Frankfurt University of Applied Sciences

OpenFaaS Installation Guide

Autor:	*Matrikelnummer:*
Marcus Legendre	1137150
Sebastian Müller	1133031
Alexander Tkachov	1134292
Alexander Seng	1057528

Summer Term 2019

Contents

1 Introduction

Serverless computing describes the concept of developing and running applications without having to worry about servers[1]. On a high level and from a developer point of view that might be the case, but the underlying serverless platform does of course still rely on servers of some kind.

In practice, serverless computing or function as a service (FaaS) is a layer of abstraction that hides away most of the operational aspects of running one or more applications. While all of the big cloud providers offer serverless products these days, one can also run a self-hosted serverless platform.

One example for an open-source serverless framework that's capable of running a private serverless platform is OpenFaaS. It is distributed under the MIT license and can be installed on any machine or cluster that's compatible with Docker Swarm and/or Kubernetes.

The aim of this document is to provide step by step instructions on how to create and operate your own serverless plattform using OpenFaaS.

2 Architecture

The API Gateway[1] of OpenFaaS gives the opportunity to access the faas-provider from the outside. Tasks of the faas-provider are for example the creation of docker images out of functions, the deployment of functions or the removal of functions from the system. The gateway is realized as an REST API and is accessible via faas-cli, the provided UI and simple http requests. With each deployed function the user can specify the amount of pods which should contain the function. One task of the gateway is to ensure autoscaling based on the users specification. During a function deployment it is possible to set some values like minimum/maximum amount of replicas or the scaling factor of functions. Those values are set in the yaml file of each function. The autoscaling is done with the help of Prometheus and the AlertManager in case of a installation in Docker Swarm. If OpenFaaS is installed inside a Kubernetes Cluster the built-in Horizontal Pod Autoscaler can be used.

Prometheus is an open source monitoring application which provides functions to get several metrics of your deployed functions inside OpenFaaS. One metric could be the rate of invocations of your function within a defined window.

The AlertManager will read from the metrics retrieved by Prometheus and will inform the gateway if there is a need to scale functions. The rules for sending an alert are defined in a configuration file.

After receiving an alert from the AlertManager the Gateway will ensure that the concerned pods get scaled based on the given scaling factor. The component which will scale the pods is the faas-provider.

There is also the possibility to use Custom Resources[2] from Kubernetes to extend your Kubernetes API. With such a extension you can use Kubernetes to check logs, debug and monitor OpenFaaS functions. The OpenFaaS Operator inside the diagram is the Custom Resource. In order that that the OpenFaaS Operator can do his job all necessary data is stored inside Secret, Deployment and Services.

[1] https://docs.openfaas.com/architecture/gateway/

[2] https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/

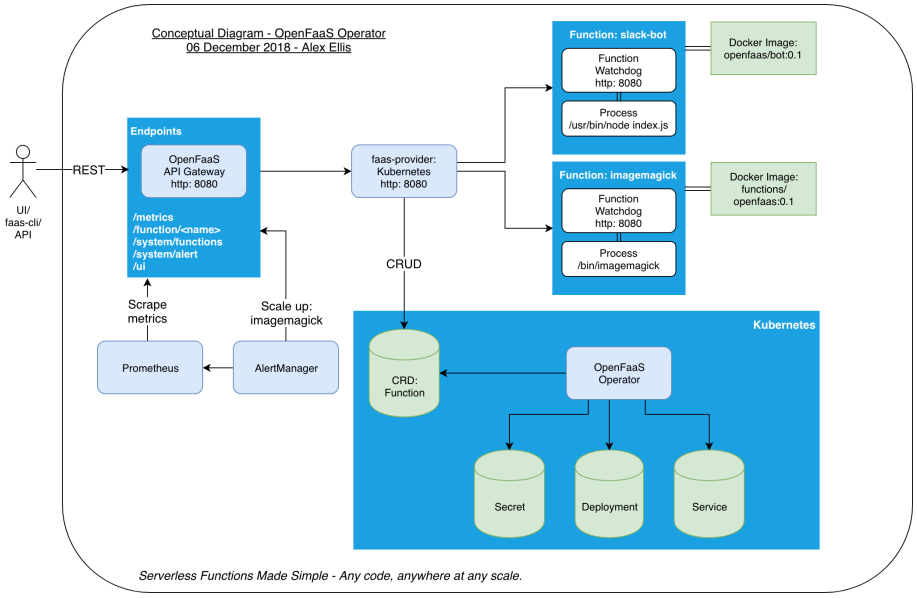

Figure 1: OpenFaaS Architecture

3 Installation

OpenFaaS can run in a variety of environments. This guide walks you through how to install it on a developer computer or in-house development/test server, as well a more production-like environment.

