OpenShift Container Platform 4.2

Serverless

OpenShift Serverless installation, usage, and release notes

Last Updated: 2019-10-16

OpenShift Container Platform 4.2 Serverless

OpenShift Serverless installation, usage, and release notes

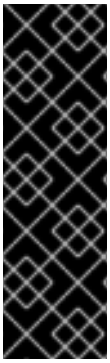
Legal Notice

Abstract

This document provides information on how to use OpenShift Serverless in OpenShift Container Platform 4.2

Table of Contents

CHAPTER 1. UNDERSTANDING OPENSHIFT SERVERLESS



IMPORTANT

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

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

OpenShift Serverless is based on the open source Knative project, which provides portability and consistency across hybrid and multi-cloud environments by enabling an enterprise-grade serverless platform. It implements the building blocks for developers to create modern, source-centric, container-based applications through a series of Custom Resource Definitions (CRDs) and associated controllers in Kubernetes.

OpenShift Serverless simplifies the process of getting code from development into production by reducing the requirement for developer input in infrastructure set up or back-end development. Developers on Red Hat OpenShift Serverless can use the provided Kubernetes-native APIs, as well as familiar languages and frameworks to deploy applications and container workloads.

Combining Operators, Knative and OpenShift Container Platform enables stateful, stateless, and serverless workloads to all run on a single multi-cloud container platform with automated operations. Developers can benefit from being able to use a single platform for hosting their microservices, legacy, and serverless applications. Applications are packaged as OCI compliant Linux containers that can be run anywhere.

Applications can be triggered by a variety of event sources, such as events from your own applications, cloud services from multiple providers, Software as a Service (SaaS) systems and Red Hat Services (AMQ Streams).

OpenShift Serverless applications can be integrated with other OpenShift Container Platform services, such as Service Mesh and cluster monitoring, delivering a complete serverless application development and deployment experience.

CHAPTER 2. OPENSHIFT SERVERLESS PRODUCT ARCHITECTURE

2.1. KNATIVE SERVING

Knative Serving on OpenShift Container Platform builds on Kubernetes and Istio to support deploying and serving serverless applications.

It creates a set of Kubernetes Custom Resource Definitions (CRDs) that are used to define and control the behavior of serverless workloads on an OpenShift Container Platform cluster.

These CRDs can be used as building blocks to address complex use cases, such as rapid deployment of serverless containers, automatic scaling of Pods, routing and network programming for Istio components, or viewing point-in-time snapshots of deployed code and configurations.

2.1.1. Knative Serving components

The components described in this section are the resources that Knative Serving requires to be configured and run correctly.

Knative service resource

The **service.serving.knative.dev** resource automatically manages the whole lifecycle of a serverless workload on a cluster. It controls the creation of other objects to ensure that an app has a route, a configuration, and a new revision for each update of the service. Services can be defined to always route traffic to the latest revision or to a pinned revision.

Knative route resource

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

Knative configuration resource

The **configuration.serving.knative.dev** resource maintains the required state for your deployment. Modifying a configuration creates a new revision.

Knative revision resource

The **revision.serving.knative.dev** resource is a point-in-time snapshot of the code and configuration for each modification made to the workload. Revisions are immutable objects and can be retained for as long as needed. Cluster administrators can modify the **revision.serving.knative.dev** resource to enable automatic scaling of Pods in your OpenShift Container Platform cluster.

2.2. KNATIVE CLIENT

The Knative Client (**kn**) extends the functionality of the **oc** or **kubectl** tools to enable interaction with Knative components on OpenShift Container Platform. **kn** allows developers to deploy and manage applications without editing YAML files directly.

CHAPTER 3. INSTALLING OPENSHIFT SERVERLESS

3.1. CLUSTER SIZE REQUIREMENTS

The cluster must be sized appropriately to ensure that OpenShift Serverless can run correctly. You can use the MachineSet API to manually scale your cluster up to the desired size.

An OpenShift cluster with 10 CPUs and 40 GB memory is the minimum requirement for getting started with your first serverless application. This usually means you must scale up one of the default MachineSets by two additional machines.

