# Red Hat OpenShift Serverless 1.31

## Knative CLI

Overview of CLI commands for Knative Functions, Serving, and Eventing

Last Updated: 2023-12-14

# Red Hat OpenShift Serverless 1.31 Knative CLI

Overview of CLI commands for Knative Functions, Serving, and Eventing

## Legal Notice



## Abstract

This document provides an overview of the CLI commands available for Knative Functions, Serving, and Eventing. It also provides information about configuring the Knative CLI and on using plugins.

# Table of Contents

# CHAPTER 1. KNATIVE SERVING CLI COMMANDS

## 1.1. KN SERVICE COMMANDS

You can use the following commands to create and manage Knative services.

### 1.1.1. Creating serverless applications by using the Knative CLI

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

#### Prerequisites

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

#### Procedure

- Create a Knative service:

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

  Where:

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

  **Example command**

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

  **Example output**

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

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

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

### 1.1.2. 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
```

### 1.1.3. 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>
```

### 1.1.4. 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:       showcase
Namespace:  default
Age:        2m
URL:        http://showcase-default.apps.ocp.example.com

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

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

**Example output with --verbose flag**

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

Revisions:
  100%  @latest (showcase-00001) [1] (3m)
        Image:  quay.io/openshift-knative/showcase (pinned to aaea76)
```

```
      Env:  GREET=Bonjour

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
```

## 1.2. KN SERVICE COMMANDS IN OFFLINE MODE

### 1.2.1. About the Knative CLI offline mode

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. When you create a service in offline mode, no changes happen on the cluster, and instead the service descriptor file is created on your local machine.

IMPORTANT

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

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

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

### 1.2.2. Creating a service using offline mode

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

**IMPORTANT**

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

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

**Prerequisites**

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

**Procedure**

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

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

**Example output**

```
Service 'showcase' created in namespace 'test'.
```

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

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

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

2. Examine the created directory structure:

```
$ tree ./
```

**Example output**

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

2 directories, 1 file
```

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

3. Examine the generated service descriptor file:

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

**Example output**

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

4. List information about the new service:

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

**Example output**

```
Name:       showcase
Namespace:  test
Age:
URL:

Revisions:

Conditions:
  OK TYPE    AGE REASON
```

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

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

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

```
$ kn service create -f test/ksvc/showcase.yaml
```

**Example output**

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

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

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

## 1.3. KN CONTAINER COMMANDS

You can use the following commands to create and manage multiple containers in a Knative service spec.

### 1.3.1. 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: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:

```
$ 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
```

## 1.4. KN DOMAIN COMMANDS

You can use the following commands to create and manage domain mappings.

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

**Prerequisites**

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

NOTE

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

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

**Procedure**

- Map a domain to a CR in the current namespace:

```
$ kn domain create <domain_mapping_name> --ref <target_name>
```

**Example command**

```
$ kn domain create example.com --ref showcase
```

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

If a prefix is not provided when using the **--ref** flag, it is assumed that the target is a Knative service in the current namespace.

- Map a domain to a Knative service in a specified namespace:

```
$ kn domain create <domain_mapping_name> --ref
<ksvc:service_name:service_namespace>
```

**Example command**

```
$ kn domain create example.com --ref ksvc:showcase:example-namespace
```

- Map a domain to a Knative route:

```
$ kn domain create <domain_mapping_name> --ref <kroute:route_name>
```

