Red Hat OpenShift Serverless 1.30

Functions

Setting up and using OpenShift Serverless Functions

Last Updated: 2023-09-19

Red Hat OpenShift Serverless 1.30 Functions

Setting up and using OpenShift Serverless Functions

Legal Notice

Copyright © 2023 Red Hat, Inc.

The text of and illustrations in this document are licensed by Red Hat under a Creative Commons Attribution–Share Alike 3.0 Unported license ("CC-BY-SA"). An explanation of CC-BY-SA is available at
http://creativecommons.org/licenses/by-sa/3.0/
. In accordance with CC-BY-SA, if you distribute this document or an adaptation of it, you must provide the URL for the original version.

Red Hat, as the licensor of this document, waives the right to enforce, and agrees not to assert, Section 4d of CC-BY-SA to the fullest extent permitted by applicable law.

Red Hat, Red Hat Enterprise Linux, the Shadowman logo, the Red Hat logo, JBoss, OpenShift, Fedora, the Infinity logo, and RHCE are trademarks of Red Hat, Inc., registered in the United States and other countries.

Linux ® is the registered trademark of Linus Torvalds in the United States and other countries.

Java ® is a registered trademark of Oracle and/or its affiliates.

XFS ® is a trademark of Silicon Graphics International Corp. or its subsidiaries in the United States and/or other countries.

MySQL ® is a registered trademark of MySQL AB in the United States, the European Union and other countries.

Node.js ® is an official trademark of Joyent. Red Hat is not formally related to or endorsed by the official Joyent Node.js open source or commercial project.

The OpenStack ® Word Mark and OpenStack logo are either registered trademarks/service marks or trademarks/service marks of the OpenStack Foundation, in the United States and other countries and are used with the OpenStack Foundation's permission. We are not affiliated with, endorsed or sponsored by the OpenStack Foundation, or the OpenStack community.

All other trademarks are the property of their respective owners.

Abstract

This document provides information on getting started with OpenShift Serverless Functions, and on developing and deploying functions using Quarkus, Node.js, TypeScript, and Python.

Table of Contents

CHAPTER 1. GETTING STARTED WITH FUNCTIONS

Function lifecycle management includes creating and deploying a function, after which it can be invoked. You can do all of these operations on OpenShift Serverless using the **kn func** tool.

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.

- You have the **oc** CLI installed.
- You have the Knative (**kn**) CLI installed. Installing the Knative 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 3.4.7 or higher.
- You have access to an available image registry, such as the OpenShift Container 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.

1.2. CREATING, DEPLOYING, AND INVOKING A FUNCTION

On OpenShift Serverless, you can use the **kn func** to create, deploy, and invoke a function.

Procedure

1. Create a function project:

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

   Example command

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

   Example output

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

2. Navigate to the function project directory:

Example command

```
$ cd examplefunc
```

3. Build and run the function locally:

   Example command

```
$ kn func run
```

4. Deploy the function to your cluster:

```
$ kn func deploy
```

   Example output

```
Function deployed at: http://func.example.com
```

5. Invoke the function:

```
$ kn func invoke
```

   This invokes either a locally or remotely running function. If both are running, the local one is invoked.

1.3. ADDITIONAL RESOURCES FOR OPENSHIFT CONTAINER PLATFORM

- Exposing a default registry manually
- Marketplace page for the Intellij Knative plugin
- Marketplace page for the Visual Studio Code Knative plugin
- Creating applications using the Developer perspective

1.4. NEXT STEPS

- See Using functions with Knative Eventing

CHAPTER 2. CREATING FUNCTIONS

Before you can build and deploy a function, you must create it. You can create functions using the Knative (**kn**) CLI.

2.1. 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
```

2.2. CREATING A FUNCTION IN THE WEB CONSOLE

You can create a function from a Git repository by using the **Developer** perspective of the Red Hat OpenShift Serverless web console.

Prerequisites

- Before you can create a function by using the web console, a cluster administrator must complete the following steps:
  - Install the OpenShift Serverless Operator and Knative Serving on the cluster.
  - Install the OpenShift Pipelines Operator on the cluster.
  - Create the following pipeline tasks so that they are available for all namespaces on the cluster:

    func-s2i task

    ```
    $ oc apply -f https://raw.githubusercontent.com/openshift-knative/kn-plugin-func/serverless-1.30/pkg/pipelines/resources/tekton/task/func-s2i/0.1/func-s2i.yaml
    ```

    func-deploy task

    ```
    $ oc apply -f https://raw.githubusercontent.com/openshift-knative/kn-plugin-func/serverless-1.30/pkg/pipelines/resources/tekton/task/func-s2i/0.1/func-s2i.yaml
    ```

    Node.js function

    ```
    $ oc apply -f https://raw.githubusercontent.com/openshift-knative/kn-plugin-func/serverless-1.30/pkg/pipelines/resources/tekton/task/func-s2i/0.1/func-s2i.yaml
    ```

- You must log into the Red Hat OpenShift Serverless web console.
- You must create a project or have access to a project with the appropriate roles and permissions to create applications and other workloads in Red Hat OpenShift Serverless.
- You must create or have access to a Git repository that contains the code for your function. The repository must contain a **func.yaml** file and use the **s2i** build strategy.

Procedure

1. In the **Developer** perspective, navigate to **+Add → Create Serverless function**. The **Create Serverless function** page is displayed.
2. Enter a **Git Repo URL** that points to the Git repository that contains the code for your function.
3. In the **Pipelines** section:
   a. Select the **Build, deploy and configure a Pipeline Repository** radio button to create a new pipeline for your function.
   b. Select the **Use Pipeline from this cluster** radio button to connect your function to an existing pipeline in the cluster.
4. Click **Create**.

Verification

- After you have created a function, you can view it in the **Topology** view of the **Developer** perspective.

CHAPTER 3. RUNNING FUNCTIONS LOCALLY

You can run a function locally by using the **kn func** tool. This can be useful, for example, for testing the function before deploying it to the cluster.

3.1. 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
```

