING

Sometimes, when Istio is used as an Ingress layer, the Ingress and Istio routing have issues. You can see the details on them by using the OpenShift CLI (**oc**).

Procedure

1. List all Ingress resources and their corresponding labels by running the following command:

```
$ oc get ingresses.networking.internal.knative.dev -o=custom-
columns='NAME:.metadata.name,LABELS:.metadata.labels'
```

Example output

```
NAME           LABELS
helloworld-go  map[serving.knative.dev/route:helloworld-go
serving.knative.dev/routeNamespace:default serving.knative.dev/service:helloworld-go]
```

In this output, labels **serving.knative.dev/route** and **serving.knative.dev/routeNamespace** indicate the **Route** where the Ingress resource resides. Your **Route** and Ingress should be listed.

If your Ingress does not exist, the route controller assumes that the **Revision** objects targeted by your **Route** or **Service** object are not ready. Proceed with other debugging procedures to diagnose **Revision** readiness status.

2. If your Ingress is listed, examine the **ClusterIngress** object created for your route by running the following command:

```
$ oc get ingresses.networking.internal.knative.dev <ingress_name> --output yaml
```

In the status section of the output, if the condition with **type=Ready** has the status of **True**, then Ingress is working correctly. Otherwise, the output contains error messages.

3. If Ingress has the status of **Ready**, then there is a corresponding **VirtualService** object. Verify the configuration of the **VirtualService** object by running the following command:

```
$ oc get virtualservice -l networking.internal.knative.dev/ingress=<ingress_name> -n
<ingress_namespace> --output yaml
```

The network configuration in the **VirtualService** object must match that of the **Ingress** and **Route** objects. Because the **VirtualService** object does not expose a **Status** field, you might need to wait for its settings to propagate.

4.6.1. Additional resources

- Maistra Service Mesh documentation

CHAPTER 5. KOURIER AND ISTIO INGRESSES

OpenShift Serverless supports the following two ingress solutions:

- Kourier
- Istio using Red Hat OpenShift Service Mesh

The default is Kourier.

5.1. KOURIER AND ISTIO INGRESS SOLUTIONS

5.1.1. Kourier

Kourier is the default ingress solution for OpenShift Serverless. It has the following properties:

- It is based on envoy proxy.
- It is simple and lightweight.
- It provides the basic routing functionality that Serverless needs to provide its set of features.
- It supports basic observability and metrics.
- It supports basic TLS termination of Knative Service routing.
- It provides only limited configuration and extension options.

5.1.2. Istio using OpenShift Service Mesh

Using Istio as the ingress solution for OpenShift Serverless enables an additional feature set that is based on what Red Hat OpenShift Service Mesh offers:

- Native mTLS between all connections
- Serverless components are part of a service mesh
- Additional observability and metrics
- Authorization and authentication support
- Custom rules and configuration, as supported by Red Hat OpenShift Service Mesh

However, the additional features come with a higher overhead and resource consumption. For details, see the Red Hat OpenShift Service Mesh documentation.

See the "Integrating Service Mesh with OpenShift Serverless" section of Serverless documentation for Istio requirements and installation instructions.

5.1.3. Traffic configuration and routing

Regardless of whether you use Kourier or Istio, the traffic for a Knative Service is configured in the **knative-serving** namespace by the **net-kourier-controller** or the **net-istio-controller** respectively.

The controller reads the **KnativeService** and its child custom resources to configure the ingress

solution. Both ingress solutions provide an ingress gateway pod that becomes part of the traffic path. Both ingress solutions are based on Envoy. By default, Serverless has two routes for each **KnativeService** object:

- A **cluster-external route** that is forwarded by the OpenShift router, for example **myapp-namespace.example.com**.
- A **cluster-local route** containing the cluster domain, for example **myapp.namespace.svc.cluster.local**. This domain can and should be used to call Knative services from Knative or other user workloads.

The ingress gateway can forward requests either in the serve mode or the proxy mode:

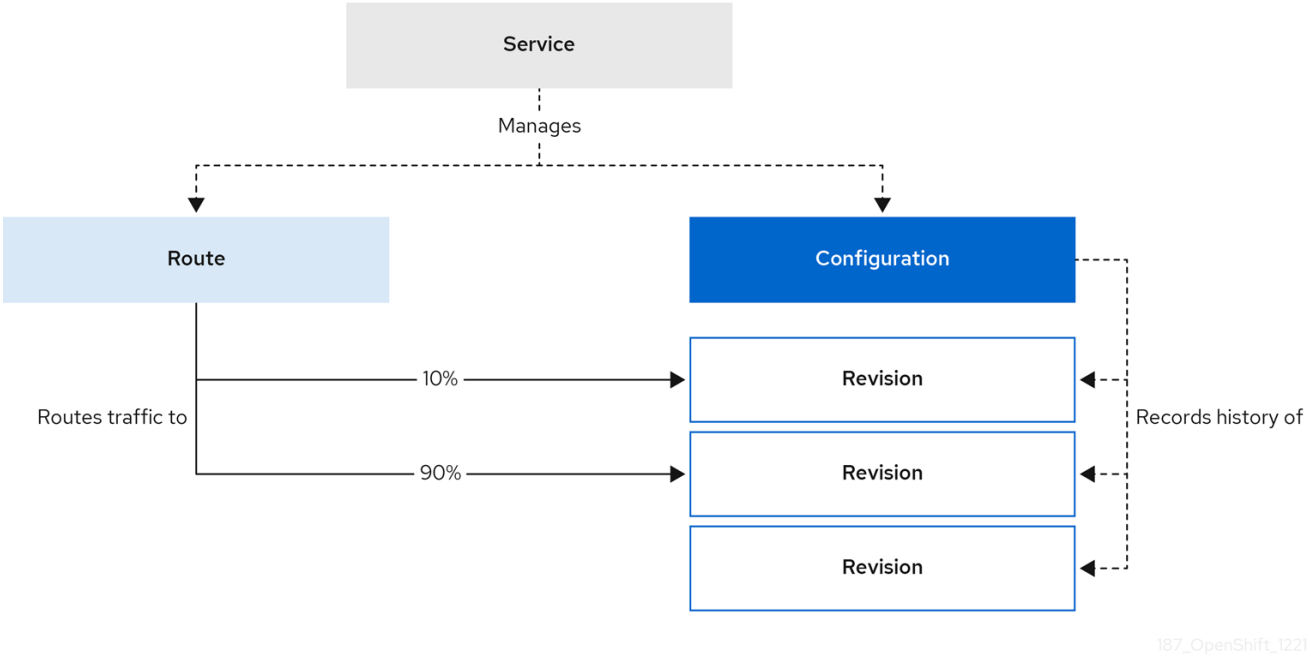
- In the serve mode, requests go directly to the Queue-Proxy sidecar container of the Knative service.
- In the proxy mode, requests first go through the Activator component in the **knative-serving** namespace.

The choice of mode depends on the configuration of Knative, the Knative service, and the current traffic. For example, if a Knative Service is scaled to zero, requests are sent to the Activator component, which acts as a buffer until a new Knative service pod is started.

CHAPTER 6. TRAFFIC SPLITTING

6.1. TRAFFIC SPLITTING OVERVIEW

In a Knative application, traffic can be managed by creating a traffic split. A traffic split is configured as part of a route, which is managed by a Knative service.



Configuring a route allows requests to be sent to different revisions of a service. This routing is determined by the **traffic** spec of the **Service** object.

A **traffic** spec declaration consists of one or more revisions, each responsible for handling a portion of the overall traffic. The percentages of traffic routed to each revision must add up to 100%, which is ensured by a Knative validation.

The revisions specified in a **traffic** spec can either be a fixed, named revision, or can point to the “latest” revision, which tracks the head of the list of all revisions for the service. The "latest" revision is a type of floating reference that updates if a new revision is created. Each revision can have a tag attached that creates an additional access URL for that revision.

