OpenFaaS Basic Documentation (Linux)

Prerequisites

* Kubernetes cluster installed and *kubectl* CLI
* Docker installed with a DockerHub account

Installation

These instructions are meant for **installing the OpenFaaS CLI and deploying it on top of a Kubernetes cluster**. Note that this process is specific to a local Kubernetes cluster on Minikube, and processes may vary for other clusters depending on the provider.

1. Install the OpenFaaS CLI using *curl*.

curl -sSL https://cli.openfaas.com | sudo -E sh

If the process ran successfully, you will see output indicating that *faas-cli* has been installed. The result should look similar to the following picture:

1. Install OpenFaaS onto a Kubernetes cluster with a tool called *arkade* that helps integrate the two for us.

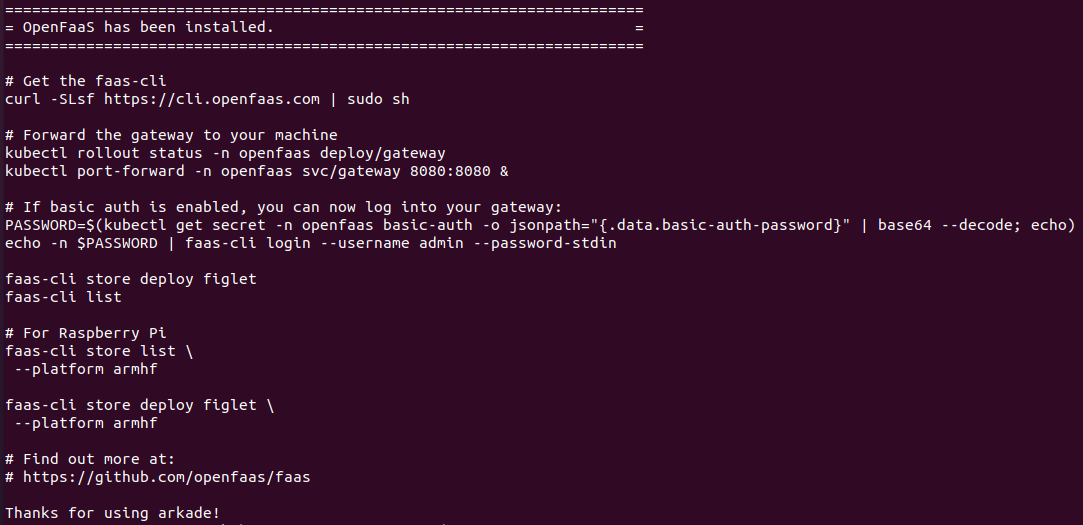
curl -SLsf https://dl.get-arkade.dev/ | sudo sh

Again, if the tool was properly installed, you should see a confirmation output as shown below:

1. Next, use the tool to install OpenFaaS onto our Kubernetes cluster. This process depends on your type of Kubernetes cluster.
   1. If using a 3rd party managed, cloud provided, or a cluster that supplies load-balancers:
   2. If using a local cluster or VM:

arkade install openfaas --load-balancer

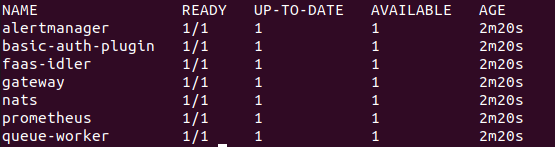
arkade install openfaas

If the installation occurred properly, you should see a success message along with a set of instructions for moving forward, which we will later use to load our saved function states.

1. Lastly, to ensure the installation had no errors, we need to check our Kubernetes cluster to see if the newly installed OpenFaaS deployments are running.

kubectl -n openfaas get deployments -l "release=openfaas, app=openfaas"

Your output should be as follows. Note that it may take a few minutes for all the deployments to become ready. If there is an error, reference the official OpenFaaS documentation for Kubernetes deployment at <https://docs.openfaas.com/deployment/kubernetes/>.