3.1 Development environment with Docker Swarm

Docker Swarm is a tool which makes it possible to create and manage multiple cluster of docker containers. Such a cluster is also called a swarm but we will use the term cluster to avoid confusion. One of Docker Swarms's features is to scale your cluster of docker containers. This means that the so called swarm manager removes or adds nodes inside the cluster for an specific service.

As you can see in Listing 1 the first step is to create a master node with the docker swarm mode but the version of the client and daemon API has to be at least 1.24. If you have more than one IP address you have to add the option `--advertise-addr`. With that option you can apply an IP address to your master node or you can specify an interface. If you specify lo as interface the master node can only be reached inside that network interface.

Listing 1: Initialize the swarm

```
> docker swarm init # if you have a single IP address
> docker swarm init --advertise-addr <network adapter> # if you have more
 ↪  than one
```

In Listing 2 we just download the git repository of OpenFaaS which can be updated with a simple git pull. After downloading the repository you have to execute the mentioned script below inside the root directory of the repository. This file deploys OpenFaaS inside your cluster and automatically generates admin credentials. If you want to set the password by yourself just change the value of `secret` inside `deploy_stack.sh`. This value can be found at line 32 but it's not recommended to modify this value. Just do it for local testing environments. After the first call of `faas-cli login` your hashed password will be written into `~/.openfaas/config.yml`.

Listing 2: Download OpenFaaS

```
> git clone https://github.com/openfaas/faas && cd faas
```

Listing 3: Deploy OpenFaaS in the swarm

```
> ./deploy_stack.sh # username: admin, password: randomly generated on the
 ↪  first run
> ./deploy_stack.sh --no-auth # no authentication
```

Now you are able to use the UI from OpenFaaS. The address shown in Listing 5 is used to access the UI.

Listing 4: Open a web browser

```
http://127.0.0.1:8080
```

After your work is done you should remove OpenFaaS and all it's container which were created during installation. But before OpenFaaS gets removed all deployed functions should be removed firstly. That's because all deployed functions run in a seperated docker container and those won't be removed with the command which will remove OpenFaaS.

Listing 5: Remove OpenFaaS and all deployed functions

```
> faas-cli remove <function-name>
> docker stack rm func
```

3.1.1 Multi-Node Cluster with Docker Swarm

For testing OpenFaas in a multiple node scenario we decide to make a cluster of Docker Swarm nodes in Virtual Machines. Using virtual machines for testing has the benefit that they can be easily ported to other physical Hosts and in case of a malfunction we can reset them to a former snapshot without doing troubleshooting. As a virtualization platform we choose Virtualbox, but it is also possible to do this with every other available virtualization software.
We create a cluster with one master and two worker nodes. For that, three virtual machines with the latest Debian OS release were used. All the machines must be part of the same Local Network, either with the Nat-Network option or the Network-Bridge option of VirtualBox. After the machines are created and the OS is installed, all of them need docker to be installed. On the master node, the swarm mode must be initialized, like it is described in Listing 1. After initializing the swarm node, Docker returns a token, like in Listing 6, which is necessary for joining workers to the cluster.

Listing 6: Join Swarm Token

```
> --token SWMTKN-1-11tgy0poz8v9c8gu0syjr6d42i8m13m3ltn0xrk6ep1djce56h-4ylp1q⌋
 ↪  qoyb71jfzv13i5j5x42
```

The installation of OpenFaas on the master node can be done similar like it is shown in Listing 2 and Listing 3. After the installation of OpenFaas is done, the worker nodes should join the swarm, to do this, the token from Listing 6 is needed. The command is shown in Listing 7.

Listing 7: Join a worker to the swarm

```
> docker swarm join --token "Token" "IP-AddressMasterNode"
```

To check if all the workers joined the swarm the command from Listing 8 can be used on the master node.