The **traffic** spec can be modified by:

- Editing the YAML of a **Service** object directly.
- Using the Knative (**kn**) CLI **--traffic** flag.
- Using the OpenShift Container Platform web console.

When you create a Knative service, it does not have any default **traffic** spec settings.

6.2. TRAFFIC SPEC EXAMPLES

The following example shows a **traffic** spec where 100% of traffic is routed to the latest revision of the service. Under **status**, you can see the name of the latest revision that **latestRevision** resolves to:

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: example-service
  namespace: default
spec:
...
  traffic:
  - latestRevision: true
    percent: 100
status:
  ...
  traffic:
  - percent: 100
    revisionName: example-service
```

The following example shows a **traffic** spec where 100% of traffic is routed to the revision tagged as **current**, and the name of that revision is specified as **example-service**. The revision tagged as **latest** is kept available, even though no traffic is routed to it:

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: example-service
  namespace: default
spec:
...
  traffic:
  - tag: current
    revisionName: example-service
    percent: 100
  - tag: latest
    latestRevision: true
    percent: 0
```

The following example shows how the list of revisions in the **traffic** spec can be extended so that traffic is split between multiple revisions. This example sends 50% of traffic to the revision tagged as **current**, and 50% of traffic to the revision tagged as **candidate**. The revision tagged as **latest** is kept available, even though no traffic is routed to it:

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: example-service
  namespace: default
spec:
...
  traffic:
  - tag: current
    revisionName: example-service-1
    percent: 50
  - tag: candidate
    revisionName: example-service-2
    percent: 50
```

```
- tag: latest
  latestRevision: true
  percent: 0
```

6.3. TRAFFIC SPLITTING USING THE KNATIVE CLI

Using the Knative (**kn**) CLI to create traffic splits provides a more streamlined and intuitive user interface over modifying YAML files directly. You can use the **kn service update** command to split traffic between revisions of a service.

6.3.1. Creating a traffic split by using the Knative CLI

Prerequisites

- The OpenShift Serverless Operator and Knative Serving are installed on your cluster.
- You have installed the Knative (**kn**) CLI.
- You have created a Knative service.

Procedure

- Specify the revision of your service and what percentage of traffic you want to route to it by using the **--traffic** tag with a standard **kn service update** command:

  Example command

  ```
  $ kn service update <service_name> --traffic <revision>=<percentage>
  ```

  Where:

  - **<service_name>** is the name of the Knative service that you are configuring traffic routing for.
  - **<revision>** is the revision that you want to configure to receive a percentage of traffic. You can either specify the name of the revision, or a tag that you assigned to the revision by using the **--tag** flag.
  - **<percentage>** is the percentage of traffic that you want to send to the specified revision.

- Optional: The **--traffic** flag can be specified multiple times in one command. For example, if you have a revision tagged as **@latest** and a revision named **stable**, you can specify the percentage of traffic that you want to split to each revision as follows:

  Example command

  ```
  $ kn service update showcase --traffic @latest=20,stable=80
  ```

  If you have multiple revisions and do not specify the percentage of traffic that should be split to the last revision, the **--traffic** flag can calculate this automatically. For example, if you have a third revision named **example**, and you use the following command:

  Example command

  ```
  -
  ```

```
$ kn service update showcase --traffic @latest=10,stable=60
```

The remaining 30% of traffic is split to the **example** revision, even though it was not specified.

6.4. CLI FLAGS FOR TRAFFIC SPLITTING

The Knative (**kn**) CLI supports traffic operations on the traffic block of a service as part of the **kn service update** command.

6.4.1. Knative CLI traffic splitting flags

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

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

6.4.1.1. Multiple flags and order 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.

You can add tags to revisions and then split traffic according to the tags you have set.

6.4.1.2. Custom URLs for revisions

Assigning a **--tag** flag to a service by using the **kn service update** command creates a custom URL for the revision that is created when you update the service. The custom URL follows the pattern **https://<tag>-<service_name>-<namespace>.<domain>** or **http://<tag>-<service_name>-<namespace>.<domain>**.

The **--tag** and **--untag** flags use the following syntax:

- Require one value.
- Denote a unique tag in the traffic block of the service.
- Can be specified multiple times in one command.

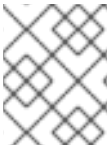
6.4.1.2.1. Example: Assign a tag to a revision

The following example assigns the tag **latest** to a revision named **example-revision**:

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

6.4.1.2.2. Example: Remove a tag from a revision

You can remove a tag to remove the custom URL, by using the **--untag** flag.



NOTE

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

The following command removes all tags from the revision named **example-revision**:

```
$ kn service update <service_name> --untag example-tag
```

6.5. SPLITTING TRAFFIC BETWEEN REVISIONS

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

Any new change in the code or the service configuration creates a new revision, which is 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.

6.5.1. Managing traffic between revisions by using the OpenShift Container Platform web console

Prerequisites

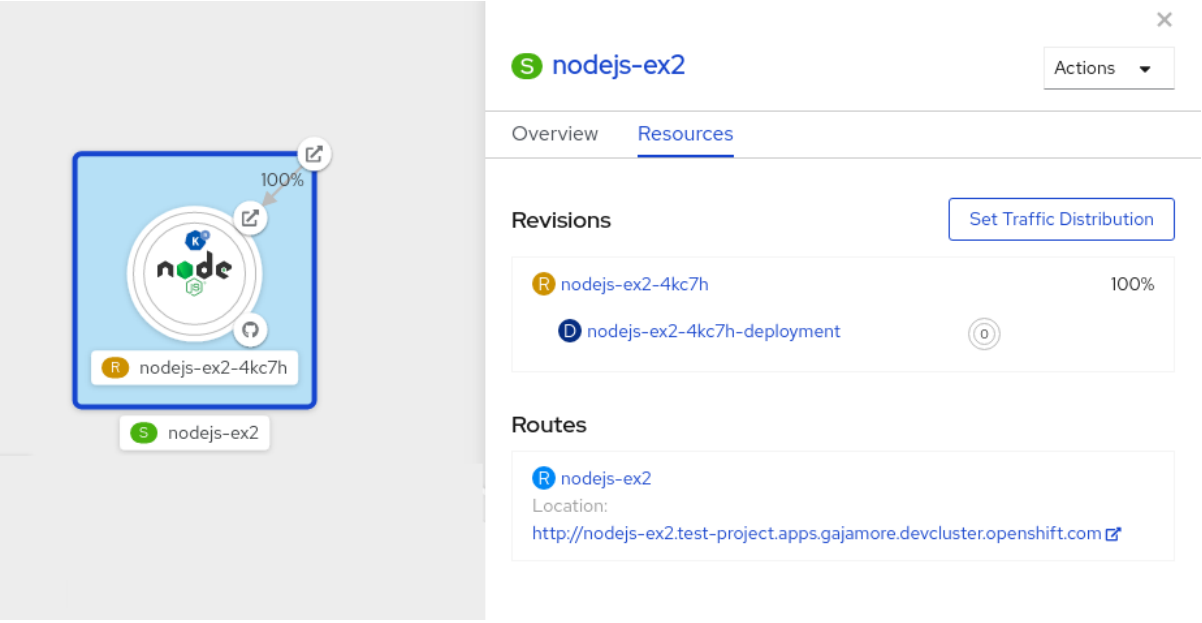
- The OpenShift Serverless Operator and Knative Serving are installed on your cluster.
- You have logged in to the OpenShift Container Platform web console.

Procedure

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

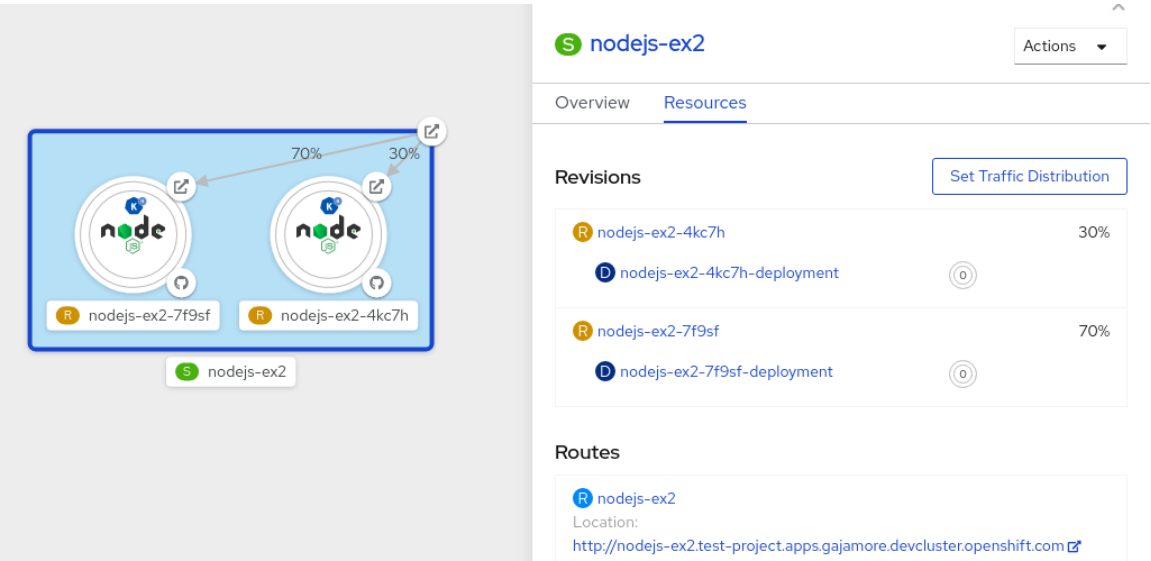
1. Click the Knative service to see its overview in the side panel.
2. Click the **Resources** tab, to see a list of **Revisions** and **Routes** for the service.

   Figure 6.1. 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 **Set Traffic Distribution** to see the traffic distribution dialog box:
   a. Add the split traffic percentage portion for the two revisions in the **Splits** field.
   b. Add tags to create custom URLs for the two revisions.
   c. Click **Save** to see two nodes representing the two revisions in the Topology view.

Figure 6.2. Serverless application revisions



6.6. REROUTING TRAFFIC USING BLUE-GREEN STRATEGY

You can safely reroute traffic from a production version of an app to a new version, by using a blue-green deployment strategy.

6.6.1. Routing and managing traffic by using a blue-green deployment strategy

Prerequisites

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

Procedure

1. Create and deploy an app as a Knative service.
2. Find the name of the first revision that was created when you deployed the service, by viewing the output from the following command:

```
$ oc get ksvc <service_name> -o=jsonpath='{.status.latestCreatedRevisionName}'
```

Example command

```
$ oc get ksvc showcase -o=jsonpath='{.status.latestCreatedRevisionName}'
```

Example output

```
$ showcase-00001
```

3. Add the following YAML to the service **spec** to send inbound traffic to the revision:

```
...
spec:
  traffic:
```

```
    - revisionName: <first_revision_name>
      percent: 100 # All traffic goes to this revision
...
```

4. Verify that you can view your app at the URL output you get from running the following command:

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

5. Deploy a second revision of your app by modifying at least one field in the **template** spec of the service and redeploying it. For example, you can modify the **image** of the service, or an **env** environment variable. You can redeploy the service by applying the service YAML file, or by using the **kn service update** command if you have installed the Knative (**kn**) CLI.

6. Find the name of the second, latest revision that was created when you redeployed the service, by running the command:

```
$ oc get ksvc <service_name> -o=jsonpath='{.status.latestCreatedRevisionName}'
```

   At this point, both the first and second revisions of the service are deployed and running.

7. Update your existing service to create a new, test endpoint for the second revision, while still sending all other traffic to the first revision:

   Example of updated service spec with test endpoint

```
...
spec:
  traffic:
    - revisionName: <first_revision_name>
      percent: 100 # All traffic is still being routed to the first revision
    - revisionName: <second_revision_name>
      percent: 0 # No traffic is routed to the second revision
      tag: v2 # A named route
...
```

   After you redeploy this service by reapplying the YAML resource, the second revision of the app is now staged. No traffic is routed to the second revision at the main URL, and Knative creates a new service named **v2** for testing the newly deployed revision.

8. Get the URL of the new service for the second revision, by running the following command:

```
$ oc get ksvc <service_name> --output jsonpath="{.status.traffic[*].url}"
```

   You can use this URL to validate that the new version of the app is behaving as expected before you route any traffic to it.

9. Update your existing service again, so that 50% of traffic is sent to the first revision, and 50% is sent to the second revision:

   Example of updated service spec splitting traffic 50/50 between revisions

```
...
spec:
```

```
  traffic:
    - revisionName: <first_revision_name>
      percent: 50
    - revisionName: <second_revision_name>
      percent: 50
      tag: v2
...
```

10. When you are ready to route all traffic to the new version of the app, update the service again to send 100% of traffic to the second revision:

    Example of updated service spec sending all traffic to the second revision

```
...
spec:
  traffic:
    - revisionName: <first_revision_name>
      percent: 0
    - revisionName: <second_revision_name>
      percent: 100
      tag: v2
...
```

    TIP

    You can remove the first revision instead of setting it to 0% of traffic if you do not plan to roll back the revision. Non-routeable revision objects are then garbage-collected.

11. Visit the URL of the first revision to verify that no more traffic is being sent to the old version of the app.

CHAPTER 7. EXTERNAL AND INGRESS ROUTING

7.1. ROUTING OVERVIEW

Knative leverages OpenShift Container Platform TLS termination to provide routing for Knative services. When a Knative service is created, an 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.

7.1.1. Additional resources for OpenShift Container Platform

- Route-specific annotations

7.2. CUSTOMIZING LABELS AND ANNOTATIONS

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.

7.2.1. Customizing labels and annotations for OpenShift Container Platform routes

Prerequisites

- You must have the OpenShift Serverless Operator and Knative Serving installed on your OpenShift Container Platform cluster.
- Install the OpenShift CLI (**oc**).

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

     ```
     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 Knative (**kn**) CLI, enter:

  Example service created by using a kn command

```
$ 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

```
$ oc get routes.route.openshift.io \
    -l serving.knative.openshift.io/ingressName=<service_name> \ 1
    -l serving.knative.openshift.io/ingressNamespace=<service_namespace> \ 2
    -n knative-serving-ingress -o yaml \
      | grep -e "<label_name>: \"<label_value>\""  -e "<annotation_name>:
<annotation_value>" 3
```

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.

7.3. CONFIGURING 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.

7.3.1. Configuring OpenShift Container Platform routes for Knative services

Prerequisites

- The OpenShift Serverless Operator and Knative Serving component must be installed on your OpenShift Container Platform cluster.
- Install the OpenShift CLI (**oc**).