CHAPTER 4. DEPLOYING FUNCTIONS

You can deploy your functions to the cluster by using the **kn func** tool.

4.1. 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.

CHAPTER 5. BUILDING FUNCTIONS

To run a function, you first must build the function project. This happens automatically when using the **kn func run** command, but you can also build a function without running it.

5.1. 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.

5.1.1. 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
```

5.1.2. 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. 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
```

5.1.4. Help command

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

Build help command

```
$ kn func help build
```

CHAPTER 6. LISTING EXISTING FUNCTIONS

You can list existing functions. You can do it using the **kn func** tool.

6.1. 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
```

CHAPTER 7. INVOKING FUNCTIONS

You can test a deployed function by invoking it. You can do it using the **kn func** tool.

7.1. 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.

CHAPTER 8. DELETING FUNCTIONS

You can delete a function. You can do it using the **kn func** tool.

8.1. 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.

CHAPTER 9. BUILDING AND DEPLOYING FUNCTIONS ON THE CLUSTER

Instead of building a function locally, you can build a function directly on the cluster. When using this workflow on a local development machine, you only need to work with the function source code. This is useful, for example, when you cannot install on-cluster function building tools, such as docker or podman.

9.1. BUILDING AND DEPLOYING A FUNCTION ON THE CLUSTER

You can use the Knative (**kn**) CLI to initiate a function project build and then deploy the function directly on the cluster. To build a function project in this way, the source code for your function project must exist in a Git repository branch that is accessible to your cluster.

Prerequisites

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

Procedure

1. Create the following resources:
   a. Create the **s2i** Tekton task to be able to use Source-to-Image in the pipeline:

      ```
      $ oc apply -f https://raw.githubusercontent.com/openshift-knative/kn-plugin-func/serverless-1.29.0/pkg/pipelines/resources/tekton/task/func-s2i/0.1/func-s2i.yaml
      ```

   b. Create the **kn func** deploy Tekton task to be able to deploy the function in the pipeline:

      ```
      $ oc apply -f https://raw.githubusercontent.com/openshift-knative/kn-plugin-func/serverless-1.29.0/pkg/pipelines/resources/tekton/task/func-deploy/0.1/func-deploy.yaml
      ```

2. Create a function:

   ```
   $ kn func create <function_name> -l <runtime>
   ```

3. Implement the business logic of your function. Then, use Git to commit and push the changes.
4. Deploy your function:

   ```
   $ kn func deploy --remote
   ```

   If you are not logged into the container registry referenced in your function configuration, you are prompted to provide credentials for the remote container registry that hosts the function image:

   #### Example output and prompts

   ```
    Creating Pipeline resources
   ```

```
Please provide credentials for image registry used by Pipeline.
? Server: https://index.docker.io/v1/
? Username: my-repo
? Password: ********
  Function deployed at URL: http://test-function.default.svc.cluster.local
```

5. To update your function, commit and push new changes by using Git, then run the **kn func deploy --remote** command again.

6. Optional. You can configure your function to be built on the cluster after every Git push by using pipelines-as-code:

   a. Generate the Tekton **Pipelines** and **PipelineRuns** configuration for your function:

   ```
   $ kn func config git set
   ```

   Apart from generating configuration files, this command connects to the cluster and validates that the pipeline is installed. By using the token, it also creates, on behalf of the user, a webhook on the function repository. That webhook triggers the pipeline on the cluster every time changes are pushed to the repository.

   You need to have a valid GitHub personal access token with the repository access to use this command.

   b. Commit and push the generated **.tekton/pipeline.yaml** and **.tekton/pipeline-run.yaml** files:

   ```
   $ git add .tekton/pipeline.yaml .tekton/pipeline-run.yaml
   $ git commit -m 'Add the Pipelines and PipelineRuns configuration'
   $ git push
   ```

   c. After you make a change to your function, commit and push it. The function is rebuilt automatically by using the created pipeline.

9.2. SPECIFYING FUNCTION REVISION

When building and deploying a function on the cluster, you must specify the location of the function code by specifying the Git repository, branch, and subdirectory within the repository. You do not need to specify the branch if you use the **main** branch. Similarly, you do not need to specify the subdirectory if your function is at the root of the repository. You can specify these parameters in the **func.yaml** configuration file, or by using flags with the **kn func deploy** command.

Prerequisites

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

Procedure

- Deploy your function:

  ```
  $ kn func deploy --remote \ 1
  ```

```
            --git-url <repo-url> \ (2)
            [--git-branch <branch>] \ (3)
            [--git-dir <function-dir>] (4)
```

1. With the **--remote** flag, the build runs remotely.
2. Substitute **<repo-url>** with the URL of the Git repository.
3. Substitute **<branch>** with the Git branch, tag, or commit. If using the latest commit on the **main** branch, you can skip this flag.
4. Substitute **<function-dir>** with the directory containing the function if it is different than the repository root directory.

For example:

```
$ kn func deploy --remote \
            --git-url https://example.com/alice/myfunc.git \
            --git-branch my-feature \
            --git-dir functions/example-func/
```

CHAPTER 10. CONNECTING AN EVENT SOURCE TO A FUNCTION

Functions are deployed as Knative services on an OpenShift Container Platform cluster. You can connect functions to Knative Eventing components so that they can receive incoming events.

10.1. CONNECT AN EVENT SOURCE TO A FUNCTION USING THE DEVELOPER PERSPECTIVE

Functions are deployed as Knative services on an OpenShift Container Platform cluster. When you create an event source by using the OpenShift Container Platform web console, you can specify a deployed function that events are sent to from that source.

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 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.
- You have created and deployed a function.

Procedure

1. Create an event source of any type, by navigating to **+Add → Event Source** and selecting the event source type that you want to create.
2. In the **Target** section of the **Create Event Source** form view, select your function in the **Resource** list.
3. Click **Create**.

Verification

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

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

CHAPTER 11. FUNCTIONS DEVELOPMENT REFERENCE GUIDE

11.1. DEVELOPING QUARKUS FUNCTIONS

After you have created a Quarkus function project , you can modify the template files provided to add business logic to your function. This includes configuring function invocation and the returned headers and status codes.

11.1.1. Prerequisites

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

11.1.2. Quarkus function template structure

When you create a Quarkus function by using the Knative (**kn**) CLI, the project directory looks similar to a typical Maven project. Additionally, the project contains the **func.yaml** file, which is used for configuring the function.

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

Template structure

```
.
├── func.yaml (1)
├── mvnw
├── mvnw.cmd
├── pom.xml (2)
├── README.md
└── src
    ├── main
    │   ├── java
    │   │   └── functions
    │   │       ├── Function.java (3)
    │   │       ├── Input.java
    │   │       └── Output.java
    │   └── resources
    │       └── application.properties
    └── test
        └── java
            └── functions (4)
                ├── FunctionTest.java
                └── NativeFunctionIT.java
```

(1) Used to determine the image name and registry.

(2) The Project Object Model (POM) file contains project configuration, such as information about dependencies. You can add additional dependencies by modifying this file.

Example of additional dependencies

```
...
<dependencies>
```

```
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.13</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>org.assertj</groupId>
      <artifactId>assertj-core</artifactId>
      <version>3.8.0</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
...
```

Dependencies are downloaded during the first compilation.

3. The function project must contain a Java method annotated with **@Funq**. You can place this method in the **Function.java** class.

4. Contains simple test cases that can be used to test your function locally.

11.1.3. 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 11.1. Function invocation options

Invocation method	Data type contained in the instance	Example of data
HTTP POST request	JSON object in the body of the request	**{ "customerId": "0123456", "productId": "6543210" }**
HTTP GET request	Data in the query string	**?customerId=0123456&productId=6543210**
CloudEvent	JSON object in the **data** property	**{ "customerId": "0123456", "productId": "6543210" }**

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

```
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

```
public class Purchase {
    private long customerId;
    private long productId;
    // getters and setters
}
```

11.1.3.1. Invocation examples

The following example code defines three functions named **withBeans**, **withCloudEvent**, and **withBinary**;

Example

```
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:

```
$ curl "http://localhost:8080/withBeans?message=Hello%20there." -X GET
```

- 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:

```
$ curl "http://localhost:8080/" -X POST \
  -H "Content-Type: application/octet-stream" \
  -H "Ce-SpecVersion: 1.0"\
  -H "Ce-Type: withBinary" \
  -H "Ce-Source: cURL" \
  -H "Ce-Id: 42" \
  --data-binary '@img.jpg'
```

- A **CloudEvent** object in structured encoding:

```
$ curl http://localhost:8080/ \
  -H "Content-Type: application/cloudevents+json" \
  -d "{ \"data_base64\": \"$(base64 --wrap=0 img.jpg)\",
```