NOTE

For this configuration, the requirements depend on the deployed applications. By default, each pod requests ~400m of CPU and recommendations are based on this value. In the given recommendation, an application can scale up to 10 replicas. Lowering the actual CPU request of the application further pushes the boundary.

NOTE

The numbers given only relate to the pool of worker machines of the OpenShift cluster. Master nodes are not used for general scheduling and are omitted.

For more advanced use-cases, such as using OpenShift logging, monitoring, metering, and tracing, you must deploy more resources. Recommended requirements for such use-cases are 24 vCPUs and 96GB of memory.

Additional resources

For more information on using the MachineSet API, read about Creating MachineSets.

3.1.1. Scaling a MachineSet manually

If you must add or remove an instance of a machine in a MachineSet, you can manually scale the MachineSet.

Prerequisites

- Install an OpenShift Container Platform cluster and the **oc** command line.
- Log in to **oc** as a user with **cluster-admin** permission.

Procedure

1. View the MachineSets that are in the cluster:

```
$ oc get machinesets -n openshift-machine-api
```

   The MachineSets are listed in the form of **<clusterid>-worker-<aws-region-az>**.

2. Scale the MachineSet:

```
$ oc scale --replicas=2 machineset <machineset> -n openshift-machine-api
```

Or:

```
$ oc edit machineset <machineset> -n openshift-machine-api
```

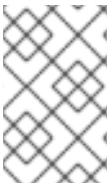
You can scale the MachineSet up or down. It takes several minutes for the new machines to be available.

IMPORTANT

By default, the OpenShift Container Platform router pods are deployed on workers. Because the router is required to access some cluster resources, including the web console, do not scale the worker MachineSet to **0** unless you first relocate the router pods.

3.2. INSTALLING SERVICE MESH

An installed version of Service Mesh is required for the installation of OpenShift Serverless. For details, see the OpenShift Container Platform documentation on Installing Service Mesh.



NOTE

Use the Service Mesh documentation for Operator installation only. Once you install the Operators, use the documentation below to install the Service Mesh Control Plane and Member Roll.

3.2.1. Installing the ServiceMeshControlPlane

Service Mesh is comprised of a data plane and a control plane. After you install the ServiceMesh operator, you can install the control plane. The control plane manages and configures the sidecar proxies to enforce policies and collect telemetry. The following procedure installs a version of the ServiceMesh control plane that acts as an ingress to your applications.

NOTE

You must install the control plane into the **istio-system** namespace. Other namespaces are currently not supported.

Sample YAML file

```
apiVersion: maistra.io/v1
kind: ServiceMeshControlPlane
metadata:
  name: basic-install
  namespace: istio-system
spec:
  istio:
    global:
      multitenant: true
      proxy:
        autoInject: disabled
      omitSidecarInjectorConfigMap: true
      disablePolicyChecks: false
      defaultPodDisruptionBudget:
```

```
    enabled: false
istio_cni:
  enabled: true
gateways:
  istio-ingressgateway:
    autoscaleEnabled: false
    type: LoadBalancer
  istio-egressgateway:
    enabled: false
  cluster-local-gateway:
    autoscaleEnabled: false
    enabled: true
    labels:
      app: cluster-local-gateway
      istio: cluster-local-gateway
    ports:
      - name: status-port
        port: 15020
      - name: http2
        port: 80
        targetPort: 8080
      - name: https
        port: 443
mixer:
  enabled: false
  policy:
    enabled: false
  telemetry:
    enabled: false
pilot:
  autoscaleEnabled: false
  sidecar: false
kiali:
  enabled: false
tracing:
  enabled: false
prometheus:
  enabled: false
grafana:
  enabled: false
sidecarInjectorWebhook:
  enabled: false
```

NOTE

Autoscaling is disabled in this version. This release is not intended for production use.

NOTE

Running ServiceMesh with a sidecar injection enabled with OpenShift Serverless is currently not recommended.

Prerequisite

- An account with cluster administrator access.

- The ServiceMesh operator must be installed.