Procedure

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

IMPORTANT

The **serving.knative.openshift.io/disableRoute=true** annotation instructs OpenShift Serverless to not automatically create a route for you. However, the service still shows a URL and reaches a status of **Ready**. This URL does not work externally until you create your own route with the same hostname as the hostname in the URL.

   a. Create a Knative **Service** resource:

      Example resource

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: <service_name>
  annotations:
    serving.knative.openshift.io/disableRoute: "true"
spec:
  template:
    spec:
      containers:
      - image: <image>
...
```

   b. Apply the **Service** resource:

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

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

      Example kn command

```
$ 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

```
$ $ 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:

```
apiVersion: route.openshift.io/v1
kind: Route
metadata:
  annotations:
    haproxy.router.openshift.io/timeout: 600s (1)
  name: <route_name> (2)
  namespace: knative-serving-ingress (3)
spec:
  host: <service_host> (4)
  port:
    targetPort: http2
  to:
    kind: Service
    name: kourier
    weight: 100
  tls:
    insecureEdgeTerminationPolicy: Allow
    termination: edge (5)
    key: |-
      -----BEGIN PRIVATE KEY-----
      [...]
      -----END PRIVATE KEY-----
    certificate: |-
      -----BEGIN CERTIFICATE-----
      [...]
      -----END CERTIFICATE-----
    caCertificate: |-
      -----BEGIN CERTIFICATE-----
      [...]
      -----END CERTIFICATE----
  wildcardPolicy: None
```

(1) The timeout value for the OpenShift Container Platform route. You must set the same value as the **max-revision-timeout-seconds** setting (**600s** by default).

(2) The name of the OpenShift Container Platform route.

(3) The namespace for the OpenShift Container Platform route. This must be **knative-serving-ingress**.

(4) The hostname for external access. You can set this to **<service_name>-<service_namespace>.<domain>**.

(5) The certificates you want to use. Currently, only **edge** termination is supported.

4. Apply the **Route** resource:

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

7.4. GLOBAL HTTPS REDIRECTION

HTTPS redirection provides redirection for incoming HTTP requests. These redirected HTTP requests are encrypted. You can enable HTTPS redirection for all services on the cluster by configuring the **httpProtocol** spec for the **KnativeServing** custom resource (CR).

7.4.1. HTTPS redirection global settings

Example KnativeServing CR that enables HTTPS redirection

```
apiVersion: operator.knative.dev/v1beta1
kind: KnativeServing
metadata:
  name: knative-serving
spec:
  config:
    network:
      httpProtocol: "redirected"
...
```

7.5. URL SCHEME FOR EXTERNAL ROUTES

The URL scheme of external routes defaults to HTTPS for enhanced security. This scheme is determined by the **default-external-scheme** key in the **KnativeServing** custom resource (CR) spec.

7.5.1. Setting the URL scheme for external routes

Default spec

```
...
spec:
  config:
    network:
      default-external-scheme: "https"
...
```

You can override the default spec to use HTTP by modifying the **default-external-scheme** key:

HTTP override spec

```
...
spec:
  config:
    network:
      default-external-scheme: "http"
...
```

7.6. HTTPS REDIRECTION PER SERVICE

You can enable or disable HTTPS redirection for a service by configuring the **networking.knative.dev/http-option** annotation.

7.6.1. Redirecting HTTPS for a service

The following example shows how you can use this annotation in a Knative **Service** YAML object:

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: example
  namespace: default
  annotations:
    networking.knative.dev/http-protocol: "redirected"
spec:
  ...
```

7.7. CLUSTER LOCAL AVAILABILITY

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.

7.7.1. Setting cluster availability to cluster local

Prerequisites

- The OpenShift Serverless Operator and Knative Serving are installed on the cluster.
- You have created a Knative service.

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:

```
$ oc get ksvc
```

Example output

```
NAME          URL                                        LATESTCREATED
LATESTREADY       READY   REASON
hello         http://hello.default.svc.cluster.local     hello-tx2g7       hello-
tx2g7       True
```

7.7.2. Enabling TLS authentication for cluster local services

For cluster local services, the Kourier local gateway **kourier-internal** is used. If you want to use TLS traffic against the Kourier local gateway, you must configure your own server certificates in the local gateway.

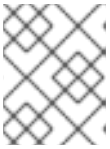
Prerequisites

- You have installed the OpenShift Serverless Operator and Knative Serving.
- You have administrator permissions.
- You have installed the OpenShift (**oc**) CLI.

Procedure

1. Deploy server certificates in the **knative-serving-ingress** namespace:

```
$ export san="knative"
```



NOTE

Subject Alternative Name (SAN) validation is required so that these certificates can serve the request to **<app_name>.<namespace>.svc.cluster.local**.

2. Generate a root key and certificate:

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

3. Generate a server key that uses SAN validation:

```
$ openssl req -out tls.csr -newkey rsa:2048 -nodes -keyout tls.key \
  -subj "/CN=Example/O=Example" \
  -addext "subjectAltName = DNS:$san"
```

4. Create server certificates:

```
$ openssl x509 -req -extfile <(printf "subjectAltName=DNS:$san") \
  -days 365 -in tls.csr \
  -CA ca.crt -CAkey ca.key -CAcreateserial -out tls.crt
```

5. Configure a secret for the Kourier local gateway:

a. Deploy a secret in **knative-serving-ingress** namespace from the certificates created by the previous steps:

```
$ oc create -n knative-serving-ingress secret tls server-certs \
    --key=tls.key \
    --cert=tls.crt --dry-run=client -o yaml | oc apply -f -
```

b. Update the **KnativeServing** custom resource (CR) spec to use the secret that was created by the Kourier gateway:

Example KnativeServing CR

```
...
spec:
  config:
    kourier:
      cluster-cert-secret: server-certs
...
```

The Kourier controller sets the certificate without restarting the service, so that you do not need to restart the pod.

You can access the Kourier internal service with TLS through port **443** by mounting and using the **ca.crt** from the client.

7.8. KOURIER GATEWAY SERVICE TYPE

The Kourier Gateway is exposed by default as the **ClusterIP** service type. This service type is determined by the **service-type** ingress spec in the **KnativeServing** custom resource (CR).

Default spec

```
...
spec:
  ingress:
    kourier:
      service-type: ClusterIP
...
```

7.8.1. Setting the Kourier Gateway service type

You can override the default service type to use a load balancer service type instead by modifying the **service-type** spec:

LoadBalancer override spec

```
...
spec:
  ingress:
    kourier:
      service-type: LoadBalancer
...
```

7.9. 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.

7.9.1. Interacting with a serverless application using HTTP2 and gRPC

IMPORTANT

This method applies to OpenShift Container Platform 4.10 and later. For older versions, see the following section.

Prerequisites

- Install OpenShift Serverless Operator and Knative Serving on your cluster.
- Install the OpenShift CLI (**oc**).
- Create a Knative service.
- Upgrade OpenShift Container Platform 4.10 or later.
- Enable HTTP/2 on OpenShift Ingress controller.

Procedure

1. Add the **serverless.openshift.io/default-enable-http2=true** annotation to the **KnativeServing** Custom Resource:

```
$ oc annotate knativeserving <your_knative_CR> -n knative-serving
serverless.openshift.io/default-enable-http2=true
```

2. After the annotation is added, you can verify that the **appProtocol** value of the Kourier service is **h2c**:

```
$ oc get svc -n knative-serving-ingress kourier -o jsonpath="{.spec.ports[0].appProtocol}"
```

Example output

```
h2c
```

3. Now you can use the gRPC framework over the HTTP/2 protocol for external traffic, for example:

```
import "google.golang.org/grpc"

grpc.Dial(
  YOUR_URL, (1)
  grpc.WithTransportCredentials(insecure.NewCredentials())), (2)
)
```

1. Your **ksvc** URL.
2. Your certificate.

Additional resources

- Enabling HTTP/2 Ingress connectivity

7.10. USING SERVING WITH OPENSHIFT INGRESS SHARDING

You can use Knative Serving with OpenShift ingress sharding to split ingress traffic based on domains. This allows you to manage and route network traffic to different parts of a cluster more efficiently.

NOTE

Even with OpenShift ingress sharding in place, OpenShift Serverless traffic is still routed through a single Knative Ingress Gateway and the activator component in the **knative-serving** project.

For more information about isolating the network traffic, see Using Service Mesh to isolate network traffic with OpenShift Serverless.

Prerequisites

- You have installed the OpenShift Serverless Operator and Knative Serving.
- You have cluster administrator permissions on OpenShift Container Platform, or you have cluster or dedicated administrator permissions on Red Hat OpenShift Service on AWS or OpenShift Dedicated.

7.10.1. Configuring OpenShift ingress shards

Before configuring Knative Serving, you must configure OpenShift ingress shards.

