Function as a Service

General Principles, Container Virtualization, OpenFaaS, OpenWhisk – Winter Term 2020

Henry-Norbert Cocos
cocos@fb2.fra-uas.de

Computer Science
Faculty of Computer Science and Engineering
Frankfurt University of Applied Sciences

Contents

FRANKFURT UNIVERSITY OF APPLIED SCIENCES

About Myself

Henry-Norbert Cocos

- Bachelor of Science in Computer Science, Frankfurt University of Applied Sciences
- Currently Master program Allgemeine Informatik, Frankfurt University of Applied Sciences

Research Interests:

- Cloud Computing
- Cluster Computing
- Edge Computing
- Function as a Service
- Internet of Things

FRANKFURT UNIVERSITY OF APPLIED SCIENCES

Container Virtualization

Container Virtualization capsulates Applications in virtual environments!

This technology has a long going history!
(`chroot` was first implemented in 1979 [1])

Containers are more efficient than Hypervisor-based Virtualization or Paravirtualization!

Containerization vs application virtualization

In contrast to application virtualization the libraries needed for the application in a container environment are run inside the container, whereas in application virtualization the OS user-libraries are used. Both depend on the systems Kernel.

Container Virtualization and Software Engineering

Container Virtualization sets the basis for Continuous Integration and Continuous Delivery (CI/CD) [2]!

Applications can be divided into smaller modules!

This has the following benefits:

1. **The division makes development cycles shorter and integration of new features faster (CI)!**
2. **The division makes the application deployment cycles shorter (CD)!**

Book: Skalierbare Container-Infrastrukturen [2]

The Book *Skalierbare Container-Infrastrukturen* offers a great source for the integration of containers in application development. The focus of the book is on DevOps (specifically CI/CD) and the orchestration of Docker containers with Kubernetes and the use inside of OpenShift! Unfortunately the book is only available in german language :-(

CI/CD Lifecycle

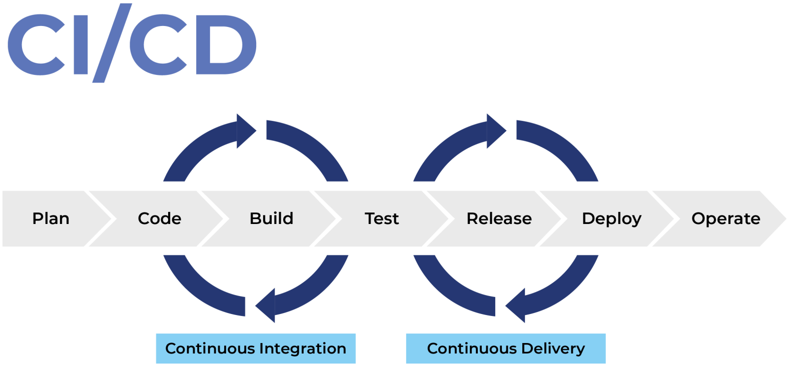


Figure: CI/CD Lifecycle

Source: `https://medium.com/faun/most-popular-ci-cd-pipelines-and-tools-ccfdce429867`

Docker

Figure: Docker

Source:
`https://www.docker.com/brand-guidelines`

Docker

- Released by dotCloud 2013
- Enables Container Virtualization
- A more advanced form of Application Virtualization
- Available for:
  Linux, MacOS, Windows

FRANKFURT UNIVERSITY OF APPLIED SCIENCES

Docker Architecture

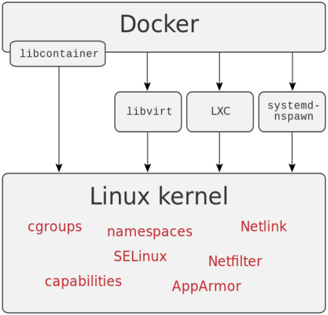


Figure: Docker Architecture

Source: https://en.wikipedia.org/wiki/Docker_(software)

Docker Architecture

- Docker uses the Linux Kernel
- `libcontainer` creates containers
- `libvirt` manages Virtual Environments
- LXC was replaced by `libcontainer` (from version Docker Version 0.9 on!)