Accessing the OpenFaaS Gateway

OpenFaaS **requires basic authentication for accessing your functions** upon re-entry, so you must load your state every time you need to use it again after exiting or logging out. To load your OpenFaaS environment, follow the next steps:

1. For the functions to properly be stored, accessed, and called upon, **an API gateway must be setup**. Thus, verify that this gateway was pre-installed when we deployed OpenFaaS onto Kubernetes.

kubectl rollout status -n openfaas deploy/gateway

The output should say ‘*deployment "gateway" successfully rolled out*’.

1. Because the gateway is deployed on top of our Kubernetes cluster, we do not have the access permission to immediately step into it. To overcome this, open a ‘tunnel’ from the cluster to your local machine with port forwarding.

kubectl port-forward -n openfaas svc/gateway 8080:8080 &

1. Now that the gateway is set, you need a password to login. The username is always *“admin”*, but the password is stored in a *secret* and must be indirectly accessed.

PASSWORD=$(kubectl get secret -n openfaas basic-auth -o jsonpath="{.data.basic-auth-password}" | base64 --decode; echo)

1. Login to the gateway and store the password information for later usage.

echo -n $PASSWORD | faas-cli login --username admin --password-stdin

You can now create, access, and deploy functions through the OpenFaaS CLI. Try typing faas-cli in the terminal for different CLI commands. Note that these steps must be repeated **every time you access OpenFaaS after exiting the current environment** you are in (i.e. after logging out of your machine, stopping your Kubernetes cluster, etc.).

Accessing the OpenFaaS UI

OpenFaaS provides a breadth of different methods for interacting with functions. After logging into the gateway, you have opened access to use the CLI to create functions. In addition, this also allows you to **view, create, and deploy functions through the UI**.

1. Get the OpenFaaS gateway URL
   1. If you are using a local or on-premise Kubernetes cluster, your URL is set to <http://127.0.0.1:8080> or <http://localhost:8080> by default.
   2. If you are using a 3rd party cloud cluster, get the external IP or DNS entry of your LoadBalancer.

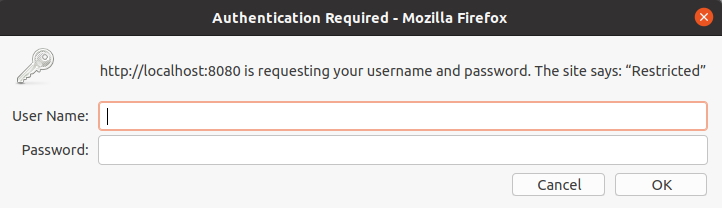
kubectl get svc -o wide gateway-external -n openfaas

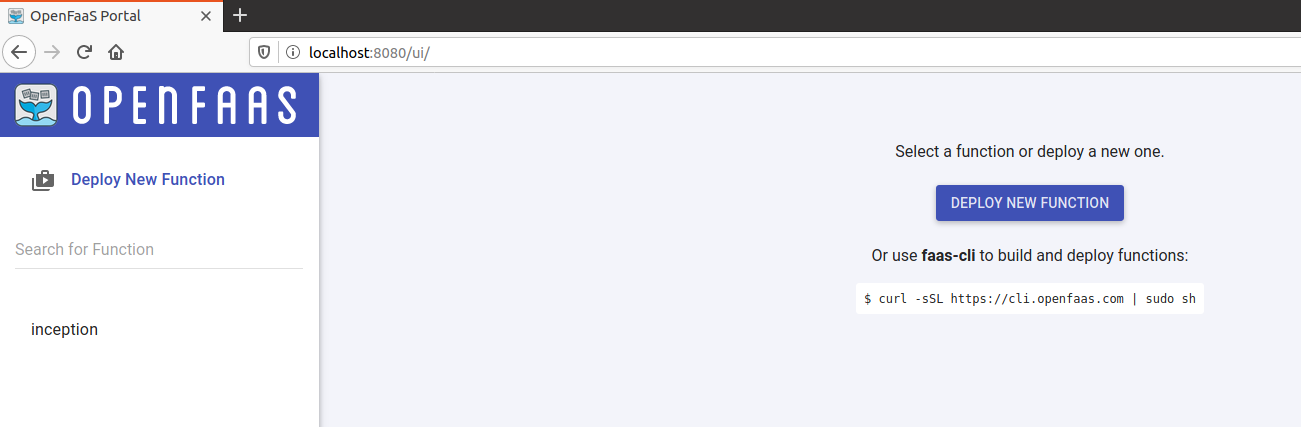
Your output should display the following information, but the ‘*EXTERNAL-IP*’ field should show the IP configured by your 3rd party manager that you need for accessing the gateway UI.