**Example command**

```
$ kn domain create example.com --ref kroute:example-route
```

### 1.4.2. 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 Knative (**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 Knative (**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>
```

# CHAPTER 2. CONFIGURING THE KNATIVE CLI

You can customize your Knative (**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 Knative (**kn**) CLI looks for is **$XDG_CONFIG_HOME/kn**.
- If the **XDG_CONFIG_HOME** environment variable is not set, the Knative ( **kn**) CLI looks for the configuration in the home directory of the user at **$HOME/.config/kn/config.yaml**.

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

### Example configuration file

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

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

(2) Specifies the directory where the Knative (**kn**) CLI looks for plugins. The default path depends on the operating system, as described previously. 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 Knative ( **kn**) CLI command.

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

(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**.

# CHAPTER 3. KNATIVE CLI PLUGINS

The Knative (**kn**) CLI supports the use of plugins, 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. Knative (**kn**) CLI plugins are used in the same way as the main **kn** functionality.

Currently, Red Hat supports the **kn-source-kafka** plugin and the **kn-event** plugin.

IMPORTANT

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

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

## 3.1. BUILDING EVENTS BY USING THE KN-EVENT PLUGIN

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.

### Prerequisites

- You have installed the Knative (**kn**) CLI.

### Procedure

- Build an event:

```
$ 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**

```
$ kn event build -o yaml
```

    **Resultant event in the YAML format**

```
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**

```
$ 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**

```
{
  "specversion": "1.0",
  "id": "RjtL8UH66X+UJg==",
  "source": "kn-event/v0.4.0",
  "type": "org.example.bank.bar",
  "datacontenttype": "application/json",
  "time": "2021-10-15T10:43:23.113187943Z",
  "data": {
    "automated": true,
    "operation": {
      "amount": "2345.40",
      "from": 87656231,
      "to": 2344121,
      "type": "local-wire-transfer"
    },
    "signature": "FGzCPLvYWdEgsdpb3qXkaVp7Da0="
  }
}
```

## 3.2. SENDING EVENTS BY USING THE KN-EVENT PLUGIN

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.

**Prerequisites**

- You have installed the Knative (**kn**) CLI.

**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 \
    --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/
```

  The following example shows sending an event to an in-cluster resource:

  **Example command**

```
$ 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
```

# CHAPTER 4. KNATIVE EVENTING CLI COMMANDS

## 4.1. KN SOURCE COMMANDS

You can use the following commands to list, create, and manage Knative event sources.

### 4.1.1. 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 Knative (**kn**) CLI.

#### Procedure

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

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

**Example output**

```
TYPE              NAME                                    DESCRIPTION
ApiServerSource   apiserversources.sources.knative.dev     Watch and send Kubernetes API events to a sink
PingSource        pingsources.sources.knative.dev         Periodically send ping events to a sink
SinkBinding       sinkbindings.sources.knative.dev        Binding for connecting a PodSpecable to a sink
```

2. Optional: On OpenShift Container Platform, you can also list the available event source types in YAML format:

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

### 4.1.2. Knative CLI sink flag

When you create an event source 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 sink can be any addressable or callable resource that can receive incoming events from other resources.

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

**Example command using the sink flag**

```
$ kn source binding create bind-heartbeat \
  --namespace sinkbinding-example \
  --subject "Job:batch/v1:app=heartbeat-cron" \
```

```
--sink http://event-display.svc.cluster.local \ 1
--ce-override "sink=bound"
```

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

### 4.1.3. Creating and managing container sources by using the Knative CLI

You can use the **kn source container** commands to create and manage container sources by using the Knative (**kn**) CLI. Using the Knative CLI to create event sources provides a more streamlined and intuitive user interface than modifying YAML files directly.

#### 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>
```

### 4.1.4. Creating an API server source by using the Knative CLI

You can use the **kn source apiserver create** command to create an API server source by using the **kn** CLI. Using the **kn** CLI to create an API server source provides a more streamlined and intuitive user interface than modifying YAML files directly.