Docker Application Architecture I

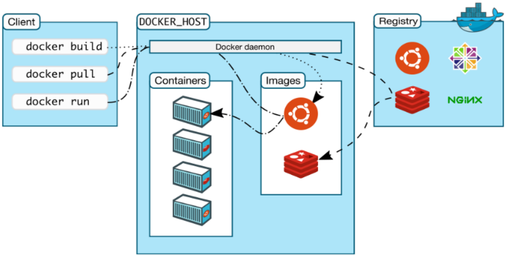


Figure: Docker Application Architecture

Source: `https://docs.docker.com/engine/docker-overview/#docker-architecture`

Applications in Docker [3]

- Client-Server Architecture
- Docker Client `docker`
- Docker Daemon `dockerd`

Docker Objects

- Images
- Containers

Docker Application Architecture II

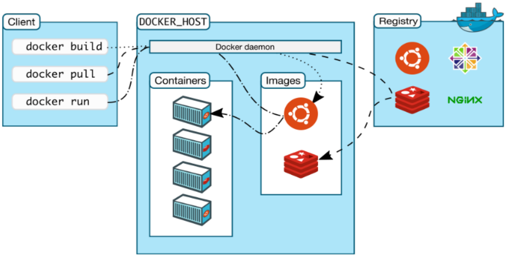


Figure: Docker Application Architecture

Source: `https://docs.docker.com/engine/docker-overview/#docker-architecture`

Docker Client `docker`

- Manages Docker Daemon/s

Docker Daemon `dockerd`

- Listens to Requests
- Manages Docker Objects (images, containers, etc.)

FRANKFURT UNIVERSITY OF APPLIED SCIENCES

Docker Application Architecture III

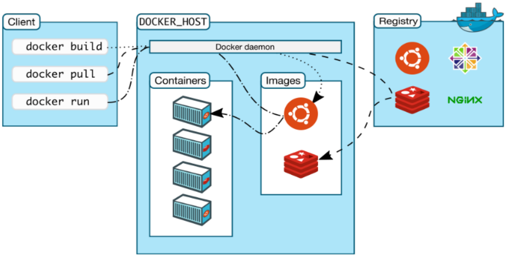


Figure: Docker Application Architecture

Source: `https://docs.docker.com/engine/docker-overview/#docker-architecture`

Docker Objects

- **Containers**
  - Runnable Instance
  - Isolated from other containers
- **Images**
  - Read-Only File
  - Defines an Application

CI/CD Pipeline with Docker and Kubernetes

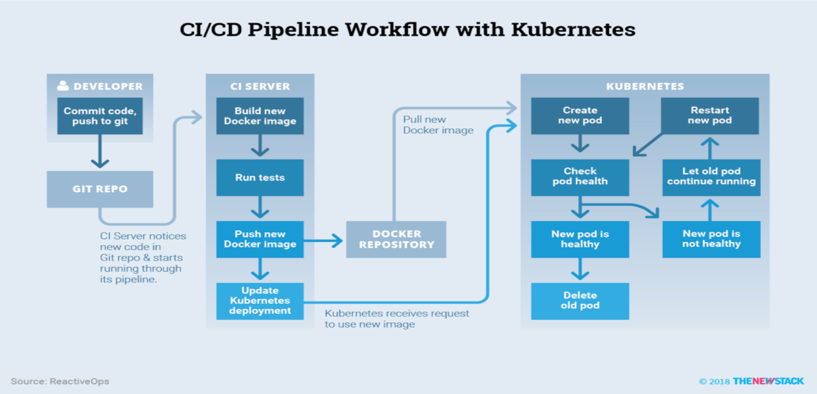


Figure: CI/CD Pipeline with Docker and Kubernetes

Source: `https://thenewstack.io/ci-cd-with-kubernetes-tools-and-practices/`

Microservice Architectures

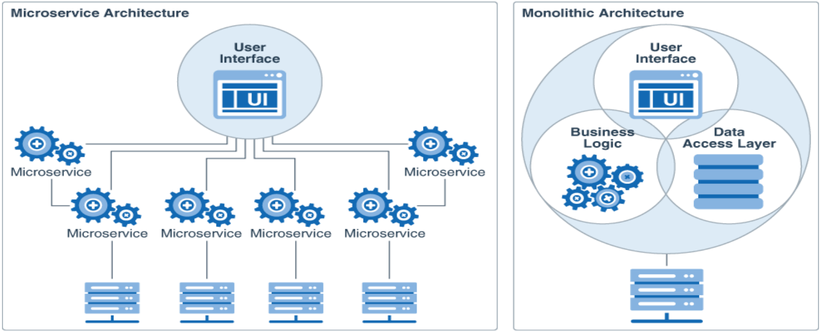


Figure: Microservice Architectures vs Monolithic Architectures

Source: `https://docs.oracle.com/de/solutions/learn-architect-microservice/index.html#GUID-BDCEFE30-C883-45D5-B2E6-325C241388A5`

Microservice Architecture Benefits

Microservice Architectures have the following Benefits:

- Functional Decomposition – *Do one thing and do it well!* [1]
- Independent Developement of Services – Teams can work on Services independently
- Increased overall robustness – If one Service fails it fails! Not the whole Application!
- Increased Compatibility – All Services are independent and need to work together (usually through REST-APIs)
- Increased Scalability – Services can be scaled independent from other Services