1. Before accessing the gateway, we need to explicitly find the password data we indirectly dealt with in the gateway step. If the previous steps were followed correctly, the data should be stored as the environment variable ‘*PASSWORD*’.

echo $PASSWORD

1. Upon entering the URL for your OpenFaaS gateway, a popup authentication box will appear. Input the required fields (remember that your username is ‘*admin’* by default).



If your information is correctly inputted, you should have full access to the OpenFaaS UI. From here, you can create, deploy, monitor/log, and even download functions from the store.

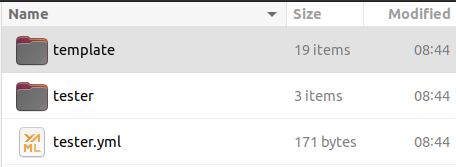
Creating and Deploying a Custom Function

The two main options for creating your own function are through the CLI or the UI. This documentation will only go over the CLI, since the UI is extremely straightforward; it simply requires clicking the “*Deploy New Function”* button and filling in the necessary fields in the UI. There are two parts to successfully developing a usable function: creating it through templates and deploying it through the gateway.

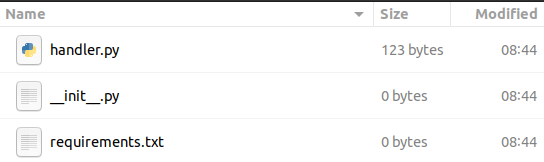
1. After creating a folder and stepping into it with the command line prompt ‘*cd’*, create a new function from an existing template.

faas-cli new --lang <specify programming language> <function name> --prefix="<your-docker-username-here>"

This command will create a template folder, a function folder, and a .yml file within your specified directory. The following image shows the output for creating an example python function called ‘*tester*’.



1. Open the function folder (in the example above, the folder is labelled ‘*tester*’) and fill in the ‘*handler’* file with your function. Any additional dependencies like modules and imports should be included within the ‘*requirements.txt*’ file.



1. Once the function is built and fully complete, the next step is to upload it to the OpenFaaS gateway for deployment. The easiest option is to build, push, and deploy the function all at once with the ‘*up*’ command.

faas-cli up -f <path to the function’s .yml file>

If there are no errors during this step, you should see a deployment confirmation, and a URL for triggering the function.

