OpenShift Container Platform 4.9 Serverless

OpenShift Serverless installation, usage, and release notes
Abstract

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

CHAPTER 1. RELEASE NOTES .......................................................... 11
1.1. ABOUT API VERSIONS ......................................................... 11
1.2. DEPRECATED AND REMOVED FEATURES ................................. 11
1.3. RELEASE NOTES FOR RED HAT OPENSHIFT SERVERLESS 1.20.0 .... 12
   1.3.1. New features .......................................................... 12
   1.3.2. Known issues .......................................................... 12
1.4. RELEASE NOTES FOR RED HAT OPENSHIFT SERVERLESS 1.19.0 .... 14
   1.4.1. New features .......................................................... 14
   1.4.2. Fixed issues ............................................................ 14
   1.4.3. Known issues .......................................................... 14
1.5. RELEASE NOTES FOR RED HAT OPENSHIFT SERVERLESS 1.18.0 .... 15
   1.5.1. New features .......................................................... 15
   1.5.2. Fixed issues ............................................................ 16
   1.5.3. Known issues .......................................................... 16

CHAPTER 2. DISCOVER ............................................................... 17
2.1. ABOUT OPENSHIFT SERVERLESS ............................................. 17
   2.1.1. Knative Serving ....................................................... 17
      2.1.1.1. Knative Serving resources .................................... 17
   2.1.2. Knative Eventing ..................................................... 17
      2.1.3. Supported configurations ....................................... 18
2.2. ABOUT OPENSHIFT SERVERLESS FUNCTIONS ............................. 18
   2.2.1. Supported runtimes ................................................ 19
      2.2.2. Next steps .......................................................... 19
2.3. EVENT SOURCES ........................................................... 19
2.4. CHANNELS AND SUBSCRIPTIONS ......................................... 20
   2.4.1. Channel implementation types .................................... 21
   2.4.2. Next steps ............................................................ 21

CHAPTER 3. INSTALL ............................................................... 22
3.1. INSTALLING THE OPENSHIFT SERVERLESS OPERATOR ............... 22
   3.1.1. Before you begin ..................................................... 22
      3.1.1.1. Defining cluster size requirements ........................... 22
      3.1.1.2. Scaling your cluster using machine sets .................... 22
   3.1.2. Installing the OpenShift Serverless Operator .................... 22
      3.1.3. Additional resources ............................................. 23
      3.1.4. Next steps .......................................................... 23
3.2. INSTALLING KNATIVE SERVING ........................................... 24
   3.2.1. Installing Knative Serving by using the web console ............. 24
   3.2.2. Installing Knative Serving by using YAML ....................... 26
   3.2.3. Advanced configuration options .................................. 27
      3.2.3.1. Controller custom certs ...................................... 27
      3.2.3.2. High availability .............................................. 28
   3.2.4. Next steps ............................................................ 28
3.3. INSTALLING KNATIVE EVENTING ......................................... 28
   3.3.1. Installing Knative Eventing by using the web console .......... 29
   3.3.2. Installing Knative Eventing by using YAML ..................... 31
   3.3.3. Next steps ............................................................ 32
3.4. REMOVING OPENSHIFT SERVERLESS ..................................... 32
   3.4.1. Uninstalling Knative Serving ..................................... 32
   3.4.2. Uninstalling Knative Eventing .................................... 32
3.4.3. Removing the OpenShift Serverless Operator
  3.4.3.1. Deleting Operators from a cluster using the web console
  3.4.3.2. Deleting Operators from a cluster using the CLI
  3.4.3.3. Refreshing failing subscriptions
  3.4.4. Deleting OpenShift Serverless custom resource definitions

CHAPTER 4. DEVELOP ......................................................... 37
4.1. SERVERLESS APPLICATIONS 37
  4.1.1. Creating serverless applications 37
    4.1.1.1. Creating serverless applications by using the Knative CLI 37
    4.1.1.1.1. Knative client multi-container support 38
    4.1.1.2. Creating serverless applications using YAML 39
  4.1.2. Updating serverless applications by using the Knative CLI 40
  4.1.3. Applying service declarations 40
  4.1.4. Describing serverless applications by using the Knative CLI 41
  4.1.5. Verifying your serverless application deployment 42
  4.1.6. Interacting with a serverless application using HTTP2 and gRPC 43
  4.1.7. Enabling communication with Knative applications on a cluster with restrictive network policies 45
  4.1.8. HTTPS redirection per service 46
  4.1.9. Using kn CLI in offline mode 46
    4.1.9.1. About offline mode 47
    4.1.9.2. Creating a service using offline mode 47
  4.2. AUTOSCALING 50
    4.2.1. Scale bounds 50
      4.2.1.1. Minimum scale bounds 50
      4.2.1.1.1. Setting the minScale annotation by using the Knative CLI 51
      4.2.1.2. Maximum scale bounds 51
      4.2.1.2.1. Setting the maxScale annotation by using the Knative CLI 51
    4.2.2. Concurrency 52
      4.2.2.1. Concurrency limits and targets 52
      4.2.2.2. Configuring a soft concurrency target 52
      4.2.2.3. Configuring a hard concurrency limit 53
      4.2.2.4. Concurrency target utilization 54
      4.2.3. Additional resources 54
  4.3. TRAFFIC MANAGEMENT 54
    4.3.1. Traffic routing examples 55
      4.3.1.1. Traffic routing between multiple revisions 55
      4.3.1.2. Traffic routing to the latest revision 55
      4.3.1.3. Traffic routing to the current revision 56
    4.3.2. Managing traffic between revisions by using the OpenShift Container Platform web console 56
    4.3.3. Traffic management using the Knative CLI 58
      4.3.3.1. Multiple flags and order precedence 58
      4.3.3.2. Example traffic management commands 59
      4.3.3.3. Knative CLI traffic management flags 59
      4.3.3.4. Custom URLs for revisions 60
        4.3.3.4.1. Example: Assign a tag to a revision 60
        4.3.3.4.2. Example: Remove a tag from a revision 60
    4.3.4. Routing and managing traffic by using a blue-green deployment strategy 60
  4.4. ROUTING 63
    4.4.1. Configuring OpenShift Container Platform routes for Knative services 63
    4.4.2. Setting cluster availability to cluster local 65
    4.4.3. Additional resources 66
5.4.6. Additional resources
5.5. CREATING KNATIVE SERVING COMPONENTS IN THE ADMINISTRATOR PERSPECTIVE
5.5.1. Creating serverless applications using the Administrator perspective
5.6. CONFIGURING THE KNATIVE SERVING CUSTOM RESOURCE
5.6.1. Overriding system deployment configurations
5.6.2. Configuring the EmptyDir extension
5.6.3. HTTPS redirection global settings
5.6.4. Additional resources
5.7. AUTOSCALING
5.7.1. Enabling scale-to-zero
5.7.2. Configuring the scale-to-zero grace period
5.8. INTEGRATING SERVICE MESH WITH OPENSIFT SERVERLESS
5.8.1. Integrating Service Mesh with OpenShift Serverless natively
5.8.1.1. Creating a certificate to encrypt incoming external traffic
5.8.1.2. Integrating Service Mesh with OpenShift Serverless
5.8.1.3. Enabling Knative Serving metrics when using Service Mesh with mTLS
5.8.2. Integrating Service Mesh with OpenShift Serverless when Kourier is enabled
5.9. MONITORING SERVERLESS COMPONENTS
5.9.1. Monitoring the overall health status of Knative components
5.9.2. Monitoring Knative Serving revision CPU and memory usage
5.9.3. Monitoring Knative Eventing source CPU and memory usage
5.9.4. Monitoring event sources
5.9.5. Monitoring Knative Eventing brokers and triggers
5.9.6. Monitoring Knative Eventing channels
5.10. METRICS
5.10.1. Prerequisites
5.10.2. Controller metrics
5.10.3. Webhook metrics
5.10.4. Knative Eventing metrics
5.10.4.1. Broker ingress metrics
5.10.4.2. Broker filter metrics
5.10.4.3. InMemoryChannel dispatcher metrics
5.10.4.4. Event source metrics
5.10.5. Knative Serving metrics
5.10.5.1. Activator metrics
5.10.5.2. Autoscaler metrics
5.10.5.3. Go runtime metrics
5.11. HIGH AVAILABILITY ON OPENSIFT SERVERLESS
5.11.1. Configuring high availability replicas on OpenShift Serverless
5.11.1.1. Configuring high availability replicas for Knative Serving
5.11.1.2. Configuring high availability replicas for Knative Eventing
5.11.1.3. Configuring high availability replicas for Knative Kafka

CHAPTER 6. MONITOR

6.1. USING OPENSIFT LOGGING WITH OPENSIFT SERVERLESS
6.1.1. About deploying OpenShift Logging
6.1.2. About deploying and configuring OpenShift Logging
6.1.2.1. Configuring and Tuning OpenShift Logging
6.1.2.2. Sample modified ClusterLogging custom resource
6.1.3. Using OpenShift Logging to find logs for Knative Serving components
6.1.4. Using OpenShift Logging to find logs for services deployed with Knative Serving
6.2. TRACING REQUESTS USING JAEGER
6.2.1. Configuring Jaeger for use with OpenShift Serverless
6.3. METRICS
6.3.1. Prerequisites
6.3.2. Queue proxy metrics
6.4. MONITORING KNATIVE SERVICES
6.4.1. Knative service metrics exposed by default
6.4.2. Knative service with custom application metrics
6.4.3. Configuration for scraping custom metrics
6.4.4. Examining metrics of a service
6.4.5. Examining metrics of a service in the dashboard
6.4.6. Additional resources
6.5. AUTOSCALING DASHBOARD
6.5.1. Navigating to the autoscaling dashboard
6.5.2. Pod information
6.5.3. Observed concurrency
6.5.4. Scrape time
6.5.5. Panic mode
6.5.6. Activator metrics
6.5.7. Requests per second

CHAPTER 7. OPENSHIFT SERVERLESS SUPPORT
7.1. GETTING SUPPORT
7.2. GATHERING DIAGNOSTIC INFORMATION FOR SUPPORT
7.2.1. About the must-gather tool
7.2.2. About collecting OpenShift Serverless data

CHAPTER 8. SECURITY
8.1. CONFIGURING JSON WEB TOKEN AUTHENTICATION FOR KNATIVE SERVICES
8.1.1. Enabling sidecar injection for a Knative service
8.1.2. Using JSON Web Token authentication with Service Mesh 2.x and OpenShift Serverless
8.1.3. Using JSON Web Token authentication with Service Mesh 1.x and OpenShift Serverless
8.2. CONFIGURING A CUSTOM DOMAIN FOR A KNATIVE SERVICE
8.2.1. Creating a custom domain mapping
8.2.2. Creating a custom domain mapping by using the Knative CLI
8.2.3. Creating a custom domain mapping by using the web console
8.2.3.1. Mapping a custom domain to a service by using the Administrator perspective
8.2.3.2. Mapping a custom domain to a service by using the Developer perspective
8.3. USING A CUSTOM TLS CERTIFICATE FOR DOMAIN MAPPING
8.3.1. Adding a custom TLS certificate to a DomainMapping CR
8.4. SECURITY CONFIGURATION FOR KNATIVE KAFKA
8.4.1. Configuring TLS authentication
8.4.2. Configuring SASL authentication
8.4.3. Configuring SASL authentication using public CA certificates
8.4.4. Additional resources

CHAPTER 9. FUNCTIONS
9.1. SETTING UP OPENSHIFT SERVERLESS FUNCTIONS
9.1.1. Prerequisites
9.1.2. Using podman
9.1.3. Next steps
9.2. GETTING STARTED WITH FUNCTIONS
9.2.1. Prerequisites
9.2.2. Creating functions
9.2.3. Building functions
9.2.4. Deploying functions
9.2.5. Emitting a test event to a deployed function
9.2.6. Building and deploying functions with OpenShift Container Registry
9.2.7. Additional resources
9.2.8. Next steps

9.3. DEVELOPING NODE.JS FUNCTIONS
9.3.1. Prerequisites
9.3.2. Node.js function template structure
9.3.3. About invoking Node.js functions
  9.3.3.1. Node.js context objects
    9.3.3.1.1. Context object methods
    9.3.3.1.2. CloudEvent data
  9.3.3.2. Node.js function return values
    9.3.3.1. Returning headers
    9.3.3.2. Returning status codes
  9.3.5. Testing Node.js functions
9.3.6. Next steps

9.4. DEVELOPING TYPESCRIPT FUNCTIONS
9.4.1. Prerequisites
9.4.2. TypeScript function template structure
9.4.3. About invoking TypeScript functions
  9.4.3.1. TypeScript context objects
    9.4.3.1.1. Context object methods
    9.4.3.1.2. Context types
    9.4.3.1.3. CloudEvent data
  9.4.4. TypeScript function return values
    9.4.4.1. Returning headers
    9.4.4.2. Returning status codes
  9.4.5. Testing TypeScript functions
9.4.6. Next steps

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

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

9.7. DEVELOPING QUARKUS FUNCTIONS
9.7.1. Prerequisites
9.7.2. Quarkus function template structure
9.7.3. About invoking Quarkus functions
  9.7.3.1. Invocation examples
9.7.4. CloudEvent attributes
11.1. INSTALLING THE KNATIVE CLI
11.1.1. Installing the Knative CLI using the OpenShift Container Platform web console
11.1.2. Installing the Knative CLI for Linux using an RPM
11.1.3. Installing the Knative CLI for Linux
11.1.4. Installing the Knative CLI for Linux on IBM Power using an RPM
11.1.5. Installing the Knative CLI for Linux on IBM Power
11.1.6. Installing the Knative CLI for Linux on IBM Z and LinuxONE using an RPM
11.1.7. Installing the Knative CLI for Linux on IBM Z and LinuxONE
11.1.8. Installing the Knative CLI for macOS
11.1.9. Installing the Knative CLI for Windows
11.2. KNATIVE CLI ADVANCED CONFIGURATION
11.2.1. Customizing the Knative CLI
11.2.2. Knative CLI plug-ins

CHAPTER 12. REFERENCE
12.1. KNATIVE CLI FLAGS
12.1.1. Knative CLI --sink flag
12.2. KNATIVE SERVING CLI COMMANDS
12.2.1. kn service commands
12.2.1.1. Creating serverless applications by using the Knative CLI
12.2.1.2. Updating serverless applications by using the Knative CLI
12.2.1.3. Applying service declarations
12.2.1.4. Describing serverless applications by using the Knative CLI
12.2.2. kn container commands
12.2.2.1. Knative client multi-container support
12.2.3. kn domain commands
12.2.3.1. Creating a custom domain mapping by using the Knative CLI
12.2.3.2. Managing custom domain mappings by using the Knative CLI
12.3. KNATIVE EVENTING CLI COMMANDS
12.3.1. kn source commands
12.3.1.1. Listing available event source types by using the Knative CLI
12.3.1.2. Creating and managing container sources by using the Knative CLI
12.3.1.3. Creating an API server source by using the Knative CLI
12.3.1.4. Creating a ping source by using the Knative CLI
12.3.1.5. Creating a Kafka event source by using the Knative CLI
12.4. FUNCTIONS COMMANDS
12.4.1. Creating functions
12.4.2. Building functions
12.4.3. Deploying functions
12.4.4. Listing existing functions
12.4.5. Describing a function
12.4.6. Emitting a test event to a deployed function
12.4.6.1. kn func emit optional parameters
12.4.7. Deleting a function
12.5. KN EVENT
12.5.1. Building events
12.5.2. Sending events
CHAPTER 1. RELEASE NOTES

This topic includes information about new features, changes, and known issues that pertain to the latest OpenShift Serverless release on OpenShift Container Platform.

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

NOTE

OpenShift Serverless is based on the open source Knative project.

For details about the latest Knative component releases, see the Knative blog.

1.1. ABOUT API VERSIONS

The OpenShift Serverless Operator automatically upgrades older resources that use deprecated versions of APIs to use the latest version.

For example, if you have created resources on your cluster that use older versions of the `ApiServerSource` API, such as `v1beta1`, the OpenShift Serverless Operator automatically updates these resources to use the `v1` version of the API when this is available and the `v1beta1` version is deprecated.

After they have been deprecated, older versions of APIs might be removed in any upcoming release. Using deprecated versions of APIs does not cause resources to fail. However, if you try to use a version of an API that has been removed, it will cause resources to fail. Ensure that your manifests are updated to use the latest version to avoid issues.

1.2. DEPRECATED AND REMOVED FEATURES

Some features available in previous releases have been deprecated or removed.

 Deprecated functionality is still included in OpenShift Serverless and continues to be supported; however, it will be removed in a future release of this product and is not recommended for new deployments. For the most recent list of major functionality deprecated and removed within OpenShift Serverless, refer to the table below.

In the table, features are marked with the following statuses:

- `-`: Not yet available
- `TP`: Technology Preview
- `GA`: General Availability
- `DEP`: Deprecated
- `REM`: Removed

Table 1.1. Deprecated and removed features tracker

<table>
<thead>
<tr>
<th>Feature</th>
<th>1.18</th>
<th>1.19</th>
<th>1.20</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>kn func emit</code> (<code>kn func invoke</code> in 1.21+)</td>
<td>TP</td>
<td>TP</td>
<td>TP</td>
</tr>
</tbody>
</table>
1.3. RELASE NOTES FOR RED HAT OPENSÅFT SERVERLESS 1.20.0

OpenShift Serverless 1.20.0 is now available. New features, changes, and known issues that pertain to OpenShift Serverless on OpenShift Container Platform are included in this topic.

1.3.1. New features

- OpenShift Serverless now uses Knative Serving 0.26.
- OpenShift Serverless now uses Knative Eventing 0.26.
- OpenShift Serverless now uses Kourier 0.26.
- OpenShift Serverless now uses Knative `kn` CLI 0.26.
- OpenShift Serverless now uses Knative Kafka 0.26.
- The `kn func` CLI plug-in now uses `func` 0.20.
- The Kafka broker is now available as a Technology Preview.
- The `kn event` plug-in is now available as a Technology Preview.

1.3.2. Known issues

- OpenShift Serverless deploys Knative services with a default address that uses HTTPS. When sending an event to a resource inside the cluster, the sender does not have the cluster certificate authority (CA) configured. This causes event delivery to fail, unless the cluster uses globally accepted certificates. For example, an event delivery to a publicly accessible address works:

  $ kn event send --to-url https://ce-api.foo.example.com/

  On the other hand, this delivery fails if the service uses a public address with an HTTPS certificate issued by a custom CA:
Sending an event to other addressable objects, such as brokers or channels, is not affected by this issue and works as expected.

- The Kafka broker currently does not work on a cluster with Federal Information Processing Standards (FIPS) mode enabled.

- If you create a Springboot function project directory with the `kn func create` command, subsequent running of the `kn func build` command fails with this error message:

  ```
  [analyzer] no stack metadata found at path "
  [analyzer] ERROR: failed to : set API for buildpack 'paketo-buildpacks/ca-certificates@3.0.2':
  buildpack API version '0.7' is incompatible with the lifecycle
  ```

  As a workaround, you can change the `builder` property to `gcr.io/paketo-buildpacks/builder:base` in the function configuration file `func.yaml`.

- Deploying a function using the `gcr.io` registry fails with this error message:

  ```
  Error: failed to get credentials: failed to verify credentials: status code: 404
  ```

  As a workaround, use a different registry than `gcr.io`, such as `quay.io` or `docker.io`.

- TypeScript functions created with the `http` template fail to deploy on the cluster. As a workaround, in the `func.yaml` file, replace the following section:

  ```
  buildEnvs: []
  ```

  with this:

  ```
  buildEnvs:
  - name: BP_NODE_RUN_SCRIPTS
    value: build
  ```

- In `func` version 0.20, some runtimes might be unable to build a function by using `podman`. You might see an error message similar to the following:

  ```
  ERROR: failed to image: error during connect: Get "http://%2FVar%2Frun%2Fdocker.sock/v1.40/info": EOF
  ```

  The following workaround exists for this issue:

  a. Update the podman service by adding `--time=0` to the service `ExecStart` definition:

  ```
  Example service configuration
  ExecStart=/usr/bin/podman $LOGGING system service --time=0
  ```

  b. Restart the podman service by running the following commands:

  ```
  $ systemctl --user daemon-reload
  ```
Alternatively, you can expose the podman API by using TCP:

$ systemctl restart --user podman.socket

- Alternatively, you can expose the podman API by using TCP:

$ podman system service --time=0 tcp:127.0.0.1:5534 &
export DOCKER_HOST=tcp://127.0.0.1:5534

1.4. RELEASE NOTES FOR RED HAT OPENSIFT SERVERLESS 1.19.0

OpenShift Serverless 1.19.0 is now available. New features, changes, and known issues that pertain to OpenShift Serverless on OpenShift Container Platform are included in this topic.

1.4.1. New features

- OpenShift Serverless now uses Knative Serving 0.25.
- OpenShift Serverless now uses Knative Eventing 0.25.
- OpenShift Serverless now uses Kourier 0.25.
- OpenShift Serverless now uses Knative kn CLI 0.25.
- OpenShift Serverless now uses Knative Kafka 0.25.
- The kn func CLI plug-in now uses func 0.19.
- The KafkaBinding API is deprecated in OpenShift Serverless 1.19.0 and will be removed in a future release.
- HTTPS redirection is now supported and can be configured either globally for a cluster or per each Knative service.

1.4.2. Fixed issues

- In previous releases, the Kafka channel dispatcher waited only for the local commit to succeed before responding, which might have caused lost events in the case of an Apache Kafka node failure. The Kafka channel dispatcher now waits for all in-sync replicas to commit before responding.

1.4.3. Known issues

- In func version 0.19, some runtimes might be unable to build a function by using podman. You might see an error message similar to the following:

```
ERROR: failed to image: error during connect: Get "http://%2Fvar%2Frun%2Fdocker.sock/v1.40/info": EOF
```

- The following workaround exists for this issue:

  a. Update the podman service by adding --time=0 to the service ExecStart definition:

  Example service configuration

  ```
  ExecStart=/usr/bin/podman $LOGGING system service --time=0
  ```
b. Restart the podman service by running the following commands:

```
$ systemctl --user daemon-reload
$ systemctl restart --user podman.socket
```

- Alternatively, you can expose the podman API by using TCP:

```
$ podman system service --time=0 tcp:127.0.0.1:5534 &
export DOCKER_HOST=tcp://127.0.0.1:5534
```

### 1.5. RELEASE NOTES FOR RED HAT OPENSSHIFT SERVERLESS 1.18.0

OpenShift Serverless 1.18.0 is now available. New features, changes, and known issues that pertain to OpenShift Serverless on OpenShift Container Platform are included in this topic.

#### 1.5.1. New features

- OpenShift Serverless now uses Knative Serving 0.24.0.
- OpenShift Serverless now uses Knative Eventing 0.24.0.
- OpenShift Serverless now uses Kourier 0.24.0.
- OpenShift Serverless now uses Knative `kn` CLI 0.24.0.
- OpenShift Serverless now uses Knative Kafka 0.24.7.
- The `kn func` CLI plug-in now uses `func` 0.18.0.
- In the upcoming OpenShift Serverless 1.19.0 release, the URL scheme of external routes will default to HTTPS for enhanced security. If you do not want this change to apply for your workloads, you can override the default setting before upgrading to 1.19.0, by adding the following YAML to your `KnativeServing` custom resource (CR):

```yaml
...  
spec:  
  config:  
    network:  
      defaultExternalScheme: "http"
...  
```

If you want the change to apply in 1.18.0 already, add the following YAML:

```yaml
...  
spec:  
  config:  
    network:  
      defaultExternalScheme: "https"
...  
```
In the upcoming OpenShift Serverless 1.19.0 release, the default service type by which the Kourier Gateway is exposed will be **ClusterIP** and not **LoadBalancer**. If you do not want this change to apply to your workloads, you can override the default setting before upgrading to 1.19.0, by adding the following YAML to your **KnativeServing** custom resource (CR):

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

- You can now use **emptyDir** volumes with OpenShift Serverless. See the OpenShift Serverless documentation about Knative Serving for details.

- Rust templates are now available when you create a function using **kn func**.

### 1.5.2. Fixed issues

- The prior 1.4 version of Camel-K was not compatible with OpenShift Serverless 1.17.0. The issue in Camel-K has been fixed, and Camel-K version 1.4.1 can be used with OpenShift Serverless 1.17.0.

- Previously, if you created a new subscription for a Kafka channel, or a new Kafka source, a delay was possible in the Kafka data plane becoming ready to dispatch messages after the newly created subscription or sink reported a ready status. As a result, messages that were sent during the time when the data plane was not reporting a ready status, might not have been delivered to the subscriber or sink. In OpenShift Serverless 1.18.0, the issue is fixed and the initial messages are no longer lost. For more information about the issue, see [Knowledgebase Article #6343981](#).

### 1.5.3. Known issues

- Older versions of the Knative **kn** CLI might use older versions of the Knative Serving and Knative Eventing APIs. For example, version 0.23.2 of the **kn** CLI uses the **v1alpha1** API version. On the other hand, newer releases of OpenShift Serverless might no longer support older API versions. For example, OpenShift Serverless 1.18.0 no longer supports version **v1alpha1** of the **kafkasources.sources.knative.dev** API.

    Consequently, using an older version of the Knative **kn** CLI with a newer OpenShift Serverless might produce an error because the **kn** cannot find the outdated API. For example, version 0.23.2 of the **kn** CLI does not work with OpenShift Serverless 1.18.0.

    To avoid issues, use the latest **kn** CLI version available for your OpenShift Serverless release. For OpenShift Serverless 1.18.0, use Knative **kn** CLI 0.24.0.
2.1. ABOUT OPENSOURCEServerless

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

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

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

2.1.1. Knative Serving

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

Knative Serving supports deploying and managing cloud-native applications. It provides a set of objects as Kubernetes custom resource definitions (CRDs) that define and control the behavior of serverless workloads on an OpenShift Container Platform cluster.

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

- Rapidly deploying serverless containers.
- Automatically scaling pods.

2.1.1.1. Knative Serving resources

**Service**

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

**Revision**

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

**Route**

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

**Configuration**

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

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

An event-driven architecture is based on the concept of decoupled relationships between event producers and event consumers. Event producers create events, and event sinks, or consumers, receive events.

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

Knative Eventing supports the following use cases:

**Publish an event without creating a consumer**

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

**Consume an event without creating a publisher**

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

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

**Addressable resources**

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

**Callable resources**

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

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

- channels and subscriptions, or
- brokers and triggers.

### 2.1.3. Supported configurations

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

### 2.2. ABOUT OPENSHIFT SERVERLESS FUNCTIONS
IMPORTANT

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

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

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

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

2.2.1. Supported runtimes

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

- Node.js
- Python
- Golang
- Quarkus
- TypeScript

2.2.2. Next steps

- See Getting started with functions.

2.3. EVENT SOURCES

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

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

Currently, OpenShift Serverless supports the following event source types:

**API server source**

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

**Ping source**

Produces events with a fixed payload on a specified cron schedule.
**Kafka event source**

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

You can also create a custom event source.

### 2.4. CHANNELS AND SUBSCRIPTIONS

Channels are custom resources that define a single event-forwarding and persistence layer. After events have been sent to a channel from an event source or producer, these events can be sent to multiple Knative services, or other sinks, by using a subscription.

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

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

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

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

**NOTE**

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

When this mechanism is used with the preceding example, two objects are created: a generic backing channel and an InMemoryChannel channel. If you are using a different default channel implementation, the InMemoryChannel is replaced with one that is specific to your implementation. For example, with Knative Kafka, the KafkaChannel channel is created.
The backing channel acts as a proxy that copies its subscriptions to the user-created channel object, and sets the user-created channel object status to reflect the status of the backing channel.

### 2.4.1. Channel implementation types

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

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

- No event persistence is available. If a pod goes down, events on that pod are lost.
- **InMemoryChannel** channels do not implement event ordering, so two events that are received in the channel at the same time can be delivered to a subscriber in any order.
- If a subscriber rejects an event, there are no re-delivery attempts by default. You can configure re-delivery attempts by modifying the **delivery** spec in the **Subscription** object.

For more information about Kafka channels, see the **Knative Kafka** documentation.

### 2.4.2. Next steps

- Create a channel and a **subscription** that allows event sinks to subscribe to channels and receive events.

- If you are a cluster administrator, you can configure default settings for channels. See **Configuring channel defaults**.
CHAPTER 3. INSTALL

3.1. INSTALLING THE OPENSOURCE SERVERLESS OPERATOR

You can install the OpenShift Serverless Operator to an OpenShift Container Platform cluster by following the procedures in this document.

3.1.1. Before you begin

Read the following information about supported configurations and prerequisites before you install OpenShift Serverless.

- OpenShift Serverless is supported for installation in a restricted network environment.
- OpenShift Serverless currently cannot be used in a multi-tenant configuration on a single cluster.

3.1.1.1. Defining cluster size requirements

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

**NOTE**

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

The total size requirements to run OpenShift Serverless are dependent on the applications deployed. By default, each pod requests approximately 400m of CPU, so the minimum requirements are based on this value. Lowering the actual CPU request of applications can increase the number of possible replicas.

If you have high availability (HA) enabled on your cluster, this requires between 0.5 - 1.5 cores and between 200MB - 2GB of memory for each replica of the Knative Serving control plane.

3.1.1.2. Scaling your cluster using machine sets

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

3.1.2. Installing the OpenShift Serverless Operator

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

**Prerequisites**

- You have access to an OpenShift Container Platform account with cluster administrator access.
- You have logged in to the OpenShift Container Platform web console.

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

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

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

4. On the Install Operator page:
   a. The Installation Mode is All namespaces on the cluster (default) This mode installs the Operator in the default openshift-serverless namespace to watch and be made available to all namespaces in the cluster.
   b. The Installed Namespace will be openshift-serverless.
   c. Select the stable channel as the Update Channel. The stable channel will enable installation of the latest stable release of the OpenShift Serverless Operator.
   d. Select Automatic or Manual approval strategy.

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

6. From the Catalog → Operator Management page, you can monitor the OpenShift Serverless Operator subscription’s installation and upgrade progress.
   a. If you selected a Manual approval strategy, the subscription’s upgrade status will remain Upgrading until you review and approve its install plan. After approving on the Install Plan page, the subscription upgrade status moves to Up to date.
   b. If you selected an Automatic approval strategy, the upgrade status should resolve to Up to date without intervention.

Verification

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

If it does not:

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

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

3.1.3. Additional resources

- For more information about using OpenShift Serverless on a restricted network environment, see Using Operator Lifecycle Manager on restricted networks.
- HA is enabled for some Knative components by default. You can configure HA by following the documentation on Configuring high availability replicas on OpenShift Serverless.

3.1.4. Next steps
After the OpenShift Serverless Operator is installed, you can install Knative Serving or install Knative Eventing.

3.2. INSTALLING KNATIVE SERVING

This guide provides information about installing Knative Serving using the default settings. However, you can configure more advanced settings in the KnativeServing custom resource (CR). For more information about configuration options for the KnativeServing CR, see Advanced configuration options.

3.2.1. Installing Knative Serving by using the web console

You can use the following procedure to install Knative Serving by using the OpenShift Container Platform web console.

Prerequisites

- You have access to an OpenShift Container Platform account with cluster administrator access.
- You have logged in to the OpenShift Container Platform web console.
- You have installed the OpenShift Serverless Operator.

Procedure

1. In the Administrator perspective of the OpenShift Container Platform web console, navigate to Operators → Installed Operators.
2. Check that the Project dropdown at the top of the page is set to Project: knative-serving.
3. Click Knative Serving in the list of Provided APIs for the OpenShift Serverless Operator to go to the Knative Serving tab.
4. Click Create Knative Serving.
5. In the Create Knative Serving page, you can install Knative Serving using the default settings by clicking Create. You can also modify settings for the Knative Serving installation by editing the KnativeServing object using either the form provided, or by editing the YAML.

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

NOTE

For more information about configuration options for the KnativeServing custom resource definition, see the documentation on Advanced installation configuration options.
6. After you have installed Knative Serving, the **KnativeServing** object is created, and you are automatically directed to the **Knative Serving** tab. You will see the **knative-serving** custom resource in the list of resources.

**Verification**

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

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

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

4. You should see a list of conditions with a status of **True**, as shown in the example image.
5. If the conditions have a status of **Unknown** or **False**, wait a few moments and then check again after you have confirmed that the resources have been created.

### 3.2.2. Installing Knative Serving by using YAML

You can use the following procedure to install Knative Serving by applying YAML files.