```
\"datacontenttype\": \"application/octet-stream\",
\"id\": \"42\",
\"source\": \"curl\",
\"type\": \"withBinary\",
\"specversion\": \"1.0\"}"
```

11.1.4. 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:

```
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();
    }
}
```

11.1.5. Quarkus function return values

Functions can return an instance of any type from the list of permitted types. Alternatively, they can return 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

```
public class Functions {
    @Funq
```

```
    public List<Purchase> getPurchasesByName(String name) {
      // logic to retrieve purchases
    }
}
```

- Invoking this function through an HTTP request produces an HTTP response that contains a list of purchases in the body of the response.
- Invoking this function through an incoming **CloudEvent** object produces a **CloudEvent** response with a list of purchases in the **data** property.

11.1.5.1. Permitted types

The input and output of a function can be any of the **void**, **String**, or **byte[]** types. Additionally, they can be primitive types and their wrappers, for example, **int** and **Integer**. They can also be the following complex objects: Javabeans, maps, lists, arrays, and the special **CloudEvents<T>** type.

Maps, lists, arrays, the **<T>** type parameter of the **CloudEvents<T>** type, and attributes of Javabeans can only be of types listed here.

Example

```
public class Functions {
    public List<Integer> getIds();
    public Purchase[] getPurchasesByName(String name);
    public String getNameById(int id);
    public Map<String,Integer> getNameIdMapping();
    public void processImage(byte[] img);
}
```

11.1.6. Testing Quarkus functions

Quarkus 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 the **src/test/** directory, which contains basic Maven tests. These tests can be extended as needed.

Prerequisites

- You have created a Quarkus function.
- You have installed the Knative (**kn**) CLI.

Procedure

1. Navigate to the project folder for your function.
2. Run the Maven tests:

```
$ ./mvnw test
```

11.1.7. Overriding liveness and readiness probe values

You can override **liveness** and **readiness** probe values for your Quarkus functions. This allows you to configure health checks performed on the function.

Prerequisites

- The OpenShift Serverless Operator and Knative Serving are installed on the cluster.
- You have installed the Knative (**kn**) CLI.
- You have created a function by using **kn func create**.

Procedure

1. Override the **/health/liveness** and **/health/readiness** paths with your own values. You can do this either by changing properties in the function source or by setting the **QUARKUS_SMALLRYE_HEALTH_LIVENESS_PATH** and **QUARKUS_SMALLRYE_HEALTH_READINESS_PATH** environment variables on **func.yaml** file.

   a. To override the paths using the function source, update the path properties in the **src/main/resources/application.properties** file:

   ```
   quarkus.smallrye-health.root-path=/health (1)
   quarkus.smallrye-health.liveness-path=alive (2)
   quarkus.smallrye-health.readiness-path=ready (3)
   ```

   (1) The root path, which is automatically prepended to the **liveness** and **readiness** paths.

   (2) The liveness path, set to **/health/alive** here.

   (3) The readiness path, set to **/health/ready** here.

   b. To override the paths using environment variables, define the path variables in the **build** block of the **func.yaml** file:

   ```
   build:
     builder: s2i
     buildEnvs:
     - name: QUARKUS_SMALLRYE_HEALTH_LIVENESS_PATH
       value: alive (1)
     - name: QUARKUS_SMALLRYE_HEALTH_READINESS_PATH
       value: ready (2)
   ```

   (1) The liveness path, set to **/health/alive** here.

   (2) The readiness path, set to **/health/ready** here.

2. Add the new endpoints to the **func.yaml** file, so that they are properly bound to the container for the Knative service:

   ```
   deploy:
     healthEndpoints:
       liveness: /health/alive
       readiness: /health/ready
   ```

11.1.8. Next steps

- Build and deploy a function.

11.2. DEVELOPING NODE.JS FUNCTIONS

After you have created a Node.js function project , you can modify the template files provided to add business logic to your function. This includes configuring function invocation and the returned headers and status codes.

11.2.1. Prerequisites

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

11.2.2. Node.js function template structure

When you create a Node.js function using the Knative (**kn**) CLI, the project directory looks like a typical Node.js project. The only exception is the additional **func.yaml** file, which is used to configure the function.

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

Template structure

```
.
├── func.yaml (1)
├── index.js (2)
├── package.json (3)
├── README.md
└── test (4)
    ├── integration.js
    └── unit.js
```

1. The **func.yaml** configuration file is used to determine the image name and registry.
2. Your project must contain an **index.js** file which exports a single function.
3. 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.
4. Integration and unit test scripts are provided as part of the function template.

11.2.3. About invoking Node.js functions

When using the Knative (**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.

11.2.3.1. Node.js context objects

Functions are invoked by providing a **context** object as the first parameter. This object provides access to the incoming HTTP request information.

Example context object

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

This information includes 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**.

11.2.3.1.1. 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

```
// 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();
}
```

11.2.3.1.2. 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:

```
{
  "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

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

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

11.2.4. 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

```
function handle(context, customer) {
  // process customer and return a new CloudEvent
  return new CloudEvent({
    source: 'customer.processor',
    type: 'customer.processed'
  })
}
```

11.2.4.1. 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

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

11.2.4.2. 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

```
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

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

11.2.5. 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 Knative (**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
```

11.2.6. Overriding liveness and readiness probe values

You can override **liveness** and **readiness** probe values for your Node.js functions. This allows you to configure health checks performed on the function.