1. Alternatively, you can separately execute the build, push, and deploy tasks if necessary.

faas-cli build -f <path to the function’s .yml file>

1. Verify that the function was successfully deployed

faas-cli deploy -f <path to the function’s .yml file>

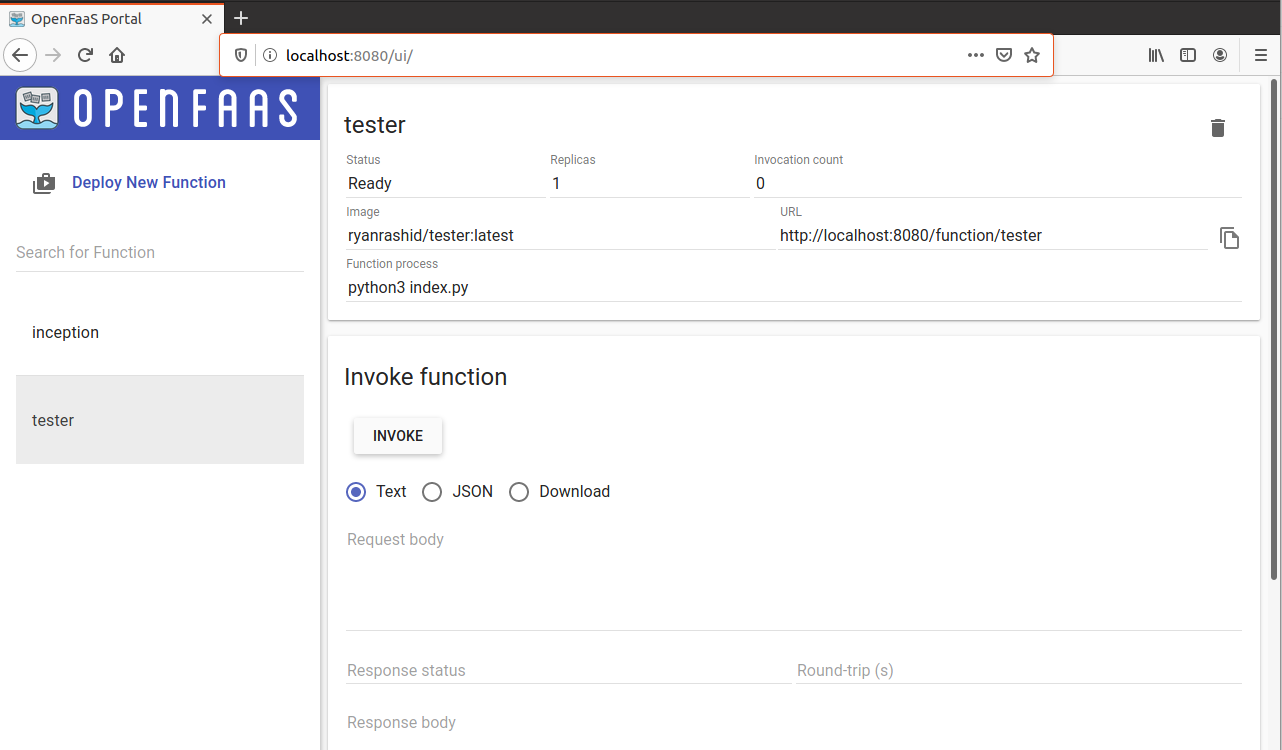
faas-cli push -f <path to the function’s .yml file>

* 1. Use the CLI with the command:

faas-cli list

The output should show the list of your functions, the amount of times each has been invoked, and the number of replicas created.



* 1. Use the UI by accessing the gateway IP (see previous stage for more details):

Sample App: Spreadsheet Parser Function

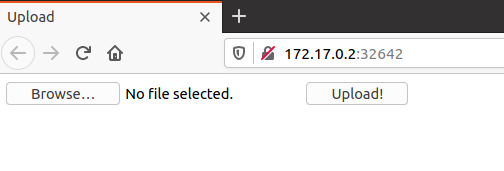
To see these steps in action, we will now create a simple function that **analyses a spreadsheet and returns the highest sales** value among all named employees. The application will be split into two parts: the base application that calls on a function trigger, and the function that will be executed upon being called. For the purpose of this exercise, the focus will be on the processes involving the function, as opposed to the overall application itself.

1. Before dealing with the function, create a Kubernetes deployment and service for the web application itself, which can be pulled from Docker Hub repository ‘*ryanrashid/parse-app:v1*’.

kubectl create deployment parser --image=docker.io/ryanrashid/parse-app:v1

kubectl expose deployment/parser --type=LoadBalancer --port=5000

If you access the service’s URL, a basic web app will appear with options to select and upload a spreadsheet. However, the buttons will not work until the OpenFaaS function is deployed.



1. Make a directory to store OpenFaaS function files and create a new function.

mkdir parse && cd parse

faas-cli new --lang python3 parse --prefix="<your-docker-username-here>"

1. Open the folder ‘*parse*’ and copy the following function code into the ‘*handler.py*’ file.

import json

import operator

def handle(req):

loaded = json.loads(req)

biggest = max(loaded.get("Sales").values())

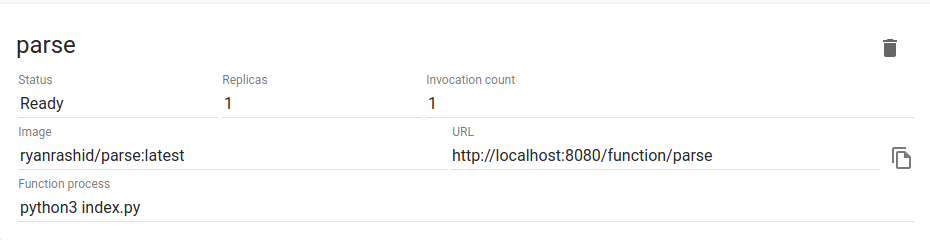
index = max(loaded.get("Sales").items(), key=operator.itemgetter(1))[0]

page = "Your best employee is %s, with $%i in sales!" % (loaded.get("Name").get(index), biggest)

return page

1. Build, push, and deploy the function to the OpenFaaS gateway. This process will only work when logged into the gateway; follow the steps isolated previously if not logged in.