#### 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 OpenShift CLI (**oc**).
- You have installed the Knative (**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:

```
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:

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

3. Create an API server source that has an event sink. In the following example, the sink is a broker:

```
$ 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 event-display --image quay.io/openshift-knative/showcase
```

5. If you used a broker as an event sink, create a trigger to filter events from the **default** broker to the service:

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

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

```
$ oc create deployment event-origin --image quay.io/openshift-knative/showcase
```

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
```

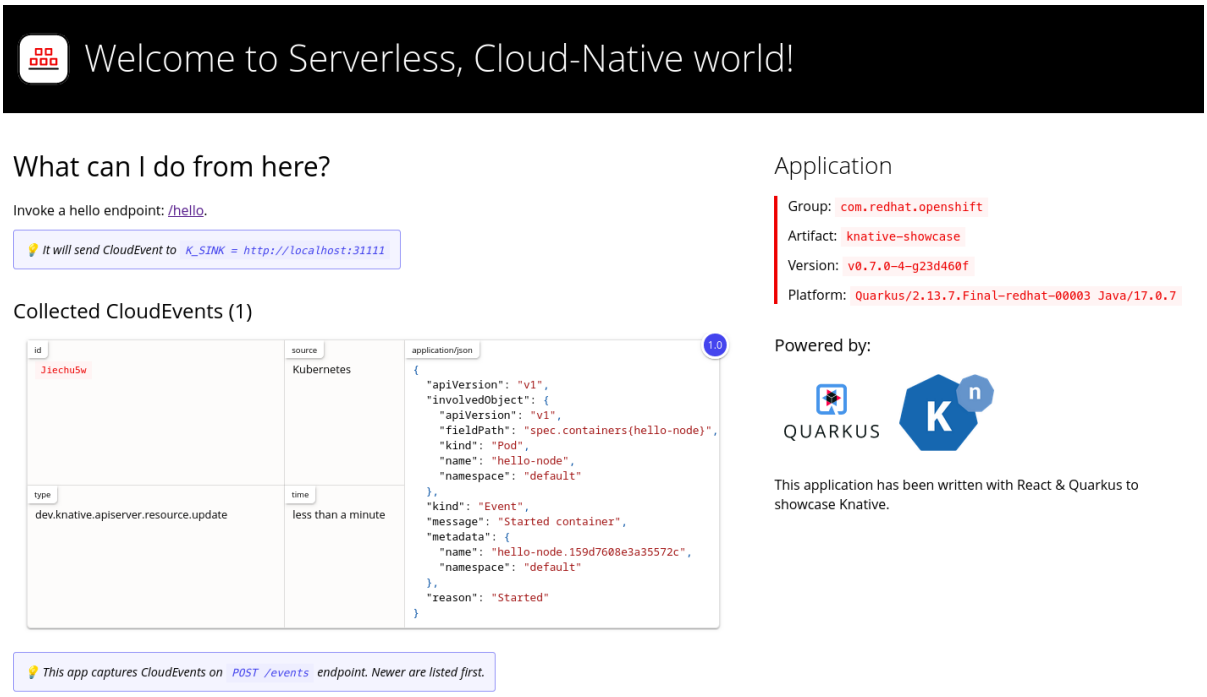
**Verification**

To verify that the Kubernetes events were sent to Knative, look at the event-display logs or use web browser to see the events.

- To view the events in a web browser, open the link returned by the following command:

```
$ kn service describe event-display -o url
```

**Figure 4.1. Example browser page**




- Alternatively, to see the logs in the terminal, view the event-display logs for the pods by entering the following 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.apiserver.resource.update
  datacontenttype: application/json
  ...
Data,
  {
    "apiVersion": "v1",
    "involvedObject": {
      "apiVersion": "v1",
      "fieldPath": "spec.containers{event-origin}",
      "kind": "Pod",
      "name": "event-origin",
      "namespace": "default",
       .....
    },
    "kind": "Event",
```

```
  "message": "Started container",
  "metadata": {
    "name": "event-origin.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
```

### 4.1.5. Creating a ping source by using the Knative CLI

You can use the **kn source ping create** command to create a ping source by using the Knative ( **kn**) CLI. Using the Knative CLI to create event sources provides a more streamlined and intuitive user interface than modifying YAML files directly.

**Prerequisites**

- The OpenShift Serverless Operator, Knative Serving and Knative Eventing are installed on the cluster.
- You have installed the Knative (**kn**) CLI.
- You have created a project or have access to a project with the appropriate roles and permissions to create applications and other workloads in OpenShift Container Platform.
- Optional: If you want to use the verification steps for this procedure, install the OpenShift CLI (**oc**).

**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/showcase
```