Listing 8: Show worker nodes

```
> docker node ls
```

The master node should now return the names of the joined workers, if it does, the setup of the cluster is done and functions can be deployed.
The described steps of installing and deploying OpenFaas can be transferred 1:1 to physical hosts insted of virtual machines, which makes it a good simulation of a own private hardware-based cluster. To remove a worker node from the Cluster, the following command(Listing 9) must be used on the node:

Listing 9: Node leave swarm

```
> docker swarm leave
```

After that, the command from listing 10 should be used on the master node.

Listing 10: Master remove node

```
> docker node rm "NameOfTheNode"
```

There are two possible options to run this test cluster on host systems. In the first one, all the virtual machines are running on the same physical host as it is shown in figure 2.

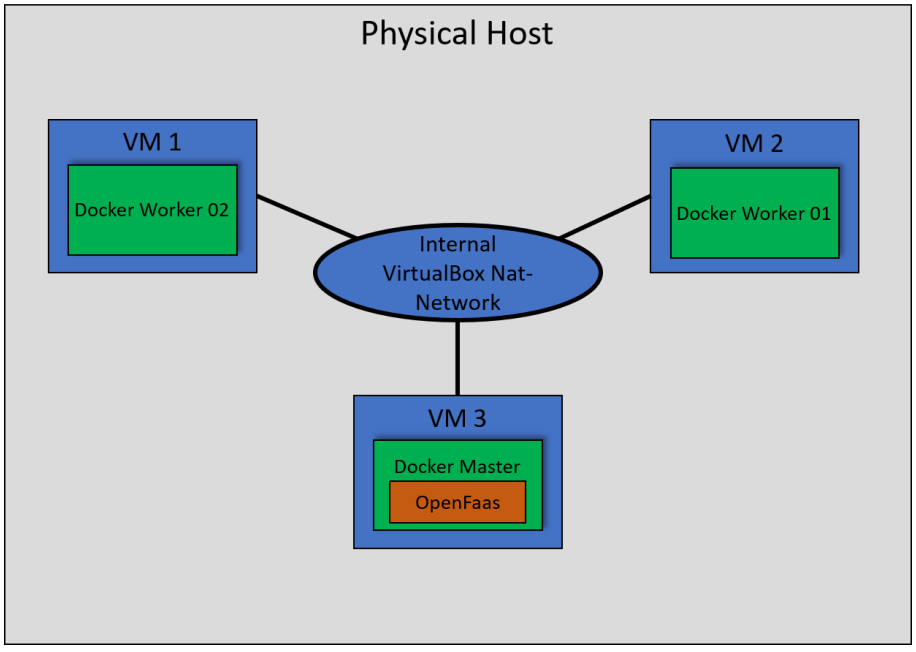

Figure 2: One host Setup for the test cluster

With this setup it is possible to simulate a cluster of multiple nodes on different machines on one physical host. The virtual machines are connected with a Nat-Network which was created with VirtualBox. With this configuration, the machines can communicate with each other and with the host network, but from the host network, they are invincible.
The second option is to run the virtual machines on different physical hosts systems.