[1] Quote from Ken Thompson
https://en.wikipedia.org/wiki/Ken_Thompson

Docker Benefits

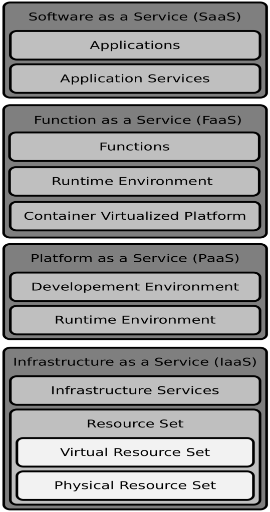
Docker has the following benefits:

- Less resource consumption than OS Virtualization
- Isolation of Applications
- Fast deployment
- Perfect for testing purposes
- Containers can be restarted

Docker Swarm and Kubernetes

The Docker Engine has a build in solution for Cluster deployment and management. The `swarm` mode enables the control over multiple Docker hosts and is crucial for the scalability of applications [4]. Kubernetes is a different system that enables deployment over multiple hosts.

Cloud Service Layers



FRANKFURT UNIVERSITY OF APPLIED SCIENCES

Function as a Service

Function as a Service (FaaS) has emerged as a new platform service in Cloud Computing!

FaaS reduces administration tasks and brings the focus back to the Source Code! [5]

FaaS enables more effective event-driven applications!

Function as a Service

(a) (b)

(c) (d)

Figure: Popular FaaS Offerings:
(a) AWS Lambda [6]
(b) Google Cloud Functions [7]
(c) IBM Cloud Functions [8]
(d) Apache OpenWhisk [9]

Function as a Service (FaaS)

- Event-driven
- Scalable
- Fast deployment of code
- Payment per invocation

Amazon Alexa

Alexa Skills are executed in AWS Lambda!

FRANKFURT UNIVERSITY OF APPLIED SCIENCES

Public FaaS offerings – AWS Lambda

Figure: AWS Lambda [6]

AWS Lambda

- Released in 2014
- Fully automated administration
- Automated Scaling
- Built in fault tolerance
- Support for multiple languages: Java, Node.js, C# and Python

Public FaaS offerings – IBM Cloud Functions

Figure: IBM Cloud Functions [8]

IBM Cloud Functions

- Released in 2016
- Event-driven Architecture
- Automated Scaling
- Apache OpenWhisk is basis of IBM Cloud Functions (No Vendor Lock-in!)
- Support for multiple languages: JavaScript, Python, Ruby, ...[2]

[2] More on that in Section 4

FaaS Generic Architecture I

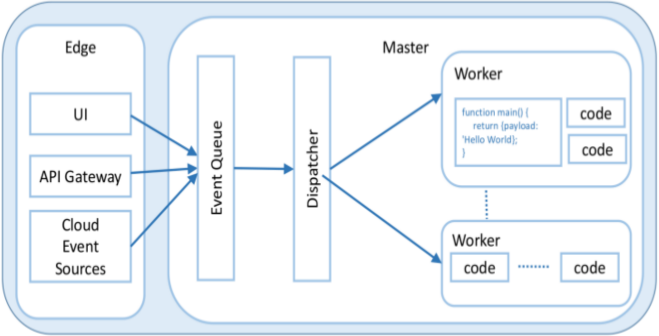


Figure: Generic FaaS Architecture [10]

FaaS Generic Architecture II

Edge

- **UI** – An UI for the management of functions
- **API Gateway** – The general API for the implemented functions

Event Queue/Dispatcher

- **Event Queue** – Manages the triggered Events
- **Dispatcher** – Manages the scaling of invocations

Worker

- **Worker Processes/Containers** – Execute the function invocations

Interesting Paper

Figure 12 and the explanation of the architecture are taken from the paper of Baldini et.al. [10]

Function as a Service

The Service consists of:

- Scalability – Reaction to large number of Requests
- Environment – Running the code on a Platform
- Virtualization – Capsulation of running code

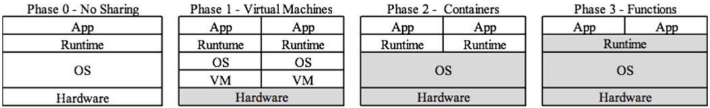


Figure: Evolution of Virtualization [11]

OpenFaaS

Figure: OpenFaaS

Source:
`https://github.com/openfaas`

OpenFaaS

- Open Source Platform
- Functions can be deployed and scaled
- Event-driven
- Lightweight
- Support for multiple languages: C#, Node.js, Python, Ruby