Procedure

- Use a label selector in the **IngressController** CR to configure OpenShift Serverless to match specific ingress shards with different domains:

Example IngressController CR

```
apiVersion: operator.openshift.io/v1
kind: IngressController
metadata:
  name: ingress-dev 1
  namespace: openshift-ingress-operator
spec:
  routeSelector:
    matchLabels:
      router: dev 2
  domain: "dev.serverless.cluster.example.com" 3
  # ...
---
```

```
apiVersion: operator.openshift.io/v1
kind: IngressController
metadata:
  name: ingress-prod 4
  namespace: openshift-ingress-operator
spec:
  routeSelector:
    matchLabels:
      router: prod 5
  domain: "prod.serverless.cluster.example.com" 6
  # ...
```

1. Name of the first ingress shard.
2. A label selector to match the **ingress-dev** shard.
3. A custom domain for the **ingress-dev** shard.
4. Name of the second ingress shard.
5. A label selector to match the **ingress-prod** shard.
6. A custom domain for the **ingress-prod** shard.

7.10.2. Configuring custom domains in the KnativeServing CR

After configuring OpenShift ingress shards, you must configure Knative Serving to match them.

Procedure

- In the **KnativeServing** CR, configure Serving to use the same domains and labels as your ingress shards by adding the **spec.config.domain** field:

  Example KnativeServing CR

```
spec:
  config:
    domain: 1
      dev.serverless.cluster.example.com: |
        selector:
          router: dev
      prod.serverless.cluster.example.com: |
        selector:
          router: prod
  # ...
```

1. These values need to match the values in the ingress shard configuration.

7.10.3. Targeting a specific ingress shard in the Knative Service

After configuring ingress sharding and Knative Serving, you can target a specific ingress shard in your Knative Service resources using a label.

Procedure

- In your **Service** CR, add the label selector that matches a specific shard:

  Example Service CR

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: hello-dev
  labels:
    router: dev (1)
spec:
  template:
    spec:
      containers:
      - image: docker.io/openshift/hello-openshift
---
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: hello-prod
  labels:
    router: prod (2)
spec:
  template:
    spec:
      containers:
      - image: docker.io/openshift/hello-openshift
  # ...
```

(1) (2) The labels must match the configuration in the **KnativeServing** CR.

7.10.4. Verifying Serving with OpenShift ingress sharding configuration

After configuring ingress sharding, Knative Serving, and your service, you can verify that your service uses the correct route and the selected ingress shard.

Procedure

1. Print information about the services in the cluster by running the following command:

```
$ oc get ksvc
```

Example output

```
NAME        URL                                                LATESTCREATED
LATESTREADY       READY   REASON
hello-dev   https://hello-dev-default.dev.serverless.cluster.example.com    hello-dev-00001
hello-dev-00001    True
hello-prod   https://hello-prod-default.prod.serverless.cluster.example.com  hello-prod-00001
hello-prod-00001   True
```

2. Verify that your service uses the correct route and the selected ingress shard by running the following command:

```
$ oc get route -n knative-serving-ingress -o jsonpath='{range .items[*]}{@.metadata.name}{"
"}{@.spec.host}{" "}{@.status.ingress[*].routerName}{"\n"}{end}'
```

Example output

```
route-19e6628b-77af-4da0-9b4c-1224934b2250-323461616533 hello-prod-
default.prod.serverless.cluster.example.com ingress-prod
route-cb5085d9-b7da-4741-9a56-96c88c6adaaa-373065343266 hello-dev-
default.dev.serverless.cluster.example.com ingress-dev
```

CHAPTER 8. CONFIGURING ACCESS TO KNATIVE SERVICES

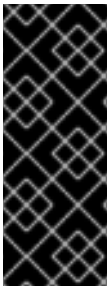
8.1. CONFIGURING JSON WEB TOKEN AUTHENTICATION FOR KNATIVE SERVICES

OpenShift Serverless does not currently have user-defined authorization features. To add user-defined authorization to your deployment, you must integrate OpenShift Serverless with Red Hat OpenShift Service Mesh, and then configure JSON Web Token (JWT) authentication and sidecar injection for Knative services.

8.2. USING JSON WEB TOKEN AUTHENTICATION WITH SERVICE MESH 2.X

You can use JSON Web Token (JWT) authentication with Knative services by using Service Mesh 2.x and OpenShift Serverless. To do this, you must create authentication requests and policies in the application namespace that is a member of the **ServiceMeshMemberRoll** object. You must also enable sidecar injection for the service.

8.2.1. Configuring JSON Web Token authentication for Service Mesh 2.x and OpenShift Serverless



IMPORTANT

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

For OpenShift Container Platform, if you require sidecar injection for pods in these namespaces, see the OpenShift Serverless documentation on *Integrating Service Mesh with OpenShift Serverless natively*.

Prerequisites

- You have installed the OpenShift Serverless Operator, Knative Serving, and Red Hat OpenShift Service Mesh on your cluster.
- Install the OpenShift CLI (**oc**).
- You have created a project or have access to a project with the appropriate roles and permissions to create applications and other workloads in OpenShift Container Platform.

Procedure

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

   Example service

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

2. Apply the **Service** resource:

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

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

```
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
```

4. Apply the **RequestAuthentication** resource:

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

5. Allow access to the **RequestAuthenticaton** resource from system pods for each serverless application namespace that is a member in the **ServiceMeshMemberRoll** object, by creating the following **AuthorizationPolicy** resource:

```
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: allowlist-by-paths
  namespace: <namespace>
spec:
  action: ALLOW
  rules:
  - to:
    - operation:
        paths:
        - /metrics 1
        - /healthz 2
```

1. The path on your application to collect metrics by system pod.

2 The path on your application to probe by system pod.

6. Apply the **AuthorizationPolicy** resource:

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

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

```
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"]
```

8. 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:

```
$ curl -H "Authorization: Bearer $TOKEN"  http://hello-example-1-
default.apps.example.com
```

   The request is now allowed:

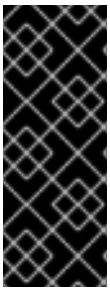
Example output

```
Hello OpenShift!
```

8.3. USING JSON WEB TOKEN AUTHENTICATION WITH SERVICE MESH 1.X

You can use JSON Web Token (JWT) authentication with Knative services by using Service Mesh 1.x and OpenShift Serverless. To do this, you must create a policy in the application namespace that is a member of the **ServiceMeshMemberRoll** object. You must also enable sidecar injection for the service.

8.3.1. Configuring JSON Web Token authentication for Service Mesh 1.x and OpenShift Serverless



IMPORTANT

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

For OpenShift Container Platform, if you require sidecar injection for pods in these namespaces, see the OpenShift Serverless documentation on *Integrating Service Mesh with OpenShift Serverless natively*.

Prerequisites

- You have installed the OpenShift Serverless Operator, Knative Serving, and Red Hat OpenShift Service Mesh on your cluster.
- Install the OpenShift CLI (**oc**).
- You have created a project or have access to a project with the appropriate roles and permissions to create applications and other workloads in OpenShift Container Platform.

Procedure

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

Example service

```
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 You must set the annotation **sidecar.istio.io/rewriteAppHTTPProbers: "true"** in your Knative service, because OpenShift Serverless versions 1.14.0 and higher use an HTTP

2. Apply the **Service** resource:

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

3. Create a policy in a serverless application namespace which is a member in the **ServiceMeshMemberRoll** object, that only allows requests with valid JSON Web Tokens (JWT):

IMPORTANT

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

```
apiVersion: authentication.istio.io/v1alpha1
kind: Policy
metadata:
  name: default
  namespace: <namespace>
spec:
  origins:
  - jwt:
      issuer: testing@secure.istio.io
      jwksUri: "https://raw.githubusercontent.com/istio/istio/release-
1.6/security/tools/jwt/samples/jwks.json"
      triggerRules:
      - excludedPaths:
        - prefix: /metrics 1
        - prefix: /healthz 2
  principalBinding: USE_ORIGIN
```

1 The path on your application to collect metrics by system pod.

2 The path on your application to probe by system pod.

4. Apply the **Policy** resource:

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

Verification

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

```
$ curl http://hello-example-default.apps.mycluster.example.com/
```

Example output

```
Origin authentication failed.
```

2. Verify the request with a valid JWT.

   a. Get the valid JWT token:

   ```
   $ TOKEN=$(curl https://raw.githubusercontent.com/istio/istio/release-1.6/security/tools/jwt/samples/demo.jwt -s) && echo "$TOKEN" | cut -d '.' -f2 - | base64 --decode -
   ```

   b. Access the service by using the valid token in the **curl** request header:

   ```
   $ curl http://hello-example-default.apps.mycluster.example.com/ -H "Authorization: Bearer $TOKEN"
   ```

   The request is now allowed:

   Example output

   ```
   Hello OpenShift!
   ```

CHAPTER 9. CONFIGURING KUBE-RBAC-PROXY FOR SERVING

The **kube-rbac-proxy** component provides internal authentication and authorization capabilities for Knative Serving.