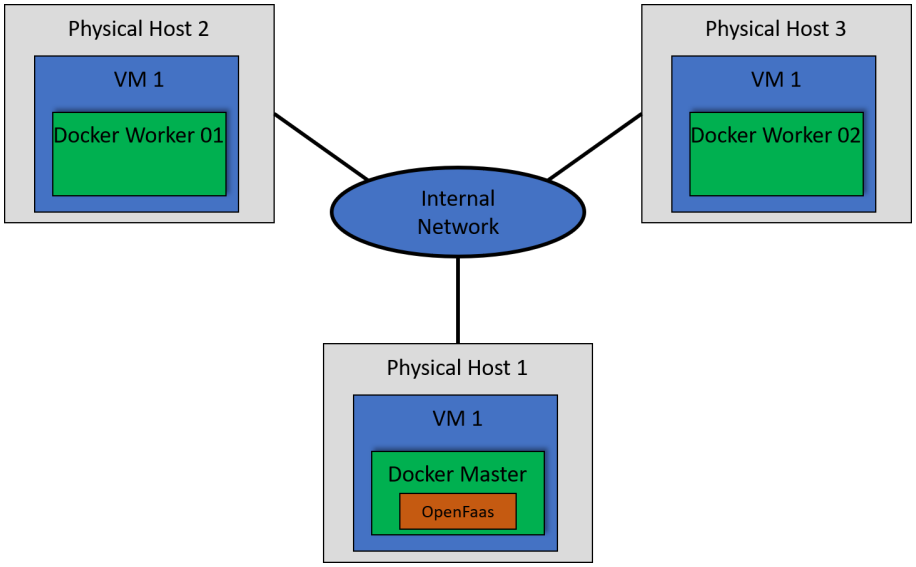

Figure 3: Multiple host setup for the test cluster

This is more close to a own hardware based cluster, but it is independent from the hosts OS. The principle arcitecture is shown in figure 3. In this picture all virtual, machines are on different physical hosts, but other configurations are also possible. For this setup it is important, to choose the Network-Bridge functionality on the VM, with this option the machine acts like a independent host in the network. This is important, because the machines must communicate in the host network.

3.2 Production environment with Kubernetes

OpenFaaS is fully compatible with Kubernetes and can leverage many of its features for improved scalability and configurability. A production ready Kubernetes cluster typically provides a solid foundation for vertical scaling. If the cluster also has horizontal scaling in the form of cluster auto scaling[3], OpenFaaS can virtually scale to infinity. (Provided you are willing to pay for it and your cloud provider doesn't run out of virtual machines.)

FaaS is often hailed for its automatic scaling. But when you decide to run your own FaaS plattform, scalability is likely limited by the underlying hardware. When combining OpenFaaS with Kubernetes and a cloud provider. you can cover the full spectrum of resource requirements, from zero to serving one of the biggest websites on the internet or running compute-intensive parallelized scientific calculations.

This section will walk you through how to get a Kubernetes cluster with OpenFaaS up and running. This involves two major steps:

1. Creating a Kubernetes cluster.
2. Installing the necessary software components inside the Kubernetes cluster.

[3]Cluster auto scaling means worker nodes are created and destroyed automatically as workloads fluctuate

The process described in this installation guide will create a managed Kubernetes cluster on Microsoft Azure. However we will give pointers what needed to be changed to use another cloud provider. The steps involving the installation of the actual software components for OpenFaaS should work on any Kubernetes cluster.

This guide will walk you through the process command by command. If you're looking for a simpler and faster way you can find a mostly automated solution using Bash scripts in the accompanying GitHub repository[4]).

3.2.1 Prerequisites / Tool Chain

To follow along a Microsoft Azure account with a valid subscription is needed (a free trial is sufficient). Additionally a small selection of command line tools are needed:

- Azure CLI[2]
- Terraform[3]
- kubectl[4]
- helm[5]

Azure CLI is needed to access your Microsoft Azure account from the command line. The simplest, although not recommended, way to install Azure CLI is to run `curl -L https://aka.ms/InstallAzureCli | bash`.

Terraform is an *Infrastructure as Code* tool. It can be used to create and destroy infrastructure like virtual machines, load balancers, block storage devices and many more with a range of providers such as Amazon Web Services, Google Cloud, Microsoft Azure, IBM Cloud, Scaleway and others. Although Terraform is not strictly necessary for a guide like this, using a Infrastructure as Code solution is more befitting to a production like setup. If preferred you can use the graphical web interface of your cloud provider (`https://portal.azure.com/` in our case) to create the required resources.
Terraform is typically not provided by package managers and has to be downloaded as a binary from the official website (see listing 11).

Listing 11: Install the Terraform binary on Linux

```
> curl -O https://releases.hashicorp.com/terraform/0.12.0/terraform_0.12.0_l⌋
 ↪  inux_amd64.zip
> unzip terraform_0.12.0_linux_amd64.zip
> sudo mv terraform /usr/local/bin/
```

kubectl is the Kubernetes command line tool that allows you to interact with Kubernetes clusters. Some Linux distributions provide a package for kubectl. There's also a Snap package. More information for your particular plattform can be found in the Kubernetes documentation[4].

Helm is a package manager for Kubernetes. It simplifies the installation of software that's comprised of multiple standalone components and assists with configuration

[4] `https://github.com/orangefoil/azure-kubernetes-openfaas`

management. E.g. an application might consist of multple pods, which are collections of containers, some of which may require persistence storage and/or specific network configuration like exposing ports to a public network or load balancers. Helm packages are called charts and OpenFaaS provides an official Helm Chart which is used in this guide.
Helm is also not provided by package managers and has to be downloaded from the official website (see listing 12).