Prerequisites

- The OpenShift Serverless Operator and Knative Serving are installed on the cluster.
- You have installed the Knative (**kn**) CLI.
- You have created a function by using **kn func create**.

Procedure

1. In your function code, create the **Function** object, which implements the following interface:

```
export interface Function {
  init?: () => any; (1)

  shutdown?: () => any; (2)

  liveness?: HealthCheck; (3)

  readiness?: HealthCheck; (4)

  logLevel?: LogLevel;

  handle: CloudEventFunction | HTTPFunction; (5)
}
```

(1) The initialization function, called before the server is started. This function is optional and should be synchronous.

(2) The shutdown function, called after the server is stopped. This function is optional and should be synchronous.

(3) The liveness function, called to check if the server is alive. This function is optional and should return 200/OK if the server is alive.

(4) The readiness function, called to check if the server is ready to accept requests. This function is optional and should return 200/OK if the server is ready.

(5) The function to handle HTTP requests.

For example, add the following code to the **index.js** file:

```
const Function = {

  handle: (context, body) => {
    // The function logic goes here
    return 'function called'
  },

  liveness: () => {
    process.stdout.write('In liveness\n');
    return 'ok, alive';
  }, (1)

  readiness: () => {
    process.stdout.write('In readiness\n');
    return 'ok, ready';
  } (2)
};

Function.liveness.path = '/alive'; (3)
Function.readiness.path = '/ready'; (4)

module.exports = Function;
```

1. Custom **liveness** function.
2. Custom **readiness** function.
3. Custom **liveness** endpoint.
4. Custom **readiness** endpoint.

As an alternative to **Function.liveness.path** and **Function.readiness.path**, you can specify custom endpoints using the **LIVENESS_URL** and **READINESS_URL** environment variables:

```
run:
  envs:
  - name: LIVENESS_URL
    value: /alive (1)
  - name: READINESS_URL
    value: /ready (2)
```

1. The liveness path, set to **/alive** here.
2. The readiness path, set to **/ready** here.

2. Add the new endpoints to the **func.yaml** file, so that they are properly bound to the container for the Knative service:

```
deploy:
  healthEndpoints:
    liveness: /alive
    readiness: /ready
```

11.2.7. Node.js context object reference

The **context** object has several properties that can be accessed by the function developer. Accessing these properties can provide information about HTTP requests and write output to the cluster logs.

11.2.7.1. 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

```
function handle(context) {
  context.log.info("Processing customer");
}
```

You can access the function by using the **kn func invoke** command:

Example command

```
$ kn func invoke --target 'http://example.function.com'
```

Example output


```
{"level":30,"time":1604511655265,"pid":3430203,"hostname":"localhost.localdomain","reqId":1,"msg":"Pr
ocessing 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.

11.2.7.2. query

Returns the query string for the request, if any, as key-value pairs. These attributes are also found on the context object itself.

Example query

```
function handle(context) {
  // Log the 'name' query parameter
  context.log.info(context.query.name);
  // Query parameters are also attached to the context
  context.log.info(context.name);
}
```

You can access the function by using the **kn func invoke** command:

Example command

```
$ kn func invoke --target 'http://example.com?name=tiger'
```

Example output


```
{"level":30,"time":1604511655265,"pid":3430203,"hostname":"localhost.localdomain","reqId":1,"msg":"tig
er"}
```


11.2.7.3. body

Returns the request body if any. If the request body contains JSON code, this will be parsed so that the attributes are directly available.

Example body

```
function handle(context) {
  // log the incoming request body's 'hello' parameter
  context.log.info(context.body.hello);
}
```

You can access the function by using the **curl** command to invoke it:

Example command

```
$ kn func invoke -d '{"Hello": "world"}'
```

Example output

```
{"level":30,"time":1604511655265,"pid":3430203,"hostname":"localhost.localdomain","reqId":1,"msg":"w
orld"}
```

11.2.7.4. headers

Returns the HTTP request headers as an object.

Example header

```
function handle(context) {
  context.log.info(context.headers["custom-header"]);
}
```

You can access the function by using the **kn func invoke** command:

Example command

```
$ kn func invoke --target 'http://example.function.com'
```

Example output

```
{"level":30,"time":1604511655265,"pid":3430203,"hostname":"localhost.localdomain","reqId":1,"msg":"so
me-value"}
```

11.2.7.5. 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.

11.2.8. Next steps

- Build and deploy a function.

11.3. DEVELOPING TYPESCRIPT FUNCTIONS

After you have created a TypeScript function project , you can modify the template files provided to add business logic to your function. This includes configuring function invocation and the returned headers and status codes.

11.3.1. Prerequisites

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

11.3.2. TypeScript function template structure

When you create a TypeScript function using the Knative (**kn**) CLI, the project directory looks like a typical TypeScript project. The only exception is the additional **func.yaml** file, which is used for configuring the function.

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

Template structure

