
Project Report

Use OpenWhisk to create a function-as-a-service (FaaS) environment

Version 1

23/05/2019

Mr Leite

Auchabie Antoine

Ballu Timothée

Paul Yves-Marie

Pescio Vincent

Rouleau Valentin

Tremenbert Alban

Table of Contents

1. Introduction.....	3
2. Background.....	3
3. Specification and Design	8
4. Implementation.....	9
5. Results and Evaluation	11
6. Conclusions.....	14
7. Reflections on Learning	14
8. References.....	14

Table of Figures

Figure 1 - A view of cloud computing system	3
Figure 2 – Three main cloud computing models.....	4
Figure 3 - Cloud computing models with FaaS.....	5
Figure 4 - Cloud computing models services.....	5
Figure 5 - OpenWhisk programming model	6
Figure 6 - OpenWhisk system overview	7
Figure 7 - Internal flow of processing.....	8
Figure 8 - Minikube VM creation and initialization	9
Figure 9 - Helm initialization.....	9
Figure 10 - Kubernetes nodes configuration.....	10
Figure 11 - mycluster.yaml file	10
Figure 12 - OpenWhisk deployment with Helm	10
Figure 13 - OpenWhisk CLI configuration.....	11
Figure 14 - Node.js function	11
Figure 15 - Pods status	12
Figure 16 - Helm test	12
Figure 17 - OpenWhisk CLI properties.....	12
Figure 18 – Creation of the Node.js action.....	13
Figure 19 - OpenWhisk entities list in guest namespace	13
Figure 20 - Invocation of the Node.js action	13

Table of Abbreviations

FaaS: Function-as-a-Service

NIST: National Institute of Standards and Technology

IaaS: Infrastructure-as-a-Service

PaaS: Platform-as-a-Service

SaaS: Service-as-a-Service

CLI: Command Line Interface

VM: Virtual Machine

1. Introduction

To begin with, the goal of our project is to use OpenWhisk to create a function-as-a-service (FaaS) environment. Our work allowed us to understand the cloud computing better and to improve our technical skills. During the presentation in front of the class at the end of our project, we hope our classmates will also improve their understanding of cloud computing. To consider our project as successful, our environment created using OpenWhisk should support code written in Java, Python, Node.js and PHP and external resources such as RabbitMQ and MariaDB. To carry out our project, we started with a big research work about Faas cloud and more particularly the OpenWhisk operation. Then, we were able to develop our environment. Thanks to everyone's investment and team organisation, we managed to develop a local Faas environment using OpenWhisk. This report presents our work.

2. Background

According to the National Institute of Standards and Technology (NIST), "Cloud computing is a model for enabling convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction. This cloud model promotes availability and is composed of five essential characteristics, three service models, and four deployment models". This model offers a low operating cost, a high scalability, an easy access and a business risk reduction to cloud users. That's why the cloud computing model is being more and more used today.