Procedure

1. Log in to your OpenShift Container Platform installation as a cluster administrator.
2. Run the following command to create the **istio-system** namespace:

```
$ oc new-project istio-system
```

3. Copy the sample YAML file into a **smcp.yaml** file.
4. Apply the YAML file using the command:

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

5. Run this command to watch the progress of the pods during the installation process:

```
$ oc get pods -n istio-system -w
```

3.2.2. Installing a ServiceMeshMemberRoll

You must have a Service Mesh Member Roll for the control plane namespace, if the Service Mesh is configured for multi-tenancy. For applications to use the deployed control plane and ingress, their namespaces must be part of a member roll.

A multi-tenant control plane installation only affects namespaces configured as part of the Service Mesh. You must specify the namespaces associated with the Service Mesh in a **ServiceMeshMemberRoll** resource located in the same namespace as the **ServiceMeshControlPlane** resource and name it **default**.

ServiceMeshMemberRoll Custom Resource Example

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

Prerequisites

- Installed Service Mesh Operator.
- A custom resource file that defines the parameters of your Red Hat OpenShift Service Mesh control plane.

Procedure

1. Create a YAML file that replicates the ServiceMeshMemberRoll Custom Resource sample.

2. Configure the YAML file to include relevant namespaces.

NOTE

Add all namespaces to which you want to deploy serverless applications. Ensure you retain the **knative-serving** namespace in the member roll.

3. Copy the configured YAML into a file **smmr.yaml** and apply it using:

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

3.3. INSTALLING THE OPENSHIFT SERVERLESS OPERATOR

The OpenShift Serverless Operator can be installed using the OpenShift Container Platform instructions for installing Operators.

You can install the OpenShift Serverless Operator in the host cluster by following the OpenShift Container Platform instructions on installing an Operator.

NOTE

The OpenShift Serverless Operator only works for OpenShift Container Platform versions 4.1.13 and later.

For details, see the OpenShift Container Platform documentation on adding Operators to a cluster.

3.4. INSTALLING KNATIVE SERVING

You must create a **KnativeServing** object to install Knative Serving using the OpenShift Serverless Operator.

Sample serving.yaml

```
apiVersion: v1
kind: Namespace
metadata:
 name: knative-serving
---
apiVersion: serving.knative.dev/v1alpha1
kind: KnativeServing
metadata:
 name: knative-serving
 namespace: knative-serving
```

Prerequisite

- An account with cluster administrator access.
- Installed OpenShift Serverless Operator.

Procedure

1. Copy the sample YAML file into **serving.yaml** and apply it using:

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

2. Verify the installation is complete by using the command:

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

Results should be similar to:

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

3.5. UNINSTALLING KNATIVE SERVING

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

Prerequisite

- Installed Knative Serving

Procedure

1. To remove Knative Serving, use the following command:

```
$ oc delete knativeserving knative-serving -n knative-serving
```

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

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

3.6. DELETING THE OPENSHIFT SERVERLESS OPERATOR

You can remove the OpenShift Serverless Operator from the host cluster by following the OpenShift Container Platform instructions on deleting an Operator.

For details, see the OpenShift Container Platform documentation on deleting Operators from a cluster.

3.7. DELETING KNATIVE SERVING CRDS FROM THE OPERATOR

After uninstalling the OpenShift Serverless Operator, the Operator CRDs and API services remain on the cluster. Use this procedure to completely uninstall the remaining components.

Prerequisite

- You have uninstalled Knative Serving and removed the OpenShift Serverless Operator using the previous procedure.

Procedure

1. Run the following command to delete the remaining Knative Serving CRDs:

```
$ oc delete crd knativeservings.serving.knative.dev
```

2. Delete the Knative Serving API by using command:

```
$ oc delete apiservice v1alpha1.serving.knative.dev
```

CHAPTER 4. GETTING STARTED WITH KNATIVE SERVICES

Knative services are Kubernetes services that a user creates to deploy a serverless application. Each Knative service is defined by a route and a configuration, contained in a **.yaml** file.

4.1. CREATING A KNATIVE SERVICE

To create a service, you must create the **service.yaml** file.