**Prerequisites**

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

**Procedure**

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

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

2. Apply the `serving.yaml` file:

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

**Verification**

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

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

**Example output**

```
DependenciesInstalled=True
DeploymentsAvailable=True
InstallSucceeded=True
Ready=True
```
2. Check that the Knative Serving resources have been created:

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

**Example output**

<table>
<thead>
<tr>
<th>NAME</th>
<th>READY</th>
<th>STATUS</th>
<th>RESTARTS</th>
<th>AGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>activator-67ddf8c9d7-p7rm5</td>
<td>2/2</td>
<td>Running</td>
<td>0</td>
<td>4m</td>
</tr>
<tr>
<td>activator-67ddf8c9d7-q84fz</td>
<td>2/2</td>
<td>Running</td>
<td>0</td>
<td>4m</td>
</tr>
<tr>
<td>autoscaler-5d87bc6dbf-6nqc6</td>
<td>2/2</td>
<td>Running</td>
<td>0</td>
<td>3m59s</td>
</tr>
<tr>
<td>autoscaler-5d87bc6dbf-h64rl</td>
<td>2/2</td>
<td>Running</td>
<td>0</td>
<td>3m59s</td>
</tr>
<tr>
<td>autoscaler-hpa-77f85f5cc4-1r7ts</td>
<td>2/2</td>
<td>Running</td>
<td>0</td>
<td>3m57s</td>
</tr>
<tr>
<td>autoscaler-hpa-77f85f5cc4-zx7hl</td>
<td>2/2</td>
<td>Running</td>
<td>0</td>
<td>3m56s</td>
</tr>
<tr>
<td>controller-5fcf7cb8db-nlccl</td>
<td>2/2</td>
<td>Running</td>
<td>0</td>
<td>3m50s</td>
</tr>
<tr>
<td>controller-5fcf7cb8db-rmv7r</td>
<td>2/2</td>
<td>Running</td>
<td>0</td>
<td>3m18s</td>
</tr>
<tr>
<td>domain-mapping-86d84bb6b4-r746m</td>
<td>2/2</td>
<td>Running</td>
<td>0</td>
<td>3m58s</td>
</tr>
<tr>
<td>domain-mapping-86d84bb6b4-v7nh8</td>
<td>2/2</td>
<td>Running</td>
<td>0</td>
<td>3m58s</td>
</tr>
<tr>
<td>domainmapping-webhook-769d679d45-bkcnj</td>
<td>2/2</td>
<td>Running</td>
<td>0</td>
<td>3m58s</td>
</tr>
<tr>
<td>domainmapping-webhook-769d679d45-ff68</td>
<td>2/2</td>
<td>Running</td>
<td>0</td>
<td>3m58s</td>
</tr>
<tr>
<td>storage-version-migration-serving-serving-0.26.0--1-6qlk8</td>
<td>0/1</td>
<td>Completed</td>
<td>0</td>
<td>3m56s</td>
</tr>
<tr>
<td>webhook-5fb774f8d8-6bqrt</td>
<td>2/2</td>
<td>Running</td>
<td>0</td>
<td>3m57s</td>
</tr>
<tr>
<td>webhook-5fb774f8d8-b8lt5</td>
<td>2/2</td>
<td>Running</td>
<td>0</td>
<td>3m57s</td>
</tr>
</tbody>
</table>

3. Check that the necessary networking components have been installed to the automatically created **knative-serving-ingress** namespace:

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

**Example output**

<table>
<thead>
<tr>
<th>NAME</th>
<th>READY</th>
<th>STATUS</th>
<th>RESTARTS</th>
<th>AGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>net-kourier-controller-7d4b6c5d95-62mkf</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>76s</td>
</tr>
<tr>
<td>net-kourier-controller-7d4b6c5d95-qmgm2</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>76s</td>
</tr>
<tr>
<td>3scale-kourier-gateway-6688b49568-987qz</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>75s</td>
</tr>
<tr>
<td>3scale-kourier-gateway-6688b49568-b5tnp</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>75s</td>
</tr>
</tbody>
</table>

### 3.2.3. Advanced configuration options

This section provides information about advanced configuration options for Knative Serving.

#### 3.2.3.1. Controller custom certs

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

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

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

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

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

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

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

**IMPORTANT**

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

3.2.3.2. High availability

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

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

Example YAML

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

3.2.4. Next steps

- If you want to use Knative event-driven architecture you can install Knative Eventing.

3.3. INSTALLING KNATIVE EVENTING

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

This guide provides information about installing Knative Eventing using the default settings.
3.3.1. Installing Knative Eventing by using the web console

You can use the following procedure to install Knative Eventing by using the OpenShift Container Platform web console.

Prerequisites

- You have access to an OpenShift Container Platform account with cluster administrator access.
- You have logged in to the OpenShift Container Platform web console.
- You have installed the OpenShift Serverless Operator.

Procedure

1. In the Administrator perspective of the OpenShift Container Platform web console, navigate to Operators → Installed Operators.
2. Check that the Project dropdown at the top of the page is set to Project: knative-eventing.
3. Click Knative Eventing in the list of Provided APIs for the OpenShift Serverless Operator to go to the Knative Eventing tab.
4. Click Create Knative Eventing.
5. In the Create Knative Eventing page, you can choose to configure the KnativeEventing object by using either the default form provided, or by editing the YAML.
   - Using the form is recommended for simpler configurations that do not require full control of KnativeEventing object creation.
     Optional. If you are configuring the KnativeEventing object using the form, make any changes that you want to implement for your Knative Eventing deployment.
6. Click Create.
   - Editing the YAML is recommended for more complex configurations that require full control of KnativeEventing object creation. You can access the YAML by clicking the edit YAML link in the top right of the Create Knative Eventing page.
     Optional. If you are configuring the KnativeEventing object by editing the YAML, make any changes to the YAML that you want to implement for your Knative Eventing deployment.
7. Click Create.
8. After you have installed Knative Eventing, the KnativeEventing object is created, and you are automatically directed to the Knative Eventing tab. You will see the knative-eventing custom resource in the list of resources.

Verification

1. Click on the knative-eventing custom resource in the Knative Eventing tab.
2. You are automatically directed to the Knative Eventing Overview page.
3. Scroll down to look at the list of Conditions.

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

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

5. If the conditions have a status of Unknown or False, wait a few moments and then check again after you have confirmed that the resources have been created.
3.3.2. Installing Knative Eventing by using YAML

You can use the following procedure to install Knative Eventing by applying YAML files.

Prerequisites

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

Procedure

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

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

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

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

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

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

Verification

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

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

   **Example output**

   ![Example output](example-output.png)

   **NOTE**

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

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

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

   ```bash
   $ oc get pods -n knative-eventing
   ```
3.3.3. Next steps

- If you want to use Knative services you can install Knative Serving.

3.4. REMOVING OPENSIFT SERVERLESS

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

**NOTE**

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

3.4.1. Uninstalling Knative Serving

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

**Prerequisites**

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

**Procedure**

1. Delete the knative-serving CR:

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

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

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

3.4.2. Uninstalling Knative Eventing

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

**Prerequisites**

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

1. Delete the `knative-eventing` CR:

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

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

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

3.4.3. Removing the OpenShift Serverless Operator

You can remove the OpenShift Serverless Operator from the host cluster by using the web console or the `oc` CLI.

**NOTE**

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

3.4.3.1. Deleting Operators from a cluster using the web console

Cluster administrators can delete installed Operators from a selected namespace by using the web console.

**Prerequisites**

- Access to an OpenShift Container Platform cluster web console using an account with `cluster-admin` permissions.

**Procedure**

1. Navigate to the **Operators → Installed Operators** page.

2. Scroll or enter a keyword into the **Filter by name** field to find the Operator that you want to remove. Then, click on it.

3. On the right side of the **Operator Details** page, select **Uninstall Operator** from the **Actions** drop-down menu. An **Uninstall Operator?** dialog box is displayed.

4. Select **Uninstall** to remove the Operator, Operator deployments, and pods. Following this action, the Operator stops running and no longer receives updates.

**NOTE**

This action does not remove resources managed by the Operator, including custom resource definitions (CRDs) and custom resources (CRs). Dashboards and navigation items enabled by the web console and off-cluster resources that continue to run might need manual clean up. To remove these after uninstalling the Operator, you might need to manually delete the Operator CRDs.
3.4.3.2. Deleting Operators from a cluster using the CLI

Cluster administrators can delete installed Operators from a selected namespace by using the CLI.

**Prerequisites**

- Access to an OpenShift Container Platform cluster using an account with `cluster-admin` permissions.
- `oc` command installed on workstation.

**Procedure**

1. Check the current version of the subscribed Operator (for example, `jaeger`) in the `currentCSV` field:
   
   ```bash
   $ oc get subscription jaeger -n openshift-operators -o yaml | grep currentCSV
   
   currentCSV: jaeger-operator.v1.8.2
   ```

   **Example output**

2. Delete the subscription (for example, `jaeger`):

   ```bash
   $ oc delete subscription jaeger -n openshift-operators
   
   subscription.operators.coreos.com "jaeger" deleted
   ```

   **Example output**

3. Delete the CSV for the Operator in the target namespace using the `currentCSV` value from the previous step:

   ```bash
   $ oc delete clusterserviceversion jaeger-operator.v1.8.2 -n openshift-operators
   
   clusterserviceversion.operators.coreos.com "jaeger-operator.v1.8.2" deleted
   ```

3.4.3.3. Refreshing failing subscriptions

In Operator Lifecycle Manager (OLM), if you subscribe to an Operator that references images that are not accessible on your network, you can find jobs in the `openshift-marketplace` namespace that are failing with the following errors:

**Example output**

```
ImagePullBackOff for
Back-off pulling image "example.com/openshift4/ose-elasticsearch-operator-bundle@sha256:6d2587129c846ec28d384540322b40b05833e7e00b25cca584e004af9a1d292e"

Example output
```
As a result, the subscription is stuck in this failing state and the Operator is unable to install or upgrade.

You can refresh a failing subscription by deleting the subscription, cluster service version (CSV), and other related objects. After recreating the subscription, OLM then reinstall the correct version of the Operator.

Prerequisites

- You have a failing subscription that is unable to pull an inaccessible bundle image.
- You have confirmed that the correct bundle image is accessible.

Procedure

1. Get the names of the Subscription and ClusterServiceVersion objects from the namespace where the Operator is installed:

   ```
   $ oc get sub,csv -n <namespace>
   ```

   **Example output**

<table>
<thead>
<tr>
<th>NAME</th>
<th>PACKAGE</th>
<th>SOURCE</th>
<th>CHANNEL</th>
</tr>
</thead>
<tbody>
<tr>
<td>subscription/operators.coreos.com/elasticsearch-operator</td>
<td>elasticsearch-operator</td>
<td>redhat-operators</td>
<td>5.0</td>
</tr>
</tbody>
</table>

2. Delete the subscription:

   ```
   $ oc delete subscription <subscription_name> -n <namespace>
   ```

3. Delete the cluster service version:

   ```
   $ oc delete csv <csv_name> -n <namespace>
   ```

4. Get the names of any failing jobs and related config maps in the openshift-marketplace namespace:

   ```
   $ oc get job,configmap -n openshift-marketplace
   ```

   **Example output**

<table>
<thead>
<tr>
<th>NAME</th>
<th>COMPLETIONS</th>
<th>DURATION</th>
<th>AGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>job.batch/1de9443b6324e629ddf31fed0a853a121275806170e34c926d69e53a7fcbc6</td>
<td>1/1</td>
<td>9m30s</td>
<td>1/1</td>
</tr>
<tr>
<td>26s</td>
<td>9m30s</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
5. Delete the job:

   $ oc delete job <job_name> -n openshift-marketplace

   This ensures pods that try to pull the inaccessible image are not recreated.

6. Delete the config map:

   $ oc delete configmap <configmap_name> -n openshift-marketplace

7. Reinstall the Operator using OperatorHub in the web console.

**Verification**

- Check that the Operator has been reinstalled successfully:

   $ oc get sub, csv, installplan -n <namespace>

### 3.4.4. Deleting OpenShift Serverless custom resource definitions

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

**IMPORTANT**

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

**Prerequisites**

- You have access to an OpenShift Container Platform account with cluster administrator access.
- You have uninstalled Knative Serving and removed the OpenShift Serverless Operator.

**Procedure**

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

  $ oc get crd -oname | grep 'knative.dev' | xargs oc delete
CHAPTER 4. DEVELOP

4.1. SERVERLESS APPLICATIONS

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

Example Knative service YAML

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

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.

4.1.1. Creating serverless applications

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

- Create a Knative service from the OpenShift Container Platform web console. See the documentation about Creating applications using the Developer perspective.
- Create a Knative service using the kn CLI.
- Create and apply a YAML file.

4.1.1.1. Creating serverless applications by using the Knative CLI

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

Prerequisites

- OpenShift Serverless Operator and Knative Serving are installed on your cluster.
- You have installed the kn CLI.
• 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> --env <key=value>

Example command

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

Example output

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

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

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

4.1.1.1. Knative client multi-container support

You can use the kn container add command to print YAML container spec to standard output. This command is useful for multi-container use cases because it can be used along with other standard kn flags to create definitions. The kn container add command accepts all container-related flags that are supported for use with the kn service create command. The kn container add command can also be chained by using UNIX pipes (|) to create multiple container definitions at once.

Example commands

• Add a container from an image and print it to standard output:

  $ kn container add <container_name> --image <image_uri>

Example command

  $ kn container add sidecar --image docker.io/example/sidecar

Example output

  containers:
      - image: docker.io/example/sidecar
        name: sidecar
        resources: {}
Chain two `kn container add` commands together, and then pass them to a `kn service create` command to create a Knative service with two containers:

```bash
$ kn container add <first_container_name> --image <image_uri> | \
kn container add <second_container_name> --image <image_uri> | \
kn service create <service_name> --image <image_uri> --extra-containers -
```

`--extra-containers` specifies a special case where `kn` reads the pipe input instead of a YAML file.

**Example command**

```bash
$ kn container add sidecar --image docker.io/example/sidecar:first | \
kn container add second --image docker.io/example/sidecar:second | \
kn service create my-service --image docker.io/example/my-app:latest --extra-containers -
```

The `--extra-containers` flag can also accept a path to a YAML file:

```bash
$ kn service create <service_name> --image <image_uri> --extra-containers <filename>
```

**Example command**

```bash
$ kn service create my-service --image docker.io/example/my-app:latest --extra-containers my-extra-containers.yaml
```

### 4.1.1.2. Creating serverless applications using YAML

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

**Procedure**

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

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

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

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

### 4.1.2. Updating serverless applications by using the Knative CLI

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

**Example commands**

- Update a service by adding a new environment variable:
  
  $ kn service update <service_name> --env <key>=<value>

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

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

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

- Update a tag from `testing` to `staging` for the latest `READY` revision of a service:
  
  $ kn service update <service_name> --untag testing --tag @latest=staging

- Add the `test` tag to a revision that receives 10% of traffic, and send the rest of the traffic to the latest `READY` revision of a service:
  
  $ kn service update <service_name> --tag <revision_name>=test --traffic test=10,@latest=90

### 4.1.3. Applying service declarations

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

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

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

- Create a service:
  
  $ kn service apply <service_name> --image <image>

- Add an environment variable to a service:
  
  $ kn service apply <service_name> --image <image> --env <key>=<value>

- Read the service declaration from a JSON or YAML file:
  
  $ kn service apply <service_name> -f <filename>

4.1.4. Describing serverless applications by using the Knative CLI

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

Example commands

- Describe a service:
  
  $ kn service describe --verbose <service_name>

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

Example output without `--verbose` flag

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

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

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

Example output with `--verbose` flag

```
Name:         hello
Namespace:    default
Annotations:  serving.knative.dev/creator=system:admin
              serving.knative.dev/lastModifier=system:admin
Age:          3m
URL:          http://hello-default.apps.ocp.example.com
Cluster:      http://hello.default.svc.cluster.local
```
Revisions:
100% @latest (hello-00001) [1] (3m)
  Image: docker.io/openshift/hello-openshift (pinned to aaea76)
  Env: RESPONSE=Hello Serverless!

Conditions:
OK TYPE AGEl REASON
++ Ready 3m
++ ConfigurationsReady 3m
++ RoutesReady 3m

- Describe a service in YAML format:
  
  ```
  $ kn service describe <service_name> -o yaml
  ```

- Describe a service in JSON format:
  
  ```
  $ kn service describe <service_name> -o json
  ```

- Print the service URL only:
  
  ```
  $ kn service describe <service_name> -o url
  ```

### 4.1.5. Verifying your serverless application deployment

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

**NOTE**

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

**Procedure**

1. Find the application URL:
   
   ```
   $ oc get ksvc <service_name>
   ```

   **Example output**

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

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

   **Example HTTP request**

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

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

Example output

Hello Serverless!

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

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

Example output

Hello Serverless!

\textbf{IMPORTANT}

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

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

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

Example output

Hello Serverless!

\textbf{4.1.6. Interacting with a serverless application using HTTP2 and gRPC}

OpenShift Serverless supports only insecure or edge-terminated routes.

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

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

This method needs to expose Kourier Gateway using the LoadBalancer service type. You can configure this by adding the following YAML to your KnativeServing custom resource definition (CRD):

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

Procedure

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

2. Find the ingress gateway’s public address:

   ```sh
   $ oc -n knative-serving-ingress get svc kourier
   
   NAME                   TYPE           CLUSTER-IP      EXTERNAL-IP          PORT(S)                        AGE
   kourier   LoadBalancer   172.30.51.103   a83e86291bcdd11e993af02b7a65e514-33544245.us-east-1.elb.amazonaws.com   80:31380/TCP,443:31390/TCP   67m
   
   (Output truncated for clarity)
   
   The public address is surfaced in the EXTERNAL-IP field, and in this case is 
   a83e86291bcdd11e993af02b7a65e514-33544245.us-east-1.elb.amazonaws.com.
   
3. Manually set the host header of your HTTP request to the application’s host, but direct the request itself against the public address of the ingress gateway.

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

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

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

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

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

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

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

Example NetworkPolicy object that denies all requests to the namespace

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

To allow access to your applications from Knative system pods, you must add a label to each of the Knative system namespaces, and then create a `NetworkPolicy` object in your application namespace that allows access to the namespace for other namespaces that have this label.

IMPORTANT

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

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

Procedure

1. Add the `knative.openshift.io/system-namespace=true` label to each Knative system namespace that requires access to your application:
   a. Label the `knative-serving` namespace:

   ```bash
   $ oc label namespace knative-serving knative.openshift.io/system-namespace=true
   ```
   b. Label the `knative-serving-ingress` namespace:

   ```bash
   $ oc label namespace knative-serving-ingress knative.openshift.io/system-namespace=true
   ```
c. Label the **knative-eventing** namespace:

   ```sh
   $ oc label namespace knative-eventing knative.openshift.io/system-namespace=true
   ```

d. Label the **knative-kafka** namespace:

   ```sh
   $ oc label namespace knative-kafka knative.openshift.io/system-namespace=true
   ```

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

   **Example NetworkPolicy object**

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

   **1.** Provide a name for your network policy.

   **2.** The namespace where your application exists.

**Additional resources**

- See [Configuring JSON Web Token authentication for Knative services](#).

**4.1.8. HTTPS redirection per service**

You can enable or disable HTTPS redirection for a service by configuring the `networking.knative.dev/httpOption` annotation, as shown in the following example:

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

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

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

### 4.1.9.1. About offline mode

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

1. When you create a service in offline mode, no changes happen on the cluster. Instead, the only thing that happens is the creation of the service descriptor file on your local machine.
2. After the descriptor file is created, you can manually modify it and track it in a version control system.
3. Finally, you can propagate changes to the cluster by using the `kn service create -f`, `kn service apply -f`, or `oc apply -f` commands on the descriptor files.

The offline mode has several uses:

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

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

### 4.1.9.2. Creating a service using offline mode

**Prerequisites**

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

**Procedure**

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

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

Example output

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

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

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

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

2. Examine the created directory structure:

```bash
$ tree ./
.
  ├── test
  │   └── ksvc
  │       └── event-display.yaml
  └── 2 directories, 1 file
```

Example output

- The current `./` directory specified with `--target` contains the new `test/` directory that is named after the specified namespace.
- The `test/` directory contains the `ksvc` directory, named after the resource type.
- The `ksvc` directory contains the descriptor file `event-display.yaml`, named according to the specified service name.

3. Examine the generated service descriptor file:

```bash
$ cat test/ksvc/event-display.yaml
```

Example output

```yaml
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  creationTimestamp: null
name: event-display
namespace: test
spec:
```
4. List information about the new service:

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

**Example output**

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

Revisions: 

Conditions: 
  OK TYPE AGE REASON

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

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

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

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

**Example output**

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

- 0.058s The Route is still working to reflect the latest desired specification.
- 0.098s ...
- 0.168s Configuration "event-display" is waiting for a Revision to become ready.
- 23.377s ...
- 23.419s Ingress has not yet been reconciled.
- 23.534s Waiting for load balancer to be ready
4.2. 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 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 `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.

4.2.1. 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.

4.2.1.1. Minimum scale bounds

The minimum number of replicas that can serve an application is determined by the `minScale` annotation.

The `minScale` value defaults to 0 replicas if the following conditions are met:

- The `minScale` annotation is not set
- Scaling to zero is enabled
- The class KPA is used

If scale to zero is not enabled, the `minScale` value defaults to 1.

Example service spec with `minScale` annotation

```yaml
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: example-service
  namespace: default
```
4.2.1.1. Setting the minScale annotation by using the Knative CLI

You can use the `kn service` command with the `--min-scale` flag to create or modify the `--min-scale` value for a service.

Prerequisites

- Knative Serving is installed on the cluster.
- You have installed the `kn` CLI.

Procedure

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

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

Example command

  ```
  $ kn service create example-service --image quay.io/openshift-knative/knative-eventing-sources-event-display:latest --min-scale 2
  ```

4.2.1.2. Maximum scale bounds

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

Example service spec with `maxScale` annotation

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: example-service
  namespace: default
spec:
  template:
    metadata:
      annotations:
        autoscaling.knative.dev/maxScale: "10"
...
```

4.2.1.2.1. Setting the maxScale annotation by using the Knative CLI

You can use the `kn service` command with the `--max-scale` flag to create or modify the `--max-scale` value for a service.
Prerequisites

- Knative Serving is installed on the cluster.
- You have installed the `kn` CLI.

Procedure

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

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

Example command

```bash
$ kn service create example-service --image quay.io/openshift-knative/knative-eventing-sources-event-display:latest --max-scale 10
```

4.2.2. Concurrency

Concurrency determines the number of simultaneous requests that can be processed by each replica of an application at any given time.

4.2.2.1. Concurrency limits and targets

Concurrency can be configured as either 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.

4.2.2.2. Configuring a soft concurrency target

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: example-service
    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 example-service --image quay.io/openshift-knative/knative-eventing-sources-event-display:latest --concurrency-target 50
  ```

4.2.2.3. Configuring a hard concurrency limit

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: example-service
    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 example-service --image quay.io/openshift-knative/knative-eventing-sources-event-display:latest --concurrency-limit 50
```

### 4.2.2.4. 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` annotation value is set to 10, and the `targetUtilizationPercentage` 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` annotation limit is reached.

**Example service configured using the targetUtilizationPercentage annotation**

```yaml
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: example-service
  namespace: default
spec:
template:
  metadata:
    annotations:
      autoscaling.knative.dev/targetUtilizationPercentage: "70"
...
```

### 4.2.3. Additional resources

- Scale-to-zero can be enabled or disabled for the cluster by cluster administrators. For more information, see [Enabling scale-to-zero](#).

### 4.3. TRAFFIC MANAGEMENT

A revision is a point-in-time snapshot of the code and configuration for each modification made to a Knative service. Each time the configuration of a service is updated, a new revision for the service is created. Revisions are immutable objects and can be retained for as long as they are required or used. Knative Serving revisions can be automatically scaled up and down according to incoming traffic.

You can manage traffic routing to different revisions of a Knative service by modifying the `traffic` spec of the service resource.
4.3.1. Traffic routing examples

When you create a Knative service, it does not have any default traffic spec settings. By setting the traffic spec, you can split traffic over any number of fixed revisions, or send traffic to the latest revision.

4.3.1.1. Traffic routing between multiple revisions

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`:

```yaml
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
```

4.3.1.2. Traffic routing to the latest revision
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:

```yaml
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
```

### 4.3.1.3. Traffic routing to the current revision

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:

```yaml
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
```

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

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.

**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.

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.
4.3.3. Traffic management using the Knative CLI

You can use the `kn service update` command to split traffic between revisions of a service.

**Example command**

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

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.
- `<percent>` is the percentage of traffic that you want to send to the specified revision.

### 4.3.3.1. 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.

**IMPORTANT**

The `--traffic` flag can be specified multiple times in one command, and is valid only if the sum of the Percent values in all flags totals 100.
You can add tags to revisions and then split traffic according to the tags you have set.

### 4.3.3.2. Example traffic management commands

In the following command, the `@latest` tag means that `blue` resolves to the latest revision of the service:

```bash
$ kn service update example-service --tag green=revision-0001 --tag blue=@latest
```

After the `green` and `blue` tags have been applied, you can run the following command to split traffic for the service named `example-service`, by sending 80% of traffic to the revision `green` and 20% of traffic to the revision `blue`:

```bash
$ kn service update example-service --traffic green=80 --traffic blue=20
```

Alternatively, you could use the following command to send 80% of traffic to the latest revision and 20% to a revision named `v1`, without using tags:

```bash
$ kn service update example-service --traffic @latest=80 --traffic v1=20
```

**NOTE**

You can only use the identifier `@latest` once per command with the `--traffic` flag.

### 4.3.3.3. Knative CLI traffic management flags

The `kn` CLI supports traffic operations on the traffic block of a service as part of the `kn service update` command.

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

<table>
<thead>
<tr>
<th>Flag</th>
<th>Value(s)</th>
<th>Operation</th>
<th>Repetition</th>
</tr>
</thead>
<tbody>
<tr>
<td>--traffic</td>
<td>RevisionName=Percent</td>
<td>Gives Percent traffic to RevisionName</td>
<td>Yes</td>
</tr>
<tr>
<td>--traffic</td>
<td>Tag=Percent</td>
<td>Gives Percent traffic to the revision having Tag</td>
<td>Yes</td>
</tr>
<tr>
<td>--traffic</td>
<td>@latest=Percent</td>
<td>Gives Percent traffic to the latest ready revision</td>
<td>No</td>
</tr>
<tr>
<td>--tag</td>
<td>RevisionName=Tag</td>
<td>Gives Tag to RevisionName</td>
<td>Yes</td>
</tr>
<tr>
<td>--tag</td>
<td>@latest=Tag</td>
<td>Gives Tag to the latest ready revision</td>
<td>No</td>
</tr>
</tbody>
</table>
4.3.3.4. 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.

### 4.3.3.4.1. Example: Assign a tag to a revision

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

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

### 4.3.3.4.2. 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`:

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

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

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

#### Prerequisites

- The OpenShift Serverless Operator and Knative Serving are installed on the cluster.

#### 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:

   ```bash
   $ oc get ksvc <service_name> -o=jsonpath='{.status.latestCreatedRevisionName}'
   
   Example command
   $ oc get ksvc example-service -o=jsonpath='{.status.latestCreatedRevisionName}''
   
   Example output
   $ example-service-00001
   
3. Add the following YAML to the service `spec` to send inbound traffic to the revision:

   ```yaml
   ... 
   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:

   ```bash
   $ oc get ksvc <service_name>
   
   Example of updated service spec with test endpoint
   ... 
   spec: 
     traffic: 
     - revisionName: <first_revision_name> 
       percent: 100 # All traffic goes to this revision
     - revisionName: <second_revision_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 `kn` CLI.

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

   ```bash
   $ 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:

   ```yaml
   ... 
   spec: 
     traffic: 
     - revisionName: <first_revision_name> 
       percent: 100 # All traffic is still being routed to the first revision
     - revisionName: <second_revision_name>
   ```
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:

   ```bash
   $ 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**

   ```yaml
   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**

    ```yaml
    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.
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.

### 4.4.1. Configuring OpenShift Container Platform routes for Knative services

If you want to configure a Knative service to use your TLS certificate on OpenShift Container Platform, you must disable the automatic creation of a route for the service by the OpenShift Serverless Operator and instead manually create a route for the service.

**NOTE**

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

#### Prerequisites

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

#### 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:

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

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

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:

```yaml
apiVersion: route.openshift.io/v1
kind: Route
metadata:
  annotations:
    haproxy.router.openshift.io/timeout: 600s
  name: <route_name>
  namespace: knative-serving-ingress
spec:
  host: <service_host>
  port:
    targetPort: http2
to:
  kind: Service
  name: kourier
  weight: 100
tls:
  insecureEdgeTerminationPolicy: Allow
termination: edge
```

1. `600s`
2. `<route_name>`
3. `knative-serving-ingress`
4. `<service_host>`
5. `edge`
---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>
```

### 4.4.2. Setting cluster availability to cluster local

By default, Knative services are published to a public IP address. Being published to a public IP address means that Knative services are public applications, and have a publicly accessible URL.

Publicly accessible URLs are accessible from outside of the cluster. However, developers may need to build back-end services that are only be accessible from inside the cluster, known as private services. Developers can label individual services in the cluster with the `networking.knative.dev/visibility=cluster-local` label to make them private.

**IMPORTANT**

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

**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:

```bash
$ oc label ksvc <service_name> networking.knative.dev/visibility=cluster-local
```

**Verification**

• Check that the URL for your service is now in the format `http://<service_name>.<namespace>.svc.cluster.local`, by entering the following command and reviewing the output:

```bash
$ oc get ksvc
```

**Example output**

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

### 4.4.3. Additional resources

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

### 4.5. EVENT SINKS

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

#### 4.5.1. Knative CLI `--sink` flag

When you create an event-producing custom resource by using the Knative (`kn`) CLI, you can specify a sink where events are sent to from that resource, by using the `--sink` flag.

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

**Example command using the `--sink` flag**

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

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

**TIP**

You can configure which CRs can be used with the `--sink` flag for `kn` CLI commands by [Customizing `kn`].
4.5.2. Connect an event source to a sink using the Developer perspective

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

Prerequisites

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

Procedure

1. Create an event source of any type, by navigating to +Add → Event Sources and then selecting the event source type that you want to create.
2. In the Sink section of the Event Sources form view, select Resource. Then use the drop-down list to select your sink.
3. Click Create.

Verification

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

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

4.5.3. Connecting a trigger to a sink

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

Example of a trigger connected to a Kafka sink

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

---

1. Replace `<trigger_name>` with the name of your trigger.
2. Replace `<kafka_sink_name>` with the name of your Kafka sink.
4.6. EVENT DELIVERY

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

4.6.1. Event delivery behavior for Knative Eventing channels

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

4.6.1.1. Event delivery behavior for Knative Kafka channels

If an event is successfully delivered to a Kafka channel or broker receiver, the receiver responds with a 202 status code, which means that the event has been safely stored inside a Kafka topic and is not lost. If the receiver responds with any other status code, the event is not safely stored, and steps must be taken by the user to resolve this issue.

4.6.1.2. Delivery failure status codes

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

500
This is a generic status code which means that the event was not delivered successfully.

404
This status code means that the channel or broker the event is being delivered to does not exist, or that the Host header is incorrect.

400
This status code means that the event being sent to the receiver is invalid.

4.6.2. Configurable event delivery parameters

The following parameters can be configured for event delivery.

Dead letter sink
You can configure the deadLetterSink delivery parameter so that if an event fails to be delivered it is sent to the specified event sink.

Retries
You can set a minimum number of times that the delivery must be retried before the event is sent to the dead letter sink, by configuring the retry delivery parameter with an integer value.

Back off delay
You can set the `backoffDelay` delivery parameter to specify the time delay before an event delivery retry is attempted after a failure. The duration of the `backoffDelay` parameter is specified using the ISO 8601 format.

**Back off policy**

The `backoffPolicy` delivery parameter can be used to specify the retry back off policy. The policy can be specified as either linear or exponential. When using the linear back off policy, the back off delay is the time interval specified between retries. When using the exponential backoff policy, the back off delay is equal to `backoffDelay*2^<numberOfRetries>`.

### 4.6.3. Configuring event delivery failure parameters using subscriptions

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

**Example subscription YAML**

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

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

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

2. You can set the `backoffDelay` delivery parameter to specify the time delay before an event delivery retry is attempted after a failure. The duration of the `backoffDelay` parameter is specified using the ISO 8601 format. For example, `PT1S` specifies a 1 second delay.

3. The `backoffPolicy` delivery parameter can be used to specify the retry back off policy. The policy can be specified as either linear or exponential. When using the linear back off policy, the back off delay is the time interval specified between retries. When using the exponential back off policy, the back off delay is equal to `backoffDelay*2^<numberOfRetries>`.

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

### 4.7. LISTING EVENT SOURCES AND EVENT SOURCE TYPES

You can use the `kn` CLI or the Developer perspective in the OpenShift Container Platform web console to list and manage available event sources or event source types.
4.7.1. Listing available event source types by using the Knative CLI

You can list event source types that can be created and used on your cluster by using the `kn source list-types` CLI command.

**Prerequisites**

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

**Procedure**

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

   ```
   $ kn source list-types
   ```

   **Example output**

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

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

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

4.7.2. Viewing available event source types within the Developer perspective

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

**Prerequisites**

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

**Procedure**

1. Access the **Developer** perspective.

2. Click **+Add**.

3. Click **Event source**.

4. View the available event source types.
4.7.3. Listing available event sources by using the Knative CLI

You can list available event sources by using the `kn source list` command.

```
$ kn source list
```

**Example output**

```
NAME   TYPE              RESOURCE                               SINK         READY
a1     ApiServerSource   apiserversources.sources.knative.dev   ksvc:eshow2   True
b1     SinkBinding       sinkbindings.sources.knative.dev       ksvc:eshow3   False
p1     PingSource        pingsources.sources.knative.dev        ksvc:eshow1   True
```

### 4.7.3.1. Listing event sources of a specific type only

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

```
$ kn source list --type <event_source_type>
```

**Example command**

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

**Example output**

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

### 4.8. CREATING AN API SERVER SOURCE

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

#### 4.8.1. Creating an API server source by using the web console

You can use the following procedure to create an API server source by using the OpenShift Container Platform web console.

**Prerequisites**

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

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

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

```yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  name: events-sa
namespace: default

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

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

1.2.3.4 Change this namespace to the namespace that you have selected for installing the event source.

2. Apply the YAML file:

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

3. In the **Developer** perspective, navigate to **Add → Event Source**. The **Event Sources** page is displayed.
4. Optional: If you have multiple providers for your event sources, select the required provider from the Providers list to filter the available event sources from the provider.

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

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

   NOTE

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

   a. Enter v1 as the APIVERSION and Event as the KIND.
   
   b. Select the Service Account Name for the service account that you created.
   
   c. Select the Sink for the event source. A Sink can be either a Resource, such as a channel, broker, or service, or a URI.

7. Click Create.

Verification

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

   ![Topology view](image)

   NOTE

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

Deleting the API server source

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

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

You can use the following procedure to create an API server source by using the **kn** CLI.

**Prerequisites**

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

**PROCEDURE**

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

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

   ```yaml
   apiVersion: v1
   kind: ServiceAccount
   metadata:
     name: events-sa
     namespace: default
   ---
   apiVersion: rbac.authorization.k8s.io/v1
   ```
1. Change this namespace to the namespace that you have selected for installing the event source.