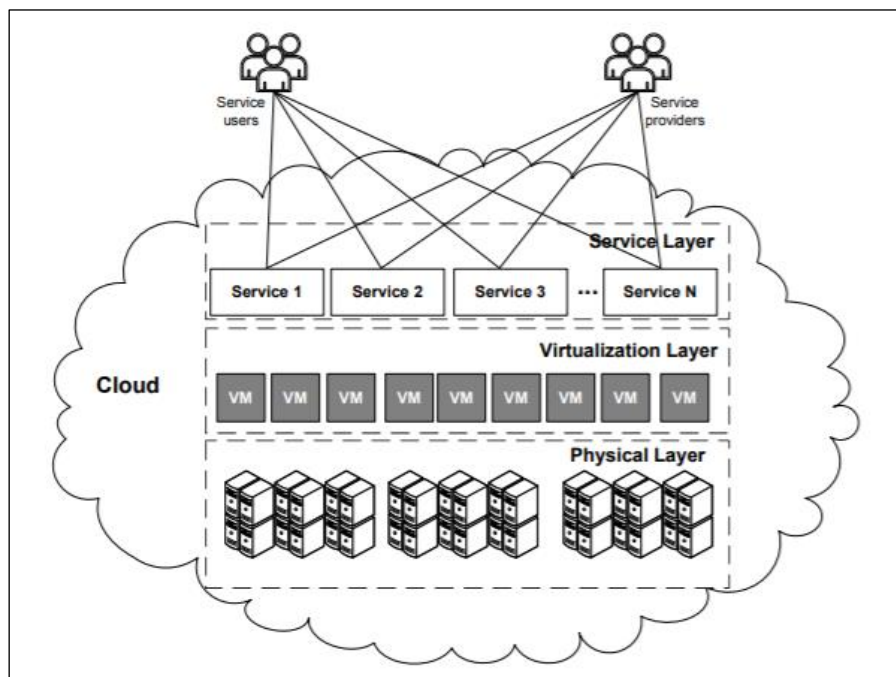


Figure 1 - A view of cloud computing system

Cloud computing providers offer their services according to different models. The NIST defines three standards models. Firstly, the Infrastructure-as-a-Service (IaaS) model provides virtualized computing resources over the internet. The cloud provider hosts the infrastructure components traditionally present in an on-premises data center, including servers, storage and networking hardware, as well as the virtualization or hypervisor layer. Secondly, the Platform-as-a-Service (PaaS) model provides a platform allowing customers to develop, run, and manage applications without the complexity of building and maintaining the infrastructure typically associated with developing and launching an application. In this way, customers can focus on creating and running applications rather than constructing and maintaining the underlying infrastructure and services. Thirdly, the Software-as-a-Service (SaaS) model hosts applications and makes them available to customers over the Internet. The provider hosts the customer's software and delivers it to approved end users over the internet. These three models are the main categories of cloud computing.

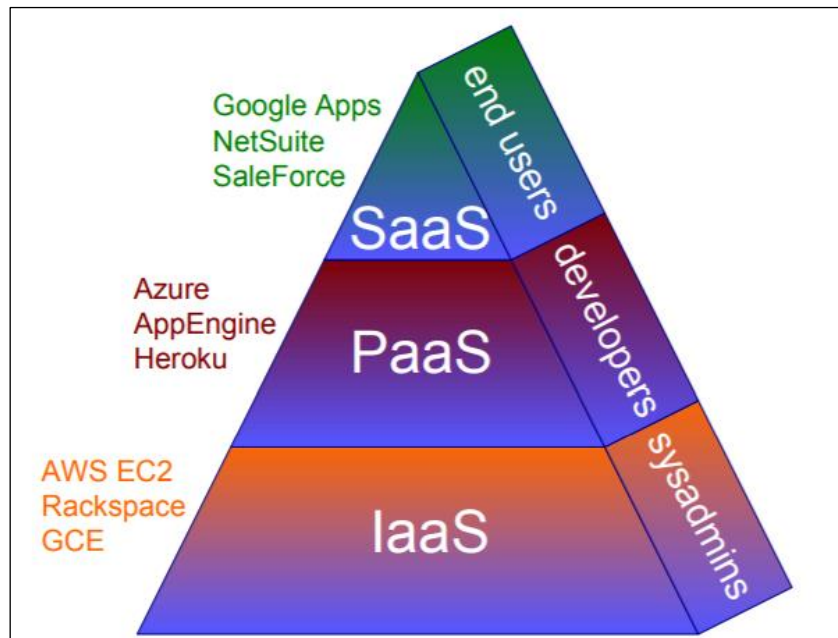


Figure 2 – Three main cloud computing models

The Function-as-a-Service (FaaS) model is another cloud computing model. This model is close to the PaaS model: it also provides a platform allowing customers to develop, run, and manage application functionalities without the complexity of building and maintaining the infrastructure typically associated with developing and launching an app. However, hosting services typically always have at least one server process running that receives external requests in the PaaS model whereas the FaaS model does not require any server process constantly being running. The FaaS model is included under the broader term serverless computing. The FaaS model offers some advantages to customers. Actually, developers can focus on coding and running their functions rather than constructing and maintaining the underlying infrastructure and services. Then, cloud customers only pay for the execution time of their functions.