You can copy the sample below. This sample will create a sample golang application called **helloworld-go** and allows you to specify the image for that application.

```
apiVersion: serving.knative.dev/v1alpha1 # Current version of Knative
kind: Service
metadata:
  name: helloworld-go # The name of the app
  namespace: default # The namespace the app will use
spec:
  template:
    spec:
      containers:
        - image: gcr.io/knative-samples/helloworld-go # The URL to the image of the app
          env:
            - name: TARGET # The environment variable printed out by the sample app
              value: "Go Sample v1"
```

4.2. DEPLOYING A SERVERLESS APPLICATION

To deploy a serverless application, you must apply the **service.yaml** file.

Procedure

1. Navigate to the directory where the **service.yaml** file is contained.
2. Deploy the application by applying the **service.yaml** file.

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

Now that service has been created and the application has been deployed, Knative will create a new immutable revision for this version of the application.

Knative will also perform network programming to create a route, ingress, service, and load balancer for your application, and will automatically scale your pods up and down based on traffic, including inactive pods.

CHAPTER 5. MONITORING OPENSHIFT SERVERLESS COMPONENTS

As an OpenShift Container Platform cluster administrator, you can deploy the OpenShift Container Platform monitoring stack and monitor the metrics of OpenShift Serverless components.

When using the OpenShift Serverless Operator, the required ServiceMonitor objects are created automatically for monitoring the deployed components.

OpenShift Serverless components, such as Knative Serving, expose metrics data. Administrators can monitor this data by using the OpenShift Container Platform web console.

5.1. CONFIGURING CLUSTER FOR APPLICATION MONITORING

Before application developers can monitor their applications, the human operator of the cluster needs to configure the cluster accordingly. This procedure shows how to.

Prerequisites

- You must log in as a user that belongs to a role with administrative privileges for the cluster.

Procedure

1. In the OpenShift Container Platform web console, navigate to the **Operators → OperatorHub** page and install the Prometheus Operator in the namespace where your application is.

2. Navigate to the **Operators → Installed Operators** page and install Prometheus, Alertmanager, Prometheus Rule, and Service Monitor in the same namespace.

5.2. VERIFYING AN OPENSHIFT CONTAINER PLATFORM MONITORING INSTALLATION FOR USE WITH KNATIVE SERVING

Manual configuration for monitoring by an administrator is not required, but you can carry out these steps to verify that monitoring is installed correctly.

Procedure

1. Verify that the ServiceMonitor objects are deployed.

```
$ oc get servicemonitor -n knative-serving
NAME         AGE
activator    11m
autoscaler   11m
controller   11m
```

2. Verify that the **openshift.io/cluster-monitoring=true** label has been added to the Knative Serving namespace:

```
$ oc get namespace knative-serving --show-labels
NAME              STATUS   AGE   LABELS
knative-serving   Active   4d    istio-injection=enabled,openshift.io/cluster-
monitoring=true,serving.knative.dev/release=v0.7.0
```

5.3. MONITORING KNATIVE SERVING USING THE OPENSHIFT CONTAINER PLATFORM MONITORING STACK

This section provides example instructions for the visualization of Knative Serving Pod autoscaling metrics by using the OpenShift Container Platform monitoring tools.

Prerequisites

- You must have the OpenShift Container Platform monitoring stack installed.

Procedure

1. Navigate to the OpenShift Container Platform web console and authenticate.
2. Navigate to **Monitoring → Metrics**.
3. Enter the **Expression** and select **Run queries**. To monitor Knative Serving autoscaler Pods, use this example expression.

   ```
   autoscaler_actual_pods
   ```

   You will now see monitoring information for the Knative Serving autoscaler Pods in the console.

CHAPTER 6. CLUSTER LOGGING WITH OPENSHIFT SERVERLESS

6.1. ABOUT CLUSTER LOGGING

As an OpenShift Container Platform cluster administrator, you can deploy cluster logging to aggregate logs for a range of OpenShift Container Platform services. As an OpenShift Container Platform cluster administrator, you can deploy cluster logging to aggregate logs for a range of OpenShift Container Platform services.

The cluster logging components are based upon Elasticsearch, Fluentd or Rsyslog, and Kibana. The collector, Fluentd, is deployed to each node in the OpenShift Container Platform cluster. It collects all node and container logs and writes them to Elasticsearch (ES). Kibana is the centralized, web UI where users and administrators can create rich visualizations and dashboards with the aggregated data.