Listing 12: Install the Helm binary on Linux

```
> curl -O https://storage.googleapis.com/kubernetes-helm/helm-v2.14.0-linux-
↪  amd64.tar.gz
> tar xfvz helm-v2.14.0-linux-amd64.tar.gz
> sudo mv linux-amd64/helm /usr/local/bin/
```

3.2.2 Creating the Kubernetes cluster

First we need to log into Microsoft Azure using their command line tool by typing:

Listing 13: Log in to Microsoft Azure

```
> az login
```

This will open the Azure website in a browser window and ask you to enter your login information and confirm that you want to grant access to the Azure CLI.

Next we need to create an Azure Service Principal. Microsoft defines a Service Principal as follows[6]:

> An Azure service principal is an identity created for use with applications, hosted services, and automated tools to access Azure resources. This access is restricted by the roles assigned to the service principal, giving you control over which resources can be accessed and at which level. For security reasons, it's always recommended to use service principals with automated tools rather than allowing them to log in with a user identity.

This Service Principal requires a name and a password. It can be created with the command:

Listing 14: Create a Service Principal on Azure

```
> az ad sp create-for-rbac \
  --name my-kubernetes-openfaas \
  --password my-insecure-password
```

If you are copying and pasting commands from this guide please make sure to choose a more secure password. The output of that command will look something like this:

Listing 15: Example output after creating the Service Principal

```
{
  "appId": "ff7043b8-99f3-9640-a673-e54f06e1416b",
  "displayName": "my-kubernetes-openfaas",
  "name": "http://my-kubernetes-openfaas",
  "password": "my-insecure-password",
  "tenant": "3ea17ddb-4ef6-4220-r2d2-c709799fb49a"
}
```

With the Service Principal created we can start configuring the Kubernetes cluster or "Azure Kubernetes Service" (AKS) as it's called my Microsoft. Instead of using the Azure website to create the cluster we will use Terraform (see 3.2.1). This follows the paradigm of having infrastructure as code. Create a folder for the Terraform files we are about to create. We are going to follow the best practices for Terraform and create the files *main.tf*, *output.tf*, *provider.tf* and *variables.tf*. For the contents of these files see Listing 16, Listing 17, Listing 18 and Listing 19 respectively.

Listing 16: Terraform main.tf

```
resource "azurerm_resource_group" "resource_group" {
  name     = "cc-project-resources"
  location = "${var.location}"
}

resource "azurerm_kubernetes_cluster" "kubernetes_cluster" {
  name                = "cc-project-kubernetes"
  location            = "${azurerm_resource_group.resource_group.location}"
  resource_group_name = "${azurerm_resource_group.resource_group.name}"
  dns_prefix          = "cc-project"

  agent_pool_profile {
    name            = "default"
    count           = "${var.worker_pool_size}"
    vm_size         = "${var.worker_instance_type}"
    os_type         = "Linux"
    os_disk_size_gb = 30
  }

  service_principal {
    client_id     = "${var.kubernetes_client_id}"
    client_secret = "${var.kubernetes_client_secret}"
  }

  tags = {
    Environment = "Production"
  }
}
```

Listing 17: Terraform output.tf

```
output "client_certificate" {
  value = "${azurerm_kubernetes_cluster.kubernetes_cluster.kube_config.0.cli
   ↪  ent_certificate}"
}

output "kube_config" {
  value = "${azurerm_kubernetes_cluster.kubernetes_cluster.kube_config_raw}"
}

resource "local_file" "kube_config" {
    content     =
     ↪  "${azurerm_kubernetes_cluster.kubernetes_cluster.kube_config_raw}"
    filename = "${path.module}//kubeconfig"
}
```

Listing 18: Terraform provider.tf

```
provider "azurerm" {
  version = "=1.24.0"
}
```

Listing 19: Terraform variables.tf

```
variable "location" {
  description = "The Azure Region in which all resources in this example
   ↪  should be provisioned"
}

variable "kubernetes_client_id" {
  description = "The Client ID for the Service Principal to use for this
   ↪  Managed Kubernetes Cluster"
}

variable "kubernetes_client_secret" {
  description = "The Client Secret for the Service Principal to use for this
   ↪  Managed Kubernetes Cluster"
}

variable "worker_instance_type" {
  default = "Standard_D2s_v3"
  description = "Type of Azure Compute instance you want as worker nodes"
}


variable "worker_pool_size" {
  default = 1
  description = "Number of Azure Compute instances you want running as
   ↪  worker nodes"
}
```