2. Apply the YAML file:

   $ oc apply -f <filename>

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

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

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

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

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

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

6. Create events by launching a pod in the default namespace:
$ oc create deployment hello-node --image quay.io/openshift-knative/knative-eventing-sources-event-display:latest

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

$ kn source apiserver describe <source_name>

**Example output**

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

**Verification**

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

1. Get the pods:

   $ oc get pods

2. View the message dumper function logs for the pods:

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

**Example output**

```cloudevents.Event
Validation: valid
Context Attributes,
  specversion: 1.0
type: dev.knative.apiserver.resource.update
datacontenttype: application/json
...```
Deleting the API server source

1. Delete the trigger:

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

2. Delete the event source:

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

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

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

4.8.2.1. Knative CLI --sink flag

When you create an event-producing custom resource by using the Knative (kn) CLI, you can specify a sink where events are sent from that resource, by using the --sink flag.

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

**Example command using the --sink flag**

```bash
$ kn source binding create bind-heartbeat \  
   --namespace sinkbinding-example \  
   --subject "Job:batch/v1:app=heartbeat-cron" \  
   --sink http://event-display.svc.cluster.local \  
   --ce-override "sink=bound"
```
svc in `http://event-display.svc.cluster.local` determines that the sink is a Knative service. Other default sink prefixes include channel, and broker.

### 4.8.3. Creating an API server source by using YAML files

You can use the following procedure to create an API server source by using YAML files.

**Prerequisites**

- The OpenShift Serverless Operator and Knative Eventing are installed on the cluster.
- You have created a project or have access to a project with the appropriate roles and permissions to create applications and other workloads in OpenShift Container Platform.
- You have created the default broker in the same namespace as the one defined in the API server source YAML file.
- You have installed the `oc` CLI.

**PROCEDURE**

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

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

```yaml
apiVersion: v1
class: ServiceAccount
metadata:
  name: events-sa
  namespace: default

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

---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: k8s-ra-event-watcher
  namespace: default
subjects:
- kind: ServiceAccount
  name: events-sa
  namespace: default
```

---
Change this namespace to the namespace that you have selected for installing the event source.

2. Apply the YAML file:

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

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

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

4. Apply the `ApiServerSource` YAML file:

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

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

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

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

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

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

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

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

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

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

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

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

**Example output**

```
apiVersion: sources.knative.dev/v1alpha1
kind: ApiServerSource
metadata:
  annotations:
  creationTimestamp: "2020-04-07T17:24:54Z"
  generation: 1
  name: testevents
  namespace: default
  resourceVersion: "62868"
  selfLink:
    /apis/sources.knative.dev/v1alpha1/namespaces/default/apiserversources/testevents
  uid:
    1603d863-bb06-4d1c-b371-f580b4db99fa
spec:
  mode: Resource
  resources:
  - apiVersion: v1
    controller: false
    controllerSelector:
```
Verification

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

1. Get the pods by entering the following command:

   ```
   $ oc get pods
   ```

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

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

Example output

```json
cloudevents.Event
Validation: valid
Context Attributes,
  specversion: 1.0
  type: dev.knative.apiserver.resource.update
datacontenttype: application/json
...
Data,
{
  "apiVersion": "v1",
  "involvedObject": {
    "apiVersion": "v1",
    "fieldPath": "spec.containers\{hello-node\}"
  }
}
"kind": "Event",
"message": "Started container",
"metadata": {
  "name": "hello-node.159d7608e3a3572c",
  "namespace": "default",
  ...
}
```
Deleting the API server source

1. Delete the trigger:

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

2. Delete the event source:

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

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

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

4.9. CREATING A PING SOURCE

A ping source is used to periodically send ping events with a constant payload to an event consumer.

A ping source can be used to schedule sending events, similar to a timer.

Example ping source YAML

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

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

4.9.1. Creating a ping source by using the web console

You can use the following procedure to create a ping source by using the OpenShift Container Platform web console.
Prerequisites

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

Procedure

1. To verify that the ping source is working, create a simple Knative service that dumps incoming messages to the logs of the service.
   a. In the Developer perspective, navigate to +Add → YAML.
   b. Copy the example YAML:
      ```yaml
      apiVersion: serving.knative.dev/v1
      kind: Service
      metadata:
        name: event-display
      spec:
        template:
          spec:
            containers:
              - image: quay.io/openshift-knative/knative-eventing-sources-event-display:latest
      ```
   c. Click Create.

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

   **NOTE**

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

   d. Enter a value for Schedule. In this example, the value is */2 * * * *, which creates a PingSource that sends a message every two minutes.
   e. Optional: You can enter a value for Data, which is the message payload.
   f. Select a Sink. This can be either a Resource or a URI. In this example, the event-display service created in the previous step is used as the Resource sink.
g. Click Create.

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

1. In the Developer perspective, navigate to Topology.

2. View the ping source and sink.

Deleting the ping source

1. Navigate to the Topology view.

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

4.9.2. Creating a ping source by using the Knative CLI
You can use the following procedure to create a ping source by using the kn CLI.

Prerequisites

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

Procedure

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

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

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

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

   $ kn service describe test-ping-source
3. Check that the controller is mapped correctly by entering the following command and inspecting the output:

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

**Example output**

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

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

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

**Verification**

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

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

1. Watch for new pods created:

```bash
$ watch oc get pods
```

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

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

**Example output**

```
   cloudevents.Event
Validation: valid
Context Attributes,
   specversion: 1.0
```
Deleting the ping source

- Delete the ping source:

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

4.9.2.1. Knative CLI --sink flag

When you create an event-producing custom resource by using the Knative (kn) CLI, you can specify a sink where events are sent to from that resource, by using the --sink flag.

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

Example command using the --sink flag

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

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

4.9.3. Creating a ping source by using YAML

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

Prerequisites

- The OpenShift Serverless Operator, Knative Serving and Knative Eventing are installed on the cluster.

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

Procedure

1. To verify that the ping source is working, create a simple Knative service that dumps incoming messages to the service’s logs.
   a. Copy the example YAML into a file named service.yaml:
b. Create the service:

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

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

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

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

b. Create the ping source:

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

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

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

**Example output**

```yaml
apiVersion: sources.knative.dev/v1alpha2
kind: PingSource
metadata:
  annotations:
    sources.knative.dev/creator: developer
    sources.knative.dev/lastModifier: developer
  creationTimestamp: "2020-04-07T16:11:14Z"
  generation: 1
  name: test-ping-source
  namespace: default
  resourceVersion: "55257"
  selfLink: /apis/sources.knative.dev/v1alpha2/namespaces/default/pingsources/test-ping-
```
You can verify that the Kubernetes events were sent to the Knative event sink by looking at the sink pod’s logs.

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

1. Watch for new pods created:

   ```bash
   $ watch oc get pods
   ```

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

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

**Example output**

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

**Deleting the ping source**

- Delete the ping source:

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

**Example command**

```bash
$ oc delete -f ping-source.yaml
```
4.10. CUSTOM EVENT SOURCES

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

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

4.10.1. Sink binding

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

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

**K_SINK**
The URL of the resolved sink.

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

4.10.1.1. Creating a sink binding by using YAML

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

**Prerequisites**

- The OpenShift Serverless Operator, Knative Serving and Knative Eventing are installed on the cluster.
- You have installed the oc 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**

1. To check that sink binding is set up correctly, create a Knative event display service, or event sink, that dumps incoming messages to its log.
   a. Copy the following sample YAML into a file named `service.yaml`:

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: event-display
spec:
template:
```
After you have created the `service.yaml` file, apply it by entering:

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

2. Create a sink binding instance that directs events to the service.
   a. Create a file named `sinkbinding.yaml` and copy the following sample code into it:

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

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

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

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

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

   ```yaml
   apiVersion: batch/v1
   kind: CronJob
   metadata:
     name: heartbeat-cron
   spec:
     # Run every minute
     schedule: "** * * * *"
     jobTemplate:
       metadata:
         labels:
           app: heartbeat-cron
       bindings.knative.dev/include: "true"
   ```
IMPORTANT

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

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

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

b. After you have created the `heartbeats-cronjob.yaml` file, apply it by entering:

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

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

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

Example output

```yaml
spec:
sink:
  ref:
    apiVersion: serving.knative.dev/v1
    kind: Service
    name: event-display
    namespace: default
  subject:
```
Verification

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

1. Enter the command:

   $ oc get pods

2. Enter the command:

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

Example output

```
☁ cloudevents.Event
Validation: valid
Context Attributes,
  specversion: 1.0
  type: dev.knative.eventing.samples.heartbeat
  source: https://knative.dev/eventing-contrib/cmd/heartbeats/#event-test/mypod
  id: 2b72d7bf-c38f-4a98-a433-608fcdd2596
  contenttype: application/json
Extensions,
  beats: true
  heart: yes
  the: 42
Data,
  {
    "id": 1,
    "label": ""
  }
```

4.10.1.2. Creating a sink binding by using the Knative CLI

This guide describes the steps required to create a sink binding instance using \texttt{kn} commands.

Prerequisites

- The OpenShift Serverless Operator, Knative Serving and Knative Eventing are installed on the cluster.

- You have installed the \texttt{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.
The following procedure requires you to create YAML files.

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

Procedure

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

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

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

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

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

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

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

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

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

b. After you have created the `heartbeats-cronjob.yaml` file, apply it by entering:

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

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

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

Example output

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

Conditions:
  OK TYPE AGE REASON
    ++ Ready 2m

Verification

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

- View the message dumper function logs by entering the following commands:
4.10.1.2.1. Knative CLI --sink flag

When you create an event-producing custom resource by using the Knative `(kn)` CLI, you can specify a sink where events are sent to from that resource, by using the `--sink` flag.

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

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

4.10.1.3. Creating a sink binding by using the web console

You can create and verify a basic sink binding from the OpenShift Container Platform web console.

**Prerequisites**

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

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

Procedure

1. Create a Knative service to use as a sink:
   a. In the **Developer** perspective, navigate to **+Add → YAML**.
   b. Copy the example YAML:

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

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

   ```yaml
   apiVersion: batch/v1
   kind: CronJob
   metadata:
     name: heartbeat-cron
   spec:
     # Run every minute
     schedule: "*/1 * * * *"
   jobTemplate:
     metadata:
       labels:
         app: heartbeat-cron
       bindings.knative.dev/include: true
   spec:
     template:
       spec:
         restartPolicy: Never
         containers:
           - name: single-heartbeat
             image: quay.io/openshift-knative/heartbeats
             args:
               - --period=1
             env:
               - name: ONE_SHOT
                 value: "true"
   ```
Ensure that you include the `bindings.knative.dev/include: true` label. The default namespace selection behavior of OpenShift Serverless uses inclusion mode.

c. Click Create.

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

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

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

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

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

NOTE

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

d. In the apiVersion field enter batch/v1.

e. In the Kind field enter Job.

```
NOTE

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

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

g. In the Match labels section:

i. Enter app in the Name field.

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

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

h. Click Create.

Verification

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

1. In the Developer perspective, navigate to Topology.

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

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

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

4.10.1.4. Sink binding reference

This topic provides reference information about the configurable parameters for SinkBinding objects.

SinkBinding objects support the following parameters:

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>Required or optional</th>
</tr>
</thead>
<tbody>
<tr>
<td>apiVersion</td>
<td>Specifies the API version, for example <code>sources.knative.dev/v1</code>.</td>
<td>Required</td>
</tr>
<tr>
<td>kind</td>
<td>Identifies this resource object as a SinkBinding object.</td>
<td>Required</td>
</tr>
</tbody>
</table>
### Field | Description | Required or optional
--- | --- | ---
**metadata** | Specifies metadata that uniquely identifies the **SinkBinding** object. For example, a **name**. | Required
**spec** | Specifies the configuration information for this **SinkBinding** object. | Required
**spec.sink** | A reference to an object that resolves to a URI to use as the sink. | Required
**spec.subject** | References the resources for which the runtime contract is augmented by binding implementations. | Required
**spec.ceOverrides** | Defines overrides to control the output format and modifications to the event sent to the sink. | Optional

#### 4.10.1.4.1. Subject parameter

The **Subject** parameter references the resources for which the runtime contract is augmented by binding implementations.

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

### Field | Description | Required or optional
--- | --- | ---
**apiVersion** | API version of the referent. | Required
**kind** | Kind of the referent. | Required
**namespace** | Namespace of the referent. If omitted, this defaults to the namespace of the object. | Optional
**name** | Name of the referent. Do not use if you configure **selector**. | Optional
**selector** | Selector of the referents. Do not use if you configure **name**. | Optional
**selector.matchExpressions** | A list of label selector requirements. Only use one of either **matchExpressions** or **matchLabels**. | Optional
<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>Required or optional</th>
</tr>
</thead>
<tbody>
<tr>
<td>selector.matchExpressions.key</td>
<td>The label key that the selector applies to.</td>
<td>Required if using matchExpressions.</td>
</tr>
<tr>
<td>selector.matchExpressions.operator</td>
<td>Represents a key’s relationship to a set of values. Valid operators are <code>In</code>, <code>NotIn</code>, <code>Exists</code> and <code>DoesNotExist</code>.</td>
<td>Required if using matchExpressions.</td>
</tr>
<tr>
<td>selector.matchExpressions.values</td>
<td>An array of string values. If the <code>operator</code> parameter value is <code>In</code> or <code>NotIn</code>, the values array must be non-empty. If the <code>operator</code> parameter value is <code>Exists</code> or <code>DoesNotExist</code>, the values array must be empty. This array is replaced during a strategic merge patch.</td>
<td>Required if using matchExpressions.</td>
</tr>
<tr>
<td>selector.matchLabels</td>
<td>A map of key-value pairs. Each key-value pair in the <code>matchLabels</code> map is equivalent to an element of <code>matchExpressions</code>, where the key field is <code>matchLabels.&lt;key&gt;</code>, the <code>operator</code> is <code>In</code>, and the <code>values</code> array contains only <code>matchLabels.&lt;value&gt;</code>.</td>
<td>Only use one of either <code>matchExpressions</code> or <code>matchLabels</code>.</td>
</tr>
</tbody>
</table>

### 4.10.14.11. Subject parameter examples

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

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

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

```yaml
apiVersion: sources.knative.dev/v1
kind: SinkBinding
```
Given the following YAML, any Pod object with the label `working=example` or `working=sample` in the `default` namespace is selected:

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

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

```
apiVersion: sources.knative.dev/v1
kind: SinkBinding
metadata:
  name: bind-heartbeat
spec:
  subject:
    apiVersion: v1
    kind: Pod
    namespace: default
    selector:
      - matchExpression:
          key: working
          operator: In
          values:
            - example
            - sample
...
```

### 4.10.1.4.2. CloudEvent overrides

A `ceOverrides` definition provides overrides that control the CloudEvent’s output format and modifications sent to the sink.

A `ceOverrides` definition supports the following fields:

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>Required or optional</th>
</tr>
</thead>
<tbody>
<tr>
<td>extensions</td>
<td>Specifies which attributes are added or overridden on the outbound event. Each <code>extensions</code> key-value pair is set independently on the event as an attribute extension.</td>
<td>Optional</td>
</tr>
</tbody>
</table>
Only valid CloudEvent attribute names are allowed as extensions. You cannot set the spec defined attributes from the extensions override configuration. For example, you cannot modify the type attribute.

CloudEvent Overrides example

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

This sets the K_CE_OVERRIDES environment variable on the subject:

Example output

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

4.10.1.4.3. The include label

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

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

4.10.2. Container source

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

4.10.2.1. Guidelines for creating a container image

- Two environment variables are injected by the container source controller: K_SINK and K_CE_OVERRIDES. These variables are resolved from the sink and ceOverrides spec, respectively.
- Event messages are sent to the sink URI specified in the K_SINK environment variable. The event message can be in any format; however, using the CloudEvent spec is recommended.

4.10.2.1.1. Example container images

The following is an example of a heartbeats container image:

```go
package main
```
import (  
    "context"  
    "encoding/json"  
    "flag"  
    "fmt"  
    "log"  
    "os"  
    "strconv"  
    "time"  
)

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

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


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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```go
hb.Sequence++

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

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

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

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

if env.OneShot {
    return
}

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

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

```yaml
apiVersion: sources.knative.dev/v1
kind: ContainerSource
metadata:
  name: test-heartbeats
spec:
  template:
    spec:
      containers:
        - image: gcr.io/knative-releases/knative.dev/eventing/cmd/heartbeats
          name: heartbeats
          args:
            - --period=1
          env:
            - name: POD_NAME
              value: "example-pod"
            - name: POD_NAMESPACE
              value: "event-test"
        
sink:
      ref:
```
apiVersion: serving.knative.dev/v1
kind: Service
name: example-service
...

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

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

Create a container source

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

Delete a container source

$ kn source container delete <container_source_name>

Describe a container source

$ kn source container describe <container_source_name>

List existing container sources

$ kn source container list

List existing container sources in YAML format

$ kn source container list -o yaml

Update a container source

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

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

4.10.2.3. Creating a container source by using the web console

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

Prerequisites

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

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

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

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

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

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

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

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

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

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

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

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

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

4.10.2.4. **Container source reference**

This topic provides reference information about the configurable fields for the **ContainerSource** object.

**ContainerSource** objects support the following fields:

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>Required or optional</th>
</tr>
</thead>
<tbody>
<tr>
<td>apiVersion</td>
<td>Specifies the API version, for example sources.knative.dev/v1.</td>
<td>Required</td>
</tr>
<tr>
<td>kind</td>
<td>Identifies this resource object as a <strong>ContainerSource</strong> object.</td>
<td>Required</td>
</tr>
<tr>
<td>metadata</td>
<td>Specifies metadata that uniquely identifies the <strong>ContainerSource</strong> object. For example, a name.</td>
<td>Required</td>
</tr>
</tbody>
</table>
### Field Description

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>Required or optional</th>
</tr>
</thead>
<tbody>
<tr>
<td>spec</td>
<td>Specifies the configuration information for this <code>ContainerSource</code> object.</td>
<td>Required</td>
</tr>
<tr>
<td>spec.sink</td>
<td>A reference to an object that resolves to a URI to use as the sink.</td>
<td>Required</td>
</tr>
<tr>
<td>spec.template</td>
<td>A <code>template</code> spec for the <code>ContainerSource</code> object.</td>
<td>Required</td>
</tr>
<tr>
<td>spec.ceOverrides</td>
<td>Defines overrides to control the output format and modifications to the event sent to the sink.</td>
<td>Optional</td>
</tr>
</tbody>
</table>

#### 4.10.2.4.1. Template parameter example

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

#### 4.10.2.4.2. CloudEvent overrides

A `ceOverrides` definition provides overrides that control the CloudEvent’s output format and modifications sent to the sink.

A `ceOverrides` definition supports the following fields:

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>Required or optional</th>
</tr>
</thead>
</table>
### Field Descriptions

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>Required or optional</th>
</tr>
</thead>
<tbody>
<tr>
<td>extensions</td>
<td>Specifies which attributes are added or overridden on the outbound event. Each extensions key-value pair is set independently on the event as an attribute extension.</td>
<td>Optional</td>
</tr>
</tbody>
</table>

**NOTE**

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

#### CloudEvent Overrides example

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

This sets the K_CE_OVERRIDES environment variable on the subject.

#### Example output

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

### 4.11. CREATING CHANNELS

You can use the following procedures to create channels.

#### 4.11.1. Creating a channel by using the web console

You can create a channel by using the OpenShift Container Platform web console.

**Prerequisites**

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

**Procedure**

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

**Verification**

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

**4.11.2. Creating a channel by using the Knative CLI**

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

**Prerequisites**

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

**Procedure**

- Create a channel:
  
  ```
  $ kn channel create <channel_name> --type <channel_type>
  ```

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

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

  **Example output**

  Channel ‘mychannel’ created in namespace ‘default’.
Verification

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

  ```
  $ kn channel list
  ```

Example output

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

Deleting a channel

- Delete a channel:

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

4.11.3. Creating a default implementation channel by using YAML

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

Prerequisites

- The OpenShift Serverless Operator and Knative Eventing are installed on the cluster.
- You have installed the `oc` 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

1. Create a Channel object as a YAML file:

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

2. Apply the YAML file:

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

4.11.4. Creating a Kafka channel by using YAML

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

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

• You have installed the **oc** 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.

• You have installed the **oc** CLI.

**Procedure**

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

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

   **IMPORTANT**

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

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

   `$ oc apply -f <filename>`

**4.11.5. Next steps**

• After you have created a channel, [create a subscription](#) that allows event sinks to subscribe to channels and receive events.

**4.12. CREATING SUBSCRIPTIONS**

You can use the following procedures to create subscriptions that allow event sinks to subscribe to channels and receive events.

**4.12.1. Creating a subscription by using the web console**

You can create a subscription to connect a channel to a sink by using the OpenShift Container Platform web console.

**Prerequisites**

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

• You have created an event sink, such as a Knative service, and a channel.

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

Procedure

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

2. Create a subscription using one of the following methods:
   a. Hover over the channel that you want to create a subscription for, and drag the arrow. The Add Subscription option is displayed.

   i. Select your sink as a subscriber from the drop-down list.

   ii. Click Add.

   b. If the service is available in the Topology view under the same namespace or project as the channel, click on the channel that you want to create a subscription for, and drag the arrow directly to a service to immediately create a subscription from the channel to that service.

Verification

• After the subscription has been created, you can see it represented as a line that connects the channel to the service in the Topology view:
4.12.2. Creating a subscription by using YAML

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

**Prerequisites**

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

**Procedure**

- Create a **Subscription** object:
  - Create a YAML file and copy the following sample code into it:

```
apiVersion: messaging.knative.dev/v1beta1
kind: Subscription
metadata:
  name: my-subscription
  namespace: default
spec:
  channel:
    apiVersion: messaging.knative.dev/v1beta1
    kind: Channel
    name: example-channel
delivery:
  deadLetterSink:
    ref:
      apiVersion: serving.knative.dev/v1
      kind: Service
      name: error-handler
subscriber:
  ref:
```

OpenShift Container Platform 4.9 Serverless
Name of the subscription.

Configuration settings for the channel that the subscription connects to.

Configuration settings for event delivery. This tells the subscription what happens to events that cannot be delivered to the subscriber. When this is configured, events that failed to be consumed are sent to the **deadLetterSink**. The event is dropped, no re-delivery of the event is attempted, and an error is logged in the system. The **deadLetterSink** value must be a **Destination**.

Configuration settings for the subscriber. This is the event sink that events are delivered to from the channel.

- Apply the YAML file:

  $ oc apply -f <filename>

### 4.12.3. Creating a subscription by using the Knative CLI

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

**Prerequisites**

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

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

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

**Procedure**

- Create a subscription to connect a sink to a channel:

  ```bash
  $ kn subscription create <subscription_name> \ 
  --channel <group:version:kind>:<channel_name> \ 1
  --sink <sink_prefix>:<sink_name> \ 2
  --sink-dead-letter <sink_prefix>:<sink_name> 3
  ```

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

  2. **--sink** specifies the target destination to which the event should be delivered. By default, the **<sink_name>** is interpreted as a Knative service of this name, in the same namespace as the subscription. You can specify the type of the sink by using one of the following prefixes:
ksvc
A Knative service.

channel
A channel that should be used as destination. Only default channel types can be referenced here.

broker
An Eventing broker.

Optional: --sink-dead-letter is an optional flag that can be used to specify a sink which events should be sent to in cases where events fail to be delivered. For more information, see the OpenShift Serverless Event delivery documentation.

Example command

$ kn subscription create mysubscription --channel mychannel --sink ksvc:event-display

Example output

Subscription 'mysubscription' created in namespace 'default'.

Verification

- To confirm that the channel is connected to the event sink, or subscriber, by a subscription, list the existing subscriptions and inspect the output:

  $ kn subscription list

  Example output

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

Deleting a subscription

- Delete a subscription:

  $ kn subscription delete <subscription_name>

4.12.4. Describing subscriptions by using the Knative CLI

You can print information about a subscription in the terminal by using the kn CLI.

Prerequisites

- You have installed the kn CLI.
- You have created a subscription in your cluster.

Procedure
- Describe a subscription:

```bash
$ kn subscription describe <subscription_name>
```

**Example output**

```plaintext
Name:          my-subscription
Namespace:     default
Annotations:   messaging.knative.dev/creator=openshift-user,
messaging.knative.dev/lastModifier=min ...
Age:           43s
Channel:       Channel:my-channel (messaging.knative.dev/v1)
Subscriber:
  URI:         http://edisplay.default.example.com
Reply:
  Name:       default
  Resource:   Broker (eventing.knative.dev/v1)
DeadLetterSink:
  Name:       my-sink
  Resource:   Service (serving.knative.dev/v1)

Conditions:
  OK TYPE                  AGE REASON
  ++ Ready                 43s
  ++ AddedToChannel        43s
  ++ ChannelReady          43s
  ++ ReferencesResolved    43s
```

4.12.5. Listing subscriptions by using the Knative CLI

You can list existing subscriptions on your cluster by using the **kn** CLI.

**Prerequisites**

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

**Procedure**

- List subscriptions on your cluster:

```bash
$ kn subscription list
```

**Example output**

```plaintext
<table>
<thead>
<tr>
<th>NAME</th>
<th>CHANNEL</th>
<th>SUBSCRIBER</th>
<th>REPLY</th>
<th>DEAD LETTER SINK</th>
</tr>
</thead>
<tbody>
<tr>
<td>mysubscription</td>
<td>Channel:mychannel</td>
<td>ksvc:event-display</td>
<td>True</td>
<td></td>
</tr>
</tbody>
</table>

4.12.6. Updating subscriptions by using the Knative CLI

You can update a subscription by using the **kn** CLI.
Prerequisites

- You have installed the **kn** CLI.
- You have created a subscription.

Procedure

- Update a subscription:

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

  **--sink** specifies the updated target destination to which the event should be delivered. You can specify the type of the sink by using one of the following prefixes:

  - **ksvc**
    A Knative service.
  - **channel**
    A channel that should be used as destination. Only default channel types can be referenced here.
  - **broker**
    An Eventing broker.

  **Optional:** **--sink-dead-letter** is an optional flag that can be used to specify a sink which events should be sent to in cases where events fail to be delivered. For more information, see the OpenShift Serverless *Event delivery* documentation.

Example command

```
$ kn subscription update mysubscription --sink ksvc:event-display
```

4.12.7. Configuring event delivery failure parameters using subscriptions

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

Example subscription YAML

```yaml
apiVersion: messaging.knative.dev/v1
kind: Subscription
metadata:
  name: <subscription_name>
  namespace: <subscription_namespace>
spec:
delivery:
deadLetterSink: 1
  ref:
    apiVersion: serving.knative.dev/v1
    kind: Service
    name: <sink_name>
```
Configuration settings to enable using a dead letter sink. This tells the subscription what happens to events that cannot be delivered to the subscriber.

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

You can set the `backoffDelay` delivery parameter to specify the time delay before an event delivery retry is attempted after a failure. The duration of the `backoffDelay` parameter is specified using the ISO 8601 format. For example, `PT1S` specifies a 1 second delay.

The `backoffPolicy` delivery parameter can be used to specify the retry back off policy. The policy can be specified as either linear or exponential. When using the linear back off policy, the back off delay is the time interval specified between retries. When using the exponential back off policy, the back off delay is equal to `backoffDelay*2^<numberOfRetries>`.

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

**4.13. BROKERS**

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

**4.13.1. Broker types**

There are multiple broker implementations available for use with OpenShift Serverless, each of which have different event delivery guarantees and use different underlying technologies. You can choose the broker implementation when creating a broker by specifying a broker class, otherwise the default broker class is used. The default broker class can be configured by cluster administrators.

**4.13.1.1. Channel-based broker**

Events can be sent from an event source to a broker as an HTTP POST request. After events have entered the broker, they can be filtered by CloudEvent attributes using triggers, and sent as an HTTP POST request to an event sink.
The channel-based broker implementation internally uses channels for event delivery. Channel-based brokers provide different event delivery guarantees based on the channel implementation a broker instance uses, for example:

- A broker using the `InMemoryChannel` implementation is useful for development and testing purposes, but does not provide adequate event delivery guarantees for production environments.
- A broker using the `KafkaChannel` implementation provides the event delivery guarantees required for a production environment.

### 4.13.1.2. Kafka broker

The Kafka broker is a broker implementation that uses Kafka internally to provide at-least once delivery guarantees. It supports multiple Kafka versions, and has a native integration with Kafka for storing and routing events.

**IMPORTANT**

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

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

**Additional resources**

- See the [Event delivery](#) documentation for more information about delivery guarantees.
- See the [Kafka broker](#) documentation for information about using Kafka brokers.

### 4.13.2. Creating a broker that uses default settings

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

#### 4.13.2.1. Creating a broker by using the Knative CLI

**Prerequisites**

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

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

   ```
   $ kn broker list
   ```

Example output

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

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

![Topology view](image)

### 4.13.2.2. Creating a broker by annotating a trigger

You can create a broker by adding the **eventing.knative.dev/injection: enabled** annotation to a **Trigger** object.

**IMPORTANT**

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

**Prerequisites**

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

- You have installed the **oc** 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

1. Create a **Trigger** object as a YAML file that has the `eventing.knative.dev/injection: enabled` annotation:

   ```yaml
   apiVersion: eventing.knative.dev/v1
   kind: Trigger
   metadata:
     annotations:
       eventing.knative.dev/injection: enabled
   spec:
     broker: default
     subscriber: 1
     ref:
       apiVersion: serving.knative.dev/v1
       kind: Service
       name: <service_name>
   ```

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

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

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

Verification

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

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

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

   **Example output**

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

2. Navigate to the **Topology** view in the web console, and observe that the broker exists:
4.13.2.3. Creating a broker by labeling a namespace

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

**NOTE**

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

**Prerequisites**

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

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

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

**Verification**

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

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

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

**Example command**

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

**Example output**

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

2. Navigate to the **Topology** view in the web console, and observe that the broker exists:
4.13.2.4. Deleting a broker that was created by injection

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

Procedure

1. Remove the `eventing.knative.dev/injection=enabled` label from the namespace:
   
   ```
   $ oc label namespace <namespace> eventing.knative.dev/injection-
   
   Removing the annotation prevents Knative from recreating the broker after you delete it.
   ```

2. Delete the broker from the selected namespace:
   
   ```
   $ oc -n <namespace> delete broker <broker_name>
   ```

Verification

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

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

Example command

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

Example output

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

4.13.3. Managing brokers

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

4.13.3.1. Listing existing brokers by using the Knative CLI
**Prerequisites**

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

**Procedure**

- List all existing brokers:
  
  ```
  $ kn broker list
  ```

**Example output**

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

**4.13.3.2. Describing an existing broker by using the Knative CLI**

**Prerequisites**

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

**Procedure**

- Describe an existing broker:
  
  ```
  $ kn broker describe <broker_name>
  ```

**Example command using default broker**

```
$ kn broker describe default
```

**Example output**

```
Name:         default
Namespace:    default
Annotations:  eventing.knative.dev/broker.class=MTChannelBasedBroker, eventing.knative.dev/... Age:         22s
Address:      
URL:    http://broker-ingress.knative-eventing.svc.cluster.local/default/default

Conditions:
OK TYPE         AGE REASON
++ Ready       22s
```
4.14. TRIGGERS

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

4.14.1. Creating a trigger by using the web console

After you have created a broker, you can create a trigger by using the web console.

Prerequisites

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

Procedure

1. In the Developer perspective, navigate to the Topology page.
2. Hover over the broker that you want to create a trigger for, and drag the arrow. The Add Trigger option is displayed.
3. Click Add Trigger.
4. Select your sink as a Subscriber from the drop-down list.
5. Click Add.

Verification

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

Deleting a trigger

1. In the Developer perspective, navigate to the Topology page.
2. Click on the trigger that you want to delete.
3. In the Actions context menu, select Delete Trigger.

4.14.2. Creating a trigger by using the Knative CLI

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

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

Procedure

1. Create a trigger:

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

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

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

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

4.14.3. Listing triggers by using the Knative CLI

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

Prerequisites

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

Procedure

1. Print a list of available triggers:

   ```bash
   $ kn trigger list
   ```

   **Example output**

<table>
<thead>
<tr>
<th>NAME</th>
<th>BROKER</th>
<th>SINK</th>
<th>AGE</th>
<th>CONDITIONS</th>
<th>READY</th>
<th>REASON</th>
</tr>
</thead>
<tbody>
<tr>
<td>email</td>
<td>default</td>
<td>ksvc:edisplay</td>
<td>4s</td>
<td>5 OK / 5</td>
<td>True</td>
<td></td>
</tr>
<tr>
<td>ping</td>
<td>default</td>
<td>ksvc:edisplay</td>
<td>32s</td>
<td>5 OK / 5</td>
<td>True</td>
<td></td>
</tr>
</tbody>
</table>

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

   ```bash
   $ kn trigger list -o json
   ```
4.14.4. Describing a trigger by using the Knative CLI

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

**Prerequisites**

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

**Procedure**

- Enter the command:

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

**Example output**

```
Name:         ping
Namespace:    default
Labels:       eventing.knative.dev/broker=default
Annotations:  eventing.knative.dev/creator=kube:admin,
              eventing.knative.dev/lastModifier=kube:admin
Age:          2m
Broker:       default
Filter:
type:       dev.knative.event
Sink:
Name:       edisplay
Namespace:  default
Resource:   Service (serving.knative.dev/v1)

Conditions:
OK TYPE                  AGE REASON
++ Ready                  2m
++ BrokerReady            2m
++ DependencyReady        2m
++ Subscribed             2m
++ SubscriberResolved     2m
```
4.14.6. Updating a trigger by using the Knative CLI

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

**Prerequisites**

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

**Procedure**

- Update a trigger:

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

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

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

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

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

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

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

4.14.7. Deleting a trigger by using the Knative CLI

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

**Prerequisites**

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

- Delete a trigger:
  
  ```
  $ kn trigger delete <trigger_name>
  ```

**Verification**

1. List existing triggers:

   ```
   $ kn trigger list
   ```

2. Verify that the trigger no longer exists:

   **Example output**

   ```
   No triggers found.
   ```

**4.15. KNATIVE KAFKA**

Knative Kafka functionality is available in an OpenShift Serverless installation if a cluster administrator has installed the KnativeKafka custom resource.

**NOTE**

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

Knative Kafka provides additional options, such as:

- Kafka source
- Kafka channel
- Kafka broker

**IMPORTANT**

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

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

**4.15.1. Kafka event delivery and retries**

Using Kafka components in an event-driven architecture provides "at least once" event delivery. This means that operations are retried until a return code value is received. This makes applications more resilient to lost events; however, it might result in duplicate events being sent.
For the Kafka event source, there is a fixed number of retries for event delivery by default. For Kafka channels, retries are only performed if they are configured in the Kafka channel Delivery spec.