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

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

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

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

**Example output**

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

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

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

### Verification

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

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

1. Watch for new pods created:

```
$ watch oc get pods
```

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

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

**Example output**

```
☁  cloudevents.Event
```

```
Validation: valid
Context Attributes,
  specversion: 1.0
  type: dev.knative.sources.ping
  source: /apis/v1/namespaces/default/pingsources/test-ping-source
  id: 99e4f4f6-08ff-4bff-acf1-47f61ded68c9
  time: 2020-04-07T16:16:00.000601161Z
  datacontenttype: application/json
Data,
  {
    "message": "Hello world!"
  }
```

**Deleting the ping source**

- Delete the ping source:

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

### 4.1.6. Creating an Apache Kafka event source by using the Knative CLI

You can use the **kn source kafka create** command to create a Kafka source by using the Knative ( **kn**) CLI. Using the Knative CLI to create event sources provides a more streamlined and intuitive user interface than modifying YAML files directly.

**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 Knative (**kn**) CLI.
- Optional: You have installed the OpenShift CLI (**oc**) if you want to use the verification steps in this procedure.

**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/showcase
```

2. Create a **KafkaSource** CR:

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

```
    --sink event-display
```

NOTE

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

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

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

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

**Example output**

```
Name:              example-kafka-source
Namespace:         kafka
Age:               1h
BootstrapServers:  example-cluster-kafka-bootstrap.kafka.svc:9092
Topics:            example-topic
ConsumerGroup:     example-consumer-group

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

Conditions:
  OK TYPE           AGE REASON
  ++ 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
  subject: partition:46#0
  id: partition:46/offset:0
  time: 2021-03-10T11:21:49.4Z
Extensions,
  traceparent: 00-161ff3815727d8755848ec01c866d1cd-7ff3916c44334678-00
Data,
  Hello!
```

# CHAPTER 5. KNATIVE FUNCTIONS CLI COMMANDS

## 5.1. KN FUNCTIONS COMMANDS

### 5.1.1. Creating a function by using the Knative CLI

You can specify the path, runtime, template, and image registry for a function 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 Knative (**kn**) CLI.

**Procedure**

- Create a function project:

```
$ kn func create -r <repository> -l <runtime> -t <template> <path>
```

  - Accepted runtime values include **quarkus**, **node**, **typescript**, **go**, **python**, **springboot**, and **rust**.
  - Accepted template values include **http** and **cloudevents**.

    **Example command**

```
$ kn func create -l typescript -t cloudevents examplefunc
```

    **Example output**

```
Created typescript function in /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**

```
Created node function in /home/user/demo/examplefunc
```

### 5.1.2. Running a function locally

You can use the **kn func run** command to run a function locally in the current directory or in the directory specified by the **--path** flag. If the function that you are running has never previously been built, or if the project files have been modified since the last time it was built, the **kn func run** command builds the function before running it by default.

#### Example command to run a function in the current directory

```
$ kn func run
```

#### Example command to run a function in a directory specified as a path

```
$ kn func run --path=<directory_path>
```

You can also force a rebuild of an existing image before running the function, even if there have been no changes to the project files, by using the **--build** flag:

#### Example run command using the build flag

```
$ kn func run --build
```

If you set the **build** flag as false, this disables building of the image, and runs the function using the previously built image:

#### Example run command using the build flag

```
$ kn func run --build=false
```

You can use the help command to learn more about **kn func run** command options:

#### Build help command

```
$ kn func help run
```

### 5.1.3. Building a function

Before you can run a function, you must build the function project. If you are using the **kn func run** command, the function is built automatically. However, you can use the **kn func build** command to build a function without running it, which can be useful for advanced users or debugging scenarios.