With all the necessary files created run:

Listing 20: Initialize Terraform

```
> terraform init
```

This will perform several initialization steps, e.g. downloading the Terraform module for Microsoft Azure.
For the next command there are three required and two optional parameters:

1. "kubernetes_client_id" - The ID of our Service Principal that was displayed after creating the Service Principal.
2. "kubernetes_client_secret" - The password of our Service Principal that we have chosen earlier.
3. "location" - An Azure region.
4. (optional) "worker_pool_size" - Number of Azure Compute instances you want running as worker nodes.
5. (optional) "worker_instance_type" - Type of Azure Compute instance you want as worker nodes.

The command will be similar to Listing 21. Before Terraform creates the resources we defined, it will present an overview of what resources are going to be created and prompt you to confirm their creation. The creation process typically takes between 5 and 10 minutes.

Listing 21: Let Terraform create the Kubernetes cluster

```
> terraform apply \
  -var 'location = "North Europe"' \
  -var 'kubernetes_client_id = "ff7043b8-99f3-9640-a673-e54f06e1416b"' \
  -var 'kubernetes_client_secret = "my-insecure-password"'
```

You may start using your Kubernetes cluster now using the *kubeconfig* file that was created in the same directory as your Terraform files. For a quick check run:

Listing 22: Display a list of worker nodes (default is just a single node

```
> kubectl --kubeconfig=kubeconfig get node
```

Congratulations! You now have your own Kubernetes cluster. Before moving on to install OpenFaaS it is recommended to configure kubectl where to look for the *kubeconfig* of the cluster we just created. This can be done in either of the following ways:

- Run `export KUBECONFIG=/path/to/your/kubeconfig`. This is limited to your current terminal window. When opening a new terminal window you would have to run the command again.
- Copy the *kubeconfig* file to "~/.kube/config" to permanently configure kubectl for your Azure Kubernetes cluster. Warning: this may overwrite an existing configuration.

3.2.3 Installing OpenFaaS using Helm

Now that we have a Kubernetes cluster up and running we may proceed to install OpenFaaS. But there is one last intermediate step necessary. Installing the server-side component for Helm, the package management tool needed for installing OpenFaaS.

Azure Kubernetes Service uses Kubernetes' role-based access control (RBAC) by default. Thus we have to create a Service Account[5] and a Cluster Role Binding[6]. The respective commands as well as the command for installing the server-side Helm component are listed in Listing 23. Note that Helm 3.0, which at the time of writing is still in alpha, will no longer have a server-side component. This guide is written for Helm versions 2.x.

Listing 23: Installing the server-side Helm component

```
> kubectl -n kube-system create sa tiller
> kubectl create clusterrolebinding tiller \
  --clusterrole cluster-admin \
  --serviceaccount=kube-system:tiller
> helm init --wait --service-account tiller
```

For Helm to be able to find the OpenFaaS charts we need to add their repository by running:

Listing 24: Add the OpenFaaS repository to Helm

```
> helm repo add openfaas https://openfaas.github.io/faas-netes/
```

Now we can finally install OpenFaaS itself. As shown in listing 25, we will create a namespace, configure an admin password for OpenFaaS and deploy its Helm chart.

Listing 25: Installing OpenFaaS using Helm

```
> kubectl apply -f https://raw.githubusercontent.com/openfaas/faas-netes/mas⌋
 ↪  ter/namespaces.yml
> kubectl -n openfaas create secret generic basic-auth \
  --from-literal=basic-auth-user=admin \
  --from-literal=basic-auth-password="my-openfaas-password"
> helm upgrade openfaas --install openfaas/openfaas \
  --namespace openfaas  \
  --set basic_auth=true \
  --set functionNamespace=openfaas-fn \
  --set serviceType=LoadBalancer
```

To check whether OpenFaaS is running you may run:

Listing 26: Display a list of pods associated with the OpenFaaS Helm chart

```
> kubectl --namespace=openfaas get deployments -l "release=openfaas,
 ↪  app=openfaas"
```

[5] Kubernetes Authorization Overview https://kubernetes.io/docs/reference/access-authn-authz/authorization/

[6] Kubernetes RBAC https://kubernetes.io/docs/reference/access-authn-authz/rbac/

If the numbers in the columns "desired" and "available" are identical your OpenFaaS is ready.