9.1. CONFIGURING KUBE-RBAC-PROXY RESOURCES FOR SERVING

You can globally override resource allocation for the **kube-rbac-proxy** container by using the OpenShift Serverless Operator CR.

NOTE

You can also override resource allocation for a specific deployment.

The following configuration sets Knative Serving **kube-rbac-proxy** minimum and maximum CPU and memory allocation:

KnativeServing CR example

```
apiVersion: operator.knative.dev/v1beta1
kind: KnativeServing
metadata:
  name: knative-serving
  namespace: knative-serving
spec:
  config:
    deployment:
      "kube-rbac-proxy-cpu-request": "10m" 1
      "kube-rbac-proxy-memory-request": "20Mi" 2
      "kube-rbac-proxy-cpu-limit": "100m" 3
      "kube-rbac-proxy-memory-limit": "100Mi" 4
```

1. Sets minimum CPU allocation.
2. Sets minimum RAM allocation.
3. Sets maximum CPU allocation.
4. Sets maximum RAM allocation.

CHAPTER 10. CONFIGURING BURST AND QPS FOR NET-KOURIER

The queries per second (QPS) and burst values determine the frequency of requests or API calls to the API server.

10.1. CONFIGURING BURST AND QPS VALUES FOR NET-KOURIER

The queries per second (QPS) value determines the number of client requests or API calls that are sent to the API server.

The burst value determines how many requests from the client can be stored for processing. Requests exceeding this buffer will be dropped. This is helpful for controllers that are bursty and do not spread their requests uniformly in time.

When the **net-kourier-controller** restarts, it parses all **ingress** resources deployed on the cluster, which leads to a significant number of API calls. Due to this, the **net-kourier-controller** can take a long time to start.

You can adjust the QPS and burst values for the **net-kourier-controller** in the KnativeServing CR:

KnativeServing CR example

```
apiVersion: operator.knative.dev/v1beta1
kind: KnativeServing
metadata:
  name: knative-serving
  namespace: knative-serving
spec:
  workloads:
  - name: net-kourier-controller
    env:
    - container: controller
      envVars:
      - name: KUBE_API_BURST
        value: "200" (1)
      - name: KUBE_API_QPS
        value: "200" (2)
```

1. The QPS rate of communication between controller and the API Server. The default value is 200.
2. The burst capacity of communication between Kubelet and the API Server. The default value is 200.

CHAPTER 11. CONFIGURING CUSTOM DOMAINS FOR KNATIVE SERVICES

11.1. 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 mapping a custom domain name that you own to a Knative service.

You can do this 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.

11.2. CUSTOM DOMAIN MAPPING

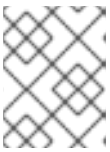
You can customize the domain for your Knative service by mapping a custom domain name that you own to a Knative service. To map a custom domain name to a custom resource (CR), you must create a **DomainMapping** CR that maps to an Addressable target CR, such as a Knative service or a Knative route.

11.2.1. Creating a custom domain mapping

You can customize the domain for your Knative service by mapping a custom domain name that you own to a Knative service. 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.
- Install the OpenShift CLI (**oc**).
- You have created a project or have access to a project with the appropriate roles and permissions to create applications and other workloads in OpenShift Container Platform.
- You have 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:

```
apiVersion: serving.knative.dev/v1beta1
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

```
apiVersion: serving.knative.dev/v1beta1
kind: DomainMapping
metadata:
 name: example.com
 namespace: default
spec:
 ref:
   name: showcase
   kind: Service
   apiVersion: serving.knative.dev/v1
```

Example route domain mapping

```
apiVersion: serving.knative.dev/v1beta1
kind: DomainMapping
metadata:
 name: example.com
 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>
```

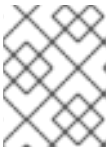
11.3. CUSTOM DOMAINS FOR KNATIVE SERVICES USING THE KNATIVE CLI

You can customize the domain for your Knative service by mapping a custom domain name that you own to a Knative service. You can use the Knative (**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.

11.3.1. Creating a custom domain mapping by using the Knative CLI

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 Knative (**kn**) CLI.
- You have created a project or have access to a project with the appropriate roles and permissions to create applications and other workloads in OpenShift Container Platform.

Procedure

- 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.com --ref showcase
```

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.

- 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.com --ref ksvc:showcase: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.com --ref kroute:example-route
```

11.4. DOMAIN MAPPING USING THE DEVELOPER PERSPECTIVE

You can customize the domain for your Knative service by mapping a custom domain name that you own to a Knative service. You can use the **Developer** perspective of the OpenShift Container Platform web console to map a **DomainMapping** custom resource (CR) to a Knative service.

11.4.1. Mapping a custom domain to a service by using the Developer perspective

Prerequisites

- You have logged in to the web console.
- You are in the **Developer** perspective.
- The OpenShift Serverless Operator and Knative Serving are installed on your cluster. This must be completed by a cluster administrator.
- 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 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. Navigate to the **Topology** page.
2. Right-click the service you want to map to a domain, and select the **Edit** option that contains the service name. For example, if the service is named **showcase**, select the **Edit showcase** option.
3. In the **Advanced options** section, click **Show advanced Routing options**.
   a. If the domain mapping CR that you want to map to the service already exists, you can select it in the **Domain mapping** list.
   b. If you want to create a new domain mapping CR, type the domain name into the box, and select the **Create** option. For example, if you type in **example.com**, the **Create** option is **Create "example.com"**.
4. Click **Save** to save the changes to your service.

Verification

1. Navigate to the **Topology** page.
2. Click on the service that you have created.
3. In the **Resources** tab of the service information window, you can see the domain you have mapped to the service listed under **Domain mappings**.

11.5. DOMAIN MAPPING USING THE ADMINISTRATOR PERSPECTIVE

If you do not want to switch to the **Developer** perspective in the OpenShift Container Platform web console or use the Knative (**kn**) CLI or YAML files, you can use the **Administator** perspective of the OpenShift Container Platform web console.

11.5.1. Mapping a custom domain to a service by using the Administrator perspective

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 mapping a custom domain name that you own to a Knative service.

You can do this 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.

If you have cluster administrator permissions on OpenShift Container Platform (or cluster or dedicated administrator permissions on OpenShift Dedicated or Red Hat OpenShift Service on AWS), you can create a **DomainMapping** custom resource (CR) by using the **Administrator** perspective in the web console.

Prerequisites

- You have logged in to the web console.
- You are in the **Administrator** perspective.
- You have installed the OpenShift Serverless Operator.
- You have installed Knative Serving.
- You have created a project or have access to a project with the appropriate roles and permissions to create applications and other workloads.
- 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 cluster.

Procedure

1. Navigate to **CustomResourceDefinitions** and use the search box to find the **DomainMapping** custom resource definition (CRD).
2. Click the **DomainMapping** CRD, then navigate to the **Instances** tab.
3. Click **Create DomainMapping**.
4. Modify the YAML for the **DomainMapping** CR so that it includes the following information for your instance:

```
apiVersion: serving.knative.dev/v1beta1
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 domain mapping to a Knative service

```
apiVersion: serving.knative.dev/v1beta1
kind: DomainMapping
metadata:
 name: custom-ksvc-domain.example.com
 namespace: default
spec:
 ref:
   name: showcase
   kind: Service
   apiVersion: serving.knative.dev/v1
```

Verification

- Access the custom domain by using a **curl** request. For example:

Example command

```
$ curl custom-ksvc-domain.example.com
```

Example output

```
{"artifact":"knative-showcase","greeting":"Welcome"}
```

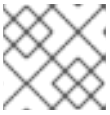
11.5.2. Restricting cipher suites by using the Administrator perspective

When you specify **net-kourier** for ingress and use **DomainMapping**, the TLS for OpenShift routing is set to passthrough, and TLS is handled by the Kourier Gateway. In such cases, you might need to restrict which TLS cipher suites for Kourier are allowed for users.