```
.
├── func.yaml (1)
├── package.json (2)
├── package-lock.json
├── README.md
├── src
│   └── index.ts (3)
├── test (4)
│   ├── integration.ts
│   └── unit.ts
└── tsconfig.json
```

1. The **func.yaml** configuration file is used to determine the image name and registry.
2. 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.
3. Your project must contain an **src/index.js** file which exports a function named **handle**.
4. Integration and unit test scripts are provided as part of the function template.

11.3.3. About invoking TypeScript functions

When using the Knative (**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.

11.3.3.1. TypeScript context objects

To invoke a function, you provide a **context** object as the first parameter. Accessing properties of the **context** object can provide information about the incoming HTTP request.

Example context object

```
function handle(context:Context): string
```

This information includes 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**.

11.3.3.1.1. 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

```
// 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();
}
```

11.3.3.1.2. Context types

The TypeScript type definition files export the following types for use in your functions.

Exported type definitions

```
// Invokable is the expeted 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;
    version(version: string): CloudEventResponse;
    response(): CloudEvent;
}
```

11.3.3.1.3. 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:

```
{
  "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

```
function handle(context: Context, cloudevent?: CloudEvent): CloudEvent
```

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

11.3.4. 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

```
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'
    })
  );
};
```

11.3.4.1. 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

```
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 } };
}
```

11.3.4.2. 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

```
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

```
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;
  }
}
```

11.3.5. 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 Knative (**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
   ```

11.3.6. Overriding liveness and readiness probe values

You can override **liveness** and **readiness** probe values for your TypeScript functions. This allows you to configure health checks performed on the function.

Prerequisites

- The OpenShift Serverless Operator and Knative Serving are installed on the cluster.
- You have installed the Knative (**kn**) CLI.
- You have created a function by using **kn func create**.

Procedure

1. In your function code, create the **Function** object, which implements the following interface:

```
export interface Function {
  init?: () => any; (1)

  shutdown?: () => any; (2)

  liveness?: HealthCheck; (3)

  readiness?: HealthCheck; (4)

  logLevel?: LogLevel;

  handle: CloudEventFunction | HTTPFunction; (5)
}
```

(1) The initialization function, called before the server is started. This function is optional and should be synchronous.

(2) The shutdown function, called after the server is stopped. This function is optional and should be synchronous.

(3) The liveness function, called to check if the server is alive. This function is optional and should return 200/OK if the server is alive.

(4) The readiness function, called to check if the server is ready to accept requests. This function is optional and should return 200/OK if the server is ready.

(5) The function to handle HTTP requests.

For example, add the following code to the **index.js** file:

```
const Function = {

  handle: (context, body) => {
    // The function logic goes here
    return 'function called'
  },

  liveness: () => {
    process.stdout.write('In liveness\n');
    return 'ok, alive';
  }, (1)

  readiness: () => {
    process.stdout.write('In readiness\n');
    return 'ok, ready';
  } (2)
};

Function.liveness.path = '/alive'; (3)
```

```
Function.readiness.path = '/ready'; (4)

module.exports = Function;
```

1. Custom **liveness** function.
2. Custom **readiness** function.
3. Custom **liveness** endpoint.
4. Custom **readiness** endpoint.

As an alternative to **Function.liveness.path** and **Function.readiness.path**, you can specify custom endpoints using the **LIVENESS_URL** and **READINESS_URL** environment variables:

```
run:
  envs:
  - name: LIVENESS_URL
    value: /alive (1)
  - name: READINESS_URL
    value: /ready (2)
```

1. The liveness path, set to **/alive** here.
2. The readiness path, set to **/ready** here.

2. Add the new endpoints to the **func.yaml** file, so that they are properly bound to the container for the Knative service:

```
deploy:
  healthEndpoints:
    liveness: /alive
    readiness: /ready
```

11.3.7. TypeScript context object reference

The **context** object has several properties that can be accessed by the function developer. Accessing these properties can provide information about incoming HTTP requests and write output to the cluster logs.

11.3.7.1. 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

```
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 invoke** command:

Example command

```
$ kn func invoke --target 'http://example.function.com'
```

Example output

```
{"level":30,"time":1604511655265,"pid":3430203,"hostname":"localhost.localdomain","reqId":1,"msg":"Pr
ocessing 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.