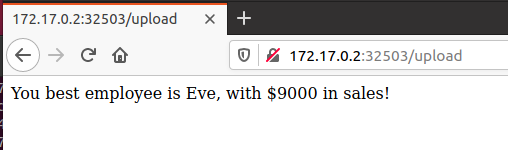
faas-cli up -f ./parse.yml

If the uploading process worked, then the OpenFaaS UI should show ‘*Ready*’ under the status of the function. The invocation count should be at 0, unless you have already invoked it previously.

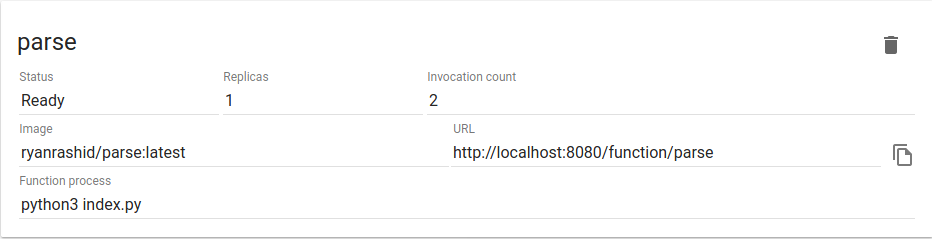
1. Now that the application is fully functional, it needs to be tested with sample spreadsheet data. The function searches for a column labelled ‘*Sales*’ and parses it for the maximum value, so the spreadsheet must follow this format. Create a simple Excel file with the following fields (don’t forget to include the ‘Name’ and ‘Sales’ fields):

|  |  |
| --- | --- |
| **Name** | **Sales** |
| Alice | 750 |
| Bob | 600 |
| Carol | 3500 |
| Dave | 1200 |
| Eve | 9000 |

1. The browse and upload buttons should now be working properly. Select the newly created spreadsheet from the location it is saved and upload it. **The function is immediately triggered once the upload button is clicked.** The output should then return the highest earner, which is Eve in the case of this example.



To verify that the function triggered, the UI should show that the invocation count has increased. In this case, the picture now shows ‘2’ because it has been invoked twice already.



Triggering Functions

One important aspect excluded from the sample is the base application’s code. The sample uses the pre-built image in the Docker Hub repository, but it is still useful to understand a few snippets of it, particularly the function trigger.

jstring = workbook.to\_json()

gateway\_hostname = os.getenv("gateway\_hostname", "gateway")

**r = requests.get("http://" + gateway\_hostname + ".openfaas:8080/function/parse", data=str(jstring))**

return r.text

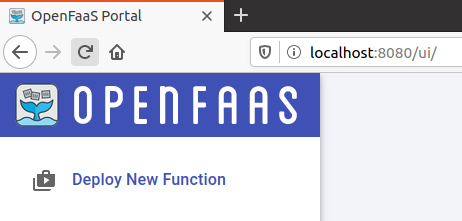
This code triggers the function once the upload button is clicked. It takes the excel file, converts it to a JSON, and **sends an HTTP ‘GET’ request to the function URL** with that JSON information passed in as the ‘*data*’ parameter. The file must be sent in this format because **OpenFaaS functions only accept plain text or JSON objects** as their parameters.

There are also other methods for triggering functions besides HTTP calls, including the UI and message queues like Kafka, Cron, etc. More details can be found in the OpenFaaS official documentation at <https://docs.openfaas.com/reference/triggers/>.