OpenFaaS Architecture I

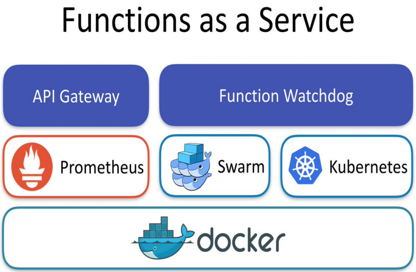


Figure: OpenFaaS Architecture [12]

OpenFaaS Architecture [12]

- **Gateway API**
  - Provides a Route to the functions
  - UI for the management of functions
  - Scales functions through Docker
- **Function Watchdog**
  - Functions are added as Docker Images
  - Entrypoint for HTTP Requests
  - In → `STDIN`
    Out → `STDOUT`

OpenFaaS Architecture II

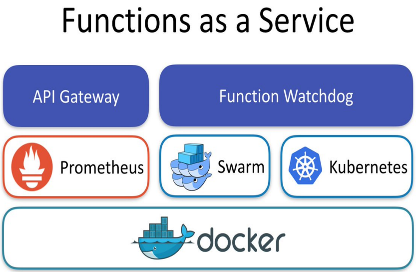


Figure: OpenFaaS Architecture [12]

OpenFaaS Architecture [12]

- **Prometheus**
  - Collects Metrics
  - Function Metrics can be inspected
  - Can be accessed through Web-UI
- **Docker**
  - Isolates Functions in Docker Images
  - Docker Swarm distributes functions
  - Kubernetes can be used to orchestrate Docker Instances

OpenFaaS Benefits

OpenFaaS has the following benefits:

- Open Source
- Low resource consumption
- Deployment of functions
- Autoscaling
- Build in Monitoring and Metrics (Prometheus)

OpenFaaS on Raspberry Pi

OpenFaaS together with Docker Swarm have a low resource consumption. Therefore OpenFaaS has been installed on a cluster of 6 Raspberry Pis. Further evaluation of the service on Raspberry Pis has to be made. More information about installation on Raspberry Pi [13].

Installing OpenFaas

In order to work with OpenFaas 3 packages need to be installed:

- **Docker**
- **OpenFaas Framework**
- **OpenFaas CLI**

Installing Docker

Install Docker:

```
$ curl -sSL https://get.docker.com | sh
```

Add `docker` **User to** `<USER>` **User Group:**

```
$ usermod <USER> -aG docker <USER>
```

Initialize Docker Swarm on Master Node:

```
$ docker swarm init
```

Command on slaves to join workers to docker swarm cluster:

```
$ docker swarm join --token <TOKEN>
```

Installing OpenFaas

Download OpenFaaS from github:

```
$ git clone https://github.com/alexellis/faas/
```

Changing into directory and deploy OpenFaaS [3]:

```
$ cd faas && ./deploy_stack.armhf.sh
```

Install OpenFaaS CLI:

```
$ curl -sSL cli.openfaas.com | sudo sh
```

[3] The script `deploy_stack.armhf.sh` is necessary for the ARM platform

Creating Functions in OpenFaas

Now that Docker and OpenFaas have been installed deployment of functions can begin!

Creating an application using Minio in OpenFaaS

Application Flow [4]:

- The Application downloads an image and stores it in a Bucket
- The image is loaded from the Bucket and then converted to Black/White
- In the last step the image is stored in another Bucket
- The Application consists of **OpenFaas** and **Minio (a private object-based storage with S3-API)**

For this Application two Functions are needed!

OpenFaaS als leichtgewichtige Basis für eigene Functions as a Service.
Henry-Norbert Cocos, Christian Baun. iX 9/2018, S.122-127, ISSN: 0935-9680

[4] Source Code and explanation available at: https://blog.alexellis.io/openfaas-storage-for-your-functions/

Creating Directory for Function

Create a functions directory:

```
$ mkdir functions
```

Change into this directory and issue the following command:

```
$ cd functions && faas-cli new --lang python-armhf
    loadimages
$ faas-cli new --lang python-armhf processimages
```

Templates for Python Functions

The command from the last slide will create the following files in the functions directory:

- `loadimages/handler.py`
- `loadimages/requirements.txt`
- `loadimages.yml`

- `processimages/handler.py`
- `processimages/requirements.txt`
- `processimages.yml`

Install Minio

Install Minio Client and Server as Docker Containers:

```
$ docker pull minio/mc
$ docker run minio/mc ls play
$ docker pull minio/minio
$ docker run -p 9000:9000 minio/minio server /data
```

Start Minio Server

Start Minio Server and get Credentials:

```
$ docker run -p 9000:9000 minio/minio server /data
...
Endpoint:  http://172.17.0.2:9000
http://127.0.0.1:9000
AccessKey: <ACCESSKEY>
SecretKey: <SECRETKEY>
...
```

Configure the Minio Client

In the next step the Minio Client has to be configured.