See the Event delivery documentation for more information about delivery guarantees.

4.15.2. Kafka source

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

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

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

Prerequisites

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

Procedure

1. In the Developer perspective, navigate to the Add page and select Event Source.
2. In the Event Sources page, select Kafka Source in the Type section.
3. Configure the Kafka Source settings:
   a. Add a comma-separated list of Bootstrap Servers.
   b. Add a comma-separated list of Topics.
   c. Add a Consumer Group.
   d. Select the Service Account Name for the service account that you created.
   e. Select the Sink for the event source. A Sink can be either a Resource, such as a channel, broker, or service, or a URI.
   f. Enter a Name for the Kafka event source.
4. Click Create.

Verification

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

1. In the Developer perspective, navigate to Topology.
2. View the Kafka event source and sink.

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

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

**Prerequisites**

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

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

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

- You have installed the `kn` CLI.

**Procedure**

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

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

2. Create a **KafkaSource** CR:

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

   **NOTE**

   Replace the placeholder values in this command with values for your source name, bootstrap servers, and topics.
The `--servers`, `--topics`, and `--consumergroup` options specify the connection parameters to the Kafka cluster. The `--consumergroup` option is optional.

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

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

**Example output**

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

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

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

**Verification steps**

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

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

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

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

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

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

**Example output**

```
☁ cloudevents.Event
Validation: valid
Context Attributes,
  specversion: 1.0
type: dev.knative.kafka.event
source: /apis/v1/namespaces/default/kafkasources/example-kafka-source#example-topic
```
4.15.2.2.1. Knative CLI --sink flag

When you create an event-producing custom resource by using the Knative (kn) CLI, you can specify a sink where events are sent to from that resource, by using the --sink flag.

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

Example command using the --sink flag

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

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

4.15.2.3. Creating a Kafka event source by using YAML

You can create a Kafka event source by using YAML.

Prerequisites

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

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

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

- You have installed the oc CLI.

Procedure

1. Create a KafkaSource object as a YAML file:

   ```yaml
   apiVersion: sources.knative.dev/v1beta1
   kind: KafkaSource
   metadata:
     name: <source_name>
   spec:
   ```
A consumer group is a group of consumers that use the same group ID, and consume data from a topic.

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

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

**IMPORTANT**

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

**Example KafkaSource object**

```yaml
consumerGroup: <group_name>  
bootstrapServers:  
- <list_of_bootstrap_servers>  
topics:  
- <list_of_topics>  
sink:  
- <list_of_sinks>  

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

2. Apply the KafkaSource YAML file:

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

**Verification**

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

```
$ oc get pods
```

**Example output**
4.15.3. Kafka broker

**IMPORTANT**

Kafka broker 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/.

If a cluster administrator has configured your OpenShift Serverless deployment to use Kafka broker as the default broker type, creating a broker by using the default settings creates a Kafka-based Broker object. If your OpenShift Serverless deployment is not configured to use Kafka broker as the default broker type, you can still use the following procedure to create a Kafka-based broker.

4.15.3.1. Creating a Kafka broker by using YAML

You can create a Kafka broker by using YAML files.

**Prerequisites**

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

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

- You have installed the `oc` CLI.

**Procedure**

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

   ```yaml
   apiVersion: eventing.knative.dev/v1
   kind: Broker
   metadata:
   annotations:
     eventing.knative.dev/broker.class: Kafka
   name: example-kafka-broker
   spec:
   config:
     apiVersion: v1
     kind: ConfigMap
     name: kafka-broker-config
     namespace: knative-eventing
   ```
The broker class. If not specified, brokers use the default class as configured by cluster administrators. To use the Kafka broker, this value must be **Kafka**.

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

2. Apply the Kafka-based broker YAML file:

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

### 4.15.4. Creating a Kafka channel by using YAML

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

**Prerequisites**

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

**Procedure**

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

```
apiVersion: messaging.knative.dev/v1beta1
kind: KafkaChannel
metadata:
  name: example-channel
  namespace: default
spec:
  numPartitions: 3
  replicationFactor: 1
```

**IMPORTANT**

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

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

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

### 4.15.5. Additional resources

- See the [Red Hat AMQ Streams](https://redhat.github.io/amq-streams/docs/) documentation for more information about Kafka concepts.
- See the Red Hat AMQ Streams TLS and SASL on Kafka documentation.

- See the Event delivery documentation for information about configuring event delivery parameters.

- See the Knative Kafka cluster administrator documentation for information about installing Kafka components and setting default configurations if you have cluster administrator permissions.
CHAPTER 5. ADMINISTER

5.1. CONFIGURING OPENSSHIFT SERVERLESS

The OpenShift Serverless Operator manages the global configuration of a Knative installation, including propagating values from the **KnativeServing** and **KnativeEventing** custom resources to system **config maps**.

Any updates to config maps which are applied manually are overwritten by the Operator. However, modifying the Knative custom resources allows you to set values for these config maps.

Knative has multiple config maps that are named with the prefix **config-**.

All Knative config maps are created in the same namespace as the custom resource that they apply to. For example, if the **KnativeServing** custom resource is created in the **knative-serving** namespace, all Knative Serving config maps are also created in this namespace.

The **spec.config** in the Knative custom resources have one `<name>` entry for each config map, named `config-<name>`, with a value which is be used for the config map **data**.

**Examples of global configuration**

You can specify that the **KnativeServing** custom resource uses the **config-domain** config map as follows:

```yaml
apiVersion: operator.knative.dev/v1alpha1
kind: KnativeServing
metadata:
  name: knative-serving
  namespace: knative-serving
spec:
  config:
    domain: 1
    example.org: |
    selector:
      app: prod
    example.com: ""
```

1 Specifies the **config-domain** config map.

You can apply values to multiple config maps. This example sets **stable-window** to 60s in the **config-autoscaler** config map, as well as specifying the **config-domain** config map:

```yaml
apiVersion: operator.knative.dev/v1alpha1
kind: KnativeServing
metadata:
  name: knative-serving
  namespace: knative-serving
spec:
  config:
    domain: 1
    example.org: |
    selector:
      app: prod
```

1
5.2. CONFIGURING KNATIVE EVENTING DEFAULTS

If you have cluster administrator permissions, you can set default options for Knative Eventing components, either for the whole cluster or for a specific namespace.

5.2.1. Configuring the default channel implementation

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

You can make changes to the `knative-eventing` namespace config maps, including the `default-ch-webhook` config map, by using the OpenShift Serverless Operator to propagate changes. To do this, you must modify the `KnativeEventing` custom resource.

Prerequisites

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

Procedure

- Modify the `KnativeEventing` custom resource to add configuration details for the `default-ch-webhook` config map:

```yaml
apiVersion: operator.knative.dev/v1alpha1
kind: KnativeEventing
metadata:
  name: knative-eventing
  namespace: knative-eventing
spec:
  config:
    default-ch-webhook: 2
    default-ch-config:
      clusterDefault:
        apiVersion: messaging.knative.dev/v1
        kind: InMemoryChannel
        spec:
          delivery:
            backoffDelay: PT0.5S
            backoffPolicy: exponential
            retry: 5
      namespaceDefaults:
        my-namespace:
          apiVersion: messaging.knative.dev/v1beta1
```

1. Specifies the `config-domain` config map.
2. Specifies the `stable-window` setting in the `config-autoscaler` config map.
In `spec.config`, you can specify the config maps that you want to add modified configurations for.

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

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

The namespace-scoped default channel type configuration. In this example, the default channel implementation for the `my-namespace` namespace is `KafkaChannel`.

**IMPORTANT**

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

# 5.3. KNATIVE KAFKA

In addition to the Knative Eventing components that are provided as part of a core OpenShift Serverless installation, cluster administrators can install the `KnativeKafka` custom resource (CR).

**NOTE**

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

The `KnativeKafka` CR provides users with additional options, such as:

- Kafka source
- Kafka channel
- Kafka broker

**IMPORTANT**

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

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

## 5.3.1. Installing Knative Kafka

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

apiVersion: operator.serverless.openshift.io/v1alpha1
kind: KnativeKafka
metadata:
  name: knative-kafka
  namespace: knative-eventing
spec:
  channel:
    enabled: true  
    bootstrapServers: <bootstrap_servers>
  source:
    enabled: true  
  broker:
    enabled: true  
  defaultConfig:
    bootstrapServers: <bootstrap_servers>
    numPartitions: <num_partitions>
    replicationFactor: <replication_factor>

1. Enables developers to use the KafkaChannel channel type in the cluster.
2. A comma-separated list of bootstrap servers from your AMQ Streams cluster.
3. Enables developers to use the KafkaSource event source type in the cluster.
4. Enables developers to use the Knative Kafka broker implementation in the cluster.
5. A comma-separated list of bootstrap servers from your Red Hat AMQ Streams cluster.
6. Defines the number of partitions of the Kafka topics, backed by the Broker objects. The default is 10.
7. Defines the replication factor of the Kafka topics, backed by the Broker objects. The default is 3.

**NOTE**

The replicationFactor value must be less than or equal to the number of nodes of your Red Hat AMQ Streams cluster.

**Prerequisites**

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

**Procedure**

1. In the Administrator perspective, navigate to Operators → Installed Operators.
2. Check that the Project dropdown at the top of the page is set to Project: knative-eventing.

3. In the list of Provided APIs for the OpenShift Serverless Operator, find the Knative Kafka box and click Create Instance.

4. Configure the Knative Kafka object in the Create Knative Kafka page.

   IMPORTANT
   
   To use the Kafka channel, source, or broker on your cluster, you must toggle the enabled switch for the options you want to use to true. These switches are set to false by default. Additionally, to use the Kafka channel or broker, you must specify the bootstrap servers.

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

   b. Editing the YAML is recommended for more complex configurations that require full control of Knative Kafka object creation. You can access the YAML by clicking the Edit YAML link in the top right of the Create Knative Kafka page.

5. Click Create after you have completed any of the optional configurations for Kafka. You are automatically directed to the Knative Kafka tab where knative-kafka is in the list of resources.

Verification

1. Click on the knative-kafka resource in the Knative Kafka tab. You are automatically directed to the Knative Kafka Overview page.

2. View the list of Conditions for the resource and confirm that they have a status of True.
If the conditions have a status of Unknown or False, wait a few moments to refresh the page.

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

   $ oc get pods -n knative-eventing

   **Example output**

<table>
<thead>
<tr>
<th>NAME</th>
<th>READY</th>
<th>STATUS</th>
<th>RESTARTS</th>
<th>AGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>kafka-ch-controller-85f879d577-xcbjh</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>44s</td>
</tr>
<tr>
<td>kafka-ch-dispatcher-55d76d7db8-ggqjl</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>44s</td>
</tr>
<tr>
<td>kafka-controller-manager-bc994c465-pl7qd</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>40s</td>
</tr>
<tr>
<td>kafka-webhook-54646f474f-wr7bb</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>42s</td>
</tr>
</tbody>
</table>

5.3.2. Configuring default settings for Kafka components

If you have cluster administrator permissions, you can set default options for Knative Kafka components, either for the whole cluster or for a specific namespace.

5.3.2.1. Configuring TLS authentication for Kafka brokers

As a cluster administrator, you can set up Transport Layer Security (TLS) authentication for Kafka brokers by modifying the **KnativeKafka** custom resource (CR).

**Prerequisites**
You have cluster administrator permissions on OpenShift Container Platform.

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

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

- You have a Kafka cluster CA certificate stored as a `.pem` file.

- You have a Kafka cluster client certificate and a key stored as `.pem` files.

### Procedure

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

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

   **IMPORTANT**

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

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

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

5.3.2.2. Configuring SASL authentication for Kafka brokers

As a cluster administrator, you can set up *Simple Authentication and Security Layer* (SASL) authentication for Kafka brokers by modifying the **KnativeKafka** custom resource (CR).

### Prerequisites

- You have cluster administrator permissions on OpenShift Container Platform.

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

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

You have chosen the SASL mechanism to use, for example PLAIN, SCRAM-SHA-256, or SCRAM-SHA-512.

If TLS is enabled, you also need the ca.crt certificate file for the Kafka cluster.

**NOTE**

It is recommended to enable TLS in addition to SASL.

**Procedure**

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

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

   Use the key names ca.crt, password, and sasl.mechanism. Do not change them.

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

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

5.3.3. Additional resources

- See the Red Hat AMQ Streams documentation for more information about Kafka concepts.

**5.4. CREATING EVENTING COMPONENTS IN THE ADMINISTRATOR PERSPECTIVE**

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

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

**Prerequisites**

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

**Procedure**

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

See [Event sources](#) for more information about which event source types are supported and can be created by using OpenShift Serverless.

### 5.4.2. Creating a broker by using the Administrator perspective

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

**Prerequisites**

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

**Procedure**

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

### 5.4.3. Creating a trigger by using the Administrator perspective

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

**Prerequisites**

- The OpenShift Serverless Operator and Knative Eventing are installed on your OpenShift Container Platform cluster.
• You have cluster administrator permissions for OpenShift Container Platform.
• You have created a broker.
• You have created a Knative service to use as a subscriber.

Procedure

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

2. In the **Broker** tab, select the Options menu for the broker that you want to add a trigger to.

3. Click **Add Trigger** in the list.

4. In the **Add Trigger** dialogue box, select a **Subscriber** for the trigger. The subscriber is the Knative service that will receive events from the broker.

5. Click **Add**.

5.4.4. Creating a channel by using the Administrator perspective

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

Prerequisites

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

• You have cluster administrator permissions for OpenShift Container Platform.

Procedure

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

2. In the **Create** list, select **Channel**. You will be directed to the **Channel** page.

3. Select the type of **Channel** object that you want to create from the **Type** drop-down.

   **NOTE**

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

4. Click **Create**.

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

Prerequisites

- The OpenShift Serverless Operator and Knative Eventing are installed on your OpenShift Container Platform cluster.
- You have cluster administrator permissions for OpenShift Container Platform.
- You have created a channel.
- You have created a Knative service to use as a subscriber.

Procedure

1. In the Administrator perspective of the OpenShift Container Platform web console, navigate to Serverless → Eventing.

2. In the Channel tab, select the Options menu for the channel that you want to add a subscription to.

3. Click Add Subscription in the list.

4. In the Add Subscription dialogue box, select a Subscriber for the subscription. The subscriber is the Knative service that will receive events from the channel.

5. Click Add.

5.4.6. Additional resources

- Brokers
- Subscriptions
- Triggers
- Channels

5.5. CREATING KNATIVE SERVING COMPONENTS IN THE ADMINISTRATOR PERSPECTIVE

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

5.5.1. Creating serverless applications using the Administrator perspective

Prerequisites

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

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

Procedure

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

5.6. CONFIGURING THE KNATIVE SERVING CUSTOM RESOURCE

This guide describes how cluster administrators can manage settings for developer-created custom resources (CRs) that are created from the Knative Serving CR.

5.6.1. Overriding system deployment configurations

You can override the default configurations for some specific deployments by modifying the deployments spec in the KnativeServing custom resource (CR).

Currently, overriding default configuration settings for the replicas, labels, annotations, and nodeSelector fields are supported.

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

- The deployment has 3 replicas.
- The label is set to example-label: label.
- The label example-label: label 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

```yaml
apiVersion: operator.knative.dev/v1alpha1
kind: KnativeServing
metadata:
  name: ks
  namespace: knative-serving
spec:
  high-availability:
    replicas: 2
  deployments:
    - name: webhook
      replicas: 3
      labels: 
```

150
5.6.2. Configuring the EmptyDir extension

This extension controls whether `emptyDir` volumes can be specified.

To enable using `emptyDir` volumes, you must modify the KnativeServing custom resource (CR) to include the following YAML:

```yaml
... 
  spec: 
    config: 
      features: 
        "kubernetes.podspec-volumes-emptydir": enabled
... 
```

5.6.3. HTTPS redirection global settings

You can enable HTTPS redirection for all services on the cluster by configuring the `httpProtocol` spec for the KnativeServing custom resource, as shown in the following example:

```yaml
apiVersion: operator.knative.dev/v1alpha1
kind: KnativeServing
metadata: 
  name: knative-serving 
spec: 
  config: 
    network: 
      httpProtocol: "redirected"
... 
```

5.6.4. Additional resources

- See Managing resources from custom resource definitions.

5.7. AUTOSCALING

As a cluster administrator, you can set global and per-namespace default configurations for autoscaling features by modifying the KnativeServing custom resource (CR). This propagates changes to the relevant config maps.

5.7.1. Enabling scale-to-zero

Cluster administrators can enable or disable scale-to-zero globally for the cluster.

Prerequisites

- You have installed OpenShift Serverless Operator and Knative Serving on your cluster.
• You have cluster administrator permissions.

• 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` CR:

```
apiVersion: operator.knative.dev/v1alpha1
kind: KnativeServing
metadata:
  name: knative-serving
spec:
  config:
    autoscaler:
      enable-scale-to-zero: "false"
```

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".

5.7.2. Configuring the scale-to-zero grace period

This setting specifies an upper bound time limit that Knative waits for scale-from-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.

• 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` CR:

```
apiVersion: operator.knative.dev/v1alpha1
kind: KnativeServing
metadata:
  name: knative-serving
spec:
  config:
    autoscaler:
      scale-to-zero-grace-period: "30s"
```

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

5.8. INTEGRATING SERVICE MESH WITH OPENSIFT SERVERLESS
Using Service Mesh with OpenShift Serverless enables developers to configure additional networking and routing options.

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

**IMPORTANT**

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

### 5.8.1. Integrating Service Mesh with OpenShift Serverless natively

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

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

To complete and verify these procedures in your deployment, you need either a certificate signed by a widely trusted public CA or a CA provided by your organization. Example commands must be adjusted according to your domain, subdomain, and CA.

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

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

#### 5.8.1.1. Creating a certificate to encrypt incoming external traffic

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

**Prerequisites**

- You have access to an OpenShift Container Platform account with cluster administrator access.
- You have installed the OpenShift Serverless Operator and Knative Serving.
- You have installed the `oc` 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

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

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

2. Create a wildcard certificate:

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

3. Sign the wildcard certificate:

   ```bash
   $ openssl x509 -req -days 365 -set_serial 0
     -CA root.crt
     -CAkey root.key
     -in wildcard.csr
     -out wildcard.crt
   ```

4. Create a secret by using the wildcard certificate:

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

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

5.8.1.2. Integrating Service Mesh with OpenShift Serverless

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

Prerequisites

- You have installed Red Hat OpenShift Service Mesh. OpenShift Serverless with Service Mesh only is supported for use with Red Hat OpenShift Service Mesh version 2.0.5 or higher.
- You have access to an OpenShift Container Platform account with cluster administrator access.
- You have installed the OpenShift Serverless Operator.

IMPORTANT

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

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

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

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

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

   IMPORTANT

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

3. Apply the ServiceMeshMemberRoll resource:

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

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

   Example knative-local-gateway object using HTTP

   ```yaml
   apiVersion: networking.istio.io/v1alpha3
   kind: Gateway
   metadata:
     name: knative-ingress-gateway
     namespace: knative-serving
   spec:
     selector:
       istio: ingressgateway
     servers:
       - port:
           number: 443
           name: https
           protocol: HTTPS
           hosts:
             - "*"
           tls:
   ```
Add the name of your wildcard certificate.

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

Example knative-local-gateway object using HTTPS

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

apiVersion: v1
kind: Service
metadata:
  name: knative-local-gateway
  namespace: istio-system
labels:
  experimental.istio.io/disable-gateway-port-translation: "true"
spec:
  type: ClusterIP
  selector:
    istio: ingressgateway
  ports:
    - name: http2
      port: 80
      targetPort: 8081
```
5. Apply the **Gateway** resources:

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

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

```yaml
apiVersion: operator.knative.dev/v1alpha1
kind: KnativeServing
metadata:
  name: knative-serving
  namespace: knative-serving
spec:
  ingress:
    istio:
      enabled: true
  deployments:
  - name: activator
    annotations:
      "sidecar.istio.io/inject": "true"
      "sidecar.istio.io/rewriteAppHTTPProbers": "true"
  - name: autoscaler
    annotations:
      "sidecar.istio.io/inject": "true"
      "sidecar.istio.io/rewriteAppHTTPProbers": "true"
```

1. Enables Istio integration.
2. Enables sidecar injection for Knative Serving data plane pods.

7. Apply the **KnativeServing** resource:

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

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

```yaml
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: <service_name>
  namespace: <namespace>  # 1
annotations:
  serving.knative.openshift.io/enablePassthrough: "true"  # 2
```

1. Enables sidecar injection for the Knative data plane pods.
2. Enables pass-through route.
A namespace that is part of the Service Mesh member roll.

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

Injects Service Mesh sidecars into the Knative service pods.

9. Apply the **Service** resource:

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

**Verification**

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

```bash
$ curl --cacert root.crt <service_url>
```

**Example command**

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

**Example output**

Hello Openshift!

5.8.1.3. Enabling Knative Serving metrics when using Service Mesh with mTLS

If Service Mesh is enabled with mTLS, metrics for Knative Serving are disabled by default, because Service Mesh prevents Prometheus from scraping metrics. This section shows how to enable Knative Serving metrics when using Service Mesh and mTLS.

**Prerequisites**

- You have installed the OpenShift Serverless Operator and Knative Serving on your cluster.
- You have installed Red Hat OpenShift Service Mesh with the mTLS functionality enabled.
- You have access to an OpenShift Container Platform account with cluster administrator access.
- You have installed the `oc` 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

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

   ```yaml
   apiVersion: operator.knative.dev/v1beta1
   kind: KnativeServing
   metadata:
     name: knative-serving
   spec:
     config:
       observability:
         metrics.backend-destination: "prometheus"
   ...
   ``

   This step prevents metrics from being disabled by default.

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

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

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

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

5.8.2. Integrating Service Mesh with OpenShift Serverless when Kourier is enabled

Prerequisites

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

You have installed the OpenShift Serverless Operator and Knative Serving on your cluster.

You have installed Red Hat OpenShift Service Mesh. OpenShift Serverless with Service Mesh and Kourier is supported for use with both Red Hat OpenShift Service Mesh versions 1.x and 2.x.

**Procedure**

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

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

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

2. Apply the `ServiceMeshMemberRoll` resource:

   `$ oc apply -f <filename>`

3. Create a network policy that permits traffic flow from Knative system pods to Knative services:

   a. Add the `serving.knative.openshift.io/system-namespace=true` label to the `knative-serving` namespace:

      ```bash
      $ oc label namespace knative-serving serving.knative.openshift.io/system-namespace=true
      ```

   b. Add the `serving.knative.openshift.io/system-namespace=true` label to the `knative-serving-ingress` namespace:

      ```bash
      $ oc label namespace knative-serving-ingress serving.knative.openshift.io/system-namespace=true
      ```

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

      ```yaml
      apiVersion: networking.k8s.io/v1
      kind: NetworkPolicy
      metadata:
        name: allow-from-serving-system-namespace
      namespace: <namespace> 1
      ```

      1 A list of namespaces to be integrated with Service Mesh.
Add the namespace that you want to integrate with Service Mesh.

Apply the NetworkPolicy resource:

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

### 5.9. MONITORING SERVERLESS COMPONENTS

You can use OpenShift Container Platform monitoring dashboards to view health checks and metrics for OpenShift Serverless components.

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

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

**Prerequisites**

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

**Procedure**

1. In the Administrator perspective, navigate to Observe → Dashboards.

2. Select the Knative Health Status dashboard in the Dashboard drop-down to view the overall health status of Knative. If your Knative deployment is running as expected, the dashboard shows a status of Ready.
If you have Knative Serving or Knative Eventing installed, you can also scroll down to see the health status for each of these components.

5.9.2. Monitoring Knative Serving revision CPU and memory usage

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

Prerequisites

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

Procedure

1. In the Administrator perspective, navigate to Observe → Dashboards.
2. Select the Knative Serving - Source CPU and Memory Usage dashboard in the Dashboard drop-down list to view the following metrics:
   - Total CPU Usage (rate per minute)
   - Total Memory Usage (bytes)
   - Total Network I/O (rate per minute)
   - Total Network Errors (rate per minute)
3. Optional: You can filter this dashboard by Namespace, Configuration, or Revision, by selecting an option from the drop-down list.

5.9.3. Monitoring Knative Eventing source CPU and memory usage

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

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

You have access to the Administrator perspective in the OpenShift Container Platform web console.

You have installed the OpenShift Serverless Operator, as well as the Knative Eventing component.

The OpenShift Container Platform monitoring stack is enabled on your cluster. You can enable monitoring for OpenShift Serverless during installation by checking the box to **Enable operator recommended cluster monitoring on this namespace** when installing the OpenShift Serverless Operator.

### Procedure

1. In the Administrator perspective, navigate to Observe → Dashboards.

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

### 5.9.4. Monitoring event sources

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

### Prerequisites

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

- You have access to the Administrator perspective in the OpenShift Container Platform web console.

- You have installed the OpenShift Serverless Operator, as well as the Knative Eventing component.

- The OpenShift Container Platform monitoring stack is enabled on your cluster. You can enable monitoring for OpenShift Serverless during installation by checking the box to **Enable operator recommended cluster monitoring on this namespace** when installing the OpenShift Serverless Operator.

### Procedure

1. In the Administrator perspective, navigate to Observe → Dashboards.

2. Select the Knative Eventing - Sources dashboard in the Dashboard drop-down list.

3. You can now view the following metrics:
a. For API server sources:
   - Event Count (rate per minute)
   - Success Rate (2xx Event, fraction rate per minute)
   - Event Count by Response Code Class (rate per minute)
   - Failure Rate (non-2xx Event, fraction rate per minute)

b. For ping sources:
   - Event Count (rate per minute)
   - Success Rate (2xx Event, fraction rate per minute)
   - Event Count by Response Code Class (rate per minute)
   - Failure Rate (non-2xx Event, fraction rate per minute)

c. For Kafka sources:
   - Event Count (rate per minute)
   - Success Rate (2xx Event, fraction rate per minute)
   - Event Count by Response Code Class (rate per minute)
   - Failure Rate (non-2xx Event, fraction rate per minute)

5.9.5. Monitoring Knative Eventing brokers and triggers

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

Prerequisites

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

Procedure

1. In the Administrator perspective, navigate to Observe → Dashboards.

2. Select the Knative Eventing - Broker/Trigger dashboard in the Dashboard drop-down list.

3. You can now view the following metrics:
a. For brokers:
   - Event Count (avg/sec, over 1m window)
   - Success Rate (2xx Event, fraction rate, over 1m window)
   - Event Count by Event Type (avg/sec, over 1m window)
   - Event Count by Response Code Class (avg/sec, over 1m window)
   - Failure Rate (non-2xx Event, fraction rate, over 1m window)
   - Event Dispatch Latency (ms)

b. For triggers:
   - Event Count (avg/sec, over 1m window)
   - Success Rate (2xx Event, fraction rate, over 1m window)
   - Event Count by Event Type (avg/sec, over 1m window)
   - Event Count by Response Code Class (avg/sec, over 1m window)
   - Failure Rate (non-2xx Event, fraction rate, over 1m window)
   - Event Dispatch Latency (ms)

   **5.9.6. Monitoring Knative Eventing channels**

You can use the OpenShift Container Platform monitoring dashboards to view metrics for channels in your cluster.

**Prerequisites**

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

**Procedure**

1. In the **Administrator** perspective, navigate to **Observe → Dashboards**.
2. Select the **Knative Eventing - Channel** dashboard in the **Dashboard** drop-down list.
3. You can now view the following metrics:
a. For in-memory channels:
   - Event Count (avg/sec, over 1m window)
   - Success Rate (2xx Event, fraction rate, over 1m window)
   - Event Count by Response Code Class (avg/sec, over 1m window)
   - Failure Rate (non-2xx Event, fraction rate, over 1m window)
   - Event Dispatch Latency (ms)

b. For Kafka channels:
   - Event Count (avg/sec, over 1m window)
   - Success Rate (2xx Event, fraction rate, over 1m window)
   - Event Count by Response Code Class (avg/sec, over 1m window)
   - Failure Rate (non-2xx Event, fraction rate, over 1m window)
   - Event Dispatch Latency (ms)

5.10. METRICS

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

5.10.1. Prerequisites

- See the OpenShift Container Platform documentation on Managing metrics for information about enabling metrics for your cluster.
- You have access to an OpenShift Container Platform account with cluster administrator access.
- You have access to the Administrator perspective in the OpenShift Container Platform web console.

**WARNING**

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

For information about resolving this issue, see Integrating Service Mesh with OpenShift Serverless.

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

<table>
<thead>
<tr>
<th>Metric name</th>
<th>Description</th>
<th>Type</th>
<th>Tags</th>
<th>Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>work_queue_depth</td>
<td>The depth of the work queue.</td>
<td>Gauge</td>
<td>reconciler</td>
<td>Integer (no units)</td>
</tr>
<tr>
<td>reconcile_count</td>
<td>The number of reconcile operations.</td>
<td>Counter</td>
<td>reconciler, success</td>
<td>Integer (no units)</td>
</tr>
<tr>
<td>reconcile_latency</td>
<td>The latency of reconcile operations.</td>
<td>Histogram</td>
<td>reconciler, success</td>
<td>Milliseconds</td>
</tr>
<tr>
<td>workqueue_adds_total</td>
<td>The total number of add actions handled by the work queue.</td>
<td>Counter</td>
<td>name</td>
<td>Integer (no units)</td>
</tr>
<tr>
<td>workqueue_queue_latency_seconds</td>
<td>The length of time an item stays in the work queue before being requested.</td>
<td>Histogram</td>
<td>name</td>
<td>Seconds</td>
</tr>
<tr>
<td>workqueue_retries_total</td>
<td>The total number of retries that have been handled by the work queue.</td>
<td>Counter</td>
<td>name</td>
<td>Integer (no units)</td>
</tr>
<tr>
<td>workqueue_work_duration_seconds</td>
<td>The length of time it takes to process and item from the work queue.</td>
<td>Histogram</td>
<td>name</td>
<td>Seconds</td>
</tr>
<tr>
<td>workqueue_unfinished_work_seconds</td>
<td>The length of time that outstanding work queue items have been in progress.</td>
<td>Histogram</td>
<td>name</td>
<td>Seconds</td>
</tr>
<tr>
<td>workqueue_longest_running_processor_seconds</td>
<td>The length of time that the longest outstanding work queue items has been in progress.</td>
<td>Histogram</td>
<td>name</td>
<td>Seconds</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Metric name</th>
<th>Description</th>
<th>Type</th>
<th>Tags</th>
<th>Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>request_count</td>
<td>The number of requests that are routed to the webhook.</td>
<td>Counter</td>
<td>admission_allowed, kind_group, kind_kind, kind_version, request_operation, resource_group, resource_name, resource_resource, resource_version</td>
<td>Integer (no units)</td>
</tr>
<tr>
<td>request_latencies</td>
<td>The response time for a webhook request.</td>
<td>Histogram</td>
<td>admission_allowed, kind_group, kind_kind, kind_version, request_operation, resource_group, resource_name, resource_resource, resource_version</td>
<td>Milliseconds</td>
</tr>
</tbody>
</table>

5.10.4. Knative Eventing metrics

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

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

5.10.4.1. Broker ingress metrics

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

You can use the following metrics to debug broker filters, see how they are performing, and see which events are being dispatched by the filters. You can also measure the latency of the filtering action on an event.

<table>
<thead>
<tr>
<th>Metric name</th>
<th>Description</th>
<th>Type</th>
<th>Tags</th>
<th>Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>event_count</td>
<td>Number of events received by a broker.</td>
<td>Counter</td>
<td>broker_name, event_type, namespace_name, response_code, response_code_class, unique_name</td>
<td>Integer (no units)</td>
</tr>
<tr>
<td>event_dispatch_latencies</td>
<td>The time taken to dispatch an event to a channel.</td>
<td>Histogram</td>
<td>broker_name, event_type, namespace_name, response_code, response_code_class, unique_name</td>
<td>Milliseconds</td>
</tr>
</tbody>
</table>
5.10.4.3. InMemoryChannel dispatcher metrics

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

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

5.10.4.4. Event source metrics

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

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

#### 5.10.5.1. Activator metrics

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

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

<table>
<thead>
<tr>
<th>Metric name</th>
<th>Description</th>
<th>Type</th>
<th>Tags</th>
<th>Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>request_concurrency</td>
<td>The number of concurrent requests that are routed to the activator, or average concurrency over a reporting period.</td>
<td>Gauge</td>
<td>configuration_name, container_name, namespace_name, pod_name, revision_name, service_name</td>
<td>Integer (no units)</td>
</tr>
<tr>
<td>Metric name</td>
<td>Description</td>
<td>Type</td>
<td>Tags</td>
<td>Unit</td>
</tr>
<tr>
<td>----------------</td>
<td>------------------------------------------------------------------------------</td>
<td>----------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>request_count</td>
<td>The number of requests that are routed to activator. These are requests that have been fulfilled from the activator handler.</td>
<td>Counter</td>
<td>configuration_name, container_name, namespace_name, pod_name, response_code, response_code_class, revision_name, service_name</td>
<td>Integer (no units)</td>
</tr>
<tr>
<td>request_latencies</td>
<td>The response time in milliseconds for a fulfilled, routed request.</td>
<td>Histogram</td>
<td>configuration_name, container_name, namespace_name, pod_name, response_code, response_code_class, revision_name, service_name</td>
<td>Milliseconds</td>
</tr>
</tbody>
</table>

5.10.5.2. Autoscaler metrics

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

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

### 5.10.5.3. Go runtime metrics

Each Knative Serving control plane process emits a number of Go runtime memory statistics ([MemStats](https://golang.org/)).
The name tag for each metric is an empty tag.

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

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

HA in OpenShift Serverless is available through leader election, which is enabled by default after the Knative Serving or Eventing control plane is installed.

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

5.11.1. Configuring high availability replicas on OpenShift Serverless

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

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

These components are configured with two replicas by default.

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

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

5.11.1.1. Configuring high availability replicas for Knative Serving

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

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

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**

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

---

### CHAPTER 5. ADMINISTER

179
1. Sets the number of replicas to 3.
   - The `replicas` value sets the replica count for all HA controllers.
   - The default `replicas` value is 2.
   - You can increase the number of replicas by changing the value to 3 or more.

5.11.1.2. Configuring high availability replicas for Knative Eventing

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

**Prerequisites**

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

**Procedure**

1. In the OpenShift Container Platform web console Administrator perspective, navigate to OperatorHub → Installed Operators.
2. Select the `knative-eventing` namespace.
3. Click Knative Eventing in the list of Provided APIs for the OpenShift Serverless Operator to go to the Knative Eventing tab.
4. Click `knative-eventing`, then go to the YAML tab in the `knative-eventing` page.
5. Modify the number of replicas in the **KnativeEventing** CR:

**Example YAML**