The **kn func build** command creates an OCI container image that can be run locally on your computer or on an OpenShift Container Platform cluster. This command uses the function project name and the image registry name to construct a fully qualified image name for your function.

#### 5.1.3.1. Image container types

By default, **kn func build** creates a container image by using Red Hat Source-to-Image (S2I) technology.

#### Example build command using Red Hat Source-to-Image (S2I)

```
$ kn func build
```

#### 5.1.3.2. Image registry types

The OpenShift Container Registry is used by default as the image registry for storing function images.

#### Example build command using OpenShift Container Registry

```
$ kn func build
```

**Example output**

```
Building function image
Function image has been built, image: registry.redhat.io/example/example-function:latest
```

You can override using OpenShift Container Registry as the default image registry by using the **--registry** flag:

**Example build command overriding OpenShift Container Registry to use quay.io**

```
$ kn func build --registry quay.io/username
```

**Example output**

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

#### 5.1.3.3. Push flag

You can add the **--push** flag to a **kn func build** command to automatically push the function image after it is successfully built:

**Example build command using OpenShift Container Registry**

```
$ kn func build --push
```

#### 5.1.3.4. Help command

You can use the help command to learn more about **kn func build** command options:

**Build help command**

```
$ kn func help build
```

### 5.1.4. Deploying a function

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

- 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>]
```

**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.

NOTE

You can create a serverless function with a Git repository URL by using **Import from Git** or **Create Serverless Function** in the **+Add** view of the **Developer** perspective.

### 5.1.5. 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**

```
NAME            NAMESPACE  RUNTIME  URL
READY
example-function  default    node     http://example-function.default.apps.ci-ln-g9f36hb-
d5d6b.origin-ci-int-aws.dev.rhcloud.com  True
```

- List functions deployed as Knative services:

```
$ kn service list -n <namespace>
```

**Example output**

```
NAME            URL                                                    LATEST
AGE   CONDITIONS   READY   REASON
example-function   http://example-function.default.apps.ci-ln-g9f36hb-d5d6b.origin-ci-int-
aws.dev.rhcloud.com   example-function-gzl4c   16m   3 OK / 3      True
```

### 5.1.6. 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
```

### 5.1.7. Invoking a deployed function with a test event

You can use the **kn func invoke** CLI command to send a test request to invoke a function either locally or on your OpenShift Container Platform cluster. You can use this command to test that a function is working and able to receive events correctly. Invoking a function locally is useful for a quick test during function development. Invoking a function on the cluster is useful for testing that is closer to the production environment.

**Prerequisites**

- The OpenShift Serverless Operator and Knative Serving are installed on the cluster.
- You have installed the Knative (**kn**) CLI.
- You have created a project or have access to a project with the appropriate roles and permissions to create applications and other workloads in OpenShift Container Platform.
- You must have already deployed the function that you want to invoke.

**Procedure**

- Invoke a function:

```
$ kn func invoke
```

- The **kn func invoke** command only works when there is either a local container image currently running, or when there is a function deployed in the cluster.
- The **kn func invoke** command executes on the local directory by default, and assumes that this directory is a function project.

#### 5.1.7.1. kn func invoke optional parameters

You can specify optional parameters for the request by using the following **kn func invoke** CLI command flags.

| Flags | Description |
|---|---|
| **-t**, **--target** | Specifies the target instance of the invoked function, for example, **local** or **remote** or **https://staging.example.com/**. The default target is **local**. |
| **-f**, **--format** | Specifies the format of the message, for example, **cloudevent** or **http**. |
| **--id** | Specifies a unique string identifier for the request. |
| **-n**, **--namespace** | Specifies the namespace on the cluster. |
| **--source** | Specifies sender name for the request. This corresponds to the CloudEvent **source** attribute. |
| **--type** | Specifies the type of request, for example, **boson.fn**. This corresponds to the CloudEvent **type** attribute. |
| **--data** | Specifies content for the request. For CloudEvent requests, this is the CloudEvent **data** attribute. |
| **--file** | Specifies path to a local file containing data to be sent. |
| **--content-type** | Specifies the MIME content type for the request. |
| **-p**, **--path** | Specifies path to the project directory. |
| **-c**, **--confirm** | Enables prompting to interactively confirm all options. |
| **-v**, **--verbose** | Enables printing verbose output. |
| **-h**, **--help** | Prints information on usage of **kn func invoke**. |