Configure the Access:

```
$ ./mc config host add TestService http
    ://192.168.178.21:9000 <ACCESSKEY> <SECRETKEY>
```

Creating the Buckets

The Minio Client is used to create two Buckets.

Creating the Buckets:

```
$ ./mc mb TestService/incoming
$ ./mc mb TestService/processed
```

One Bucket for incoming Images and one for processed Images

YAML File of Function `loadimages`

```
provider:
  name: faas
  gateway: http://192.168.178.21:8080

functions:
  loadimages:
    lang: python
    handler: ./loadimages
    image: loadimages
    environment:
      minio_hostname: "192.168.178.21:9000"
      minio_access_key: <ACCESSKEY>
      minio_secret_key: <SECRETKEY>
      write_debug: true
```

Listing 1: File `loadimages.yml`

YAML File of Function processimages

```
provider:
  name: faas
  gateway: http://192.168.178.21:8080

functions:
  processimages:
    lang: python
    handler: ./processimages
    image: processimages
    environment:
      minio_hostname: "192.168.178.21:9000"
      minio_access_key: <ACCESSKEY>
      minio_secret_key: <SECRETKEY>
      write_debug: true

  convertbw:
    skip_build: true
    image: functions/resizer:latest
    fprocess: "convert – –colorspace Gray fd:1"
```

Listing 2: File `processimages.yml`

requirements.txt of the Functions

```
minio
requests
```

Listing 3: File requirements.txt

loadimages Function in Python I

```
from minio import Minio
import requests
import json
import uuid
import os

def handle(st):
 req = json.loads(st)

 mc = Minio(os.environ['minio_hostname'],
  access_key=os.environ['minio_access_key'],
  secret_key=os.environ['minio_secret_key'],
  secure=False)

 names = []
 for url in req["urls"]:
  names.append(download_push(url, mc))
 print(json.dumps(names))
```

Listing 4: File loadimages Part I

loadimages Function in Python II

```
def download_push(url, mc):
 # download file
 r = requests.get(url)

 # write to temporary file
 file_name = get_temp_file()
 f = open("/tmp/" + file_name, "wb")
 f.write(r.content)
 f.close()

 # sync to Minio
 mc.fput_object("incoming", file_name, "/tmp/"+file_name)
 return file_name

def get_temp_file():
 uuid_value = str(uuid.uuid4())
 return uuid_value
```

Listing 5: File loadimages Part II

processimages Function in Python I

```
from minio import Minio
import requests
import json
import uuid
import os

def handle(st):
 req = json.loads(st)

 mc = Minio(os.environ['minio_hostname'],
  access_key=os.environ['minio_access_key'],
  secret_key=os.environ['minio_secret_key'],
  secure=False)

 names = []
 source_bucket = "incoming"
 dest_bucket = "processed"

 for file_name in req:
  names.append(convert_push(source_bucket, dest_bucket,
     ↪ file_name, mc))

 print(json.dumps(names))
```

Listing 6: File processimages Part I

processimages Function in Python II

```
def convert_push(source_bucket, dest_bucket, file_name,
    mc):
 mc.fget_object(source_bucket, file_name, "/tmp/" +
    file_name)

 f = open("/tmp/" + file_name, "rb")
 input_image = f.read()

 # call function for b/w conversion
 r = requests.post("http://gateway:8080/function/
    convertbw", input_image)

 # write to temporary file
 dest_file_name = get_temp_file()
 f = open("/tmp/" + dest_file_name, "wb")
 f.write(r.content)
 f.close()

 # sync to Minio
 mc.fput_object(dest_bucket, dest_file_name, "/tmp/"+
    dest_file_name)

 return dest_file_name

def get_temp_file():
 uuid_value = str(uuid.uuid4())
 return uuid_value
```

Listing 7: File processimages Part II

FRANKFURT UNIVERSITY OF APPLIED SCIENCES

Building and Deploying the Functions

Build the Functions:

```
$ faas-cli build -f loadimages.yml
$ faas-cli build -f processimages.yml
```

Deploy the Functions:

```
$ faas-cli deploy -f loadimages.yml
$ faas-cli deploy -f processimages.yml
```

Downloading and Converting the images

Download images into the `incoming` Bucket:

```
$ echo '{
"urls": [
"https://images.pexels.com/photos/72161/pexels-photo
    -72161.jpeg?dl&fit=crop&w=640&h=318",
"https://images.pexels.com/photos/382167/pexels-photo
    -382167.jpeg?dl&fit=crop&w=640&h=337"]
}' | faas invoke loadimages
```

Convert the images to grey and store in `processed` Bucket:

```
$  echo '["b0f38ebc-675c-43c1-ada7-8fb95dccee57", "34
    d0ad5d-9a24-4b32-bc3e-25337f6f2f5d"]' | faas invoke
    processimages
```