```yaml
apiversion: operator.knative.dev/v1alpha1
kind: KnativeEventing
metadata:
  name: knative-eventing
  namespace: knative-eventing
spec:
  high-availability:
    replicas: 3
```

1. Sets the number of replicas to **3**.
   - The `replicas` value sets the replica count for all HA controllers.
   - The default `replicas` value is **2**.
   - You can increase the number of replicas by changing the value to **3** or more.

### 5.11.1.3. Configuring high availability replicas for Knative Kafka

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

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

The OpenShift Serverless Operator and Knative Kafka are installed on your cluster.

**Procedure**

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

2. Select the `knative-eventing` namespace.

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

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

5. Modify the number of replicas in the KnativeKafka CR:

**Example YAML**

```yaml
apiVersion: operator.serverless.openshift.io/v1alpha1
kind: KnativeKafka
metadata:
  name: knative-kafka
  namespace: knative-eventing
spec:
  high-availability:
    replicas: 3
```

   
1. Sets the number of replicas to 3.
- The `replicas` value sets the replica count for all HA controllers.
- The default `replicas` value is 2.
- You can increase the number of replicas by changing the value to 3 or more.
CHAPTER 6. MONITOR

6.1. USING OPENShift LOGGING WITH OPENShift SERVERLESS

6.1.1. About deploying OpenShift Logging

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

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

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

6.1.2. About deploying and configuring OpenShift Logging

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

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

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

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

6.1.2.1. Configuring and Tuning OpenShift Logging

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

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

Memory and CPU

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

```yaml
spec:
  logStore:
    elasticsearch:
      resources:
        limits:
          cpu: <value>
          memory: <value>
```

Elasticsearch storage

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

```
requests:
  cpu: 500m
  memory: 16Gi
type: "elasticsearch"
collection:
logs:
  fluentd:
    resources:
      limits:
        cpu:
        memory:
      requests:
        cpu:
        memory:
      type: "fluentd"
visualization:
kibana:
  resources:
    limits:
    cpu:
    memory:
    requests:
    cpu:
    memory:
type: kibana
```

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

**NOTE**

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

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

185
**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.

### 6.1.2.2. Sample modified ClusterLogging custom resource

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

**Sample modified ClusterLogging custom resource**

```json
apiVersion: "logging.openshift.io/v1"
kind: "ClusterLogging"
metadata:
    name: "instance"
    namespace: "openshift-logging"
spec:
    managementState: "Managed"
    logStore:
        type: "elasticsearch"
        retentionPolicy:
            application:
                maxAge: 1d
            infra:
                maxAge: 7d
            audit:
                maxAge: 7d
            elasticsearch:
                nodeCount: 3
        resources:
            limits:
                memory: 32Gi
            requests:
                cpu: 3
                memory: 32Gi
        storage:
            storageClassName: "gp2"
            size: "200G"
    redundancyPolicy: "SingleRedundancy"
    visualization:
        type: "kibana"
        kibana:
            resources:
                limits:
                    memory: 1Gi
```

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

**Procedure**

1. Get the Kibana route:

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

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

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

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

**NOTE**

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

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

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

**Procedure**

1. Get the Kibana route:

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

2. Use the route’s URL to navigate to the Kibana dashboard and log in.
3. Check that the index is set to .all. If the index is not set to .all, only the OpenShift system logs will be listed.

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

**Example filter**

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

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

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

**NOTE**

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

### 6.2. TRACING REQUESTS USING JAEGER

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

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

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

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

For more information about distributed tracing, see distributed tracing architecture and Installing distributed tracing.

#### 6.2.1. Configuring Jaeger for use with OpenShift Serverless

You can use the following procedure to set up Jaeger for use with OpenShift Serverless.

**Prerequisites**

- You have access to an OpenShift Container Platform account with cluster administrator access.
- You have installed the OpenShift Serverless Operator and Knative Serving.
- You have installed the Jaeger Operator.
- You have installed the `oc` 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

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

**Jaeger CR**

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

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

**Tracing YAML example**

```yaml
apiVersion: operator.knative.dev/v1alpha1
kind: KnativeServing
metadata:
  name: knative-serving
  namespace: knative-serving
spec:
  config:
    tracing:
      sample-rate: "0.1"
      backend: zipkin
      zipkin-endpoint: http://jaeger-collector.default.svc.cluster.local:9411/api/v2/spans
deploy: "false"
```

1. The **sample-rate** defines sampling probability. Using **sample-rate: "0.1"** means that 1 in 10 traces will be sampled.

2. **backend** must be set to **zipkin**.

3. The **zipkin-endpoint** must point to your **jaeger-collector** service endpoint. To get this endpoint, substitute the namespace where the Jaeger CR is applied.

4. Debugging should be set to **false**. Enabling debug mode by setting **debug: "true"** allows all spans to be sent to the server, bypassing sampling.

Verification

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

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

   ```bash
   $ oc get route jaeger
   ```

   **Example output**

<table>
<thead>
<tr>
<th>NAME</th>
<th>HOST/PORT</th>
<th>PATH</th>
<th>SERVICES</th>
<th>PORT</th>
<th>TERMINATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>WILDCARD</td>
<td>jaeger-default.apps.example.com</td>
<td>jaeger-query</td>
<td>&lt;all&gt;</td>
<td>reencrypt</td>
<td>None</td>
</tr>
</tbody>
</table>
2. Open the endpoint address in your browser to view the console.

### 6.3. METRICS

Metrics enable developers to monitor how Knative services are performing.

#### 6.3.1. Prerequisites

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

---

**WARNING**

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

For information about resolving this issue, see [Integrating Service Mesh with OpenShift Serverless](#).

#### 6.3.2. Queue proxy metrics

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

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

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

### 6.4. MONITORING KNATIVE SERVICES

You can use the OpenShift Container Platform monitoring stack to record and view health checks and metrics for your Knative services. This section describes the following topics:
• What metrics Knative services expose by default
• How to configure exposing custom metrics
• How to configure the monitoring stack to scrape the exposed metrics
• How to view the metrics of a service

NOTE
Scraping the metrics does not affect autoscaling of a Knative service, because scraping requests do not go through the activator. Consequently, no scraping takes place if no pods are running.

6.4.1. Knative service metrics exposed by default

Table 6.1. Metrics exposed by default for each Knative service on port 9090

<table>
<thead>
<tr>
<th>Metric name, unit, and type</th>
<th>Description</th>
<th>Metric tags</th>
</tr>
</thead>
</table>
| **queue_requests_per_second**  
Metric unit: dimensionless  
Metric type: gauge | Number of requests per second that hit the queue proxy.  
Formula: \( \frac{\text{stats.RequestCount}}{\text{r.reportingPeriodSeconds}} \)  
\( \text{stats.RequestCount} \) is calculated directly from the networking pkg stats for the given reporting duration. | destination_configuration="event-display", destination_namespace="pingsource1", destination_pod="event-display-00001-deployment-6b455479cb-75p6w", destination_revision="event-display-00001" |
| **queue_proxied_operations_per_second**  
Metric unit: dimensionless  
Metric type: gauge | Number of proxied requests per second.  
Formula: \( \frac{\text{stats.ProxiedRequestCount}}{\text{r.reportingPeriodSeconds}} \)  
\( \text{stats.ProxiedRequestCount} \) is calculated directly from the networking pkg stats for the given reporting duration. | |
<table>
<thead>
<tr>
<th>Metric name, unit, and type</th>
<th>Description</th>
<th>Metric tags</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>queue_average_concurrent_requests</strong></td>
<td>Number of requests currently being handled by this pod.</td>
<td>destination_configuration=&quot;event-display&quot;,</td>
</tr>
<tr>
<td>Metric unit: dimensionless</td>
<td>Average concurrency is calculated at the networking pkg side as follows:</td>
<td>destination_namespace=&quot;pingsource1&quot;, destination_pod=&quot;event-display-00001-deployment-6b455479cb-75p6w&quot;, destination_revision=&quot;event-display-00001&quot;</td>
</tr>
<tr>
<td>Metric type: gauge</td>
<td>- When a req change happens, the time delta between changes is calculated. Based on the result, the current concurrency number over delta is computed and added to the current computed concurrency. Additionally, a sum of the deltas is kept. Current concurrency over delta is computed as follows:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[ \text{global_concurrency} \times \text{delta} ]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- Each time a reporting is done, the sum and current computed concurrency are reset.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- When reporting the average concurrency the current computed concurrency is divided by the sum of deltas.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- When a new request comes in, the global concurrency counter is increased. When a request is completed, the counter is decreased.</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>queue_average_proxied_concurrent_requests</strong></td>
<td>destination_configuration=&quot;event-display&quot;,</td>
</tr>
<tr>
<td>Metric unit: dimensionless</td>
<td>Number of proxied requests currently being handled by this pod:</td>
<td>destination_namespace=&quot;pingsource1&quot;, destination_pod=&quot;event-display-00001-deployment-6b455479cb-75p6w&quot;, destination_revision=&quot;event-display-00001&quot;</td>
</tr>
<tr>
<td>Metric type: gauge</td>
<td><code>stats.AverageProxiedConcurrency</code></td>
<td></td>
</tr>
</tbody>
</table>
### process_uptime

Metric unit: seconds
Metric type: gauge

The number of seconds that the process has been up.

Metric tags:
- destination_configuration="event-display"
- destination_namespace="pingsource1"
- destination_pod="event-display-00001-deployment-6b455479cb-75p6w"
- destination_revision="event-display-00001"

### request_count

Metric unit: dimensionless
Metric type: counter

The number of requests that are routed to `queue-proxy`.

Metric tags:
- configuration_name="event-display"
- container_name="queue-proxy"
- namespace_name="apiserversource1"
- pod_name="event-display-00001-deployment-658fd4f9cf-qcnr5"
- response_code="200"
- response_code_class="2xx"
- revision_name="event-display-00001"
- service_name="event-display"

### request_latencies

Metric unit: milliseconds
Metric type: histogram

The response time in milliseconds.

Metric tags:
- configuration_name="event-display"
- container_name="queue-proxy"
- namespace_name="apiserversource1"
- pod_name="event-display-00001-deployment-658fd4f9cf-qcnr5"
- response_code="200"
- response_code_class="2xx"
- revision_name="event-display-00001"
- service_name="event-display"

### app_request_count

Metric unit: dimensionless
Metric type: counter

The number of requests that are routed to `user-container`.

Metric tags:
- configuration_name="event-display"
- container_name="queue-proxy"
- namespace_name="apiserversource1"
- pod_name="event-display-00001-deployment-658fd4f9cf-qcnr5"
- response_code="200"
- response_code_class="2xx"
- revision_name="2xx"
- service_name="event-display"
### 6.4.2. Knative service with custom application metrics

You can extend the set of metrics exported by a Knative service. The exact implementation depends on your application and the language used.

The following listing implements a sample Go application that exports the count of processed events custom metric.

```go
package main

import (
    "fmt"
    "log"
    "net/http"
    "os"
    "github.com/prometheus/client_golang/prometheus"
    "github.com/prometheus/client_golang/promauto"
    "github.com/prometheus/client_golang/promhttp"
)

var opsProcessed = promauto.NewCounter(prometheus.CounterOpts{
    Name: "myapp_processed_ops_total",
    Help: "The total number of processed events",
})
```
Including the Prometheus packages.

2 Defining the `opsProcessed` metric.

3 Incrementing the `opsProcessed` metric.

4 Configuring to use a separate server for metrics requests.

5 Configuring to use the same port as normal requests for metrics and the `metrics` subpath.

6.4.3. Configuration for scraping custom metrics
Custom metrics scraping is performed by an instance of Prometheus purposed for user workload monitoring. After you enable user workload monitoring and create the application, you need a configuration that defines how the monitoring stack will scrape the metrics.

The following sample configuration defines the `ksvc` for your application and configures the service monitor. The exact configuration depends on your application and how it exports the metrics.

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: helloworld-go
spec:
template:
  metadata:
    labels:
      app: helloworld-go
    annotations:
    spec:
      containers:
        - image: docker.io/skonto/helloworld-go:metrics
          resources:
            requests:
              cpu: "200m"
          env:
            - name: TARGET
              value: "Go Sample v1"
---
apiVersion: monitoring.coreos.com/v1
kind: ServiceMonitor
metadata:
  labels:
  name: helloworld-go-sm
spec:
  endpoints:
    - port: queue-proxy-metrics
      scheme: http
    - port: app-metrics
      scheme: http
  namespaceSelector: {}
  selector:
    matchLabels:
      name: helloworld-go-sm
---
apiVersion: v1
kind: Service
metadata:
  labels:
    name: helloworld-go-sm
spec:
  ports:
    - name: queue-proxy-metrics
      port: 9091
      protocol: TCP
      targetPort: 9091
    - name: app-metrics
```
6.4.4. Examining metrics of a service

After you have configured the application to export the metrics and the monitoring stack to scrape them, you can examine the metrics in the web console.

Prerequisites

- You have logged in to the OpenShift Container Platform web console.
- You have installed the OpenShift Serverless Operator and Knative Serving.

Procedure

1. Optional: Run requests against your application that you will be able to see in the metrics:

   ```bash
   $ hello_route=$(oc get ksvc helloworld-go -n ns1 -o jsonpath='{.status.url}') && \
   curl $hello_route
   ``

   **Example output**

   Hello Go Sample v1!

2. In the web console, navigate to the **Observe → Metrics** interface.

3. In the input field, enter the query for the metric you want to observe, for example:

   ```bash
   revision_app_request_count{namespace="ns1", job="helloworld-go-sm"}
   ```

   Another example:

   ```bash
   myapp_processed_ops_total{namespace="ns1", job="helloworld-go-sm"}
   ```

4. Observe the visualized metrics:
### 6.4.5. Examining metrics of a service in the dashboard

You can examine the metrics using a dedicated dashboard that aggregates queue proxy metrics by namespace.

**Prerequisites**

- You have logged in to the OpenShift Container Platform web console.
- You have installed the OpenShift Serverless Operator and Knative Serving.

**Procedure**

1. In the web console, navigate to the **Observe → Metrics** interface.
2. Select the **Knative User Services (Queue Proxy metrics)** dashboard.
3. Select the **Namespace, Configuration, and Revision** that correspond to your application.
4. Observe the visualized metrics:
6.4.6. Additional resources

- For more information on OpenShift Container Platform monitoring stack, see Monitoring overview.

- For information on monitoring the components of Serverless itself, as opposed to Knative services, see Monitoring serverless components.

- Enabling monitoring for user-defined projects

- Specifying how a service is monitored

6.5. AUTOSCALING DASHBOARD

You can use the Knative Serving - Scaling Debugging dashboard to examine detailed and visualized data for Knative Serving autoscaling. The dashboard is useful for several purposes:

- Troubleshooting your autoscaled workloads

- Improving understanding of how autoscaling works

- Determining why an application was autoscaled

- Evaluating resource footprint of an application, such as number of pods

**IMPORTANT**

Currently, this dashboard only supports the Knative pod autoscaler (KPA). It does not support the horizontal pod autoscaler (HPA).

The dashboard demonstrations in this section use an OpenShift Container Platform cluster with the autoscale-go sample application installed. The load is generated using the hey load generator.

The sample application has a concurrency limit of 5 requests. When the limit is exceeded, autoscaling requests additional pods for Knative from Kubernetes.

**Additional resources**
6.5.1. Navigating to the autoscaling dashboard

You can use the following steps to navigate to the autoscaling dashboard in the OpenShift Container Platform web console.

**Prerequisites**

- You have logged in to the OpenShift Container Platform web console.
- You have installed the OpenShift Serverless Operator and Knative Serving.

**Procedure**

1. In the **Developer** perspective, navigate to the **Monitoring → Dashboards** page.
2. In the **Dashboard** field, select the **Knative Serving - Scaling Debugging** dashboard.
3. Use the **Namespace**, **Configuration**, and **Revision** fields to specify the workload you want to examine.

**Additional resources**

- See detailed information about metrics.

6.5.2. Pod information

The top of the **Knative Serving - Scaling Debugging** dashboard shows the counts of the requested pods, as well as of the pods in various stages of deployment. The **Revision Pod Counts (Timeline)** graph shows the same data visualized on the timeline. This information might be useful for general assessment of autoscaling by checking for problems with pod allocation.

6.5.3. Observed concurrency
The **Observed Concurrency** graph shows the timeline of a set of concurrency-related metrics, including:

- request concurrency
- panic concurrency
- target concurrency
- excess burst capacity

Note that **ExcessBurstCapacity** is a negative number, **-200** by default, that increases when a bursty load appears. It is equal to the difference between spare capacity and the configured target burst capacity. If **ExcessBurstCapacity** is negative, then the activator is threaded in the request path by the **PodAutoscaler** controller.

### 6.5.4. Scrape time

The **Scrape Time** graph shows the timeline of scrape times for each revision. Since autoscaling makes scaling decisions based on the metrics coming from service pods, high scrape times might cause delays in autoscaling when workload changes.

### 6.5.5. Panic mode

The **Panic Mode** graph shows the timeline of times when the Knative service faces a bursty load, which causes the autoscaler to quickly adapt the service pod number.
Additional resources

- See an overview of Knative Pod Autoscaling modes.

6.5.6. Activator metrics

The Activator graphs Request Concurrency, Request Count by Response Code (last minute) and Response Time (last minute) show the timeline of requests going through the activator until the activator is removed from the request path. These graphs can be used, for example, to evaluate whether response count and the returned HTTP codes match expectations.

![Activator: Request Concurrency graph](image)

![Activator: Request Count by Response Code (last minute) graph](image)

![Activator: Response Time (last minute) graph](image)

Additional resources

- See detailed information about activator metrics.

6.5.7. Requests per second

For requests-per-second (RPS) services, an additional Observed RPS dashboard is available, which visualizes different types of requests per second:

- stable_requests_per_second
- panic_requests_per_second
- target_requests_per_second
Note that the **Observed RPS** dashboard is visible only if the requests annotations are set, for example:

```yaml
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: helloworld-go
  namespace: default
spec:
template:
  metadata:
    annotations:
      autoscaling.knative.dev/target: "10"
      autoscaling.knative.dev/metric: "rps"
spec:
  containers:
    - image: docker.io/openshift/hello-openshift
```

OpenShift Container Platform 4.9 Serverless
CHAPTER 7. OPENSHIFT SERVERLESS SUPPORT

7.1. GETTING SUPPORT

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

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

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

7.2. GATHERING DIAGNOSTIC INFORMATION FOR SUPPORT

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

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

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

7.2.1. About the must-gather tool

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

- Resource definitions
- Service logs

By default, the oc adm must-gather command uses the default plug-in image and writes into ./must-gather.local.

Alternatively, you can collect specific information by running the command with the appropriate arguments as described in the following sections:

- To collect data related to one or more specific features, use the --image argument with an image, as listed in a following section.
  For example:
  
  $ oc adm must-gather --image=registry.redhat.io/container-native-virtualization/cnv-must-gather-rhel8:v4.9.0

- To collect the audit logs, use the --/usr/bin/gather_audit_logs argument, as described in a following section.
For example:

```
$ oc adm must-gather -- /usr/bin/gather_audit_logs
```

**NOTE**

Audit logs are not collected as part of the default set of information to reduce the size of the files.

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

For example:

```
    NAMESPACE                      NAME                 READY   STATUS      RESTARTS      AGE
    ...                          ...                          ...                          ...
    openshift-must-gather-5drcj  must-gather-bklx4    2/2     Running     0             72s
    openshift-must-gather-5drcj  must-gather-s8sdh    2/2     Running     0             72s
    ...
```

### 7.2.2. About collecting OpenShift Serverless data

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

**Procedure**

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

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

**Example command**

```
$ oc adm must-gather --image=registry.redhat.io/openshift-serverless-1/svls-must-gather-rhel8:1.14.0
```
8.1. CONFIGURING JSON WEB TOKEN AUTHENTICATION FOR KNATIVE SERVICES

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

8.1.1. Enabling sidecar injection for a Knative service

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

**IMPORTANT**

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

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

Prerequisites

- You have installed the OpenShift Serverless Operator and Knative Serving.
- You have installed the `oc` 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

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

   **Example service**

   ```yaml
   apiVersion: serving.knative.dev/v1
   kind: Service
   metadata:
     name: <service_name>
   spec:
     template:
       metadata:
         annotations:
           sidecar.istio.io/inject: "true"  
           sidecar.istio.io/rewriteAppHTTPProbers: "true" 
   ...
   
   1 Add the `sidecar.istio.io/inject="true"` annotation.
   
   2 You must set the annotation `sidecar.istio.io/rewriteAppHTTPProbers: "true"` in your Knative service as OpenShift Serverless versions 1.14.0 and higher use an HTTP probe as the readiness probe for Knative services by default.
   ```
2. Apply your `Service` resource YAML file:

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

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

You can use the following procedure to enable using JSON Web Token authentication with Service Mesh 2.x and OpenShift Serverless.

#### Prerequisites

- You have installed the OpenShift Serverless Operator and Knative Serving.
- You have installed the `oc` 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

1. 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
```

2. Apply the `RequestAuthentication` resource:

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

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

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

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

4. Apply the `AuthorizationPolicy` resource:

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

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

```
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: require-jwt
  namespace: <namespace>
spec:
  action: ALLOW
  rules:
  - from:
    - source:
      requestPrincipals:
      - "testing@secure.istio.io/testing@secure.istio.io"
```

6. Apply the `AuthorizationPolicy` resource:

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

**Verification**

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

   **Example command**

   ```
   $ curl http://hello-example-1-default.apps.mycluster.example.com/
   ```

   **Example output**

   ```
   RBAC: access denied
   ```

2. Verify the request with a valid JWT.
   a. Get the valid JWT token:

   ```
   $ TOKEN=$(curl https://raw.githubusercontent.com/istio/istio/release-1.8/security/tools/jwt/samples/demo.jwt -s) && echo "$TOKEN" | cut -d '.' -f2 - | base64 --decode
   ```
   b. Access the service by using the valid token in the `curl` request header:

The request is now allowed:

Example output

Hello OpenShift!

8.1.3. Using JSON Web Token authentication with Service Mesh 1.x and OpenShift Serverless

You can use the following procedure to enable using JSON Web Token authentication with Service Mesh 1.x and OpenShift Serverless.

Prerequisites

- You have installed the OpenShift Serverless Operator and Knative Serving.
- You have installed the oc 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

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

   apiVersion: authentication.istio.io/v1alpha1
   kind: Policy
   metadata:
     name: default
     namespace: <namespace>
   spec:
     origins:
       - jwt:
           issuer: testing@secure.istio.io
         triggerRules:
           - excludedPaths:
              - prefix: /metrics
              - prefix: /healthz
         principalBinding: USE_ORIGIN

   IMPORTANT

   The paths /metrics and /healthz must be included in excludedPaths because they are accessed from system pods in the knative-serving namespace.
The path on your application to collect metrics by system pod.

The path on your application to probe by system pod.

2. Apply the **Policy** resource:

   ```
   $ 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:

   ```
   ```

   The request is now allowed:

   **Example output**

   Hello OpenShift!

### 8.2. 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, 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.

#### 8.2.1. Creating a Custom Domain Mapping

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

- The OpenShift Serverless Operator and Knative Serving are installed on your cluster.
- You have installed the `oc` 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.
- You have created a Knative service and control a custom domain that you want to map to that service.

**NOTE**

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

Procedure

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

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

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

Example service domain mapping

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

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

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

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

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

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

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

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

**Prerequisites**

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

**NOTE**

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

- You have installed the `kn` CLI tool.
- 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:

  ```bash
  $ kn domain create <domain_mapping_name> --ref <target_name>
  ```

**Example command**
8.2.3. Creating a custom domain mapping by using the web console

You can use the Administrator or Developer perspective of the OpenShift Container Platform web console to create a custom domain mapping for a Knative service.

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

If you have cluster administrator permissions, you can create a DomainMapping custom resource (CR) by using the Administrator perspective in the OpenShift Container Platform 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 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

$ kn domain create example-domain-map --ref example-service

Map a domain to a Knative service in a specified namespace:

$ kn domain create <domain_mapping_name> --ref <ksvc:service_name:service_namespace>

Example command

$ kn domain create example-domain-map --ref ksvc:example-service:example-namespace

- Map a domain to a Knative route:

$ kn domain create <domain_mapping_name> --ref <kroute:route_name>

Example command

$ kn domain create example-domain-map --ref kroute:example-route

$ kn domain create <domain_mapping_name> --ref <kroute:route_name>
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:

   ```yaml
   apiVersion: serving.knative.dev/v1alpha1
   kind: DomainMapping
   metadata:
     name: <domain_name>
     namespace: <namespace>
   spec:
     ref:
       name: <target_name>
       kind: <target_type>
   ```

   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**

```yaml
apiVersion: serving.knative.dev/v1alpha1
kind: DomainMapping
metadata:
  name: custom-ksvc-domain.example.com
  namespace: default
spec:
  ref:
    name: example-service
    kind: Service
apiVersion: serving.knative.dev/v1
```

**Verification**

- Access the custom domain by using a `curl` request. For example:

  **Example command**

  ```bash
  $ curl custom-ksvc-domain.example.com
  ```

  **Example output**

  ```
  Hello OpenShift!
  ```
8.2.3.2. Mapping a custom domain to a service by using the Developer perspective

You can use the Developer perspective of the OpenShift Container Platform web console to map a DomainMapping custom resource (CR) to a Knative service.

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 on the service that you want to map to a domain, and select the Edit option that contains the service name. For example, if the service is named example-service, select the Edit example-service 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 from the Domain mapping drop-down.
   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.

8.3. USING A CUSTOM TLS CERTIFICATE FOR DOMAIN MAPPING
You can use an existing TLS certificate with a **DomainMapping** custom resource (CR) to secure the mapped service.

**Prerequisites**

- You have completed the steps in Configuring a custom domain for a Knative service and have a working **DomainMapping** CR.

**8.3.1. Adding a custom TLS certificate to a DomainMapping CR**

You can add an existing TLS certificate with a **DomainMapping** custom resource (CR) to secure the mapped service.

**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.

**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. Update the **DomainMapping** CR to use the TLS secret you have created:

   ```
   apiVersion: serving.knative.dev/v1alpha1
   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**
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.

### 8.4. SECURITY CONFIGURATION FOR KNATIVE KAFKA

In production, Kafka clusters are often secured using the TLS or SASL authentication methods. This section shows how to configure a Kafka channel to work against a protected Red Hat AMQ Streams cluster using TLS or SASL.

**NOTE**

If you choose to enable SASL, Red Hat recommends to also enable TLS.

#### 8.4.1. Configuring TLS authentication

You can use the following procedure to configure TLS authentication for a Kafka channel.

**Prerequisites**

- You have a Kafka cluster CA certificate stored as a `.pem` file.
- You have a Kafka cluster client certificate and a key stored as `.pem` files.

**Procedure**

1. Create the certificate files as secrets in your chosen namespace:

   ```
   $ kubectl create secret -n <namespace> generic <kafka_auth_secret> \
   --from-file=ca.crt=caroot.pem \
   --from-file=user.crt=certificate.pem \
   --from-file=user.key=key.pem
   ```

   **IMPORTANT**

   Use the key names `ca.crt`, `user.crt`, and `user.key`. Do not change them.

2. Start editing the `KnativeKafka` custom resource:

   ```
   $ oc edit knativekafka
   ```

3. Reference your secret and the namespace of the secret:

   ```
   apiVersion: operator.serverless.openshift.io/v1alpha1
   kind: KnativeKafka
   metadata:
   ```
NOTE
Make sure to specify the matching port in the bootstrap server.

For example:

```
namespace: knative-eventing
name: knative-kafka
spec:
  channel:
    authSecretName: <kafka_auth_secret>
    authSecretNamespace: <kafka_auth_secret_namespace>
    bootstrapServers: <bootstrap_servers>
    enabled: true
    source:
      enabled: true
```

8.4.2. Configuring SASL authentication

You can use the following procedure to configure SASL authentication for a Kafka channel.

Prerequisites

- You have a username and password for the Kafka cluster.
- You have chosen the SASL mechanism to use, for example **PLAIN, SCRAM-SHA-256, or SCRAM-SHA-512**.
- If TLS is enabled, you also need the **ca.crt** certificate file for the Kafka cluster.

NOTE
Red Hat recommends to enable TLS in addition to SASL.

Procedure

1. Create the certificate files as secrets in your chosen namespace:

   ```
   $ oc create secret --namespace <namespace> generic <kafka_auth_secret> \
   ```
IMPORTANT

Use the key names `ca.crt`, `password`, and `saslType`. Do not change them.

2. Start editing the `KnativeKafka` custom resource:

   ```sh
   $ oc edit knativekafka
   ```

3. Reference your secret and the namespace of the secret:

   ```yaml
   apiVersion: operator.serverless.openshift.io/v1alpha1
   kind: KnativeKafka
   metadata:
     namespace: knative-eventing
     name: knative-kafka
   spec:
     channel:
       authSecretName: <kafka_auth_secret>
       authSecretNamespace: <kafka_auth_secret_namespace>
       bootstrapServers: <bootstrap_servers>
       enabled: true
     source:
       enabled: true
   ```

NOTE

Make sure to specify the matching port in the bootstrap server.

For example:

```yaml
apiVersion: operator.serverless.openshift.io/v1alpha1
kind: KnativeKafka
metadata:
  namespace: knative-eventing
  name: knative-kafka
spec:
  channel:
    authSecretName: scram-user
    authSecretNamespace: kafka
    bootstrapServers: eventing-kafka-bootstrap.kafka.svc:9093
    enabled: true
  source:
    enabled: true
```

8.4.3. Configuring SASL authentication using public CA certificates
If you want to use SASL with public CA certificates, you must use the `tls.enabled=true` flag, rather than the `ca.crt` argument, when creating the secret. For example:

```
$ oc create secret --namespace <namespace> generic <kafka_auth_secret> \
    --from-literal=tls.enabled=true \
    --from-literal=password="SecretPassword" \
    --from-literal=saslType="SCRAM-SHA-512" \
    --from-literal=user="my-sasl-user"
```

8.4.4. Additional resources

- TLS and SASL on Kafka
CHAPTER 9. FUNCTIONS

9.1. SETTING UP OPENSHIFT SERVERLESS FUNCTIONS

IMPORTANT

OpenShift Serverless Functions is a Technology Preview feature only. Technology Preview features are not supported with Red Hat production service level agreements (SLAs) and might not be functionally complete. Red Hat does not recommend using them in production. These features provide early access to upcoming product features, enabling customers to test functionality and provide feedback during the development process.

For more information about the support scope of Red Hat Technology Preview features, see https://access.redhat.com/support/offerings/techpreview/.

Before you can develop functions on OpenShift Serverless, you must complete the set up steps.

9.1.1. Prerequisites

To enable the use of OpenShift Serverless Functions on your cluster, you must complete the following steps:

- The OpenShift Serverless Operator and Knative Serving are installed on your cluster.

  NOTE

  Functions are deployed as a Knative service. If you want to use event-driven architecture with your functions, you must also install Knative Eventing.

- The oc CLI is installed on your cluster.

- The Knative (kn) CLI is installed on your cluster. Installing the kn CLI enables the use of kn func commands which you can use to create and manage functions.

- You have installed Docker Container Engine or podman version 3.3 or higher, and have access to an available image registry.

- If you are using Quay.io as the image registry, you must ensure that either the repository is not private, or that you have followed the OpenShift Container Platform documentation on Allowing pods to reference images from other secured registries.

- If you are using the OpenShift Container Registry, a cluster administrator must expose the registry.

9.1.2. Using podman

If you are using podman, you must run the following commands before getting started with OpenShift Serverless Functions:

1. Start the podman service that serves the Docker API on a UNIX socket at ${XDG_RUNTIME_DIR}/podman/podman.sock:
$ systemctl start --user podman.socket

NOTE

On most systems, this socket is located at /run/user/$(id -u)/podman/podman.sock.

2. Establish the environment variable that is used to build a function:

$ export DOCKER_HOST="unix://$(XDG_RUNTIME_DIR)/podman/podman.sock"

3. Run the build command with -v to see verbose output. You should see a connection to your local UNIX socket:

$ kn func build -v

9.1.3. Next steps

- For more information about Docker Container Engine or podman, see Container build tool options.
- See Getting started with functions.

9.2. GETTING STARTED WITH FUNCTIONS

IMPORTANT

OpenShift Serverless Functions is a Technology Preview feature only. Technology Preview features are not supported with Red Hat production service level agreements (SLAs) and might not be functionally complete. Red Hat does not recommend using them in production. These features provide early access to upcoming product features, enabling customers to test functionality and provide feedback during the development process.

For more information about the support scope of Red Hat Technology Preview features, see https://access.redhat.com/support/offerings/techpreview/.

This guide explains how you can get started with creating, building, and deploying a function on an OpenShift Serverless installation.

9.2.1. Prerequisites

Before you can complete the following procedures, you must ensure that you have completed all of the prerequisite tasks in Setting up OpenShift Serverless Functions.

9.2.2. Creating functions

You can create a basic serverless function using the kn CLI.

You can specify the path, runtime, template, and repository with the template as flags on the command line, or use the -c flag to start the interactive experience in the terminal.
Prerequisites

- The OpenShift Serverless Operator and Knative Serving are installed on the cluster.
- You have installed the `kn` CLI.

Procedure

- Create a function project:

  ```bash
  $ kn func create -r <repository> -l <runtime> -t <template> <path>
  ``

  - Supported runtimes include `node`, `go`, `python`, `quarkus`, and `typescript`.
  - Supported templates include `http` and `events`.

  **Example command**

  ```bash
  $ kn func create -l typescript -t events examplefunc
  ``

  **Example output**

  ```
  Project path: /home/user/demo/examplefunc
  Function name: examplefunc
  Runtime: typescript
  Template: events
  Writing events to /home/user/demo/examplefunc
  ```

  Alternatively, you can specify a repository that contains a custom template.