11.3.7.2. 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

```
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 invoke** command:

Example command

```
$ kn func invoke --target 'http://example.function.com' --data '{"name": "tiger"}'
```

Example output

```
{"level":30,"time":1604511655265,"pid":3430203,"hostname":"localhost.localdomain","reqId":1,"msg":"tig
er"}
{"level":30,"time":1604511655265,"pid":3430203,"hostname":"localhost.localdomain","reqId":1,"msg":"tig
er"}
```

11.3.7.3. 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

```
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 invoke** command:

Example command

```
$ kn func invoke --target 'http://example.function.com' --data '{"hello": "world"}'
```

Example output

```
{"level":30,"time":1604511655265,"pid":3430203,"hostname":"localhost.localdomain","reqId":1,"msg":"w
orld"}
```

11.3.7.4. 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"}
```

11.3.7.5. 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.

11.3.8. Next steps

- Build and deploy a function.
- See the Pino API documentation for more information about logging with functions.

11.4. DEVELOPING PYTHON FUNCTIONS

IMPORTANT

OpenShift Serverless Functions with Python 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 you have created a Python function project , you can modify the template files provided to add business logic to your function. This includes configuring function invocation and the returned headers and status codes.

11.4.1. Prerequisites

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

11.4.2. Python function template structure

When you create a Python function by using the Knative (**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.

11.4.3. 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

```
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
```

11.4.4. Python function return values

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

Example

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

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

11.4.4.1. 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

```
@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.

11.4.5. Testing Python functions

You can test Python functions locally on your computer. The default project contains a **test_func.py** file, which provides a simple unit test for functions.

NOTE

The default test framework for Python functions is **unittest**. You can use a different test framework if you prefer.

Prerequisites

- To run Python functions tests locally, you must install the required dependencies:

```
$ pip install -r requirements.txt
```

Procedure

1. Navigate to the folder for your function that contains the **test_func.py** file.
2. Run the tests:

```
$ python3 test_func.py
```

11.4.6. Next steps

- Build and deploy a function.

CHAPTER 12. CONFIGURING FUNCTIONS

12.1. ACCESSING SECRETS AND CONFIG MAPS FROM FUNCTIONS USING CLI

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.

12.1.1. 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. The available operations include listing, adding, and removing values stored in config maps and secrets as environment variables, as well as listing, adding, and removing volumes. This functionality enables you to manage what data stored on the cluster is accessible by your function.

Prerequisites

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

Procedure

1. Run the following command in the function project directory:

```
$ kn func config
```

   Alternatively, you can specify the function project directory using the **--path** or **-p** option.

2. Use the interactive interface to perform the necessary operation. For example, using the utility to list configured volumes produces an output similar to this:

```
$ kn func config
? What do you want to configure? Volumes
? What operation do you want to perform? List
Configured Volumes mounts:
- Secret "mysecret" mounted at path: "/workspace/secret"
- Secret "mysecret2" mounted at path: "/workspace/secret2"
```

   This scheme shows all operations available in the interactive utility and how to navigate to them:

```
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
```

12.1.2. 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>]
```

12.2. CONFIGURING YOUR FUNCTION PROJECT USING THE FUNC.YAML FILE

The **func.yaml** file contains the configuration for your function project. Values specified in **func.yaml** are used when you execute a **kn func** command. For example, when you run the **kn func build** command, the value in the **build** field is used. In some cases, you can override these values with command line flags or environment variables.

12.2.1. Referencing local environment variables from func.yaml fields

If you want to avoid storing sensitive information such as an API key in the function configuration, you can add a reference to an environment variable available in the local environment. You can do this by modifying the **envs** field in the **func.yaml** file.

Prerequisites

- You need to have the function project created.
- The local environment needs to contain the variable that you want to reference.

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

  ```
  name: test
  namespace: ""
  runtime: go
  ...
  envs:
  - name: MY_API_KEY
    value: '{{ env:API_KEY }}'
  ...
  ```

12.2.2. Adding annotations to functions

You can add Kubernetes annotations to a deployed Serverless function. Annotations enable you to attach arbitrary metadata to a function, for example, a note about the function’s purpose. Annotations are added to the **annotations** section of 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.

12.2.3. Adding annotations to a function

You can add annotations to a function. Similar to a label, an annotation is defined as a key-value map. Annotations are useful, for example, for providing metadata about a function, such as the function's author.

Prerequisites

- The OpenShift Serverless Operator and Knative Serving are installed on the cluster.
- You have installed the Knative (**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:

```
name: test
namespace: ""
runtime: go
...
annotations:
  <annotation_name>: "<annotation_value>" (1)
```

(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:

```
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.

12.2.4. Additional resources

- Getting started with functions
- Knative documentation on Autoscaling

- Kubernetes documentation on managing resources for containers
- Knative documentation on configuring concurrency

12.2.5. Adding function access to secrets and config maps manually

You can manually add configuration for accessing secrets and config maps to your function. This might be preferable to using the **kn func config** interactive utility and commands, for example when you have an existing configuration snippet.

12.2.5.1. Mounting a secret as a volume

You can mount a secret as a volume. Once a secret is mounted, you can access it from the function as a regular file. This enables you to store on the cluster data needed by the function, for example, a list of URIs that need to be accessed by the function.