FRANKFURT UNIVERSITY OF APPLIED SCIENCES

Incoming Bucket in Minio

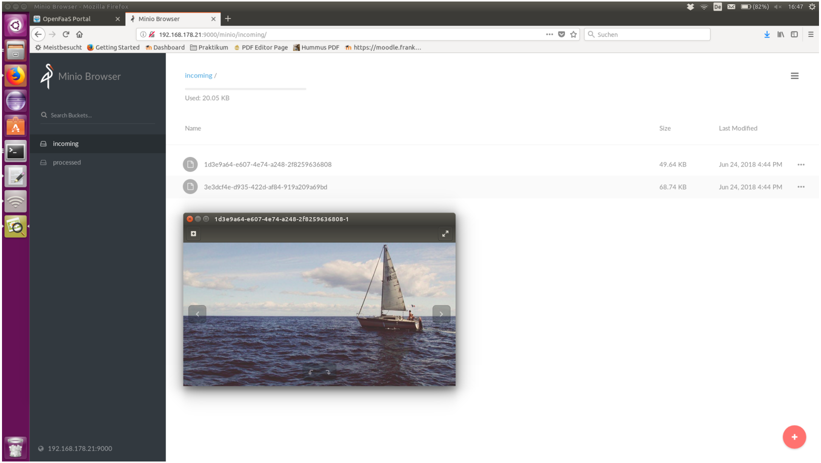


Figure: Incomming Bucket

Processed Bucket in Minio

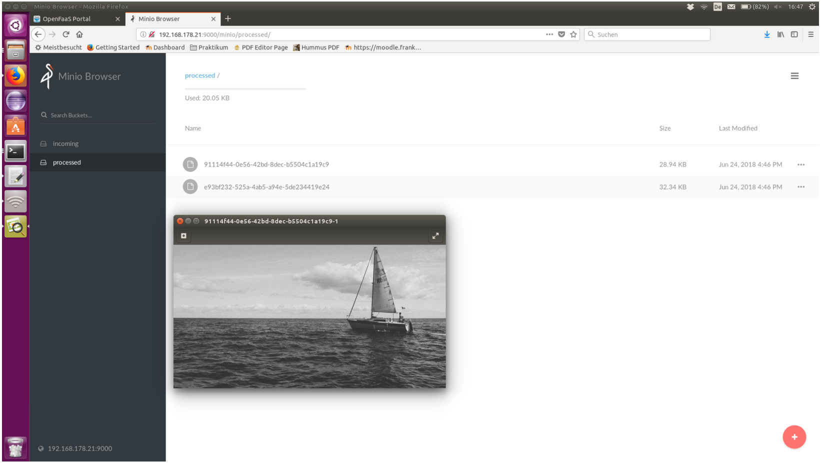


Figure: Processed Bucket

OpenWhisk

Figure: OpenWhisk [9]

OpenWhisk

- Open Source Platform
- Functions can be deployed in a production ready environment
- Support for multiple languages: JavaScript, Python 2, Python 3, PHP, Ruby, Swift
- C, C++, Go programs need to be compiled before upload, Java programs need to be uploaded as JAR-Archives

OpenWhisk Architecture I

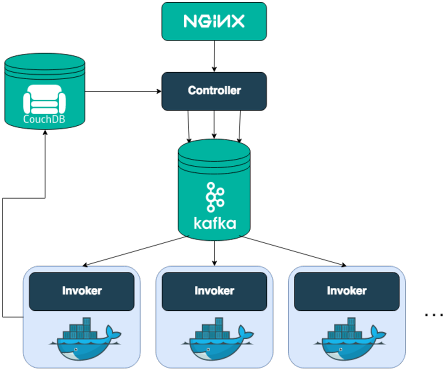


Figure: OpenWhisk Architecture

OpenWhisk Architecture [9]

Components:

- **Nginx**
- **Controller**
- **Kafka**
- **CouchDB**
- **Invoker**

Source:
`https://tinyurl.com/y7plrxbw`

OpenWhisk Architecture II

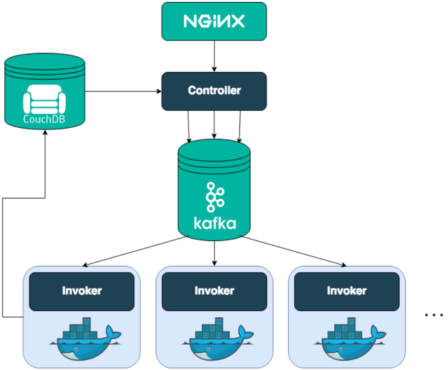


Figure: OpenWhisk Architecture

Source:
`https://tinyurl.com/y7plrxbw`

- **Nginx**
  - Loadbalancer for incoming requests
  - Forwarding requests to the controller
- **Controller**
  - Checks incoming requests
  - Controls the further action
- **Kafka**
  - Publish-Subscribe Messaging Service
  - Queues the requests