Your OpenFaaS is publicly accessible on port 8080 via a load balancer that routes requests into your Kubernetes cluster. The IP of that load balancer is listed on the helm status page and can be extracted by running:

Listing 27: Retrieve the IP address of the OpenFaaS gateway

```
> helm status openfaas | grep gateway-external
```

For instructions on how to use the OpenFaaS web interface or the OpenFaaS CLI refer to section 4.

3.2.4 Considerations

While the serverless platform we have created is a solid foundation for a production environment, there are things not in the scope of this guide, that you should consider before using this Kubernetes + OpenFaaS combination in the wild.

All communication with the OpenFaaS gateway is unencrypted, both via the command line as well as the web interface. However for advanced Kubernetes users it isn't too difficult to set up an ingress controller that supports SSL.

More things to consider are backups, disaster recovery, monitoring and alerting, fault tolerance, scalability, and others, just to name a few. Last but not least it should be carefully examined whether OpenFaaS itself is ready for production.

4 Operation of OpenFaaS

The following sections will illustrate how to create, deploy, manage and use premade or self-made OpenFaaS functions by listing the most common and important commands and providing specific examples.

This guide assumes a locally running OpenFaaS installation on port 8080 similar to the instructions of section 3.1. IP addresses and ports need to be adjusted if that is not the case.

4.1 Setup and configuration of the OpenFaaS command line tool

See listing 28 on how to install `faas-cli`. Running `sh` without root privileges will require additional manual configuration.

Listing 28: Install faas-cli

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

Alternatively you can download the latest release on the Github page of `faas-cli`[7] and place it in your `PATH` (`/usr/local/bin` for example.)

See listing 29 on how to log in into OpenFaaS from your command line.

Listing 29: Login to OpenFaaS with faas-cli

```
> faas-cli login --password-stdin # username 'admin' and default gateway
 ↪  'http://127.0.0.1:8080' are implied
> faas-cli login --password-stdin --username <name> --gateway <gateway>
```

Important Warning: Do not manage functions with `faas-cli` in the OpenFaaS folder you cloned with `git`. It will deploy and try to stop functions which are included in the folder. This can lead to unexpected behavior and cli output.

4.2 OpenFaaS Store

OpenFaaS has an official function store with free functions. Listing 30 shows the cli commands which are necessary to list and deploy them. Store functions are a good way to test a new OpenFaaS installation because they can be deployed with having to know how to write a custom function.

Listing 30: Common commands for the OpenFaaS store

```
> faas-cli store list # list available functions
> faas-cli store inspect <function_name> # show information about a function
 ↪  of the store
> faas-cli store deploy <function_name> # deploy a function from the store
```

[7] `https://github.com/openfaas/faas-cli`

4.3 Management and usage of functions

Listing 31 shows the commands which are necessary for adding, removing and invoking functions.

`Deploy` scans the current working directory for functions and tries to deploy them if no specific function configuration file or docker image is specified.

`Invoke` takes the input for the function call from STDIO. To circumvent this, a simple `echo` command which pipes its output into the input of `faas-cli invoke` can be used. See listing 32 for an example.

Listing 31: Common commands for function management and usage

```
> faas-cli list # list deployed functions
> faas-cli store deploy <function_name> # deploy store function
> faas-cli deploy # deploy locally built functions or functions in docker
 ↪  images
> faas-cli invoke <function_name> # call a function
> faas-cli remove <function_name> # remove deployed functions
```

Listing 32 demonstrates a concrete example for deploying, calling and removing a function. For this example the `Figlet` function from the store is used. `Figlet` is originally a Unix command line tool which turns text into larger text banners. The `Figlet` function from the OpenFaaS store does the same.

Functions are called with a http request. Instead of the `faas-cli` it is possible to call a function with the web UI of OpenFaaS or any http client (standalone or embedded in an application). Listing 33 demonstrates how a function is called with `curl`, a command line tool for sending data over the network.