  **Example command**

  ```bash
  $ kn func create -r https://github.com/boson-project/templates/ -l node -t hello-world examplefunc
  ``

  **Example output**

  ```
  Project path: /home/user/demo/examplefunc
  Function name: examplefunc
  Runtime: node
  Template: hello-world
  Writing events to /home/user/demo/examplefunc
  ```

9.2.3. Building functions

Before you can run a function, you must build the function project by using the `kn func build` command. The build command reads the `func.yaml` file from the function project directory to determine the image name and registry.

**Example func.yaml**

```yaml
name: example-function
namespace: default
```
If the image name and registry are not set in the **func.yaml** file, you must either specify the registry flag, `-r` when using the **kn func build** command, or you are prompted to provide a registry value in the terminal when building a function. An image name is then derived from the registry value that you have provided.

### Example command using the `-r` registry flag

```bash
$ kn func build [-i <image> -r <registry> -p <path>]
```

### Example output

```
Building function image
Function image has been built, image: quay.io/username/example-function:latest
```

This command creates an OCI container image that can be run locally on your computer, or on a Kubernetes cluster.

### Example using the registry prompt

```
$ kn func build
A registry for function images is required (e.g. 'quay.io/boson').

Registry for function images: quay.io/username
Building function image
Function image has been built, image: quay.io/username/example-function:latest
```

The values for image and registry are persisted to the **func.yaml** file, so that subsequent invocations do not require the user to specify these again.

### 9.2.4. Deploying functions

You can deploy a function to your cluster as a Knative service by using the **kn func deploy** command.

If the targeted function is already deployed, it is updated with a new container image that is pushed to a container image registry, and the Knative service is updated.

#### Prerequisites

- The OpenShift Serverless Operator and Knative Serving are installed on the cluster.
- You have installed the **kn** CLI.
- You have created a project or have access to a project with the appropriate roles and permissions to create applications and other workloads in OpenShift Container Platform.
You must have already created and initialized the function that you want to deploy.

Procedure

- Deploy a function:

  ```
  $ kn func deploy [-n <namespace> -p <path> -i <image> -r <registry>]
  ```

Example output

  - Function deployed at: http://func.example.com
    
    - If no namespace is specified, the function is deployed in the current namespace.
    - The function is deployed from the current directory, unless a path is specified.
    - The Knative service name is derived from the project name, and cannot be changed using this command.

9.2.5. Emitting a test event to a deployed function

You can use the `kn func emit` CLI command to emit a CloudEvent to a function that is either deployed locally or deployed to your OpenShift Container Platform cluster. This command can be used to test that a function is working and able to receive events correctly.

Example command

```
$ kn func emit
```

The `kn func emit` command executes on the local directory by default, and assumes that this directory is a function project.

9.2.6. Building and deploying functions with OpenShift Container Registry

When building and deploying functions, the resulting container image is stored in an image registry. Usually this will be a public registry, such as Quay. However, you can use the integrated OpenShift Container Registry instead if it has been exposed by a cluster administrator.

Prerequisites

- The OpenShift Serverless Operator and Knative Serving are installed on the cluster.
- OpenShift Container Registry has been exposed by a cluster administrator.
- You have installed the `kn` CLI.
- You have created a project or have access to a project with the appropriate roles and permissions to create applications and other workloads in OpenShift Container Platform.

Procedure

- Run the `kn func build` command, or the `kn func deploy` command, with the OpenShift Container Registry and the namespace specified for the `-r` parameter:
Example build command

```
$ kn func build -r $(oc get route -n openshift-image-registry)/my-namespace
```

Example deploy command

```
$ kn func deploy -r $(oc get route -n openshift-image-registry)/my-namespace
```

You can verify that the function deployed successfully by emitting a test event to it.

### 9.2.7. Additional resources

- See [Exposing a default registry manually](#).

### 9.2.8. Next steps

- See [Using functions with Knative Eventing](#).

### 9.3. DEVELOPING NODE.JS FUNCTIONS

**IMPORTANT**

OpenShift Serverless Functions is a Technology Preview feature only. Technology Preview features are not supported with Red Hat production service level agreements (SLAs) and might not be functionally complete. Red Hat does not recommend using them in production. These features provide early access to upcoming product features, enabling customers to test functionality and provide feedback during the development process.

For more information about the support scope of Red Hat Technology Preview features, see [https://access.redhat.com/support/offerings/techpreview/](https://access.redhat.com/support/offerings/techpreview/).

After you have [created a Node.js function project](#), you can modify the template files provided to add business logic to your function.

#### 9.3.1. Prerequisites

- Before you can develop functions, you must complete the steps in [Setting up OpenShift Serverless Functions](#).

#### 9.3.2. Node.js function template structure

When you create a Node.js function using the `kn` CLI, the project directory looks like a typical Node.js project, with the exception of an additional `func.yaml` configuration file.

Both `http` and `event` trigger functions have the same template structure:

**Template structure**

```
  ├── func.yaml
  │    └── index.js
  └── index.js
```
The `func.yaml` configuration file is used to determine the image name and registry.

Your project must contain an `index.js` file which exports a single function.

You are not restricted to the dependencies provided in the template `package.json` file. You can add additional dependencies as you would in any other Node.js project.

### Example of adding npm dependencies

```
npm install --save opossum
```

When the project is built for deployment, these dependencies are included in the created runtime container image.

Integration and unit test scripts are provided as part of the function template.

### 9.3.3. About invoking Node.js functions

When using the `kn` CLI to create a function project, you can generate a project that responds to CloudEvents, or one that responds to simple HTTP requests. CloudEvents in Knative are transported over HTTP as a POST request, so both function types listen for and respond to incoming HTTP events.

Node.js functions can be invoked with a simple HTTP request. When an incoming request is received, functions are invoked with a `context` object as the first parameter.

#### 9.3.3.1. Node.js context objects

Functions are invoked by providing a `context` object as the first parameter.

**Example context object**

```
function handle(context, data)
```

This object provides access to the incoming HTTP request information, including the HTTP request method, any query strings or headers sent with the request, the HTTP version, and the request body. Incoming requests that contain a CloudEvent attach the incoming instance of the CloudEvent to the context object so that it can be accessed by using `context.cloudevent`.

#### 9.3.3.1.1. Context object methods

The `context` object has a single method, `cloudEventResponse()`, that accepts a data value and returns a CloudEvent.

In a Knative system, if a function deployed as a service is invoked by an event broker sending a CloudEvent, the broker examines the response. If the response is a CloudEvent, this event is handled by the broker.
Example context object method

```javascript
// Expects to receive a CloudEvent with customer data
function handle(context, customer) {
  // process the customer
  const processed = handle(customer);
  return context.cloudEventResponse(customer)
    .source('/handle')
    .type('fn.process.customer')
    .response();
}
```

9.3.3.1.2. CloudEvent data

If the incoming request is a CloudEvent, any data associated with the CloudEvent is extracted from the event and provided as a second parameter. For example, if a CloudEvent is received that contains a JSON string in its data property that is similar to the following:

```json
{
  "customerId": "0123456",
  "productId": "6543210"
}
```

When invoked, the second parameter to the function, after the context object, will be a JavaScript object that has customerId and productId properties.

Example signature

```javascript
function handle(context, data)
```

The data parameter in this example is a JavaScript object that contains the customerId and productId properties.

9.3.4. Node.js function return values

Functions can return any valid JavaScript type or can have no return value. When a function has no return value specified, and no failure is indicated, the caller receives a 204 No Content response.

Functions can also return a CloudEvent or a Message object in order to push events into the Knative Eventing system. In this case, the developer is not required to understand or implement the CloudEvent messaging specification. Headers and other relevant information from the returned values are extracted and sent with the response.

Example

```javascript
function handle(context, customer) {
  // process customer and return a new CloudEvent
  return new CloudEvent({
    source: 'customer.processor',
    type: 'customer.processed'
  });
}
```
9.3.4.1. Returning headers

You can set a response header by adding a `headers` property to the `return` object. These headers are extracted and sent with the response to the caller.

Example response header

```javascript
function handle(context, customer) {
    // process customer and return custom headers
    // the response will be '204 No content'
    return { headers: { customerid: customer.id } };
}
```

9.3.4.2. Returning status codes

You can set a status code that is returned to the caller by adding a `statusCode` property to the `return` object:

Example status code

```javascript
function handle(context, customer) {
    // process customer
    if (customer.restricted) {
        return { statusCode: 451 }
    }
}
```

Status codes can also be set for errors that are created and thrown by the function:

Example error status code

```javascript
function handle(context, customer) {
    // process customer
    if (customer.restricted) {
        const err = new Error('Unavailable for legal reasons');
        err.statusCode = 451;
        throw err;
    }
}
```

9.3.5. Testing Node.js functions

Node.js functions can be tested locally on your computer. In the default project that is created when you create a function by using `kn func create`, there is a `test` folder that contains some simple unit and integration tests.

Prerequisites

- The OpenShift Serverless Operator and Knative Serving are installed on the cluster.
- You have installed the `kn` CLI.
- You have created a function by using `kn func create`. 
Procedure

1. Navigate to the **test** folder for your function.

2. Run the tests:

   ```
   $ npm test
   ```

9.3.6. Next steps

- See the [Node.js context object reference](#) documentation.

- Build and deploy a function.

### 9.4. DEVELOPING TYPESCRIPT FUNCTIONS

**IMPORTANT**

OpenShift Serverless Functions is a Technology Preview feature only. Technology Preview features are not supported with Red Hat production service level agreements (SLAs) and might not be functionally complete. Red Hat does not recommend using them in production. These features provide early access to upcoming product features, enabling customers to test functionality and provide feedback during the development process.

For more information about the support scope of Red Hat Technology Preview features, see [https://access.redhat.com/support/offerings/techpreview/](https://access.redhat.com/support/offerings/techpreview/).

After you have created a TypeScript function project, you can modify the template files provided to add business logic to your function.

9.4.1. Prerequisites

- Before you can develop functions, you must complete the steps in [Setting up OpenShift Serverless Functions](#).

9.4.2. TypeScript function template structure

When you create a TypeScript function using the **kn** CLI, the project directory looks like a typical TypeScript project with the exception of an additional **func.yaml** configuration file.

Both **http** and **event** trigger functions have the same template structure:

**Template structure**

```

├── func.yaml
├── package.json
├── package-lock.json
├── README.md
├── src
│   └── index.ts
├── test
```
The `func.yaml` configuration file is used to determine the image name and registry.

You are not restricted to the dependencies provided in the template `package.json` file. You can add additional dependencies as you would in any other TypeScript project.

**Example of adding npm dependencies**

```
npm install --save opossum
```

When the project is built for deployment, these dependencies are included in the created runtime container image.

Your project must contain an `src/index.js` file which exports a function named `handle`.

Integration and unit test scripts are provided as part of the function template.

### 9.4.3. About invoking TypeScript functions

When using the `kn` CLI to create a function project, you can generate a project that responds to CloudEvents or one that responds to simple HTTP requests. CloudEvents in Knative are transported over HTTP as a POST request, so both function types listen for and respond to incoming HTTP events.

TypeScript functions can be invoked with a simple HTTP request. When an incoming request is received, functions are invoked with a `context` object as the first parameter.

#### 9.4.3.1. TypeScript context objects

Functions are invoked with a `context` object as the first parameter.

**Example context object**

```typescript
function handle(context: Context): string
```

This object provides access to the incoming HTTP request information, including the HTTP request method, any query strings or headers sent with the request, the HTTP version, and the request body. Incoming requests that contain a CloudEvent attach the incoming instance of the CloudEvent to the context object so that it can be accessed by using `context.cloudevent`.

#### 9.4.3.1.1. Context object methods

The `context` object has a single method, `cloudEventResponse()`, that accepts a data value and returns a CloudEvent.

In a Knative system, if a function deployed as a service is invoked by an event broker sending a CloudEvent, the broker examines the response. If the response is a CloudEvent, this event is handled by the broker.

**Example context object method**
9.4.3.1.2. Context types

The TypeScript type definition files export the following types for use in your functions.

Exported type definitions

```typescript
// Expects to receive a CloudEvent with customer data
export function handle(context: Context, cloudevent?: CloudEvent): CloudEvent {
  // process the customer
  const customer = cloudevent.data;
  const processed = processCustomer(customer);
  return context.cloudEventResponse(customer)
    .source('/customer/process')
    .type('customer.processed')
    .response();
}

// Invokable is the expected Function signature for user functions
export interface Invokable {
  (context: Context, cloudevent?: CloudEvent): any
}

// Logger can be used for structural logging to the console
export interface Logger {
  debug: (msg: any) => void,
  info: (msg: any) => void,
  warn: (msg: any) => void,
  error: (msg: any) => void,
  fatal: (msg: any) => void,
  trace: (msg: any) => void,
}

// Context represents the function invocation context, and provides
// access to the event itself as well as raw HTTP objects.
export interface Context {
  log: Logger;
  req: IncomingMessage;
  query?: Record<string, any>;
  body?: Record<string, any>|string;
  method: string;
  headers: IncomingHttpHeaders;
  httpVersion: string;
  httpVersionMajor: number;
  httpVersionMinor: number;
  cloudevent: CloudEvent;
  cloudEventResponse(data: string|object): CloudEventResponse;
}

// CloudEventResponse is a convenience class used to create
// CloudEvents on function returns
export interface CloudEventResponse {
  id(id: string): CloudEventResponse;
  source(source: string): CloudEventResponse;
  type(type: string): CloudEventResponse;
}
```
If the incoming request is a CloudEvent, any data associated with the CloudEvent is extracted from the event and provided as a second parameter. For example, if a CloudEvent is received that contains a JSON string in its data property that is similar to the following:

```json
{
  "customerId": "0123456",
  "productId": "6543210"
}
```

When invoked, the second parameter to the function, after the `context` object, will be a JavaScript object that has `customerId` and `productId` properties.

**Example signature**

```javascript
function handle(context: Context, cloudevent?: CloudEvent): CloudEvent
```

The `cloudevent` parameter in this example is a JavaScript object that contains the `customerId` and `productId` properties.

### 9.4.4. TypeScript function return values

Functions can return any valid JavaScript type or can have no return value. When a function has no return value specified, and no failure is indicated, the caller receives a **204 No Content** response.

Functions can also return a CloudEvent or a `Message` object in order to push events into the Knative Eventing system. In this case, the developer is not required to understand or implement the CloudEvent messaging specification. Headers and other relevant information from the returned values are extracted and sent with the response.

**Example**

```javascript
export const handle: Invokable = function (context: Context, cloudevent?: CloudEvent): Message {
  // process customer and return a new CloudEvent
  const customer = cloudevent.data;
  return HTTP.binary(
    new CloudEvent({
      source: 'customer.processor',
      type: 'customer.processed'
    })
  );
};
```

#### 9.4.4.1. Returning headers
You can set a response header by adding a `headers` property to the `return` object. These headers are extracted and sent with the response to the caller.

**Example response header**

```typescript
export function handle(context: Context, cloudevent?: CloudEvent): Record<string, any> {
    // process customer and return custom headers
    const customer = cloudevent.data as Record<string, any>;
    return { headers: { 'customer-id': customer.id } };
}
```

### 9.4.4.2. Returning status codes

You can set a status code that is returned to the caller by adding a `statusCode` property to the `return` object:

**Example status code**

```typescript
export function handle(context: Context, cloudevent?: CloudEvent): Record<string, any> {
    // process customer
    const customer = cloudevent.data as Record<string, any>;
    if (customer.restricted) {
        return {
            statusCode: 451
        }
    }
    // business logic, then
    return {
        statusCode: 240
    }
}
```

Status codes can also be set for errors that are created and thrown by the function:

**Example error status code**

```typescript
export function handle(context: Context, cloudevent?: CloudEvent): Record<string, string> {
    // process customer
    const customer = cloudevent.data as Record<string, any>;
    if (customer.restricted) {
        const err = new Error('Unavailable for legal reasons');
        err.statusCode = 451;
        throw err;
    }
}
```

### 9.4.5. Testing TypeScript functions

TypeScript functions can be tested locally on your computer. In the default project that is created when you create a function using `kn func create`, there is a `test` folder that contains some simple unit and integration tests.

**Prerequisites**
The OpenShift Serverless Operator and Knative Serving are installed on the cluster.

- You have installed the `kn` CLI.
- You have created a function by using `kn func create`.

**Procedure**

1. If you have not previously run tests, install the dependencies first:
   ```
   $ npm install
   ```

2. Navigate to the `test` folder for your function.

3. Run the tests:
   ```
   $ npm test
   ```

**9.4.6. Next steps**

- See the [TypeScript context object reference](https://your-url) documentation.
- Build and deploy a function.
- See the [Pino API documentation](https://your-url) for more information on logging with functions.

**9.5. DEVELOPING GOLANG FUNCTIONS**

**IMPORTANT**

OpenShift Serverless Functions is a Technology Preview feature only. Technology Preview features are not supported with Red Hat production service level agreements (SLAs) and might not be functionally complete. Red Hat does not recommend using them in production. These features provide early access to upcoming product features, enabling customers to test functionality and provide feedback during the development process.

For more information about the support scope of Red Hat Technology Preview features, see [https://access.redhat.com/support/offerings/techpreview/](https://access.redhat.com/support/offerings/techpreview/).

After you have created a Golang function project, you can modify the template files provided to add business logic to your function.

**9.5.1. Prerequisites**

- Before you can develop functions, you must complete the steps in [Setting up OpenShift Serverless Functions](https://your-url).

**9.5.2. Golang function template structure**

When you create a Golang function using the `kn` CLI, the project directory looks like a typical Go project, with the exception of an additional `func.yaml` configuration file.
Golang functions have few restrictions. The only requirements are that your project must be defined in a function module, and must export the function `Handle()`.

Both http and event trigger functions have the same template structure:

**Template structure**

```
fn
 ├── README.md
 ├── func.yaml
 └── go.mod

1 The `func.yaml` configuration file is used to determine the image name and registry.

2 You can add any required dependencies to the `go.mod` file, which can include additional local Golang files. When the project is built for deployment, these dependencies are included in the resulting runtime container image.

Example of adding dependencies

```
$ go get gopkg.in/yaml.v2@v2.4.0
```

---

**9.5.3. About invoking Golang functions**

Golang functions are invoked by using different methods, depending on whether they are triggered by an HTTP request or a CloudEvent.

**9.5.3.1. Functions triggered by an HTTP request**

When an incoming HTTP request is received, your function is invoked with a standard Golang `Context` as the first parameter, followed by two more parameters:

- `http.ResponseWriter`
- `http.Request`

You can use standard Golang techniques to access the request, and set a proper HTTP response of your function.

**Example HTTP response**

```go
func Handle(ctx context.Context, res http.ResponseWriter, req *http.Request) {
    // Read body
    body, err := ioutil.ReadAll(req.Body)
    defer req.Body.Close()
    if err != nil {
        http.Error(res, err.Error(), 500)
        return
    }
}
```
9.5.3.2. Functions triggered by a cloud event

When an incoming cloud event is received, the event is invoked by the CloudEvents Golang SDK and the Event type as a parameter.

You can leverage the Golang Context as an optional parameter in the function contract, as shown in the list of supported function signatures:

**Supported function signatures**

```go
Handle()
Handle() error
Handle(context.Context)
Handle(context.Context) error
Handle(cloudevents.Event)
Handle(cloudevents.Event) error
Handle(context.Context, cloudevents.Event)
Handle(context.Context, cloudevents.Event) error
Handle(cloudevents.Event) *cloudevents.Event
Handle(cloudevents.Event) (*cloudevents.Event, error)
```

**9.5.3.2.1. CloudEvent trigger example**

A cloud event is received which contains a JSON string in the data property:

```json
{
  "customerId": "0123456",
  "productId": "6543210"
}
```

To access this data, a structure must be defined which maps properties in the cloud event data, and retrieves the data from the incoming event. The following example uses the Purchase structure:

```go
type Purchase struct {
  CustomerId string `json:"customerId"
  ProductId string `json:"productId"
}

func Handle(ctx context.Context, event cloudevents.Event) (err error) {
  purchase := &Purchase{}
  if err = event.DataAs(purchase); err != nil {
    fmt.Fprintf(os.Stderr, "failed to parse incoming CloudEvent %s\n", err)
    return
  }
  // ...
}
```
Alternatively, a Golang encoding/json package could be used to access the cloud event directly as JSON in the form of a bytes array:

```go
func Handle(ctx context.Context, event cloudevents.Event) {
    bytes, err := json.Marshal(event)
    // ...
}
```

### 9.5.4. Golang function return values

HTTP triggered functions can set the response directly by using the Golang http.ResponseWriter.

**Example HTTP response**

```go
func Handle(ctx context.Context, res http.ResponseWriter, req *http.Request) {
    // Set response
    res.Header().Add("Content-Type", "text/plain")
    res.Header().Add("Content-Length", "3")
    res.WriteHeader(200)
    _, err := fmt.Fprintf(res, "OK\n")
    if err != nil {
        fmt.Fprintf(os.Stderr, "error or response write: %v", err)
    }
}
```

Functions triggered by a cloud event might return nothing, error, or CloudEvent in order to push events into the Knative Eventing system. In this case, you must set a unique ID, proper Source, and a Type for the cloud event. The data can be populated from a defined structure, or from a map.

**Example CloudEvent response**

```go
func Handle(ctx context.Context, event cloudevents.Event) (resp *cloudevents.Event, err error) {
    // ...
    response := cloudevents.NewEvent()
    response.SetID("example-uuid-32943bac6fea")
    response.SetSource("purchase/getter")
    response.SetType("purchase")
    // Set the data from Purchase type
    response.SetData(cloudevents.ApplicationJSON, Purchase{
        CustomerId: custId,
        ProductId: prodId,
    })
    // OR set the data directly from map
    response.SetData(cloudevents.ApplicationJSON, map[string]string{"customerId": custId, "productId": prodId})
    // Validate the response
    resp = &response
    if err = resp.Validate(); err != nil {
        fmt.Printf("invalid event created. %v", err)
    }
    return
}
```

### 9.5.5. Testing Golang functions
Golang functions can be tested locally on your computer. In the default project that is created when you create a function using `kn func create`, there is a `handle_test.go` file which contains some basic tests. These tests can be extended as needed.

**Prerequisites**

- The OpenShift Serverless Operator and Knative Serving are installed on the cluster.
- You have installed the `kn` CLI.
- You have created a function by using `kn func create`.

**Procedure**

1. Navigate to the test folder for your function.
2. Run the tests:

   ```
   $ go test
   ```

**9.5.6. Next steps**

- **Build** and **deploy** a function.

**9.6. DEVELOPING PYTHON FUNCTIONS**

**IMPORTANT**

OpenShift Serverless Functions is a Technology Preview feature only. Technology Preview features are not supported with Red Hat production service level agreements (SLAs) and might not be functionally complete. Red Hat does not recommend using them in production. These features provide early access to upcoming product features, enabling customers to test functionality and provide feedback during the development process.

For more information about the support scope of Red Hat Technology Preview features, see [https://access.redhat.com/support/offerings/techpreview/](https://access.redhat.com/support/offerings/techpreview/).

After you have created a Python function project, you can modify the template files provided to add business logic to your function.

**9.6.1. Prerequisites**

- Before you can develop functions, you must complete the steps in [Setting up OpenShift Serverless Functions](#).

**9.6.2. Python function template structure**

When you create a Python function by using the `kn` CLI, the project directory looks similar to a typical Python project.

Python functions have very few restrictions. The only requirements are that your project contains a `func.py` file that contains a `main()` function, and a `func.yaml` configuration file.
Developers are not restricted to the dependencies provided in the template requirements.txt file. Additional dependencies can be added as they would be in any other Python project. When the project is built for deployment, these dependencies will be included in the created runtime container image.

Both http and event trigger functions have the same template structure:

**Template structure**

```
fn
├── func.py 1
├── func.yaml 2
├── requirements.txt 3
└── test_func.py 4
```

1. Contains a `main()` function.
2. Used to determine the image name and registry.
3. Additional dependencies can be added to the requirements.txt file as they are in any other Python project.
4. Contains a simple unit test that can be used to test your function locally.

### 9.6.3. About invoking Python functions

Python functions can be invoked with a simple HTTP request. When an incoming request is received, functions are invoked with a context object as the first parameter. The context object is a Python class with two attributes:

- The request attribute is always present, and contains the Flask request object.
- The second attribute, cloud_event, is populated if the incoming request is a CloudEvent object.

Developers can access any CloudEvent data from the context object.

**Example context object**

```python
def main(context: Context):
    
    # The context parameter contains the Flask request object and any CloudEvent received with the request.
    print(f"Method: {context.request.method}")
    print(f"Event data {context.cloud_event.data}")
    # ... business logic here
```

### 9.6.4. Python function return values

Functions can return any value supported by Flask because the invocation framework proxies these values directly to the Flask server.

**Example**
Functions can set both headers and response codes as secondary and tertiary response values from function invocation.

9.6.4.1. Returning CloudEvents

Developers can use the `@event` decorator to tell the invoker that the function return value must be converted to a CloudEvent before sending the response.

Example

```python
@event("event_source="/my/function", "event_type="/my.type")
def main(context):
    # business logic here
    data = do_something()
    # more data processing
    return data
```

This example sends a CloudEvent as the response value, with a type of "my.type" and a source of "my/function". The CloudEvent data property is set to the returned data variable. The `event_source` and `event_type` decorator attributes are both optional.

9.6.5. Testing Python functions

You can test Python functions locally on your computer. The default project contains a `test_func.py` file, which provides a simple unit test for functions.

**NOTE**

The default test framework for Python functions is `unittest`. You can use a different test framework if you prefer.

Prerequisites

- To run Python functions tests locally, you must install the required dependencies:

  ```bash
  $ pip install -r requirements.txt
  ```

Procedure

1. Navigate to the folder for your function that contains the `test_func.py` file.
2. Run the tests:

  ```bash
  $ python3 test_func.py
  ```

9.6.6. Next steps
9.7. DEVELOPING QUARKUS FUNCTIONS

**IMPORTANT**

OpenShift Serverless Functions is a Technology Preview feature only. Technology Preview features are not supported with Red Hat production service level agreements (SLAs) and might not be functionally complete. Red Hat does not recommend using them in production. These features provide early access to upcoming product features, enabling customers to test functionality and provide feedback during the development process.

For more information about the support scope of Red Hat Technology Preview features, see [https://access.redhat.com/support/offerings/techpreview/](https://access.redhat.com/support/offerings/techpreview/).

After you have created a Quarkus function project, you can modify the template files provided to add business logic to your function.

9.7.1. Prerequisites

- Before you can develop functions, you must complete the setup steps in Setting up OpenShift Serverless Functions.

9.7.2. Quarkus function template structure

When you create a Quarkus function by using the `kn` CLI, the project directory looks similar to a typical Maven project.

Both http and event trigger functions have the same template structure:

**Template structure**

```
├── func.yaml
├── mvnw
├── mvnw.cmd
├── pom.xml
├── README.md
└── src
    ├── main
    │   └── java
    │       └── functions
    │           └── Function.java
    │           └── Input.java
    │           └── Output.java
    │   └── resources
    │       └── application.properties
    └── test
        └── java
            └── functions
                └── FunctionTest.java
                └── NativeFunctionIT.java
```
Used to determine the image name and registry.

The Project Object Model (POM) file contains project configuration, such as information about dependencies. You can add additional dependencies by modifying this file.

**Example of additional dependencies**

```
...  
<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.11</version>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>org.assertj</groupId>
    <artifactId>assertj-core</artifactId>
    <version>3.8.0</version>
    <scope>test</scope>
  </dependency>
</dependencies>
...
```

Dependencies are downloaded during the first compilation.

The function project must contain a Java method annotated with `@Funq`. You can place this method in the `Function.java` class.

Contains simple test cases that can be used to test your function locally.

### 9.7.3. About invoking Quarkus functions

You can create a Quarkus project that responds to cloud events, or one that responds to simple HTTP requests. Cloud events in Knative are transported over HTTP as a POST request, so either function type can listen and respond to incoming HTTP requests.

When an incoming request is received, Quarkus functions are invoked with an instance of a permitted type.

**Table 9.1. Function invocation options**

<table>
<thead>
<tr>
<th>Invocation method</th>
<th>Data type contained in the instance</th>
<th>Example of data</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTTP POST request</td>
<td>JSON object in the body of the request</td>
<td><code>{ &quot;customerId&quot;: &quot;0123456&quot;, &quot;productId&quot;: &quot;6543210&quot; }</code></td>
</tr>
<tr>
<td>HTTP GET request</td>
<td>Data in the query string</td>
<td><code>?customerId=0123456&amp;productId=6543210</code></td>
</tr>
</tbody>
</table>
The following example shows a function that receives and processes the `customerId` and `productId` purchase data that is listed in the previous table:

### Example Quarkus function

```java
public class Functions {
    @Funq
    public void processPurchase(Purchase purchase) {
        // process the purchase
    }
}
```

The corresponding `Purchase` JavaBean class that contains the purchase data looks as follows:

### Example class

```java
public class Purchase {
    private long customerId;
    private long productId;
    // getters and setters
}
```

### 9.7.3.1. Invocation examples

The following example code defines three functions named `withBeans`, `withCloudEvent`, and `withBinary`:

### Example

```java
import io.quarkus.funqy.Funq;
import io.quarkus.funqy.knative.events.CloudEvent;

public class Input {
    private String message;
    // getters and setters
}

public class Output {
    private String message;
    // getters and setters
}

public class Functions {
    @Funq
```
public Output withBeans(Input in) {
    // function body
}

@Funq
public CloudEvent<Output> withCloudEvent(CloudEvent<Input> in) {
    // function body
}

@Funq
public void withBinary(byte[] in) {
    // function body
}

The `withBeans` function of the `Functions` class can be invoked by:

- An HTTP POST request with a JSON body:

  ```
  $ curl "http://localhost:8080/withBeans" -X POST \
  -H "Content-Type: application/json" \
  -d '{"message": "Hello there."}'
  ```

- An HTTP GET request with query parameters:

  ```
  ```

- A `CloudEvent` object in binary encoding:

  ```
  $ curl "http://localhost:8080/" -X POST \
  -H "Content-Type: application/json" \
  -H "Ce-SpecVersion: 1.0" \
  -H "Ce-Type: withBeans" \
  -H "Ce-Source: cURL" \
  -H "Ce-Id: 42" \
  -d '{"message": "Hello there."}'
  ```

- A `CloudEvent` object in structured encoding:

  ```
  $ curl http://localhost:8080/ \
  -H "Content-Type: application/cloudevents+json" \
  -d '{
      "data": {
          "message": "Hello there.",
          "datacontenttype": "application/json",
          "id": "42",
          "source": "curl",
          "type": "withBeans",
          "specversion": "1.0"
      }
  }
  ```

The `withCloudEvent` function of the `Functions` class can be invoked by using a `CloudEvent` object, similarly to the `withBeans` function. However, unlike `withBeans`, `withCloudEvent` cannot be invoked with a plain HTTP request.

The `withBinary` function of the `Functions` class can be invoked by:
• A **CloudEvent** object in binary encoding:

```bash
$ curl "http://localhost:8080/" -X POST \
-H "Content-Type: application/octet-stream" \
-H "Ce-SpecVersion: 1.0" \
-H "Ce-Type: withBinary" \
-H "Ce-Source: cURL" \
-H "Ce-Id: 42" \
--data-binary '@img.jpg'
```

• A **CloudEvent** object in structured encoding:

```bash
$ curl http://localhost:8080/ \
-H "Content-Type: application/cloudevents+json" \
-d '{ \"data_base64\": \"$(base64 --wrap=0 img.jpg)\", \"datacontenttype\": \"application/octet-stream\", \"id\": \"42\", \"source\": \"curl\", \"type\": \"withBinary\", \"specversion\": \"1.0\"]'
```

### 9.7.4. CloudEvent attributes

If you need to read or write the attributes of a CloudEvent, such as **type** or **subject**, you can use the **CloudEvent<T>** generic interface and the **CloudEventBuilder** builder. The `<T>` type parameter must be one of the permitted types.

In the following example, **CloudEventBuilder** is used to return success or failure of processing the purchase:

```java
public class Functions {

    private boolean _processPurchase(Purchase purchase) {
        // do stuff
    }

    public CloudEvent<Void> processPurchase(CloudEvent<Purchase> purchaseEvent) {
        System.out.println("subject is: " + purchaseEvent.subject());

        if (!_processPurchase(purchaseEvent.data())) {
            return CloudEventBuilder.create() 
                .type("purchase.error") 
                .build();
        }

        return CloudEventBuilder.create() 
            .type("purchase.success") 
            .build();
    }
}
```

### 9.7.5. Quarkus function return values

Functions can return an instance of:
• Any type from the list of permitted types.

• The Uni<T> type, where the <T> type parameter can be of any type from the permitted types.

The Uni<T> type is useful if a function calls asynchronous APIs, because the returned object is serialized in the same format as the received object. For example:

• If a function receives an HTTP request, then the returned object is sent in the body of an HTTP response.

• If a function receives a CloudEvent object in binary encoding, then the returned object is sent in the data property of a binary-encoded CloudEvent object.

The following example shows a function that fetches a list of purchases:

Example command

```java
public class Functions {
    @Funq
    public List<Purchase> get PurchasesByName(String name) {
        // logic to retrieve purchases
    }
}
```

• Invoking this function through an HTTP request produces an HTTP response that contains a list of purchases in the body of the response.

• Invoking this function through an incoming CloudEvent object produces a CloudEvent response with a list of purchases in the data property.

9.7.5.1. Permitted types

The input and output types of a function can be any of the following:

• void

• String

• byte[]

• Primitive types and their wrappers (for example, int and Integer).

• A JavaBean, if its attributes are of types listed here.

• A map, list, or array of the types in this list.

• The special CloudEvents<T> type, where the <T> type parameter is of a type in this list.

Example

```java
public class Functions {
    public List<Integer> getIds();
    public Purchase[] get PurchasesByName(String name);
    public String getNameById(int id);
}
```
9.7.6. Testing Quarkus functions

You can test Quarkus functions locally on your computer by running the Maven tests that are included in the project template.