Prerequisites

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

- You have installed Knative Serving.
- You have created a project or have access to a project with the appropriate roles and permissions to create applications and other workloads.



NOTE

Your custom domain must point to the IP address of the cluster.

Procedure

- In the **KnativeServing** CR, use the **cipher-suites** value to specify the cipher suites you want to enable:

KnativeServing CR example

```
spec:
  config:
    kourier:
      cipher-suites: ECDHE-ECDSA-AES128-GCM-SHA256,ECDHE-ECDSA-CHACHA20-
POLY1305
```

Other cipher suites will be disabled. You can specify multiple suites by separating them with commas.

NOTE

The Kourier Gateway's container image utilizes the Envoy proxy image, and the default enabled cipher suites depend on the version of the Envoy proxy.

11.6. SECURING A MAPPED SERVICE USING A TLS CERTIFICATE

11.6.1. Securing a service with a custom domain by using a TLS certificate

After you have configured a custom domain for a Knative service, you can use a TLS certificate to secure the mapped service. To do this, you must create a Kubernetes TLS secret, and then update the **DomainMapping** CR to use the TLS secret that you have created.

Prerequisites

- You configured a custom domain for a Knative service and have a working **DomainMapping** CR.
- You have a TLS certificate from your Certificate Authority provider or a self-signed certificate.
- You have obtained the **cert** and **key** files from your Certificate Authority provider, or a self-signed certificate.
- Install the OpenShift CLI (**oc**).

Procedure

1. Create a Kubernetes TLS secret:

```
$ oc create secret tls <tls_secret_name> --cert=<path_to_certificate_file> --key=
<path_to_key_file>
```

2. Add the **networking.internal.knative.dev/certificate-uid: <id>`** label to the Kubernetes TLS secret:

```
$ oc label secret <tls_secret_name> networking.internal.knative.dev/certificate-uid="<id>"
```

If you are using a third-party secret provider such as cert-manager, you can configure your secret manager to label the Kubernetes TLS secret automatically. Cert-manager users can use the secret template offered to automatically generate secrets with the correct label. In this case, secret filtering is done based on the key only, but this value can carry useful information such as the certificate ID that the secret contains.

NOTE

The cert-manager Operator for Red Hat OpenShift is a Technology Preview feature. For more information, see the **Installing the cert-manager Operator for Red Hat OpenShift** documentation.

3. Update the **DomainMapping** CR to use the TLS secret that you have created:

```
apiVersion: serving.knative.dev/v1beta1
kind: DomainMapping
metadata:
  name: <domain_name>
  namespace: <namespace>
spec:
  ref:
    name: <service_name>
    kind: Service
    apiVersion: serving.knative.dev/v1
# TLS block specifies the secret to be used
  tls:
    secretName: <tls_secret_name>
```

Verification

1. Verify that the **DomainMapping** CR status is **True**, and that the **URL** column of the output shows the mapped domain with the scheme **https**:

```
$ oc get domainmapping <domain_name>
```

Example output

```
NAME                URL                         READY   REASON
example.com         https://example.com         True
```

2. Optional: If the service is exposed publicly, verify that it is available by running the following command:

```
$ curl https://<domain_name>
```

If the certificate is self-signed, skip verification by adding the **-k** flag to the **curl** command.

11.6.2. Improving net-kourier memory usage by using secret filtering

By default, the informers implementation for the Kubernetes **client-go** library fetches all resources of a particular type. This can lead to a substantial overhead when many resources are available, which can cause the Knative **net-kourier** ingress controller to fail on large clusters due to memory leaking. However, a filtering mechanism is available for the Knative **net-kourier** ingress controller, which enables the controller to only fetch Knative related secrets.

The secret filtering is enabled by default on the OpenShift Serverless Operator side. An environment variable, **ENABLE_SECRET_INFORMER_FILTERING_BY_CERT_UID=true**, is added by default to the **net-kourier** controller pods.

IMPORTANT

If you enable secret filtering, all of your secrets need to be labeled with **networking.internal.knative.dev/certificate-uid: "<id>"**. Otherwise, Knative Serving does not detect them, which leads to failures. You must label both new and existing secrets.

Prerequisites

- You have cluster administrator permissions on OpenShift Container Platform, or you have cluster or dedicated administrator permissions on Red Hat OpenShift Service on AWS or OpenShift Dedicated.
- A project that you created or that you have roles and permissions for to create applications and other workloads.
- Install the OpenShift Serverless Operator and Knative Serving.
- Install the OpenShift CLI (**oc**).

You can disable the secret filtering by setting the **ENABLE_SECRET_INFORMER_FILTERING_BY_CERT_UID** variable to **false** by using the **workloads** field in the **KnativeServing** custom resource (CR).

Example KnativeServing CR

```
apiVersion: operator.knative.dev/v1beta1
kind: KnativeServing
metadata:
  name: knative-serving
  namespace: knative-serving
spec:
...
  workloads:
    - env:
        - container: controller
          envVars:
            - name: ENABLE_SECRET_INFORMER_FILTERING_BY_CERT_UID
              value: 'false'
      name: net-kourier-controller
```

CHAPTER 12. HIGH AVAILABILITY CONFIGURATION FOR KNATIVE SERVING

12.1. HIGH AVAILABILITY FOR KNATIVE SERVICES

High availability (HA) is a standard feature of Kubernetes APIs that helps to ensure that APIs stay operational if a disruption occurs. In an HA deployment, if an active controller crashes or is deleted, another controller is readily available. This controller takes over processing of the APIs that were being serviced by the controller that is now unavailable.

HA in OpenShift Serverless is available through leader election, which is enabled by default after the Knative Serving or Eventing control plane is installed. When using a leader election HA pattern, instances of controllers are already scheduled and running inside the cluster before they are required. These controller instances compete to use a shared resource, known as the leader election lock. The instance of the controller that has access to the leader election lock resource at any given time is called the leader.

12.2. HIGH AVAILABILITY FOR KNATIVE DEPLOYMENTS

High availability (HA) is available by default for the Knative Serving **activator**, **autoscaler**, **autoscaler-hpa**, **controller**, **webhook**, **domain-mapping**, **domainmapping-webhook**, **kourier-control**, and **kourier-gateway** components, which are configured to have two replicas each. You can change the number of replicas for these components by modifying the **spec.high-availability.replicas** value in the **KnativeServing** custom resource (CR).