OpenShift Container Platform cluster administrators can deploy cluster logging using a few CLI commands and the OpenShift Container Platform web console to install the Elasticsearch Operator and Cluster Logging Operator. When the operators are installed, create a Cluster Logging Custom Resource (CR) to schedule cluster logging pods and other resources necessary to support cluster logging. The operators are responsible for deploying, upgrading, and maintaining cluster logging.

OpenShift Container Platform cluster administrators can deploy cluster logging using a few CLI commands and the OpenShift Container Platform web console to install the Elasticsearch Operator and Cluster Logging Operator. When the operators are installed, create a Cluster Logging Custom Resource (CR) to schedule cluster logging pods and other resources necessary to support cluster logging. The operators are responsible for deploying, upgrading, and maintaining cluster logging.

You can configure cluster logging by modifying the Cluster Logging Custom Resource (CR), named **instance**. The CR defines a complete cluster logging deployment that includes all the components of the logging stack to collect, store and visualize logs. The Cluster Logging Operator watches the **ClusterLogging** Custom Resource and adjusts the logging deployment accordingly.

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

6.2. ABOUT DEPLOYING AND CONFIGURING CLUSTER LOGGING

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

The installation instructions that follow include a sample Cluster Logging Custom Resource (CR), which you can use to create a cluster logging instance and configure your cluster logging deployment.

If you want to use the default cluster logging install, you can use the sample CR directly.

If you want to customize your deployment, make changes to the sample CR as needed. The following describes the configurations you can make when installing your cluster logging instance or modify after installtion. See the Configuring sections for more information on working with each component, including modifications you can make outside of the Cluster Logging Custom Resource.

6.2.1. Configuring and Tuning Cluster Logging

You can configure your cluster logging environment by modifying the Cluster Logging Custom Resource deployed in the **openshift-logging** project.

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

Management state

The Cluster Logging Operator and Elasticsearch Operator can be in a *Managed* or *Unmanaged* state.

In managed state, the Cluster Logging Operator (CLO) responds to changes in the Cluster Logging Custom Resource (CR) and attempts to update the cluster to match the CR.

In order to modify certain components managed by the Cluster Logging Operator or the Elasticsearch Operator, you must set the operator to the *unmanaged* state.

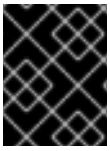
In Unmanaged state, the operators do not respond to changes in the CRs. The administrator assumes full control of individual component configurations and upgrades when in unmanaged state.

NOTE

The OpenShift Container Platform documentation indicates in a prerequisite step when you must set the cluster to Unmanaged.

```
spec:
  managementState: "Managed"
```

The OpenShift Container Platform documentation indicates in a prerequisite step when you must set the cluster to Unmanaged.



IMPORTANT

An unmanaged deployment will not receive updates until the **ClusterLogging** custom resource is placed back into a managed state.

Memory and CPU

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

```
spec:
  logStore:
    elasticsearch:
      resources:
        limits:
          cpu:
          memory:
        requests:
          cpu: 1
          memory: 16Gi
      type: "elasticsearch"
  collection:
    logs:
      fluentd:
        resources:
          limits:
            cpu:
            memory:
          requests:
```

```
          cpu:
          memory:
      type: "fluentd"
  visualization:
    kibana:
      resources:
        limits:
          cpu:
          memory:
        requests:
          cpu:
          memory:
    type: kibana
  curation:
    curator:
      resources:
        limits:
          memory: 200Mi
        requests:
          cpu: 200m
          memory: 200Mi
      type: "curator"
```

Elasticsearch storage

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

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

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

NOTE

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

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

Elasticsearch replication policy

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

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

Curator schedule

You specify the schedule for Curator in the [cron format](https://en.wikipedia.org/wiki/Cron).

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

6.2.2. Sample modified Cluster Logging Custom Resource

The following is an example of a Cluster Logging Custom Resource modified using the options previously described.