Prerequisites

- You have created a Quarkus function.

Procedure

1. Navigate to the project folder for your function.
2. Run the Maven tests:

   
   $ ./mvnw test

9.7.7. Next steps

- Build and deploy a function.

9.8. USING FUNCTIONS WITH KNATIVE EVENTING

IMPORTANT

OpenShift Serverless Functions is a Technology Preview feature only. Technology Preview features are not supported with Red Hat production service level agreements (SLAs) and might not be functionally complete. Red Hat does not recommend using them in production. These features provide early access to upcoming product features, enabling customers to test functionality and provide feedback during the development process.

For more information about the support scope of Red Hat Technology Preview features, see [https://access.redhat.com/support/offerings/techpreview/](https://access.redhat.com/support/offerings/techpreview/).

Functions are deployed as a Knative service on an OpenShift Container Platform cluster, and can be connected as a sink to Knative Eventing components.

9.8.1. Connect an event source to a sink using the Developer perspective

You can create multiple event source types in OpenShift Container Platform that can be connected to sinks.

Prerequisites

- The OpenShift Serverless Operator, Knative Serving, and Knative Eventing are installed on your OpenShift Container Platform cluster.
• You have created a sink.
• You have logged in to the web console and are in the Developer perspective.
• You have created a project or have access to a project with the appropriate roles and permissions to create applications and other workloads in OpenShift Container Platform.

Procedure

1. Create an event source of any type, by navigating to +Add → Event Sources and then selecting the event source type that you want to create.

2. In the Sink section of the Event Sources form view, select Resource. Then use the drop-down list to select your sink.

3. Click Create.

Verification

You can verify that the event source was created and is connected to the sink by viewing the Topology page.

1. In the Developer perspective, navigate to Topology.

2. View the event source and click on the connected sink to see the sink details in the side panel.

9.9. FUNCTION PROJECT CONFIGURATION IN FUNC.YAML

The func.yaml file contains the configuration for your function project. These values are used when you execute a kn func command. For example, when you run the kn func build command, the value in the builder field is used. In some cases, you can override these values with command line flags or environment variables.

9.9.1. Configurable fields in func.yaml

Many of the fields in func.yaml are generated automatically when you create, build, and deploy your function. However, there are also fields that you modify manually to change things, such as the function name or the image name.

9.9.1.1. builder

The builder field specifies the Buildpack builder image to use when building the function. In most cases, this value should not be changed. When you do change it, use a value that is listed in the builders field.

9.9.1.2. builders

Some function runtimes can be built in multiple ways. For example, a Quarkus function can be built for the JVM or as a native binary. The builders field contains all of the builders available for a given runtime.

9.9.1.3. buildEnvs

The buildEnvs field enables you to set environment variables to be available to the environment that builds your function. Unlike variables set using envs, a variable set using buildEnv is not available during function runtime.
You can set a `buildEnv` variable directly from a value. In the following example, the `buildEnv` variable named `EXAMPLE1` is directly assigned the `one` value:

```yaml
buildEnvs:
- name: EXAMPLE1
  value: one
```

You can also set a `buildEnv` variable from a local environment variable. In the following example, the `buildEnv` variable named `EXAMPLE2` is assigned the value of the `LOCAL_ENV_VAR` local environment variable:

```yaml
buildEnvs:
- name: EXAMPLE1
  value: '{{ env:LOCAL_ENV_VAR }}'
```

9.9.1.4. envs

The `envs` field enables you to set environment variables to be available to your function at runtime. You can set an environment variable in several different ways:

1. Directly from a value.

2. From a value assigned to a local environment variable. See the section "Referencing local environment variables from func.yaml fields" for more information.

3. From a key-value pair stored in a secret or config map.

4. You can also import all key-value pairs stored in a secret or config map, with keys used as names of the created environment variables.

This examples demonstrates the different ways to set an environment variable:

```yaml
name: test
namespace: ""
runtime: go
...
envs:
- name: EXAMPLE1 ¹
  value: value
- name: EXAMPLE2 ²
  value: '{{ env:LOCAL_ENV_VALUE }}'
- name: EXAMPLE3 ³
  value: '{{ secret:mysecret:key }}'
- name: EXAMPLE4 ⁴
  value: '{{ configMap:myconfigmap:key }}'
- value: '{{ secret:mysecret2 }}' ⁵
- value: '{{ configMap:myconfigmap2 }}' ⁶

¹ An environment variable set directly from a value.
² An environment variable set from a value assigned to a local environment variable.
³ An environment variable assigned from a key-value pair stored in a secret.
4. An environment variable assigned from a key-value pair stored in a config map.

5. A set of environment variables imported from key-value pairs of a secret.

6. A set of environment variables imported from key-value pairs of a config map.

9.9.1.5. volumes

The **volumes** field enables you to mount secrets and config maps as a volume accessible to the function at the specified path, as shown in the following example:

```yaml
name: test
namespace: ""
runtime: go
...
volumes:
- secret: mysecret
  path: /workspace/secret
- configMap: myconfigmap
  path: /workspace/configmap
```

1. The **mysecret** secret is mounted as a volume residing at **/workspace/secret**.

2. The **myconfigmap** config map is mounted as a volume residing at **/workspace/configmap**.

9.9.1.6. options

The **options** field enables you to modify Knative Service properties for the deployed function, such as autoscaling. If these options are not set, the default ones are used.

These options are available:

- **scale**
  - **min**: The minimum number of replicas. Must be a non-negative integer. The default is 0.
  - **max**: The maximum number of replicas. Must be a non-negative integer. The default is 0, which means no limit.
  - **metric**: Defines which metric type is watched by the Autoscaler. It can be set to `concurrency`, which is the default, or `rps`.
  - **target**: Recommendation for when to scale up based on the number of concurrently incoming requests. The **target** option can be a float value greater than 0.01. The default is 100, unless the options.resources.limits.concurrency is set, in which case target defaults to its value.
  - **utilization**: Percentage of concurrent requests utilization allowed before scaling up. It can be a float value between 1 and 100. The default is 70.

- **resources**
  - **requests**
- **cpu**: A CPU resource request for the container with deployed function.
- **memory**: A memory resource request for the container with deployed function.

- **limits**
  - **cpu**: A CPU resource limit for the container with deployed function.
  - **memory**: A memory resource limit for the container with deployed function.
  - **concurrency**: Hard Limit of concurrent requests to be processed by a single replica. It can be integer value greater than or equal to 0, default is 0 - meaning no limit.

This is an example configuration of the **scale** options:

```yaml
name: test
namespace: ""
runtime: go
...
options:
  scale:
    min: 0
    max: 10
    metric: concurrency
    target: 75
    utilization: 75
resources:
  requests:
    cpu: 100m
    memory: 128Mi
  limits:
    cpu: 1000m
    memory: 256Mi
    concurrency: 100
```

### 9.9.1.7. image

The **image** field sets the image name for your function after it has been built. You can modify this field. If you do, the next time you run `kn func build` or `kn func deploy`, the function image will be created with the new name.

### 9.9.1.8. imageDigest

The **imageDigest** field contains the SHA256 hash of the image manifest when the function is deployed. Do not modify this value.

### 9.9.1.9. labels

The **labels** field enables you to set labels on a deployed function.

You can set a label directly from a value. In the following example, the label with the **role** key is directly assigned the value of **backend**:
You can also set a label from a local environment variable. In the following example, the label with the `author` key is assigned the value of the `USER` local environment variable:

```
labels:
  - key: author
    value: '${( env:USER )}'
```

### 9.9.10. name

The `name` field defines the name of your function. This value is used as the name of your Knative service when it is deployed. You can change this field to rename the function on subsequent deployments.

### 9.9.11. namespace

The `namespace` field specifies the namespace in which your function is deployed.

### 9.9.12. runtime

The `runtime` field specifies the language runtime for your function, for example, `python`.

### 9.9.2. Referencing local environment variables from `func.yaml` fields

In the `envs` field in the `func.yaml`, you can put a reference to an environment variable available in the local environment. This can be useful for avoiding storing sensitive information, such as an API key in the function configuration.

#### Procedure

- To refer to a local environment variable, use the following syntax:

```
{{ env:ENV_VAR }}
```

Substitute `ENV_VAR` with the name of the variable in the local environment that you want to use.

For example, you might have the `API_KEY` variable available in the local environment. You can assign its value to the `MY_API_KEY` variable, which you can then directly use within your function:

```
Example function
```

```yaml
name: test
namespace: ""
runtime: go
...
envs:
  - name: MY_API_KEY
    value: '{{ env:API_KEY }}'
...
```
Additional resources

- For information on overriding values in the `func.yaml` file, see Getting started with functions.
- For more information on accessing data in secrets and config maps, see Accessing secrets and config maps from Serverless functions.
- For more information on the `scale` set of options, see the Knative documentation on Autoscaling.
- For more information on the `resources` set of options, see the Kubernetes documentation on managing resources for containers and the Knative documentation on configuring concurrency.

9.10. ACCESSING SECRETS AND CONFIG MAPS FROM FUNCTIONS

After your functions have been deployed to the cluster, they can access data stored in secrets and config maps. This data can be mounted as volumes, or assigned to environment variables. You can configure this access interactively by using the Knative CLI, or by manually by editing the function configuration YAML file.

**IMPORTANT**

To access secrets and config maps, the function must be deployed on the cluster. This functionality is not available to a function running locally.

If a secret or config map value cannot be accessed, the deployment fails with an error message specifying the inaccessible values.

9.10.1. Modifying function access to secrets and config maps interactively

You can manage the secrets and config maps accessed by your function by using the `kn func config` interactive utility.

**Prerequisites**

- The OpenShift Serverless Operator and Knative Serving are installed on the cluster.
- You have installed the `kn` CLI.
- You have created a function.

**Procedure**

1. Run the following command in the function project directory:

   ```sh
   $ kn func config
   ```

   Alternatively, you can specify the function project directory using the `--path` or `-p` option.

2. Use the interactive interface to perform the necessary operation. For example, using the utility to list configured volumes produces an output similar to this:

   ```sh
   $ kn func config
   ? What do you want to configure? Volumes
   ? What operation do you want to perform? List
   ```
This scheme shows all operations available in the interactive utility and how to navigate to them:

```
kn func config
  └── Environment variables
      ├── Add
      │    └── ConfigMap: Add all key-value pairs from a config map
      │    └── ConfigMap: Add value from a key in a config map
      │    └── Secret: Add all key-value pairs from a secret
      │    └── Secret: Add value from a key in a secret
      │    └── List: List all configured environment variables
      │    └── Remove: Remove a configured environment variable
      ├── Volumes
      │    └── Add
      │    └── ConfigMap: Mount a config map as a volume
      │    └── Secret: Mount a secret as a volume
      │    └── List: List all configured volumes
      │    └── Remove: Remove a configured volume
```

3. Optional. Deploy the function to make the changes take effect:

```
$ kn func deploy -p test
```

9.10.2. Modifying function access to secrets and config maps interactively by using specialized commands

Every time you run the `kn func config` utility, you need to navigate the entire dialogue to select the operation you need, as shown in the previous section. To save steps, you can directly execute a specific operation by running a more specific form of the `kn func config` command:

- To list configured environment variables:
  ```
  $ kn func config envs [-p <function-project-path>]
  ```

- To add environment variables to the function configuration:
  ```
  $ kn func config envs add [-p <function-project-path>]
  ```

- To remove environment variables from the function configuration:
  ```
  $ kn func config envs remove [-p <function-project-path>]
  ```

- To list configured volumes:
  ```
  $ kn func config volumes [-p <function-project-path>]
  ```

- To add a volume to the function configuration:
  ```
  $ kn func config volumes add [-p <function-project-path>]
  ```
To remove a volume from the function configuration:

```
$ kn func config volumes remove [-p <function-project-path>]
```

### 9.10.3. Adding function access to secrets and config maps manually

You can manually add configuration for accessing secrets and config maps to your function.

#### 9.10.3.1. Mounting a secret as a volume

You can use the following procedure to mount a secret as a volume.

**Prerequisites**

- The OpenShift Serverless Operator and Knative Serving are installed on the cluster.
- You have installed the `kn` CLI.
- You have created a function.

**Procedure**

1. Open the `func.yaml` file for your function.

2. For each secret you want to mount as a volume, add the following YAML to the `volumes` section:

   ```yaml
   name: test
   namespace: ""
   runtime: go
   ...
   volumes:
     - secret: mysecret
       path: /workspace/secret
   
   - Substitute **mysecret** with the name of the target secret.
   - Substitute `/workspace/secret` with the path where you want to mount the secret.

3. Save the configuration.

#### 9.10.3.2. Mounting a config map as a volume

You can use the following procedure to mount a config map as a volume.

**Prerequisites**

- The OpenShift Serverless Operator and Knative Serving are installed on the cluster.
- You have installed the `kn` CLI.
- You have created a function.

**Procedure**
1. Open the `func.yaml` file for your function.

2. For each config map you want to mount as a volume, add the following YAML to the `volumes` section:

   ```yaml
   name: test
   namespace: ""
   runtime: go
   ...
   volumes:
     - configMap: myconfigmap
       path: /workspace/configmap
   
   - Substitute `myconfigmap` with the name of the target config map.
   - Substitute `/workspace/configmap` with the path where you want to mount the config map.

3. Save the configuration.

9.10.3.3. Setting environment variable from a key value defined in a secret

You can use the following procedure to set an environment variable from a key value defined as a secret.

**Prerequisites**

- The OpenShift Serverless Operator and Knative Serving are installed on the cluster.
- You have installed the `kn` CLI.
- You have created a function.

**Procedure**

1. Open the `func.yaml` file for your function.

2. For each value from a secret key-value pair that you want to assign to an environment variable, add the following YAML to the `envs` section:

   ```yaml
   name: test
   namespace: ""
   runtime: go
   ...
   envs:
     - name: EXAMPLE
       value: '{{ secret:mysecret:key }}'
   
   - Substitute `EXAMPLE` with the name of the environment variable.
   - Substitute `mysecret` with the name of the target secret.
   - Substitute `key` with the key mapped to the target value.

3. Save the configuration.

9.10.3.4. Setting environment variable from a key value defined in a config map
You can use the following procedure to set an environment variable from a key value defined as a config map.

**Prerequisites**
- The OpenShift Serverless Operator and Knative Serving are installed on the cluster.
- You have installed the `kn` CLI.
- You have created a function.

**Procedure**
1. Open the `func.yaml` file for your function.
2. For each value from a config map key-value pair that you want to assign to an environment variable, add the following YAML to the `envs` section:

```yaml
name: test
namespace: ""
runtime: go
...  
envs:  
- name: EXAMPLE  
  value: '[[ configMap:myconfigmap:key ]]'  
```
   - Substitute `EXAMPLE` with the name of the environment variable.
   - Substitute `myconfigmap` with the name of the target config map.
   - Substitute `key` with the key mapped to the target value.
3. Save the configuration.

### 9.10.3.5. Setting environment variables from all values defined in a secret

You can use the following procedure to set an environment variable from all values defined in a secret.

**Prerequisites**
- The OpenShift Serverless Operator and Knative Serving are installed on the cluster.
- You have installed the `kn` CLI.
- You have created a function.

**Procedure**
1. Open the `func.yaml` file for your function.
2. For every secret for which you want to import all key-value pairs as environment variables, add the following YAML to the `envs` section:

```yaml
name: test
namespace: ""
```

259
runtime: go
...
envs:
- value: '{# secret:mysecret #}'

1 Substitute mysecret with the name of the target secret.

3. Save the configuration.

9.10.3.6. Setting environment variables from all values defined in a config map

You can use the following procedure to set an environment variable from all values defined in a config map.

**Prerequisites**

- The OpenShift Serverless Operator and Knative Serving are installed on the cluster.
- You have installed the kn CLI.
- You have created a function.

**Procedure**

1. Open the func.yaml file for your function.

2. For every config map for which you want to import all key-value pairs as environment variables, add the following YAML to the envs section:

```
name: test
namespace: ""
runtime: go
...
envs:
- value: '{# configMap:myconfigmap #}'

1 Substitute myconfigmap with the name of the target config map.

3. Save the file.

9.11. ADDING ANNOTATIONS TO FUNCTIONS

You can add Kubernetes annotations to a deployed Serverless function by adding them to the annotations section in the func.yaml configuration file.

There are two limitations of the function annotation feature:

- After a function annotation propagates to the corresponding Knative service on the cluster, it cannot be removed from the service by deleting it from the func.yaml file. You must remove the annotation from the Knative service by modifying the YAML file of the service directly, or by using the OpenShift Container Platform web console.

- You cannot set annotations that are set by Knative, for example, the autoscaling annotations.
9.11.1. Adding annotations to a function

You can use the following procedure to add annotations to a function.

Prerequisites

- The OpenShift Serverless Operator and Knative Serving are installed on the cluster.
- You have installed the `kn` CLI.
- You have created a function.

Procedure

1. Open the `func.yaml` file for your function.

2. For every annotation that you want to add, add the following YAML to the `annotations` section:

```yaml
name: test
namespace: ""
runtime: go
...
annotations:
  <annotation_name>: "<annotation_value>"  # 1
```

Substitute `<annotation_name>: "<annotation_value>"` with your annotation.

For example, to indicate that a function was authored by Alice, you might include the following annotation:

```yaml
name: test
namespace: ""
runtime: go
...
annotations:
  author: "alice@example.com"
```

3. Save the configuration.

The next time you deploy your function to the cluster, the annotations are added to the corresponding Knative service.

9.12. FUNCTIONS DEVELOPMENT REFERENCE GUIDE
IMPORTANT

OpenShift Serverless Functions is a Technology Preview feature only. Technology Preview features are not supported with Red Hat production service level agreements (SLAs) and might not be functionally complete. Red Hat does not recommend using them in production. These features provide early access to upcoming product features, enabling customers to test functionality and provide feedback during the development process.

For more information about the support scope of Red Hat Technology Preview features, see https://access.redhat.com/support/offerings/techpreview/.

OpenShift Serverless Functions provides templates that can be used to create basic functions for the following runtimes:

- Node.js
- Python
- Golang
- Quarkus
- TypeScript

This guide provides reference information that you can use to develop functions.

9.12.1. Node.js context object reference

The context object has several properties that can be accessed by the function developer.

9.12.1.1. log

Provides a logging object that can be used to write output to the cluster logs. The log adheres to the Pino logging API.

Example log

```javascript
function handle(context) {
    context.log.info("Processing customer");
}
```

You can access the function by using the kn func emit command to invoke it:

Example command

```
$ kn func emit --sink 'http://example.function.com'
```

Example output

```
{"level":30,"time":1604511655265,"pid":3430203,"hostname":"localhost.localdomain","reqId":1,"msg":"Processing customer"}
```
You can change the log level to one of fatal, error, warn, info, debug, trace, or silent. To do that, change the value of logLevel by assigning one of these values to the environment variable FUNC_LOG_LEVEL using the config command.

9.12.1.2. query

Returns the query string for the request, if any, as key-value pairs. These attributes are also found on the context object itself.

Example query

```javascript
function handle(context) {
  // Log the 'name' query parameter
  context.log.info(context.query.name);
  // Query parameters are also attached to the context
  context.log.info(context.name);
}
```

You can access the function by using the curl command to invoke it:

Example command

```
$ curl http://example.com?name=tiger
```

Example output

```
{"level":30,"time":1604511655265,"pid":3430203,"hostname":"localhost.localdomain","reqId":1,"msg":"tiger"}
```

9.12.1.3. body

Returns the request body if any. If the request body contains JSON code, this will be parsed so that the attributes are directly available.

Example body

```javascript
function handle(context) {
  // log the incoming request body's 'hello' parameter
  context.log.info(context.body.hello);
}
```

You can access the function by using the curl command to invoke it:

Example command

```
$ kn func emit -d '{"Hello": "world"}'
```

Example output

```
{"level":30,"time":1604511655265,"pid":3430203,"hostname":"localhost.localdomain","reqId":1,"msg":"world"}
```
9.12.1.4. headers

Returns the HTTP request headers as an object.

Example header

```javascript
function handle(context) {
  context.log.info(context.headers["custom-header"]);
}
```

You can access the function by using the `kn func emit` command to invoke it:

Example command

```bash
$ kn func emit --sink 'http://example.function.com'
```

Example output

```json
{"level":30,"time":1604511655265,"pid":3430203,"hostname":"localhost.localdomain","reqId":1,"msg":"some-value"}
```

9.12.1.5. HTTP requests

- **method**
  
  Returns the HTTP request method as a string.

- **httpVersion**
  
  Returns the HTTP version as a string.

- **httpVersionMajor**
  
  Returns the HTTP major version number as a string.

- **httpVersionMinor**
  
  Returns the HTTP minor version number as a string.

9.12.2. TypeScript context object reference

The `context` object has several properties that can be accessed by the function developer.

9.12.2.1. log

Provides a logging object that can be used to write output to the cluster logs. The log adheres to the Pino logging API.

Example log

```javascript
export function handle(context: Context): string {
  // log the incoming request body's 'hello' parameter
  if (context.body) {
    context.log.info((context.body as Record<string, string>).hello);
  } else {
    context.log.info('No data received');
  }
}
```
You can access the function by using the `kn func emit` command to invoke it:

**Example command**

```bash
$ kn func emit --sink 'http://example.function.com'
```

**Example output**

```json
{"level":30,"time":1604511655265,"pid":3430203,"hostname":"localhost.localdomain","reqId":1,"msg":"Processing customer"}
```

You can change the log level to one of `fatal`, `error`, `warn`, `info`, `debug`, `trace`, or `silent`. To do that, change the value of `logLevel` by assigning one of these values to the environment variable `FUNC_LOG_LEVEL` using the `config` command.

### 9.12.2.2. query

Returns the query string for the request, if any, as key-value pairs. These attributes are also found on the context object itself.

**Example query**

```typescript
export function handle(context: Context): string {
  // log the 'name' query parameter
  if (context.query) {
    context.log.info((context.query as Record<string, string>).name);
  } else {
    context.log.info('No data received');
  }
  return 'OK';
}
```

You can access the function by using the `kn func emit` command to invoke it:

**Example command**

```bash
$ kn func emit --sink 'http://example.function.com' --data '{"name": "tiger"}'
```

**Example output**

```json
{"level":30,"time":1604511655265,"pid":3430203,"hostname":"localhost.localdomain","reqId":1,"msg":"tiger"}
```

### 9.12.2.3. body

Returns the request body, if any. If the request body contains JSON code, this will be parsed so that the return 'OK';

```json
{"level":30,"time":1604511655265,"pid":3430203,"hostname":"localhost.localdomain","reqId":1,"msg":"tiger"}
```

9.12.2.2. query

Returns the query string for the request, if any, as key-value pairs. These attributes are also found on the context object itself.

**Example query**

```typescript
export function handle(context: Context): string {
  // log the 'name' query parameter
  if (context.query) {
    context.log.info((context.query as Record<string, string>).name);
  } else {
    context.log.info('No data received');
  }
  return 'OK';
}
```

You can access the function by using the `kn func emit` command to invoke it:

**Example command**

```bash
$ kn func emit --sink 'http://example.function.com' --data '{"name": "tiger"}'
```

**Example output**

```json
{"level":30,"time":1604511655265,"pid":3430203,"hostname":"localhost.localdomain","reqId":1,"msg":"tiger"}
```

9.12.2.3. body

- Returns the request body, if any. If the request body contains JSON code, this will be parsed so that the return 'OK';
OpenShift Container Platform 4.9 Serverless

Returns the request body, if any. If the request body contains JSON code, this will be parsed so that the
attributes are directly available.

Example body
export function handle(context: Context): string {
// log the incoming request body's 'hello' parameter
if (context.body) {
context.log.info((context.body as Record<string, string>).hello);
} else {
context.log.info('No data received');
}
return 'OK';
}
You can access the function by using the kn func emit command to invoke it:

Example command
$ kn func emit --sink 'http://example.function.com' --data '{"hello": "world"}'

Example output
{"level":30,"time":1604511655265,"pid":3430203,"hostname":"localhost.localdomain","reqId":1,"msg":"w
orld"}

9.12.2.4. headers
Returns the HTTP request headers as an object.

Example header
export function handle(context: Context): string {
// log the incoming request body's 'hello' parameter
if (context.body) {
context.log.info((context.headers as Record<string, string>)['custom-header']);
} else {
context.log.info('No data received');
}
return 'OK';
}
You can access the function by using the curl command to invoke it:

Example command
$ curl -H'x-custom-header: some-value’' http://example.function.com

Example output
{"level":30,"time":1604511655265,"pid":3430203,"hostname":"localhost.localdomain","reqId":1,"msg":"so
me-value"}

266


9.12.2.5. HTTP requests

method
   Returns the HTTP request method as a string.

httpVersion
   Returns the HTTP version as a string.

httpVersionMajor
   Returns the HTTP major version number as a string.

httpVersionMinor
   Returns the HTTP minor version number as a string.
CHAPTER 10. INTEGRATIONS

10.1. USING NVIDIA GPU RESOURCES WITH SERVERLESS APPLICATIONS

NVIDIA supports experimental use of GPU resources on OpenShift Container Platform. See OpenShift Container Platform on NVIDIA GPU accelerated clusters for more information about setting up GPU resources on OpenShift Container Platform.

10.1.1. Specifying GPU requirements for a service

After GPU resources are enabled for your OpenShift Container Platform cluster, you can specify GPU requirements for a Knative service using the `kn` CLI.

**Prerequisites**

- The OpenShift Serverless Operator, Knative Serving and Knative Eventing are installed on the cluster.
- You have installed the `kn` CLI.
- GPU resources are enabled for your OpenShift Container Platform cluster.
- You have created a project or have access to a project with the appropriate roles and permissions to create applications and other workloads in OpenShift Container Platform.

**NOTE**

Using NVIDIA GPU resources is not supported for IBM Z and IBM Power.

**Procedure**

1. Create a Knative service and set the GPU resource requirement limit to 1 by using the `--limit nvidia.com/gpu=1` flag:

   ```bash
   $ kn service create hello --image <service-image> --limit nvidia.com/gpu=1
   
   A GPU resource requirement limit of 1 means that the service has 1 GPU resource dedicated. Services do not share GPU resources. Any other services that require GPU resources must wait until the GPU resource is no longer in use.

   A limit of 1 GPU also means that applications exceeding usage of 1 GPU resource are restricted. If a service requests more than 1 GPU resource, it is deployed on a node where the GPU resource requirements can be met.

2. Optional. For an existing service, you can change the GPU resource requirement limit to 3 by using the `--limit nvidia.com/gpu=3` flag:

   ```bash
   $ kn service update hello --limit nvidia.com/gpu=3
   
   10.1.2. Additional resources

   - For more information about limits, see Setting resource quotas for extended resources.
CHAPTER 11. CLI TOOLS

11.1. INSTALLING THE KNATIVE CLI

Installation options for the oc CLI may vary depending on your operating system.

The Knative CLI (kn) does not have its own login mechanism. To log in to the cluster, you must install the oc CLI and use the oc login command.

For more information on installing the oc CLI for your operating system and logging in with oc, see the OpenShift CLI getting started documentation.

OpenShift Serverless cannot be installed using the kn CLI. A cluster administrator must install the OpenShift Serverless Operator and set up the Knative components, as described in the Installing the OpenShift Serverless Operator documentation.

IMPORTANT

If you try to use an older version of the Knative kn CLI with a newer OpenShift Serverless release, the API is not found and an error occurs.

For example, if you use the 1.16.0 release of the kn CLI, which uses version 0.22.0, with the 1.17.0 OpenShift Serverless release, which uses the 0.23.0 versions of the Knative Serving and Knative Eventing APIs, the CLI does not work because it continues to look for the outdated 0.22.0 API versions.

Ensure that you are using the latest kn CLI version for your OpenShift Serverless release to avoid issues.

11.1.1. Installing the Knative CLI using the OpenShift Container Platform web console

You can use the following procedure to install the kn CLI by using the web console.

Prerequisites

- You have logged in to the OpenShift Container Platform web console.
- The OpenShift Serverless Operator is installed on your OpenShift Container Platform cluster. After the OpenShift Serverless Operator is installed, you will see a link to download the Knative CLI (kn) for Linux (x86_64, amd64, s390x, ppc64le), macOS, or Windows from the Command Line Tools page in the OpenShift Container Platform web console.

Procedure

1. Download the kn CLI from the Command Line Tools page. You can access the Command Line Tools page by clicking the icon in the top right corner of the web console and selecting Command Line Tools in the drop-down menu.

2. Unpack the archive:

   $ tar -xf <file>

3. Move the kn binary to a directory on your PATH.
4. To check your PATH, run:

   $ echo $PATH

   **NOTE**

   If you do not use RHEL or Fedora, ensure that *libc* is installed in a directory on your library path. If *libc* is not available, you might see the following error when you run CLI commands:

   $ kn: No such file or directory

### 11.1.2. Installing the Knative CLI for Linux using an RPM

For Red Hat Enterprise Linux (RHEL), you can install the kn CLI as an RPM if you have an active OpenShift Container Platform subscription on your Red Hat account.

**Procedure**

1. Enter the command:

   # subscription-manager register

2. Enter the command:

   # subscription-manager refresh

3. Enter the command:

   # subscription-manager attach --pool=<pool_id>  

   1 Pool ID for an active OpenShift Container Platform subscription

4. Enter the command:

   # subscription-manager repos --enable="openshift-serverless-1-for-rhel-8-x86_64-rpms"

5. Enter the command:

   # yum install openshift-serverless-clients

### 11.1.3. Installing the Knative CLI for Linux

For Linux distributions, you can download the kn CLI directly as a *tar.gz* archive.

**Procedure**

1. Download the kn CLI.

2. Unpack the archive:
3. Move the **kn** binary to a directory on your **PATH**.

4. To check your **PATH**, run:

```
$ echo $PATH
```

**NOTE**
If you do not use RHEL or Fedora, ensure that **libc** is installed in a directory on your library path. If **libc** is not available, you might see the following error when you run CLI commands:

```
$ kn: No such file or directory
```

### 11.1.4. Installing the Knative CLI for Linux on IBM Power using an RPM

For Red Hat Enterprise Linux (RHEL), you can install the **kn** CLI as an RPM if you have an active OpenShift Container Platform subscription on your Red Hat account.

**Procedure**

1. Register with a Red Hat Subscription Management (RHSM) service during the firstboot process:

```
# subscription-manager register
```

2. Refresh the RHSM:

```
# subscription-manager refresh
```

3. Attach the subscription to a system by specifying ID of the subscription pool, using the **--pool** option:

```
# subscription-manager attach --pool=<pool_id>
```

1. **Pool ID for an active OpenShift Container Platform subscription**

4. Enable the repository using Red Hat Subscription Manager:

```
# subscription-manager repos --enable="openshift-serverless-1-for-rhel-8-ppc64le-rpms"
```

5. Install the **openshift-serverless-clients** on the system:

```
# yum install openshift-serverless-clients
```

### 11.1.5. Installing the Knative CLI for Linux on IBM Power

For Linux distributions, you can download the Knative CLI (**kn**) directly as a **tar.gz** archive.

**Procedure**

```
$ tar -xf <file>
```
**Procedure**

1. Download the kn CLI.

2. Unpack the archive:

   ```
   $ tar -xf <file>
   ```

3. Move the kn binary to a directory on your PATH.

4. To check your PATH, run:

   ```
   $ echo $PATH
   ```

**NOTE**

If you do not use RHEL, ensure that libc is installed in a directory on your library path.

If libc is not available, you might see the following error when you run CLI commands:

```
$ kn: No such file or directory
```  

### 11.1.6. Installing the Knative CLI for Linux on IBM Z and LinuxONE using an RPM

For Red Hat Enterprise Linux (RHEL), you can install the kn CLI as an RPM if you have an active OpenShift Container Platform subscription on your Red Hat account.

**Procedure**

1. Register with a Red Hat Subscription Management (RHSM) service during the firstboot process:

   ```
   # subscription-manager register
   ```

2. Refresh the RHSM:

   ```
   # subscription-manager refresh
   ```

3. Attach the subscription to a system by specifying ID of the subscription pool, using the `--pool` option:

   ```
   # subscription-manager attach --pool=<pool_id> ¹
   ```

   ¹ Pool ID for an active OpenShift Container Platform subscription

4. Enable the repository using Red Hat Subscription Manager:

   ```
   # subscription-manager repos --enable="openshift-serverless-1-for-rhel-8-s390x-rpms"
   ```

5. Install the openshift-serverless-clients on the system:
11.1.7. Installing the Knative CLI for Linux on IBM Z and LinuxONE

For Linux distributions, you can download the `kn` CLI directly as a `tar.gz` archive.

Procedure

1. Download the `kn` CLI.

2. Unpack the archive:

   ```
   $ tar -xf <file>
   ```

3. Move the `kn` binary to a directory on your `PATH`.

4. To check your `PATH`, run:

   ```
   $ echo $PATH
   ```

   **NOTE**

   If you do not use RHEL, ensure that `libc` is installed in a directory on your library path.

   If `libc` is not available, you might see the following error when you run CLI commands:

   ```
   $ kn: No such file or directory
   ```

11.1.8. Installing the Knative CLI for macOS

The `kn` CLI for macOS is provided as a `tar.gz` archive.

Procedure

1. Download the `kn` CLI.

2. Unpack and unzip the archive.

3. Move the `kn` binary to a directory on your `PATH`.

4. To check your `PATH`, open a terminal window and run:

   ```
   $ echo $PATH
   ```

11.1.9. Installing the Knative CLI for Windows

The `kn` CLI for Windows is provided as a zip archive.

Procedure

```
1. Download the **kn CLI**.

2. Extract the archive with a ZIP program.

3. Move the **kn** binary to a directory on your **PATH**.

4. To check your **PATH**, open the command prompt and run the command:

   ```cmd
   C:\> path
   ```

### 11.2. KNATIVE CLI ADVANCED CONFIGURATION

You can customize and extend the **kn** CLI by using advanced features, such as configuring a **config.yaml** file for **kn** or using plug-ins.

#### 11.2.1. Customizing the Knative CLI

You can customize your **kn** CLI setup by creating a **config.yaml** configuration file. You can provide this configuration by using the **--config** flag, otherwise the configuration is picked up from a default location. The default configuration location conforms to the **XDG Base Directory Specification**, and is different for Unix systems and Windows systems.