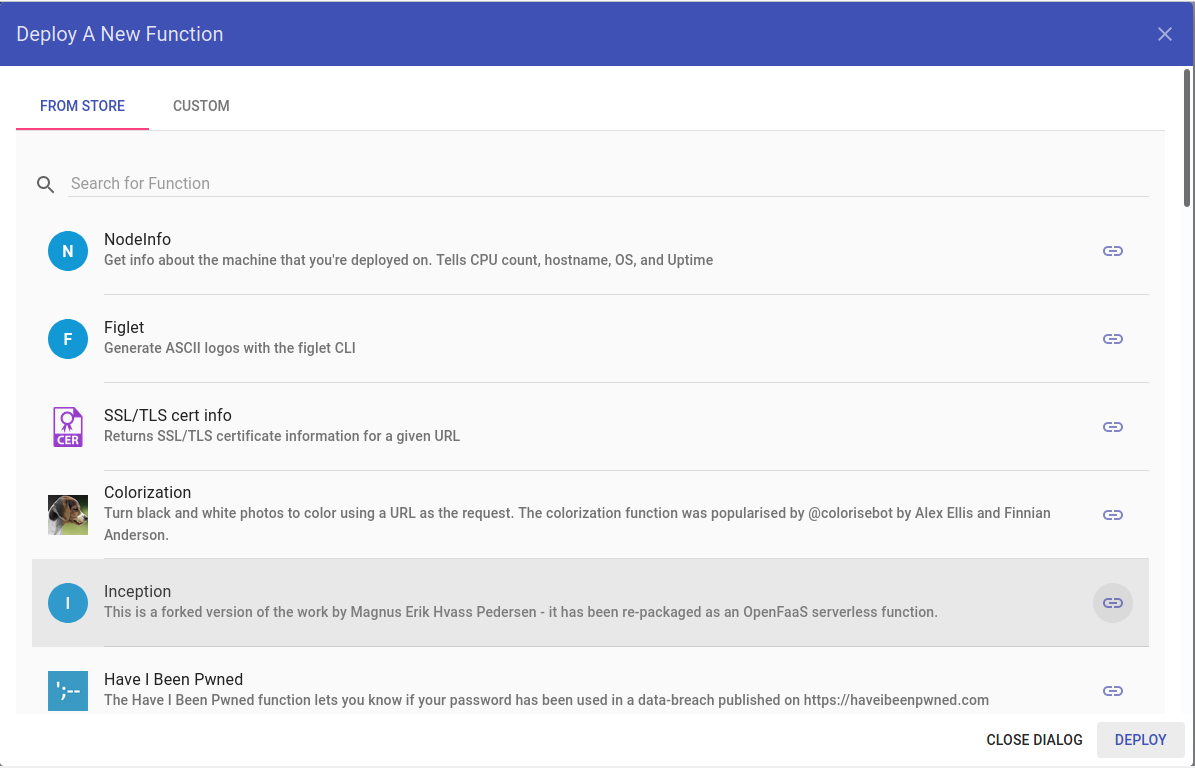
Deploying ML from the Function Store

OpenFaaS also comes with a function store where you can deploy a variety of pre-built functions from useful application tools to machine learning models. While these functions can be deployed from the CLI, it’s **much simpler through the OpenFaaS UI**, which this documentation will also cover. Before implementing the following steps, be sure to complete the prerequisite steps for **accessing the OpenFaaS Gateway and the UI first**.

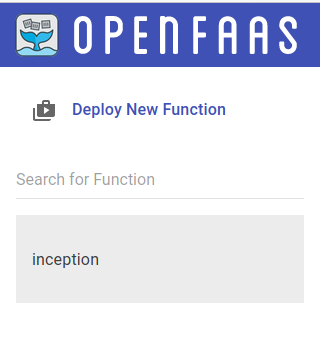
1. Press the “Deploy New Function” button at the top left of the screen.