Sample modified Cluster Logging Custom Resource

```
apiVersion: "logging.openshift.io/v1alpha1"
kind: "ClusterLogging"
metadata:
  name: "instance"
  namespace: "openshift-logging"
spec:
  managementState: "Managed"
  logStore:
    type: "elasticsearch"
    elasticsearch:
      nodeCount: 2
      resources:
        limits:
          memory: 2Gi
        requests:
          cpu: 200m
          memory: 2Gi
      storage: {}
      redundancyPolicy: "SingleRedundancy"
  visualization:
    type: "kibana"
    kibana:
      resources:
        limits:
```

```
        memory: 1Gi
      requests:
        cpu: 500m
        memory: 1Gi
    replicas: 1
curation:
  type: "curator"
  curator:
    resources:
      limits:
        memory: 200Mi
      requests:
        cpu: 200m
        memory: 200Mi
    schedule: "*/5 * * * *"
collection:
  logs:
    type: "fluentd"
    fluentd:
      resources:
        limits:
          memory: 1Gi
        requests:
          cpu: 200m
          memory: 1Gi
```

6.3. USING CLUSTER LOGGING TO FIND LOGS FOR KNATIVE SERVING COMPONENTS

Procedure

1. To open the Kibana UI, the visualization tool for Elasticsearch, use the following command to get the Kibana route:

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

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

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

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

NOTE

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

6.4. USING CLUSTER LOGGING TO FIND LOGS FOR SERVICES DEPLOYED WITH KNATIVE SERVING

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

Procedure

1. Use the following command to find the URL to Kibana:

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

2. Enter the URL in your browser to open the Kibana UI.

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

4. Filter the logs by using the Kubernetes namespace your service is deployed in. Add a filter to identify the service itself: **kubernetes.namespace_name:default AND kubernetes.labels.serving_knative_dev\/service:{SERVICE_NAME}**.

NOTE

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

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

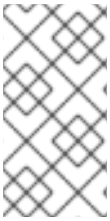
NOTE

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

CHAPTER 7. CONFIGURING KNATIVE SERVING AUTOSCALING

OpenShift Serverless provides capabilities for automatic Pod scaling, including scaling inactive Pods to zero, by enabling the Knative Serving autoscaling system in an OpenShift Container Platform cluster.

To enable autoscaling for Knative Serving, you must configure concurrency and scale bounds in the revision template.



NOTE

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

7.1. CONFIGURING CONCURRENT REQUESTS FOR KNATIVE SERVING AUTOSCALING

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

Here is an example of **target** being used in a revision template:

```
apiVersion: serving.knative.dev/v1alpha1
kind: Service
metadata:
  name: myapp
spec:
  template:
    metadata:
      annotations:
        autoscaling.knative.dev/target: 50
    spec:
      containers:
      - image: myimage
```

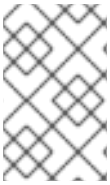
Here is an example of **containerConcurrency** being used in a revision template:

```
apiVersion: serving.knative.dev/v1alpha1
kind: Service
metadata:
  name: myapp
spec:
  template:
    metadata:
      annotations:
    spec:
      containerConcurrency: 100
      containers:
      - image: myimage
```

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

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

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



NOTE

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

7.1.1. Configuring concurrent requests using the target annotation

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

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

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

7.1.2. Configuring concurrent requests using the containerConcurrency field

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

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

0
: allows unlimited concurrent requests.

1
: guarantees that only one request is handled at a time by a given instance of the revision container.

2 or more
: will limit request concurrency to that value.

NOTE

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

7.2. CONFIGURING SCALE BOUNDS KNATIVE SERVING AUTOSCALING

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

minScale

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

maxScale

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

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

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

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

NOTE

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

CHAPTER 8. USING KNATIVE CLIENT

Knative Client (**kn**) is the Knative command line interface (CLI). The CLI exposes commands for managing your applications, as well as lower level tools to interact with components of OpenShift Container Platform. With **kn**, you can create applications and manage OpenShift Container Platform projects from the terminal.

Knative client does not have its own log in mechanism. To log in to the cluster you must install the **oc** CLI and use the **oc** login. Installation options for the CLI vary depending on your operating system.

8.1. INSTALLING THE CLI

You can install the CLI in order to interact with OpenShift Container Platform using a command-line interface.

IMPORTANT

If you installed an earlier version of **oc**, you cannot use it to complete all of the commands in OpenShift Container Platform 4.2. Download and install the new version of **oc**.