For Unix systems:

- If the **XDG_CONFIG_HOME** environment variable is set, the default configuration location that the **kn** CLI looks for is **$XDG_CONFIG_HOME/kn**.

- If the **XDG_CONFIG_HOME** environment variable is not set, the **kn** CLI looks for the configuration in the home directory of the user at **$HOME/.config/kn/config.yaml**.

For Windows systems, the default **kn** CLI configuration location is **%APPDATA%\kn**.

#### Example configuration file

```yaml
plugins:
  path-lookup: true
  directory: ~/.config/kn/plugins

eventing:
  sink-mappings:
    - prefix: svc
      group: core
      version: v1
      resource: services
```

1. Specifies whether the **kn** CLI should look for plug-ins in the **PATH** environment variable. This is a boolean configuration option. The default value is **false**.

2. Specifies the directory where the **kn** CLI will look for plug-ins. The default path depends on the operating system, as described above. This can be any directory that is visible to the user.

3. The **sink-mappings** spec defines the Kubernetes addressable resource that is used when you use the **--sink** flag with a **kn** CLI command.

4. The prefix you want to use to describe your sink. **svc** for a service, **channel**, and **broker** are predefined prefixes in **kn**.
predefined prefixes in **kn**.

5. The API group of the Kubernetes resource.
6. The version of the Kubernetes resource.
7. The plural name of the Kubernetes resource type. For example, **services** or **brokers**.

### 11.2.2. Knative CLI plug-ins

The **kn** CLI supports the use of plug-ins, which enable you to extend the functionality of your **kn** installation by adding custom commands and other shared commands that are not part of the core distribution. **kn** CLI plug-ins are used in the same way as the main **kn** functionality.

Currently, Red Hat supports the **kn-source-kafka** plug-in.
CHAPTER 12. REFERENCE

12.1. KNATIVE CLI FLAGS

This section provides information about additional, optional flags that you can use with kn commands.

12.1.1. Knative CLI --sink flag

When you create an event-producing custom resource by using the Knative (kn) CLI, you can specify a sink where events are sent to from that resource, by using the --sink flag.

The following example creates a sink binding that uses a service, http://event-display.svc.cluster.local, as the sink:

Example command using the --sink flag

```bash
$ kn source binding create bind-heartbeat \\
   --namespace sinkbinding-example \\
   --subject "Job:batch/v1:app=heartbeat-cron" \\
   --sink http://event-display.svc.cluster.local \\
   --ce-override "sink=bound"
```

svc in http://event-display.svc.cluster.local determines that the sink is a Knative service. Other default sink prefixes include channel, and broker.

12.2. KNATIVE SERVING CLI COMMANDS

You can use the following kn CLI commands to complete Knative Serving tasks on the cluster.

12.2.1. kn service commands

You can use the following commands to create and manage Knative services.

12.2.1.1. Creating serverless applications by using the Knative CLI

The following procedure describes how you can create a basic serverless application using the kn CLI.

Prerequisites

- OpenShift Serverless Operator and Knative Serving are installed on your cluster.
- You have installed the kn CLI.
- 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:

  ```bash
  $ kn service create <service-name> --image <image> --env <key=value>
  ```
Example command

```bash
$ kn service create event-display \
   --image quay.io/openshift-knative/knative-eventing-sources-event-display:latest
```

Example output

Creating service 'event-display' in namespace 'default':

0.271s The Route is still working to reflect the latest desired specification.
0.580s Configuration "event-display" is waiting for a Revision to become ready.
3.857s ...
3.861s Ingress has not yet been reconciled.
4.270s Ready to serve.

Service 'event-display' created with latest revision 'event-display-bxshg-1' and URL: http://event-display-default.apps-crc.testing

### 12.2.1.2. Updating serverless applications by using the Knative CLI

You can use the `kn service update` command for interactive sessions on the command line as you build up a service incrementally. In contrast to the `kn service apply` command, when using the `kn service update` command you only have to specify the changes that you want to update, rather than the full configuration for the Knative service.

**Example commands**

- Update a service by adding a new environment variable:
  ```bash
  $ kn service update <service_name> --env <key>=<value>
  ```

- Update a service by adding a new port:
  ```bash
  $ kn service update <service_name> --port 80
  ```

- Update a service by adding new request and limit parameters:
  ```bash
  $ kn service update <service_name> --request cpu=500m --limit memory=1024Mi --limit cpu=1000m
  ```

- Assign the `latest` tag to a revision:
  ```bash
  $ kn service update <service_name> --tag <revision_name>=latest
  ```

- Update a tag from `testing` to `staging` for the latest READY revision of a service:
  ```bash
  $ kn service update <service_name> --untag testing --tag @latest=staging
  ```

- Add the `test` tag to a revision that receives 10% of traffic, and send the rest of the traffic to the latest READY revision of a service:
  ```bash
  $ kn service update <service_name> --tag <revision_name>=test --traffic test=10,@latest=90
  ```
12.2.1.3. Applying service declarations

You can declaratively configure a Knative service by using the `kn service apply` command. If the service does not exist it is created, otherwise the existing service is updated with the options that have been changed.

The `kn service apply` command is especially useful for shell scripts or in a continuous integration pipeline, where users typically want to fully specify the state of the service in a single command to declare the target state.

When using `kn service apply` you must provide the full configuration for the Knative service. This is different from the `kn service update` command, which only requires you to specify in the command the options that you want to update.

Example commands

- Create a service:
  ```
  $ kn service apply <service_name> --image <image>
  ```

- Add an environment variable to a service:
  ```
  $ kn service apply <service_name> --image <image> --env <key>=<value>
  ```

- Read the service declaration from a JSON or YAML file:
  ```
  $ kn service apply <service_name> -f <filename>
  ```

12.2.1.4. Describing serverless applications by using the Knative CLI

You can describe a Knative service by using the `kn service describe` command.

Example commands

- Describe a service:
  ```
  $ kn service describe --verbose <service_name>
  ```

The `--verbose` flag is optional but can be included to provide a more detailed description. The difference between a regular and verbose output is shown in the following examples:

Example output without `--verbose` flag

<table>
<thead>
<tr>
<th>Name</th>
<th>hello</th>
</tr>
</thead>
<tbody>
<tr>
<td>Namespace</td>
<td>default</td>
</tr>
<tr>
<td>Age</td>
<td>2m</td>
</tr>
<tr>
<td>URL</td>
<td><a href="http://hello-default.apps.ocp.example.com">http://hello-default.apps.ocp.example.com</a></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Revisions:</th>
<th>100%</th>
</tr>
</thead>
<tbody>
<tr>
<td>@latest</td>
<td>[1] (2m)</td>
</tr>
<tr>
<td></td>
<td>Image: docker.io/openshift/hello-openshift (pinned to aaea76)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Conditions:</th>
<th>OK TYPE AGE REASON</th>
</tr>
</thead>
</table>

278
Example output with --verbose flag

Name: hello
Namespace: default
Annotations: serving.knative.dev/creator=system:admin
            serving.knative.dev/lastModifier=system:admin
Age: 3m
URL: http://hello-default.apps.ocp.example.com
Cluster: http://hello.default.svc.cluster.local

Revisions:
100% @latest (hello-00001) [1] (3m)
    Image: docker.io/openshift/hello-openshift (pinned to aaea76)
    Env: RESPONSE=Hello Serverless!

Conditions:
OK TYPE    AGE REASON
++ Ready    3m
++ ConfigurationsReady 3m
++ RoutesReady 3m

- Describe a service in YAML format:
  
  $ kn service describe <service_name> -o yaml

- Describe a service in JSON format:
  
  $ kn service describe <service_name> -o json

- Print the service URL only:
  
  $ kn service describe <service_name> -o url

12.2.2. kn container commands

You can use the following commands to create and manage multiple containers in a Knative service spec.

12.2.2.1. Knative client multi-container support

You can use the kn container add command to print YAML container spec to standard output. This command is useful for multi-container use cases because it can be used along with other standard kn flags to create definitions. The kn container add command accepts all container-related flags that are supported for use with the kn service create command. The kn container add command can also be chained by using UNIX pipes (|) to create multiple container definitions at once.

Example commands

- Add a container from an image and print it to standard output:
  
  -
$ kn container add <container_name> --image <image_uri>

Example command

$ kn container add sidecar --image docker.io/example/sidecar

Example output

containers:
  - image: docker.io/example/sidecar
    name: sidecar
    resources: {}

- Chain two `kn container add` commands together, and then pass them to a `kn service create` command to create a Knative service with two containers:

$ kn container add <first_container_name> --image <image_uri> | \
kn container add <second_container_name> --image <image_uri> | \
kn service create <service_name> --image <image_uri> --extra-containers -

`--extra-containers` - specifies a special case where `kn` reads the pipe input instead of a YAML file.

Example command

$ kn container add sidecar --image docker.io/example/sidecar:latest | \ 
kn container add second --image docker.io/example/sidecar:second | \ 
kn service create my-service --image docker.io/example/my-app:latest --extra-containers -

The `--extra-containers` flag can also accept a path to a YAML file:

$ kn service create <service_name> --image <image_uri> --extra-containers <filename>

Example command

$ kn service create my-service --image docker.io/example/my-app:latest --extra-containers my-extra-containers.yaml

12.2.3. kn domain commands

You can use the following commands to create and manage domain mappings.

12.2.3.1. Creating a custom domain mapping by using the Knative CLI

You can use the `kn` CLI to create a DomainMapping custom resource (CR) that maps to an Addressable target CR, such as a Knative service or a Knative route.

The `--ref` flag specifies an Addressable target CR for domain mapping.

If a prefix is not provided when using the `--ref` flag, it is assumed that the target is a Knative service in the current namespace. The examples in the following procedure show the prefixes for mapping to a Knative service or a Knative route.
Prerequisites

- The OpenShift Serverless Operator and Knative Serving are installed on your cluster.
- You have created a Knative service or route, and control a custom domain that you want to map to that CR.

**NOTE**

Your custom domain must point to the DNS of the OpenShift Container Platform cluster.

- You have installed the `kn` CLI tool.
- 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-domain-map --ref example-service
  ```

- Map a domain to a Knative service in a specified namespace:

  ```
  $ kn domain create <domain_mapping_name> --ref <ksvc:service_name:service_namespace>
  ```

  **Example command**

  ```
  $ kn domain create example-domain-map --ref ksvc:example-service:example-namespace
  ```

- Map a domain to a Knative route:

  ```
  $ kn domain create <domain_mapping_name> --ref <kroute:route_name>
  ```

  **Example command**

  ```
  $ kn domain create example-domain-map --ref kroute:example-route
  ```

12.2.3.2. Managing custom domain mappings by using the Knative CLI

After you have created a `DomainMapping` custom resource (CR), you can list existing CRs, view information about an existing CR, update CRs, or delete CRs by using the `kn` CLI.

Prerequisites

- The OpenShift Serverless Operator and Knative Serving are installed on your cluster.
You have created at least one **DomainMapping** CR.

You have installed the **kn** CLI tool.

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**

- List existing **DomainMapping** CRs:
  
  ```
  $ kn domain list -n <domain_mapping_namespace>
  ```

- View details of an existing **DomainMapping** CR:
  
  ```
  $ kn domain describe <domain_mapping_name>
  ```

- Update a **DomainMapping** CR to point to a new target:
  
  ```
  $ kn domain update --ref <target>
  ```

- Delete a **DomainMapping** CR:
  
  ```
  $ kn domain delete <domain_mapping_name>
  ```

### 12.3. KNATIVE EVENTING CLI COMMANDS

You can use the following **kn** CLI commands to complete Knative Eventing tasks on the cluster.

#### 12.3.1. kn source commands

You can use the following commands to list, create, and manage Knative event sources.

##### 12.3.1.1. Listing available event source types by using the Knative CLI

You can list event source types that can be created and used on your cluster by using the **kn source list-types** CLI command.

**Prerequisites**

- The OpenShift Serverless Operator and Knative Eventing are installed on the cluster.

- You have installed the **kn** CLI.

**Procedure**

1. List the available event source types in the terminal:

   ```
   $ kn source list-types
   ```

   **Example output**
Optional: You can also list the available event source types in YAML format:

```
$ kn source list-types -o yaml
```

### 12.3.1.2. Creating and managing container sources by using the Knative CLI

You can use the following `kn` commands to create and manage container sources:

**Create a container source**

```
$ kn source container create <container_source_name> --image <image_uri> --sink <sink>
```

**Delete a container source**

```
$ kn source container delete <container_source_name>
```

**Describe a container source**

```
$ kn source container describe <container_source_name>
```

**List existing container sources**

```
$ kn source container list
```

**List existing container sources in YAML format**

```
$ kn source container list -o yaml
```

**Update a container source**

This command updates the image URI for an existing container source:

```
$ kn source container update <container_source_name> --image <image_uri>
```

### 12.3.1.3. Creating an API server source by using the Knative CLI

You can use the following procedure to create an API server source by using the `kn` CLI.

**Prerequisites**

- The OpenShift Serverless Operator and Knative Eventing are installed on the cluster.
You have created a project or have access to a project with the appropriate roles and permissions to create applications and other workloads in OpenShift Container Platform.

You have installed the `oc` CLI.

You have installed the `kn` CLI.

**PROCEDURE**

If you want to re-use an existing service account, you can modify your existing `ServiceAccount` resource to include the required permissions instead of creating a new resource.

1. Create a service account, role, and role binding for the event source as a YAML file:

```yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  name: events-sa
  namespace: default 1

---
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: event-watcher
  namespace: default 2
rules:
- apiGroups: 
  - ""
  resources:
  - events
  verbs:
  - get
  - list
  - watch

---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: k8s-ra-event-watcher
  namespace: default 3
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: event-watcher
subjects:
- kind: ServiceAccount
  name: events-sa
  namespace: default 4
```

1 2 3 4 Change this namespace to the namespace that you have selected for installing the event source.
2. Apply the YAML file:

   ```bash
   $ oc apply -f <filename>
   ```

3. Create an API server source that uses a broker as a sink:

   ```bash
   $ kn source apiserver create <event_source_name> --sink broker:<broker_name> --resource "event:v1" --service-account <service_account_name> --mode Resource
   ```

4. To check that the API server source is set up correctly, create a Knative service that dumps incoming messages to its log:

   ```bash
   $ kn service create <service_name> --image quay.io/openshift-knative/knative-eventing-sources-event-display:latest
   ```

5. Create a trigger to filter events from the `default` broker to the service:

   ```bash
   $ kn trigger create <trigger_name> --sink ksvc:<service_name>
   ```

6. Create events by launching a pod in the default namespace:

   ```bash
   $ oc create deployment hello-node --image quay.io/openshift-knative/knative-eventing-sources-event-display:latest
   ```

7. Check that the controller is mapped correctly by inspecting the output generated by the following command:

   ```bash
   $ kn source apiserver describe <source_name>
   ```

**Example output**

```
Name:                mysource
Namespace:           default
Annotations:         sources.knative.dev/creator=developer,
sources.knative.dev/lastModifier=developer
Age:                 3m
ServiceAccountName:  events-sa
Mode:                Resource
Sink:
  Name:       default
  Namespace:  default
  Kind:       Broker (eventing.knative.dev/v1)
Resources:
  Kind:       event (v1)
  Controller: false
Conditions:
  OK TYPE                     AGE REASON
    ++ Ready                     3m
    ++ Deployed                  3m
    ++ SinkProvided              3m
    ++ SufficientPermissions     3m
    ++ EventTypesProvided        3m
```
Verification

You can verify that the Kubernetes events were sent to Knative by looking at the message dumper function logs.

1. Get the pods:

   $ oc get pods

2. View the message dumper function logs for the pods:

   $ oc logs $(oc get pod -o name | grep event-display) -c user-container

Example output

```yaml
   cloudevents.Event
   Validation: valid
   Context Attributes,
   specversion: 1.0
   type: dev.knative.apiserver.resource.update
   datacontenttype: application/json
   ...
   Data,
   {
      "apiVersion": "v1",
      "involvedObject": {
         "apiVersion": "v1",
         "fieldPath": "spec.containers{hello-node}"
      },
   }
   "kind": "Event",
   "message": "Started container",
   "metadata": {
      "name": "hello-node.159d7608e3a3572c",
      "namespace": "default",
      ...
   },
   "reason": "Started",
   ...
```

Deleting the API server source

1. Delete the trigger:

   $ kn trigger delete <trigger_name>

2. Delete the event source:

   $ kn source apiserver delete <source_name>
3. Delete the service account, cluster role, and cluster binding:

   $ oc delete -f authentication.yaml

12.3.1.4. Creating a ping source by using the Knative CLI

You can use the following procedure to create a ping source by using the **kn** CLI.

**Prerequisites**

- The OpenShift Serverless Operator, Knative Serving and Knative Eventing are installed on the cluster.
- You have installed the **kn** CLI.
- You have created a project or have access to a project with the appropriate roles and permissions to create applications and other workloads in OpenShift Container Platform.

**Procedure**

1. To verify that the ping source is working, create a simple Knative service that dumps incoming messages to the service logs:

   $ kn service create event-display 
   --image quay.io/openshift-knative/knative-eventing-sources-event-display:latest

2. For each set of ping events that you want to request, create a ping source in the same namespace as the event consumer:

   $ kn source ping create test-ping-source 
   --schedule "/2 * * * *" 
   --data '{"message": "Hello world!"}' 
   --sink ksvc:event-display

3. Check that the controller is mapped correctly by entering the following command and inspecting the output:

   $ kn source ping describe test-ping-source

**Example output**

Name: test-ping-source
Namespace: default
Annotations: sources.knative.dev/creator=developer, sources.knative.dev/lastModifier=developer
Age: 15s
Schedule: */2 * * * *
Data: {"message": "Hello world!"}

Sink:
Name: event-display
Namespace: default
Resource: Service (serving.knative.dev/v1)
Verification

You can verify that the Kubernetes events were sent to the Knative event sink by looking at the logs of the sink pod.

By default, Knative services terminate their pods if no traffic is received within a 60 second period. The example shown in this guide creates a ping source that sends a message every 2 minutes, so each message should be observed in a newly created pod.

1. Watch for new pods created:

   ```
   $ watch oc get pods
   ```

2. Cancel watching the pods using Ctrl+C, then look at the logs of the created pod:

   ```
   $ oc logs $(oc get pod -o name | grep event-display) -c user-container
   ```

Example output

```
cloudevents.Event
Validation: valid
Context Attributes,
   specversion: 1.0
type: dev.knative.sources.ping
source: /apis/v1/namespaces/default/pingsources/test-ping-source
id: 99e4f4f6-08ff-4bff-acf1-47f61ded68c9
time: 2020-04-07T16:16:00.000601161Z
datacontenttype: application/json
Data,
   {
      "message": "Hello world!"
   }
```

Deleting the ping source

- Delete the ping source:

  ```
  $ kn delete pingsources.sources.knative.dev <ping_source_name>
  ```

12.3.1.5. Creating a Kafka event source by using the Knative CLI

This section describes how to create a Kafka event source by using the `kn` command.
Prerequisites

- The OpenShift Serverless Operator, Knative Eventing, Knative Serving, and the `KnativeKafka` custom resource (CR) are installed on your cluster.
- You have created a project or have access to a project with the appropriate roles and permissions to create applications and other workloads in OpenShift Container Platform.
- You have access to a Red Hat AMQ Streams (Kafka) cluster that produces the Kafka messages you want to import.
- You have installed the `kn` CLI.

Procedure

1. To verify that the Kafka event source is working, create a Knative service that dumps incoming events into the service logs:

   ```
   $ kn service create event-display \
   --image quay.io/openshift-knative/knative-eventing-sources-event-display
   ```

2. Create a `KafkaSource` CR:

   ```
   $ kn source kafka create <kafka_source_name> \
   --servers <cluster_kafka_bootstrap>.kafka.svc:9092 \
   --topics <topic_name> --consumergroup my-consumer-group \
   --sink event-display
   ```

   **NOTE**

   Replace the placeholder values in this command with values for your source name, bootstrap servers, and topics.

   The `--servers`, `--topics`, and `--consumergroup` options specify the connection parameters to the Kafka cluster. The `--consumergroup` option is optional.

3. Optional: View details about the `KafkaSource` CR you created:

   ```
   $ kn source kafka describe <kafka_source_name>
   ```

Example output

```
Name:             example-kafka-source
Namespace:        kafka
Age:              1h
BootstrapServers: example-cluster-kafka-bootstrap.kafka.svc:9092
Topics:           example-topic
ConsumerGroup:    example-consumer-group

Sink:
  Name:       event-display
  Namespace:  default
  Resource:   Service (serving.knative.dev/v1)
```
Verification steps

1. Trigger the Kafka instance to send a message to the topic:

   $ oc -n kafka run kafka-producer
     -ti --image=quay.io/strimzi/kafka:latest-kafka-2.7.0 --rm=true 
     --restart=Never -- bin/kafka-console-producer.sh
     --broker-list <cluster_kafka_bootstrap>:9092 --topic my-topic

   Enter the message in the prompt. This command assumes that:
   - The Kafka cluster is installed in the kafka namespace.
   - The KafkaSource object has been configured to use the my-topic topic.

2. Verify that the message arrived by viewing the logs:

   $ oc logs $(oc get pod -o name | grep event-display) -c user-container

Example output

```
> cloudevents.Event
Validation: valid
Context Attributes,
  specversion: 1.0
  type: dev.knative.kafka.event
  source: /apis/v1/namespaces/default/kafkasources/example-kafka-source#example-topic
  subject: partition:46#0
  id: partition:46/offset:0
  time: 2021-03-10T11:21:49.4Z
Extensions,
  traceparent: 00-161ff3815727d8755848ec01c866d1cd-7ff3916c44334678-00
Data,
  Hello!
```

12.4. FUNCTIONS COMMANDS

12.4.1. Creating functions

You can create a basic serverless function using the kn CLI.

You can specify the path, runtime, template, and repository with the template as flags on the command line, or use the -c flag to start the interactive experience in the terminal.

Prerequisites

- The OpenShift Serverless Operator and Knative Serving are installed on the cluster.
You have installed the **kn** CLI.

**Procedure**

- Create a function project:
  
  ```
  $ kn func create -r <repository> -l <runtime> -t <template> <path>
  ```
  
  - Supported runtimes include **node**, **go**, **python**, **quarkus**, and **typescript**.
  - Supported templates include **http** and **events**.

**Example command**

```
$ kn func create -l typescript -t events examplefunc
```

**Example output**

```
Project path:  /home/user/demo/examplefunc
Function name: examplefunc
Runtime:       typescript
Template:      events
Writing events to /home/user/demo/examplefunc
```

- Alternatively, you can specify a repository that contains a custom template.

**Example command**

```
$ kn func create -r https://github.com/boson-project/templates/ -l node -t hello-world examplefunc
```

**Example output**

```
Project path:  /home/user/demo/examplefunc
Function name: examplefunc
Runtime:       node
Template:      hello-world
Writing events to /home/user/demo/examplefunc
```

### 12.4.2. Building functions

Before you can run a function, you must build the function project by using the **kn func build** command. The build command reads the **func.yaml** file from the function project directory to determine the image name and registry.

**Example func.yaml**

```
name: example-function
namespace: default
runtime: node
image: <image_from_registry>
imageDigest: ""
```
If the image name and registry are not set in the `func.yaml` file, you must either specify the registry flag, `-r` when using the `kn func build` command, or you are prompted to provide a registry value in the terminal when building a function. An image name is then derived from the registry value that you have provided.

**Example command using the `-r` registry flag**

```
$ kn func build [-i <image> -r <registry> -p <path>]
```

**Example output**

```
Building function image
Function image has been built, image: quay.io/username/example-function:latest
```

This command creates an OCI container image that can be run locally on your computer, or on a Kubernetes cluster.

**Example using the registry prompt**

```
$ kn func build
A registry for function images is required (e.g. 'quay.io/boson').

Registry for function images: quay.io/username
Building function image
Function image has been built, image: quay.io/username/example-function:latest
```

The values for image and registry are persisted to the `func.yaml` file, so that subsequent invocations do not require the user to specify these again.

**12.4.3. Deploying functions**

You can deploy a function to your cluster as a Knative service by using the `kn func deploy` command.

If the targeted function is already deployed, it is updated with a new container image that is pushed to a container image registry, and the Knative service is updated.

**Prerequisites**

- The OpenShift Serverless Operator and Knative Serving are installed on the cluster.
- You have installed the `kn` CLI.
- You have created a project or have access to a project with the appropriate roles and permissions to create applications and other workloads in OpenShift Container Platform.
- You must have already created and initialized the function that you want to deploy.
Procedure

- Deploy a function:

  $ kn func deploy [-n <namespace> -p <path> -i <image> -r <registry>]

Example output

Function deployed at: http://func.example.com

- If no namespace is specified, the function is deployed in the current namespace.
- The function is deployed from the current directory, unless a path is specified.
- The Knative service name is derived from the project name, and cannot be changed using this command.

12.4.4. Listing existing functions

You can list existing functions by using kn func list. If you want to list functions that have been deployed as Knative services, you can also use kn service list.

Procedure

- List existing functions:

  $ kn func list [-n <namespace> -p <path>]

Example output

<table>
<thead>
<tr>
<th>NAME</th>
<th>NAMESPACE</th>
<th>RUNTIME</th>
<th>URL</th>
</tr>
</thead>
<tbody>
<tr>
<td>example-function</td>
<td>default</td>
<td>node</td>
<td><a href="http://example-function.default.apps.ci-ln-g9f36hb-d5d6b.origin-ci-int-aws.dev.rhcloud.com">http://example-function.default.apps.ci-ln-g9f36hb-d5d6b.origin-ci-int-aws.dev.rhcloud.com</a></td>
</tr>
</tbody>
</table>

- List functions deployed as Knative services:

  $ kn service list -n <namespace>

Example output

<table>
<thead>
<tr>
<th>NAME</th>
<th>URL</th>
</tr>
</thead>
<tbody>
<tr>
<td>example-function</td>
<td><a href="http://example-function.default.apps.ci-ln-g9f36hb-d5d6b.origin-ci-int-aws.dev.rhcloud.com">http://example-function.default.apps.ci-ln-g9f36hb-d5d6b.origin-ci-int-aws.dev.rhcloud.com</a></td>
</tr>
</tbody>
</table>

12.4.5. Describing a function

The kn func info command prints information about a deployed function, such as the function name, image, namespace, Knative service information, route information, and event subscriptions.

Procedure
Describe a function:

```
$ kn func info [-f <format> -n <namespace> -p <path>]
```

Example command

```
$ kn func info -p function/example-function
```

Example output

```
Function name:
  example-function
Function is built in image:
  docker.io/user/example-function:latest
Function is deployed as Knative Service:
  example-function
Function is deployed in namespace:
  default
Routes:
  http://example-function.default.apps.ci-ln-g9f36hb-d5d6b.origin-ci-int-aws.dev.rhcloud.com
```

12.4.6. Emitting a test event to a deployed function

You can use the `kn func emit` CLI command to emit a CloudEvent to a function that is either deployed locally or deployed to your OpenShift Container Platform cluster. This command can be used to test that a function is working and able to receive events correctly.

Example command

```
$ kn func emit
```

The `kn func emit` command executes on the local directory by default, and assumes that this directory is a function project.

12.4.6.1. `kn func emit` optional parameters

You can specify optional parameters for the emitted CloudEvent by using the `kn func emit` CLI command flags.

List of flags from `--help` command output

Flags:
- `-c`, `--content-type` string The MIME Content-Type for the CloudEvent data (Env: $FUNC_CONTENT_TYPE) (default "application/json")
- `-d`, `--data` string Any arbitrary string to be sent as the CloudEvent data. Ignored if --file is provided (Env: $FUNC_DATA)
- `-f`, `--file` string Path to a local file containing CloudEvent data to be sent (Env: $FUNC_FILE)
- `-h`, `--help` help for emit
- `-i`, `--id` string CloudEvent ID (Env: $FUNC_ID) (default "306bd6a0-0b0a-48ba-b187-b633571d072a")
- `-p`, `--path` string Path to the project directory. Ignored when --sink is provided (Env: $FUNC_PATH) (default "/home/lanceball/src/github.com/nodeshift/opossum")
- `-k`, `--sink` string Send the CloudEvent to the function running at [sink]. The special value
In particular, you might find it useful to specify the following parameters:

**Event type**

The type of event being emitted. You can find information about the `type` parameter that is set for events from a certain event producer in the documentation for that event producer. For example, the API server source may set the `type` parameter of produced events as `dev.knative.apiserver.resource.update`.

**Event source**

The unique event source that produced the event. This may be a URI for the event source, for example `https://10.96.0.1/`, or the name of the event source.

**Event ID**

A random, unique ID that is created by the event producer.

**Event data**

Allows you to specify a `data` value for the event sent by the `kn func emit` command. For example, you can specify a `--data` value such as "Hello world!" so that the event contains this data string. By default, no data is included in the events created by `kn func emit`.

**Data content type**

If you are using the `--data` flag to add data for events, you can also specify what type of data is carried by the event, by using the `--content-type` flag. In the previous example, the data is plain text, so you might specify `kn func emit --data "Hello world!" --content-type "text/plain"`.

**Example commands specifying event parameters by using flags**

```bash
$ kn func emit --type <event_type> --source <event_source> --data <event_data> --content-type <content_type> -i <event_ID>

$ kn func emit --type ping --source example-ping --data "Hello world!" --content-type "text/plain" -i example-ID
```

**Example commands specifying a file on disk that contains the event parameters**

```bash
$ kn func emit --file <path>

$ kn func emit --file ./test.json
```
Example commands specifying a path to the function

You can specify a path to the function project by using the `--path` flag, or specify an endpoint for the function by using the `--sink` flag:

```
$ kn func emit --path <path_to_function>
$ kn func emit --path ./example/example-function
```

Example commands specifying a function deployed as a Knative service (sink)

```
$ kn func emit --sink <service_URL>
$ kn func emit --sink "http://example.function.com"
```

The `--sink` flag also accepts the special value `local` to send an event to a function running locally:

```
$ kn func emit --sink local
```

12.4.7. Deleting a function

You can delete a function from your cluster by using the `kn func delete` command.

Procedure

- Delete a function:

  ```
  $ kn func delete [<function_name> -n <namespace> -p <path>]
  ```

  - If the name or path of the function to delete is not specified, the current directory is searched for a `func.yaml` file that is used to determine the function to delete.

  - If the namespace is not specified, it defaults to the `namespace` value in the `func.yaml` file.

12.5. KN EVENT

IMPORTANT

The `kn event` plug-in is a Technology Preview feature only. Technology Preview features are not supported with Red Hat production service level agreements (SLAs) and might not be functionally complete. Red Hat does not recommend using them in production. These features provide early access to upcoming product features, enabling customers to test functionality and provide feedback during the development process.

For more information about the support scope of Red Hat Technology Preview features, see [https://access.redhat.com/support/offerings/techpreview/](https://access.redhat.com/support/offerings/techpreview/).

You can use the `kn event` set of commands to manage cloud events from the command line.

12.5.1. Building events
You can use the builder-like interface of the `kn event build` command to build an event. You can then send that event at a later time or use it in another context.

**Procedure**

- **Build an event:**

  ```bash
  $ kn event build --field <field-name>=<value> --type <type-name> --id <id> --output <format>
  
  where:
  
  - The `--field` flag adds data to the event as a field-value pair. You can use it multiple times.
  - The `--type` flag enables you to specify a string that designates the type of the event.
  - The `--id` flag specifies the ID of the event.
  - You can use the `json` or `yaml` arguments with the `--output` flag to change the output format of the event.
  
  All of these flags are optional.

**Building a simple event**

```bash
$ kn event build -o yaml
```

**Resultant event in the YAML format**

```yaml
data: {}
datacontenttype: application/json
id: 81a402a2-9c29-4c27-b8ed-246a253c9e58
source: kn-event/v0.4.0
specversion: "1.0"
time: "2021-10-15T10:42:57.713226203Z"
type: dev.knative.cli.plugin.event.generic
```

**Building a sample transaction event**

```bash
$ kn event build  
  --field operation.type=local-wire-transfer 
  --field operation.amount=2345.40 
  --field operation.from=87656231 
  --field operation.to=2344121 
  --field automated=true 
  --field signature='FGzCPLvYWdEgsdpb3qXkaVp7Da0=' 
  --type org.example.bank.bar 
  --id $(head -c 10 < /dev/urandom | base64 -w 0) 
  --output json
```

**Resultant event in the JSON format**

```json
{
  "specversion": "1.0",
  "id": "RjtL8UH66X+UJg==",
  "source": "kn-event/v0.4.0",
}
12.5.2. Sending events

You can use the `kn event send` command to send an event. The events can be sent either to publicly available addresses or to addressable resources inside a cluster, such as Kubernetes services, as well as Knative services, brokers, and channels. The command uses the same builder-like interface as the `kn event build` command.

Procedure

- Send an event:

  ```
  $ kn event send --field <field-name>=<value> --type <type-name> --id <id> --to-url <url> --to <cluster-resource> --namespace <namespace>
  
  where:
  ```

  - The `--field` flag adds data to the event as a field-value pair. You can use it multiple times.
  - The `--type` flag enables you to specify a string that designates the type of the event.
  - The `--id` flag specifies the ID of the event.
  - If you are sending the event to a publicly accessible destination, specify the URL using the `--to-url` flag.
  - If you are sending the event to an in-cluster Kubernetes resource, specify the destination using the `--to` flag.
    - Specify the Kubernetes resource using the `<Kind>:<ApiVersion>:<name>` format.
  - The `--namespace` flag specifies the namespace. If omitted, the namespace is taken from the current context.
    All of these flags are optional, except for the destination specification, for which you need to use either `--to-url` or `--to`.

  The following example shows sending an event to a URL:

  **Example command**

  ```
  $ kn event send
  ```
The following example shows sending an event to an in-cluster resource:

**Example command**

```bash
$ kn event send
    --type org.example.kn.ping
    --id $(uuidgen)
    --field event.type=test
    --field event.data=98765
    --to Service:serving.knative.dev/v1:event-display
```

```bash
--field player.id=6354aa60-ddb1-452e-8c13-24893667de20 \
--field player.game=2345 \
--field points=456 \
--type org.example.gaming.foo \
--to-url http://ce-api.foo.example.com/
```