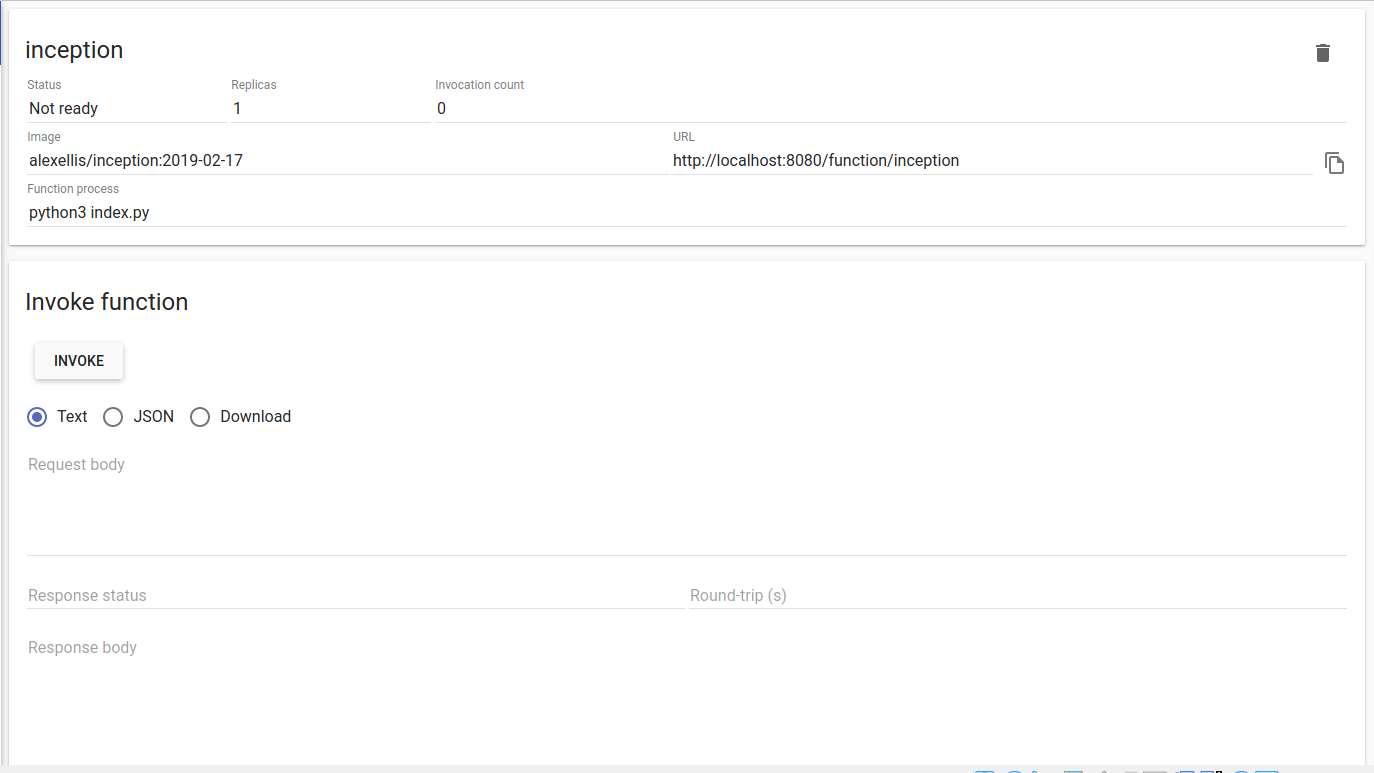
1. You should now see a list of different functions displayed. Click on the one you want to deploy and select the ‘deploy’ button at the bottom left. Here, Inception is selected.



Wait a few seconds until the function name is displayed on the left, as shown below.



1. Click on the function once it is displayed to see a dashboard with some information about the deployment. From here, you can choose to invoke the function through the UI or **build a custom runtime trigger** into an application (see ‘Trigger Functions’ section for more details).



For a sample machine learning web application, go to <https://github.hpe.com/ryan-rashid/faasml> and follow the instructions to setup a UI and machine learning functions with OpenFaaS.

Uninstalling and Cleaning Up

1. Enter the folder that stores the OpenFaaS CLI information along with other dependencies.

cd /usr/local/bin

1. Remove all the OpenFaaS files. Note that you may be denied access permissions, in which case add the keyword ‘*sudo*’ to the beginning of each command.

rm -rf faas-cli

rm -rf ~/.openfaas

rm -rf faas

1. (Optional) Uninstall the arkade installer. Note that this tool can be used to install other Kubernetes apps, so it may be better to keep this installed.

rm -rf arkade

1. Remove any OpenFaaS images leftover in docker.

docker images

rm -rf ark

docker rmi -f <insert OpenFaaS image ID>