Procedure

1. From the OpenShift Infrastructure Providers page, click **Download Command-line Tools**.
2. Click the folder for your operating system and architecture and click the compressed file.

NOTE

You can install **oc** on Linux, Windows, or macOS.

3. Save the file to your file system.
4. Extract the compressed file.
5. Place it in a directory that is on your **PATH**.

After you install the CLI, it is available using the **oc** command:

```
$ oc <command>
```

8.1.1. Installing the kn CLI for Linux

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

Procedure

1. Download the CLI.
2. Unpack the archive:

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

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

```
$ echo $PATH
```

NOTE

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

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

8.1.2. Installing the kn CLI for macOS

kn for macOS is provided as a **tar.gz** archive.

Procedure

1. Download the CLI.
2. Unpack and unzip the archive.
3. Move the **kn** binary to a directory on your PATH.
4. To check your PATH, open a terminal window and run:

```
$ echo $PATH
```

8.1.3. Installing the kn CLI for Windows

The CLI for Windows is provided as a zip archive.

Procedure

1. Download the CLI.
2. Unzip the archive with a ZIP program.
3. Move the **kn** binary to a directory on your PATH.
4. To check your PATH, open the Command Prompt and run the command:

```
C:\> path
```

8.2. LOGGING IN TO THE CLI

You can log in to the **oc** CLI to access and manage your cluster.

Prerequisites

- You must have access to an OpenShift Container Platform cluster.
- You must have installed the CLI.

Procedure

- Log in to the CLI using the **oc login** command and enter the required information when prompted.

```
$ oc login
Server [https://localhost:8443]: https://openshift.example.com:6443 (1)
The server uses a certificate signed by an unknown authority.
You can bypass the certificate check, but any data you send to the server could be intercepted by others.
Use insecure connections? (y/n): y (2)

Authentication required for https://openshift.example.com:6443 (openshift)
Username: user1 (3)
Password: (4)
Login successful.

You don't have any projects. You can try to create a new project, by running

    oc new-project <projectname>

Welcome! See 'oc help' to get started.
```

1. Enter the OpenShift Container Platform server URL.
2. Enter whether to use insecure connections.
3. Enter the user name to log in as.
4. Enter the user's password.

You can now create a project or issue other commands for managing your cluster.

8.3. BASIC WORKFLOW USING KNATIVE CLIENT

Use this basic workflow to create, read, update, delete (CRUD) operations on a service. The following example deploys a simple Hello World service that reads the environment variable **TARGET** and prints its output.

Procedure

1. Create a service in the **default** namespace from an image.

```
$ kn service create hello --image gcr.io/knative-samples/helloworld-go --env TARGET=Knative

Service 'hello' successfully created in namespace 'default'.
Waiting for service 'hello' to become ready ... OK
```

```
Service URL:
http://hello.default.apps-crc.testing
```

2. List the service.

```
$ kn service list

NAME    URL                                GENERATION   AGE   CONDITIONS   READY
REASON
hello   http://hello.default.apps-crc.testing   1          85s   3 OK / 3     True
```

3. Check if the service is working by using the **curl** service endpoint command:

```
$ curl http://hello.default.apps-crc.testing

Hello Knative!
```

4. Update the service.

```
$ kn service update hello --env TARGET=Kn

Waiting for service 'hello' to become ready ... OK
Service 'hello' updated in namespace 'default'.
```

The service's environment variable **TARGET** is now set to **Kn**.

5. Describe the service.

```
$ kn service describe hello

Name:         hello
Namespace:    default
URL:          http://hello.default.apps-crc.testing
Address:      http://hello.default.svc.cluster.local
Annotations:  serving.knative.dev/creator=kube:admin,
serving.knative.dev/lastModifier=kube:admin
Age:          34m

Revisions:
  100%  Name:    hello-fszsh-2 [2] (35s)
        Image:   gcr.io/knative-samples/helloworld-go (pinned to 5ea96ba4b872)
        Env:     TARGET=Kn
        Memory:  100M ... 200M
        CPU:     400m ... 1

Conditions:
  OK TYPE                   AGE REASON
  ++ ConfigurationsReady    22s
  ++ Ready                  22s
  ++ RoutesReady            22s ribe hello
```

6. Delete the service.

```
$ kn service delete hello

Service 'hello' successfully deleted in namespace 'default'.
```

You can then verify that the 'hello' service is deleted by attempting to **list** it.

```
$ kn service list hello

No services found.
```

8.4. AUTOSCALING WORKFLOW USING KNATIVE CLIENT

You can access autoscaling capabilities by using **kn** to modify Knative services without editing YAML files directly.