Prerequisites

- The OpenShift Serverless Operator and Knative Serving are installed on the cluster.
- You have installed the Knative (**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:

```
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.
  For example, to mount the **addresses** secret, use the following YAML:

```
name: test
namespace: ""
runtime: go
...
volumes:
- configMap: addresses
  path: /workspace/secret-addresses
```

3. Save the configuration.

12.2.5.2. Mounting a config map as a volume

You can mount a config map as a volume. Once a config map is mounted, you can access it from the function as a regular file. This enables you to store on the cluster data needed by the function, for example, a list of URIs that need to be accessed by the function.

Prerequisites

- The OpenShift Serverless Operator and Knative Serving are installed on the cluster.
- You have installed the Knative (**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:

```
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. For example, to mount the **addresses** config map, use the following YAML:

```
name: test
namespace: ""
runtime: go
...
volumes:
- configMap: addresses
  path: /workspace/configmap-addresses
```

3. Save the configuration.

12.2.5.3. Setting environment variable from a key value defined in a secret

You can set an environment variable from a key value defined as a secret. A value previously stored in a secret can then be accessed as an environment variable by the function at runtime. This can be useful for getting access to a value stored in a secret, such as the ID of a user.

Prerequisites

- The OpenShift Serverless Operator and Knative Serving are installed on the cluster.
- You have installed the Knative (**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:

```
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.
  For example, to access the user ID that is stored in **userdetailssecret**, use the following YAML:

```
name: test
namespace: ""
runtime: go
...
envs:
- value: '{{ configMap:userdetailssecret:userid }}'
```

3. Save the configuration.

12.2.5.4. Setting environment variable from a key value defined in a config map

You can set an environment variable from a key value defined as a config map. A value previously stored in a config map can then be accessed as an environment variable by the function at runtime. This can be useful for getting access to a value stored in a config map, such as the ID of a user.

Prerequisites

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

- You have installed the Knative (**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:

```
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.
  For example, to access the user ID that is stored in **userdetailsmap**, use the following YAML:

  ```
  name: test
  namespace: ""
  runtime: go
  ...
  envs:
  - value: '{{ configMap:userdetailsmap:userid }}'
  ```

3. Save the configuration.

12.2.5.5. Setting environment variables from all values defined in a secret

You can set an environment variable from all values defined in a secret. Values previously stored in a secret can then be accessed as environment variables by the function at runtime. This can be useful for simultaneously getting access to a collection of values stored in a secret, for example, a set of data pertaining to a user.

Prerequisites

- The OpenShift Serverless Operator and Knative Serving are installed on the cluster.
- You have installed the Knative (**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:

   ```
   name: test
   namespace: ""
   runtime: go
   ...
   envs:
   - value: '{{ secret:mysecret }}' (1)
   ```

1. Substitute **mysecret** with the name of the target secret.

For example, to access all user data that is stored in **userdetailssecret**, use the following YAML:

```
name: test
namespace: ""
runtime: go
...
envs:
- value: '{{ configMap:userdetailssecret }}'
```

3. Save the configuration.

12.2.5.6. Setting environment variables from all values defined in a config map

You can set an environment variable from all values defined in a config map. Values previously stored in a config map can then be accessed as environment variables by the function at runtime. This can be useful for simultaneously getting access to a collection of values stored in a config map, for example, a set of data pertaining to a user.

Prerequisites

- The OpenShift Serverless Operator and Knative Serving are installed on the cluster.
- You have installed the Knative (**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)
```

1. Substitute **myconfigmap** with the name of the target config map.

For example, to access all user data that is stored in **userdetailsmap**, use the following YAML:

```
name: test
namespace: ""
runtime: go
...
envs:
- value: '{{ configMap:userdetailsmap }}'
```

3. Save the file.

12.3. CONFIGURABLE FIELDS IN FUNC.YAML

You can configure some of the **func.yaml** fields.

12.3.1. 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.

12.3.1.1. 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:

```
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:

```
buildEnvs:
- name: EXAMPLE1
  value: '{{ env:LOCAL_ENV_VAR }}'
```

12.3.1.2. 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:

```
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.

12.3.1.3. builder

The **builder** field specifies the strategy used by the function to build the image. It accepts values of **pack** or **s2i**.

12.3.1.4. build

The **build** field indicates how the function should be built. The value **local** indicates that the function is built locally on your machine. The value **git** indicates that the function is built on a cluster by using the values specified in the **git** field.

12.3.1.5. 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:

```
name: test
namespace: ""
runtime: go
...
volumes:
- secret: mysecret (1)
  path: /workspace/secret
- configMap: myconfigmap (2)
  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**.

12.3.1.6. 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:

```
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
```

12.3.1.7. 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.

12.3.1.8. imageDigest

The **imageDigest** field contains the SHA256 hash of the image manifest when the function is deployed. Do not modify this value.

12.3.1.9. 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**:

```
labels:
- key: role
  value: 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 }}'
```

12.3.1.10. 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.

12.3.1.11. namespace

The **namespace** field specifies the namespace in which your function is deployed.

12.3.1.12. runtime

The **runtime** field specifies the language runtime for your function, for example, **python**.