12.2.1. Configuring high availability replicas for Knative Serving

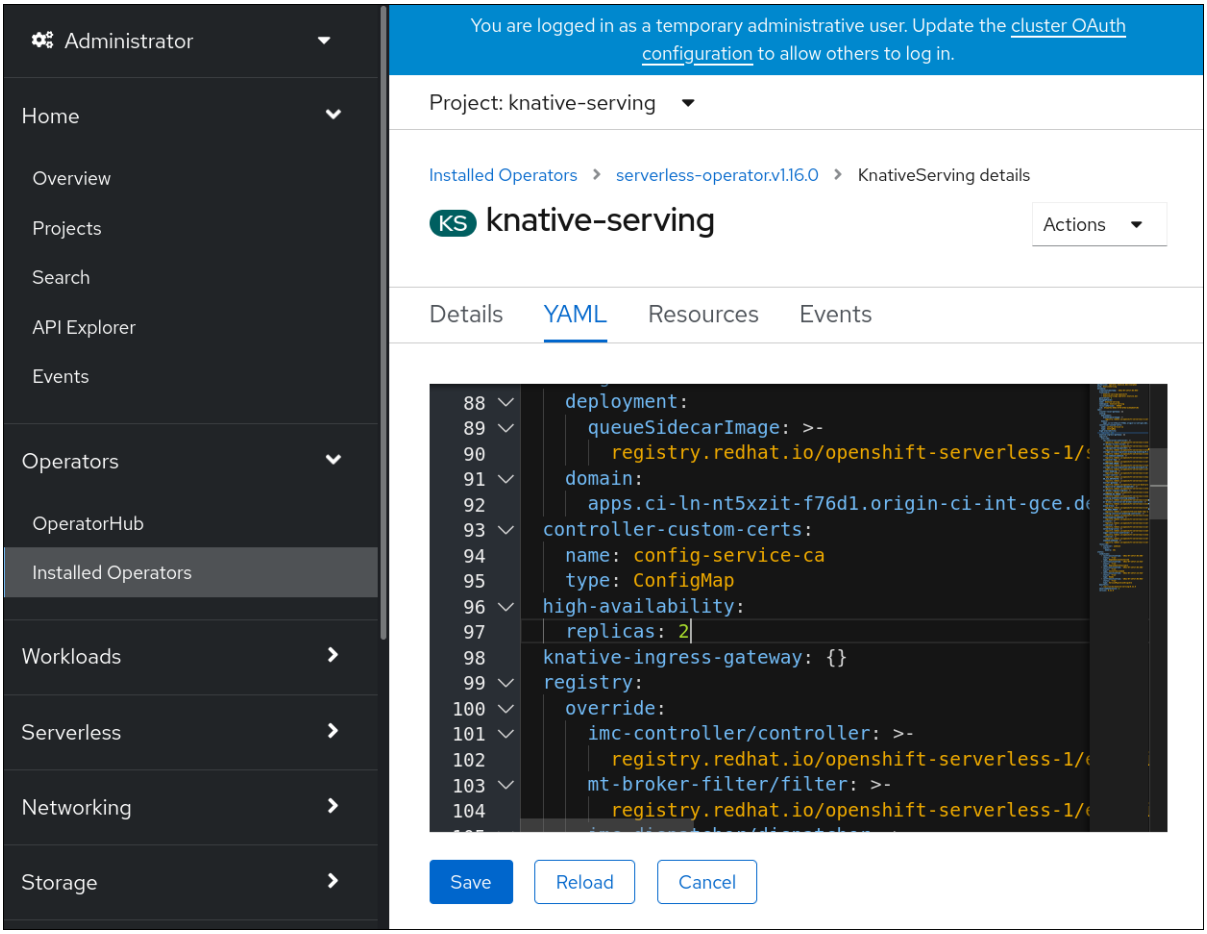
To specify three minimum replicas for the eligible deployment resources, set the value of the field **spec.high-availability.replicas** in the custom resource to **3**.

Prerequisites

- You have cluster administrator permissions on OpenShift Container Platform, or you have cluster or dedicated administrator permissions on Red Hat OpenShift Service on AWS or OpenShift Dedicated.
- The OpenShift Serverless Operator and Knative 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** CR:

Example YAML

```
apiVersion: operator.knative.dev/v1beta1
kind: KnativeServing
metadata:
  name: knative-serving
  namespace: knative-serving
spec:
  high-availability:
    replicas: 3
```

CHAPTER 13. TUNING SERVING CONFIGURATION

13.1. OVERRIDING KNATIVE SERVING SYSTEM DEPLOYMENT CONFIGURATIONS

You can override the default configurations for some specific deployments by modifying the **workloads** spec in the **KnativeServing** custom resources (CRs).

13.1.1. Overriding system deployment configurations

Currently, overriding default configuration settings is supported for the **resources**, **replicas**, **labels**, **annotations**, and **nodeSelector** fields, as well as for the **readiness** and **liveness** fields for probes.

In the following example, a **KnativeServing** CR overrides the **webhook** deployment so that:

- The **readiness** probe timeout for **net-kourier-controller** is set to be 10 seconds.
- The deployment has specified CPU and memory resource limits.
- The deployment has 3 replicas.
- The **example-label: label** label is added.
- The **example-annotation: annotation** annotation is added.
- The **nodeSelector** field is set to select nodes with the **disktype: hdd** label.

NOTE

The **KnativeServing** CR label and annotation settings override the deployment's labels and annotations for both the deployment itself and the resulting pods.

KnativeServing CR example

```
apiVersion: operator.knative.dev/v1beta1
kind: KnativeServing
metadata:
  name: ks
  namespace: knative-serving
spec:
  high-availability:
    replicas: 2
  workloads:
  - name: net-kourier-controller
    readinessProbes: 1
      - container: controller
        timeoutSeconds: 10
  - name: webhook
    resources:
    - container: webhook
      requests:
        cpu: 300m
        memory: 60Mi
```

```
    limits:
      cpu: 1000m
      memory: 1000Mi
  replicas: 3
  labels:
    example-label: label
  annotations:
    example-annotation: annotation
  nodeSelector:
    disktype: hdd
```

1 You can use the **readiness** and **liveness** probe overrides to override all fields of a probe in a container of a deployment as specified in the Kubernetes API except for the fields related to the probe handler: **exec**, **grpc**, **httpGet**, and **tcpSocket**.

Additional resources

- Probe configuration section of the Kubernetes API documentation

CHAPTER 14. CONFIGURING QUEUE PROXY RESOURCES

The Queue Proxy is a sidecar container to each application container within a service. It improves managing Serverless workloads, ensuring efficient resource usage. You can configure the Queue Proxy.

14.1. CONFIGURING QUEUE PROXY RESOURCES FOR A KNATIVE SERVICE

Apart from configuring the Queue Proxy resource requests and limits globally in the deployment configmap, you can set them at the service level using the corresponding annotations targeting CPU, memory, and ephemeral storage resource types.

Prerequisites

- Red Hat OpenShift Pipelines must be installed on your cluster.
- You have installed the OpenShift (**oc**) CLI.
- You have installed the Knative (**kn**) CLI.

Procedure

- Modify the configmap of your service with resource requests and limits:

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: example-service
  namespace: default
spec:
  template:
    metadata:
      annotations:
        queue.sidecar.serving.knative.dev/cpu-resource-request: "1"
        queue.sidecar.serving.knative.dev/cpu-resource-limit: "2"
        queue.sidecar.serving.knative.dev/memory-resource-request: "1Gi"
        queue.sidecar.serving.knative.dev/memory-resource-limit: "2Gi"
        queue.sidecar.serving.knative.dev/ephemeral-storage-resource-request: "400Mi"
        queue.sidecar.serving.knative.dev/ephemeral-storage-resource-limit: "450Mi"
```

Alternatively, you can use the special **queue.sidecar.serving.knative.dev/resource-percentage** annotation, which calculates the Queue Proxy resources as a percentage of the application container. When CPU and memory resource requirements are calculated from the application container requirements and they are outside the boundaries below, the values are adjusted to fit within the boundaries. In this case, the following minimum and maximum boundaries are applied to the CPU and memory resource requirements:

Table 14.1. Resource requirements boundaries

Resource requirements	Min	Max
CPU request	25m	100m

Resource requirements	Min	Max
CPU limit	40m	500m
Memory request	50Mi	200Mi
Memory limit	200Mi	500Mi

NOTE

If you simultaneously set a percentage annotation and a specific resource value using the corresponding resource annotation, then the latter takes precedence.

WARNING

The **queue.sidecar.serving.knative.dev/resource-percentage** annotation is now deprecated and will be removed in a future version of OpenShift Serverless.