##### 5.1.7.1.1. Main parameters

The following parameters define the main properties of the **kn func invoke** command:

**Event target (-t, --target)**

The target instance of the invoked function. Accepts the **local** value for a locally deployed function, the **remote** value for a remotely deployed function, or a URL for a function deployed to an arbitrary endpoint. If a target is not specified, it defaults to **local**.

**Event message format (-f, --format)**

The message format for the event, such as **http** or **cloudevent**. This defaults to the format of the template that was used when creating the function.

**Event type (--type)**

The type of event that is sent. You can find information about the **type** parameter that is set in the documentation for each event producer. For example, the API server source might set the **type** parameter of produced events as **dev.knative.apiserver.resource.update**.

**Event source (--source)**

The unique event source that produced the event. This might be a URI for the event source, for example **https://10.96.0.1/**, or the name of the event source.

**Event ID (--id)**

A random, unique ID that is created by the event producer.

**Event data (--data)**

Allows you to specify a **data** value for the event sent by the **kn func invoke** 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 invoke**.

NOTE

Functions that have been deployed to a cluster can respond to events from an existing event source that provides values for properties such as **source** and **type**. These events often have a **data** value in JSON format, which captures the domain specific context of the event. By using the CLI flags noted in this document, developers can simulate those events for local testing.

You can also send event data using the **--file** flag to provide a local file containing data for the event. In this case, specify the content type using **--content-type**.

**Data content type (--content-type)**

If you are using the **--data** flag to add data for events, you can use the **--content-type** flag to specify what type of data is carried by the event. In the previous example, the data is plain text, so you might specify **kn func invoke --data "Hello world!" --content-type "text/plain"**.

#### 5.1.7.1.2. Example commands

This is the general invocation of the **kn func invoke** command:

```
$ kn func invoke --type <event_type> --source <event_source> --data <event_data> --content-type
<content_type> --id <event_ID> --format <format> --namespace <namespace>
```

For example, to send a "Hello world!" event, you can run:

```
$ kn func invoke --type ping --source example-ping --data "Hello world!" --content-type "text/plain" --
id example-ID --format http --namespace my-ns
```

##### 5.1.7.1.2.1. Specifying the file with data

To specify the file on disk that contains the event data, use the **--file** and **--content-type** flags:

```
$ kn func invoke --file <path> --content-type <content-type>
```

For example, to send JSON data stored in the **test.json** file, use this command:

```
$ kn func invoke --file ./test.json --content-type application/json
```

##### 5.1.7.1.2.2. Specifying the function project

You can specify a path to the function project by using the **--path** flag:

```
$ kn func invoke --path <path_to_function>
```

For example, to use the function project located in the **./example/example-function** directory, use this command:

```
$ kn func invoke --path ./example/example-function
```

##### 5.1.7.1.2.3. Specifying where the target function is deployed

By default, **kn func invoke** targets the local deployment of the function:

```
$ kn func invoke
```

To use a different deployment, use the **--target** flag:

```
$ kn func invoke --target <target>
```

For example, to use the function deployed on the cluster, use the **--target remote** flag:

```
$ kn func invoke --target remote
```

To use the function deployed at an arbitrary URL, use the **--target <URL>** flag:

```
$ kn func invoke --target "https://my-event-broker.example.com"
```

You can explicitly target the local deployment. In this case, if the function is not running locally, the command fails:

```
$ kn func invoke --target local
```

### 5.1.8. Deleting a function

You can delete a function by using the **kn func delete** command. This is useful when a function is no longer required, and can help to save resources on your cluster.

#### 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.