OpenWhisk Architecture III

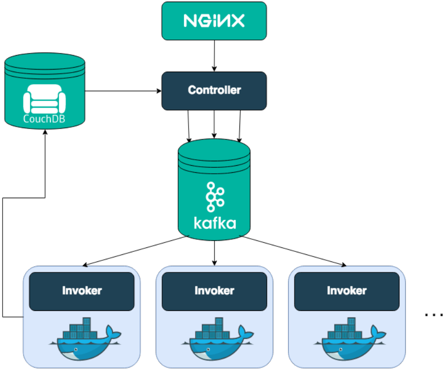


Figure: OpenWhisk Architecture

Source:
https://tinyurl.com/y7plrxbw

- **CouchDB**
  - Authentication of requests (permission checking)
  - Stores information on the imported Functions
- **Invoker**
  - Docker Container(s) running the Function
  - Each Invoker can be paused for faster request fullfilment

Installing OpenWhisk

There are 3 ways of installing OpenWhisk:

- **In a Docker Container**
- **As a virtual machine using vagrant and e.g. VirtualBox**
- **Inside a Kubernetes Cluster**

Installing OpenWhisk – Docker Container

Install OpenWhisk as Docker Containers:

```
$ git clone https://github.com/apache/incubator-
   openwhisk-devtools.git
$ cd incubator-openwhisk-devtools/docker-compose
$ make quick-start
```

Installing OpenWhisk – vagrant

Install OpenWhisk with vagrant and VirtualBox:

```
$ git clone --depth=1 https://github.com/apache/
    incubator-openwhisk.git openwhisk
$ cd openwhisk/tools/vagrant
$ ./hello
```

Installing OpenWhisk – Kubernetes

Install OpenWhisk inside a Kubernetes Cluster:

```
$ minikube start --memory 4096 --kubernetes-version v1
    .10.5
$ minikube ssh -- sudo ip link set docker0 promisc on
$ kubectl label nodes --all openwhisk-role=invoker
$ helm init --wait
$ kubectl create clusterrolebinding tiller-cluster-admin
    \
--clusterrole=cluster-admin --serviceaccount=kube-system:
    default
$ git clone https://github.com/apache/incubator-openwhisk
    -deploy-kube
$ helm install ./incubator-openwhisk-deploy-kube/helm/
    openwhisk/ \
--name openwhisk --wait --timeout 900 \
--set whisk.ingress.type=NodePort \
--set whisk.ingress.api_host_name=$(minikube ip) \
--set whisk.ingress.api_host_port=31001 \
--set nginx.httpsNodePort=31001
```

Creating an application using MongoDB in OpenWhisk

Application Flow [5]

- The Application manages the Stock of a Market
- For this task it stores the data in a database
- The Application recieves parameters for product ID and number of items
- The Application consists of **OpenWhisk** and **MongoDB** (NoSQL) database

Functions as a Service mit OpenWhisk. Henry-Norbert Cocos, Marcus Legendre, Christian Baun. iX 12/2018, S.126-130, ISSN: 0935-9680

[5] Source Code available at: `https://github.com/OrangeFoil/openwhisk-examples/tree/master/inventory`

Creating a function in OpenWhisk

```
import pymongo

mongo_url = 'mongodb+srv://user:password@example.org/
    ↪ database'
mongodb_client = pymongo.MongoClient(mongo_url)
mongodb = mongodb_client.my_database


def main(params):
product_id = params['product_id']
stock_change = int(params['stock_change'])

result = mongodb.inventory.find_one_and_update(
{'product_id': product_id},
{'$inc': {'count': stock_change}},
upsert=True,
return_document=pymongo.collection.ReturnDocument.AFTER
)

return {
'product_id': result['product_id'],
'count': result['count']
}
```

Listing 8: File __main__.py

Deploy the function in OpenWhisk

Deploy the function in OpenWhisk:

```
$ mkdir tmp-build
$ cp __main__.py tmp-build/
$ pip3 install dnspython pymongo -t tmp-build/
$ cd tmp-build
$ zip -r ../exec.zip ./*
$ cd ..
```

As a ZIP-File. . .

In order to run the function in OpenWhisk, the dependencies `dnspython` and `pymongo` need to be installed with the Python Package Manager `pip3` (Python Installs Packages)! Those dependencies are stored with the application inside a ZIP-File.

Creating Actions and Triggers

In the OpenWhisk platform events are characterized by **Trigger**. An **Action** is used to invoke the function. A **Rule** binds an **Action** to a **Trigger**.