Listing 32: Example - Figlet function from the store

```
> faas-cli store deploy figlet
Deployed. 202 Accepted.
URL: http://127.0.0.1:8080/function/figlet

> faas-cli list
Function                        Invocations     Replicas
figlet                          0               1

> echo "Cloud Computing" | faas-cli invoke figlet
  ____ _                 _
 / ___| | ___  _   _  __| |
| |   | |/ _ \| | | |/ _` |
| |___| | (_) | |_| | (_| |
 \____|_|\___/ \__,_|\__,_|

  ____                             _   _
 / ___|___  _ __ ___  _ __  _   _| |_(_)_ __   __ _
| |   / _ \| '_ ` _ \| '_ \| | | | __| | '_ \ / _` |
| |__| (_) | | | | | | |_) | |_| | |_| | | | | (_| |
 \____\___/|_| |_| |_| .__/ \__,_|\__|_|_| |_|\__, |
                     |_|                       |___/

> faas-cli remove figlet
Deleting: figlet.
Removing old function.
```

Listing 33: Example - Calling a function with curl

```
> curl -X POST --data "OpenFaaS" http://127.0.0.1:8080/function/figlet
  ___                   _____          ____
 / _ \ _ __   ___ _ __ |  ___|_ _  __ _/ ___|
| | | | '_ \ / _ \ '_ \| |_ / _` |/ _` \___ \
| |_| | |_) |  __/ | | |  _| (_| | (_| |___) |
 \___/| .__/ \___|_| |_|_|  \__,_|\__,_|____/
      |_|
```

4.4 Development of functions

The quickest way to develop a function is starting with a template provided by OpenFaaS. These exist for several languages. Listing 34 lists two way to view them: The first command just lists the available templates from the OpenFaas store (official, unofficial and incubating) and the second set of commands downloads the (official) templates into the working directory so that they can be used for the creation of a new function.

Listing 34: Listing available templates

```
> # View available templates from the store
> faas-cli template store list

> # Download the templates for offline access
> faas-cli template pull
> faas-cli new --list
```

Listing 35 illustrates how to create a new function. Create a bare function from a template first. The name can be freely chosen, the template name has to be valid template (like `python3` or `java8`). After the function is implemented, it can be build to a docker image with the second command. OpenFaaS will create a `build` directory to store intermediate files which are necessary for the build. After the image was build successfully, it can be deployed with the last command.

Listing 35: Creating a function from a template

```
> faas-cli new <function_name> --lang=<template_name>
> # Implement your function before building it
> faas-cli build -f <function_name>.yml
> faas-cli deploy -f <function_name>.yml
```

4.5 Working with docker registries

Functions can be uploaded to and deployed from docker registries as seen in listing 36. As a prerequisite, the user has to be logged in to docker on the command line and the `image` field of the yaml file of a function needs to point to a valid docker repository. It these criteria are met, the function can be uploaded as an image to the docker registry with `faas-cli up` which also deploys it after uploading.

To deploy a function from a docker registry `faas-cli deploy` can be used. The image name must point to an image with a function while the name under which the function will be deployed can be chosen freely.

Listing 36: Working with docker registries

```
> # Make sure that you are logged in to the registry
> # Adjust the image name in the yaml file of the function
> faas-cli push -f <function_name>.yml
> faas-cli deploy --image <image_name> --name <any name>
```

4.6 Web UI

Basic tasks for managing functions can be done via the Web UI. You can use it to

- view running functions
- invoke functions
- deploy functions from the store or docker images

- stop functions

References

[1] Sarah Allen et al. “CNCF WG-Serverless Whitepaper”. In: (2018). URL: https://github.com/cncf/wg-serverless/blob/master/whitepaper/cncf_serverless_whitepaper_v1.0.pdf.

[2] Microsoft Coorporation. *Microsoft Azure - Install the Azure CLI*. Feb. 12, 2019. URL: https://docs.microsoft.com/en-us/cli/azure/install-azure-cli?view=azure-cli-latest.

[3] Hashicorp. *Terraform*. May 29, 2019. URL: https://www.terraform.io/.

[4] Kubernetes. *Install and Set Up kubectl*. May 29, 2019. URL: https://kubernetes.io/docs/tasks/tools/install-kubectl/.

[5] Helm. *Helm - The Package Manager for Kubernetes*. May 29, 2019. URL: https://helm.sh/.

[6] Microsoft Coorporation. *Create an Azure service principal with Azure PowerShell*. June 3, 2019. URL: https://docs.microsoft.com/de-de/powershell/azure/create-azure-service-principal-azureps?view=azps-2.1.0.