Use the **service create** and **service update** commands with the appropriate flags to configure the autoscaling behavior.

Flag	Description
--concurrency-limit int	Hard limit of concurrent requests to be processed by a single replica.
--concurrency-target int	Recommendation for when to scale up based on the concurrent number of incoming requests. Defaults to **--concurrency-limit**.
--max-scale int	Maximum number of replicas.
--min-scale int	Minimum number of replicas.

8.5. TRAFFIC SPLITTING USING KNATIVE CLIENT

kn helps you control which revisions get routed traffic on your Knative service.

Knative service allows for traffic mapping, which is the mapping of revisions of the service to an allocated portion of traffic. It offers the option to create unique URLs for particular revisions and has the ability to assign traffic to the latest revision.

With every update to the configuration of the service, a new revision is created with the service route pointing all the traffic to the latest ready revision by default.

You can change this behavior by defining which revision gets a portion of the traffic.

Procedure

- Use the **kn service update** command with the **--traffic** flag to update the traffic.

NOTE

--traffic RevisionName=Percent uses the following syntax:

- The **--traffic** flag requires two values separated by separated by an equals sign (**=**).
- The **RevisionName** string refers to the name of the revision.
- **Percent** integer denotes the traffic portion assigned to the revision.
- Use identifier **@latest** for the RevisionName to refer to the latest ready revision of the service. You can use this identifier only once with the **--traffic** flag.
- If the **service update** command updates the configuration values for the service along with traffic flags, the **@latest** reference will point to the created revision to which the updates are applied.
- **--traffic** flag can be specified multiple times and is valid only if the sum of the **Percent** values in all flags totals 100.

NOTE

For example, to route 10% of traffic to your new revision before putting all traffic on, use the following command:

```
$ kn service update svc --traffic @latest=10 --traffic svc-vwxyz=90
```

8.5.1. Assigning tag revisions

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

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

NOTE

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

Procedure

- Use the following command:

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

NOTE

--tag RevisionName=Tag uses the following syntax:

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

8.5.2. Unassigning tag revisions

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

NOTE

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

Procedure

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

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

NOTE

--untag Tag uses the following syntax:

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

8.5.3. Traffic flag operation precedence

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

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

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

8.5.4. Traffic splitting flags

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

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

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

CHAPTER 9. OPENSHIFT SERVERLESS RELEASE NOTES

For an overview of OpenShift Serverless functionality, see Understanding OpenShift Serverless.

9.1. GETTING SUPPORT

If you experience difficulty with a procedure described in this documentation, visit the Customer Portal to learn more about support for Technology Preview features.

9.2. RELEASE NOTES FOR RED HAT OPENSHIFT SERVERLESS TECHNOLOGY PREVIEW 1.0.0

9.2.1. New features

This release of OpenShift Serverless introduces the OpenShift Serverless Operator, which supports Knative Serving 0.7.1 and is tested for OpenShift Service Mesh 1.0.

9.2.2. Known issues

The following limitations exist in OpenShift Serverless at this time:

- The Knative Serving Operator should wait for ServiceMeshMemberRoll to include the **knative-serving** namespace. The installation procedure recommends creating the **knative-serving** namespace and then installing the operator. Istio does not consider the **knative-serving** namespace to be in the ServiceMeshMemberRoll when the Knative Serving Pods are being created. Consequently, the sidecars are not injected.

- Knative service returns a 503 status code when the cluster is running for a long time. The Knative Serving Pods do not show any errors. Restarting the **istio-pilot** Pod temporarily fixes the issue.

- The gRPC and HTTP2 do not work against routes. This is a known limitation of OpenShift routes.