Creating an action for updating the database:

```
$ wsk action create updateInventory exec.zip --kind
    python:3
```

Creating Triggers for increment and decrement operations:

```
$ wsk trigger create itemSold --param stock_change -1
$ wsk trigger create itemRestocked --param stock_change
    1
```

Creating a Rule and Trigger an Event

In order to invoke an action, a trigger needs to be fired. The Rule `restockRule` and `saleRule` are bound to the `updateInventory` action.

Creating A Rule to combine Triggers and Actions:

```
$ wsk rule create restockRule itemRestocked
    updateInventory
$ wsk rule create saleRule itemSold updateInventory
```

Firing the Trigger:

```
$ wsk trigger fire itemRestocked --param product_id 1234
    --param stock_change 100
$ wsk trigger fire itemSold --param product_id 5678
```

Trigger an Event

```
$ wsk trigger fire itemRestocked -p product_id 42 -p stock_change 10
ok: triggered /_/itemRestocked with id f3204cd9dc16412ba04cd9dc16212b02
$ wsk activation result --last
{
    "count": 39,
    "product_id": 42
}
$ wsk trigger fire itemSold -p product_id 42
ok: triggered /_/itemSold with id ed476347731f4f75876347731fdf751f
$ wsk trigger fire itemSold -p product_id 42
ok: triggered /_/itemSold with id c1a15500f7e149c1a15500f7e1a9c144
$ wsk activation result --last
{
    "count": 37,
    "product_id": 42
}
```

Figure: Action Invocation

OpenWhisk Benefits

OpenWhisk has the following benefits:

- Open Source
- Deployment of functions
- Autoscaling
- Robust and flexible (ideal for production)
- Migration to public offering IBM Cloud Functions possible

OpenWhisk and IBM Cloud Functions

OpenWhisk is the basis of the public offering IBM Cloud Functions. Therefore applications developed for OpenWhisk can be ported to IBM Cloud Functions and vice versa without additional refactoring. This fact gives Enterpises more flexibility in developing their service offering!

Conclusion

Function as a Service characteristics:

- More fine grained business model (payment per invocation)
- Functions have no side effects, stateless model
- Scaling of functions with Container Virtualization (Docker)
- Shorter developement and deployment cycles (DevOps)
- Suitable technology for microservices

Outlook

FaaS is a new technology in the field of Cloud Platform Services. With the developement of IoT, Smart Homes and other event-driven technologies the number of private FaaS Frameworks and public FaaS offerings will grow in the near future!

[1] M. Eder, “Hypervisor- vs. Container-based Virtualization,” https://www.net.in.tum.de/fileadmin/TUM/NET/NET-2016-07-1/NET-2016-07-1_01.pdf, accessed December 10, 2020.

[2] O. Liebel, “Skalierbare Container-Infrastrukturen : das Handbuch für Administratoren,” Bonn, 2019. [Online]. Available: http://deposit.dnb.de/cgi-bin/dokserv?id=d3636e4510914dd8890a0bebc78593d0&prov=M&dok_var=1&dok_ext=htm

[3] “Docker Docs,” https://docs.docker.com/, accessed December 10, 2020.

[4] “Docker Swarm,” https://docs.docker.com/engine/swarm/, accessed December 10, 2020.

[5] “Golem – Mehr Zeit für den Code,” https://www.golem.de/news/serverless-computing-mehr-zeit-fuer-den-code-1811-137516.html, accessed December 10, 2020.

[6] "AWS Lambda," https://aws.amazon.com/lambda/, accessed December 10, 2020.

[7] "Google Cloud Functions," https://cloud.google.com/functions/, accessed December 10, 2020.

[8] "IBM Cloud Functions," https://www.ibm.com/cloud/functions, accessed December 10, 2020.

[9] "Apache OpenWhisk," https://openwhisk.apache.org/, accessed December 10, 2020.

[10] I. Baldini, P. C. Castro, K. S. Chang, P. Cheng, S. J. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. M. Rabbah, A. Slominski, and P. Suter, "Serverless Computing: Current Trends and Open Problems," *CoRR*, vol. abs/1706.03178, 2017. [Online]. Available: http://arxiv.org/abs/1706.03178

[11] T. Lynn, P. Rosati, A. Lejeune, and V. Emeakaroha, "A Preliminary Review of Enterprise Serverless Cloud Computing (Function-as-a-Service) Platforms," in *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, Dec 2017, pp. 162–169.

[12] "OpenFaaS - Serverless Functions Made Simple," https://docs.openfaas.com/, accessed December 10, 2020.

[13] "Your Serverless Raspberry Pi cluster with Docker," https://blog.alexellis.io/your-serverless-raspberry-pi-cluster/, accessed December 10, 2020.