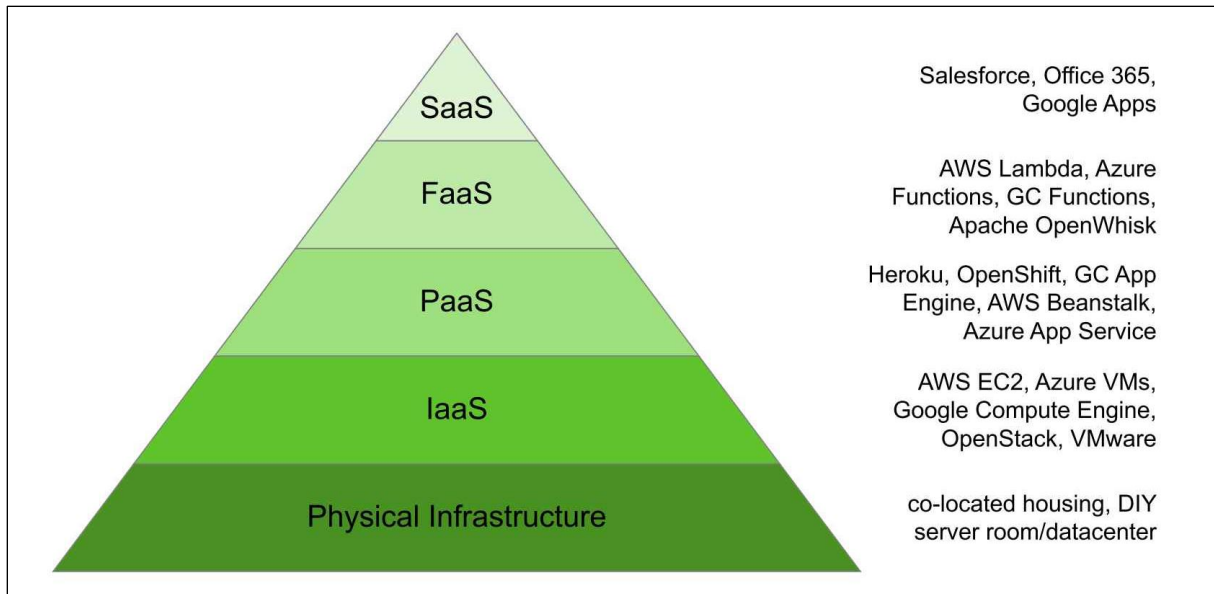


Figure 3 - Cloud computing models with FaaS

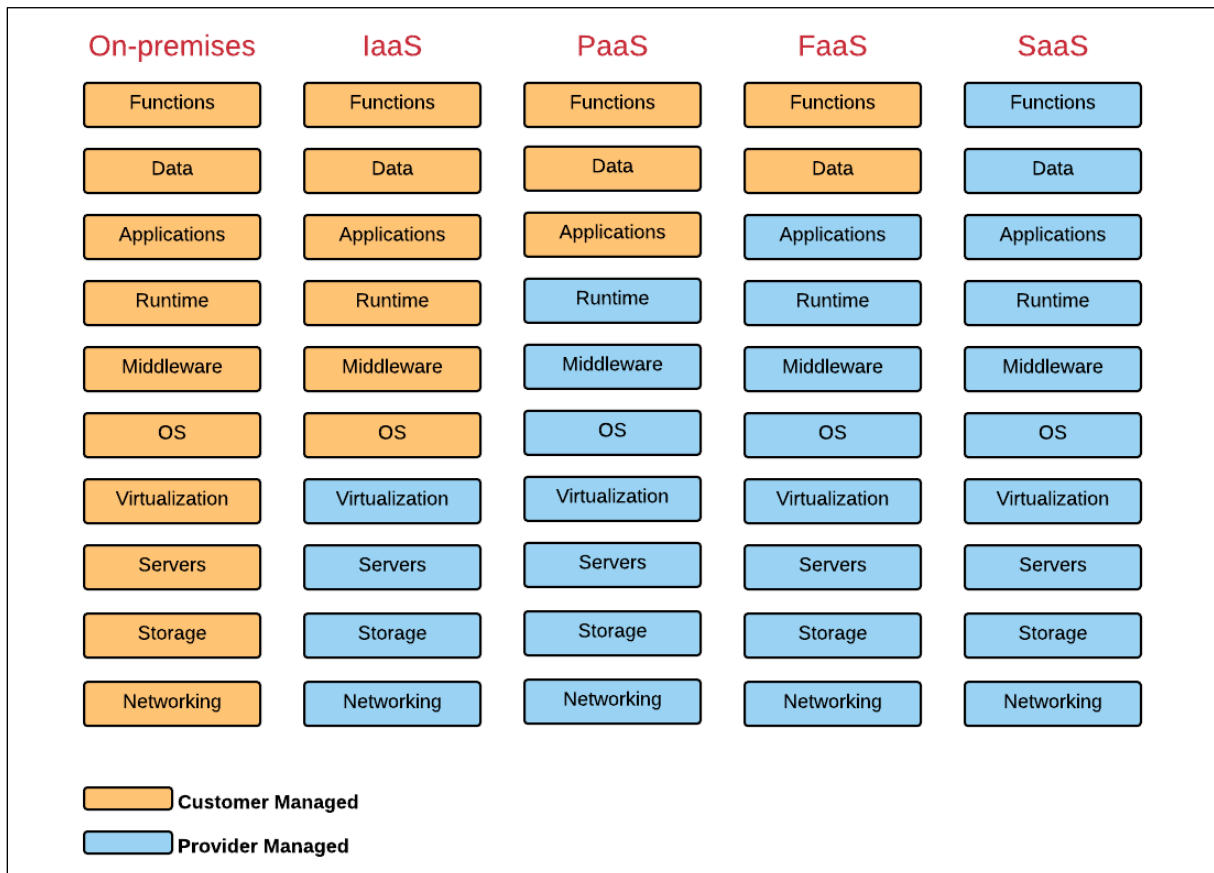


Figure 4 - Cloud computing models services

Apache OpenWhisk is an open source serverless cloud platform that executes functions in response to events at any scale. OpenWhisk manages the infrastructure, servers and scaling using Docker containers so we can focus on building applications. Then, OpenWhisk supports many deployment options both locally and within cloud infrastructure. Options include many of today's popular Container frameworks such as Kubernetes, Mesos, OpenShift and Compose. Moreover, OpenWhisk also supports a lot of languages such as NodeJS, Swift, Java, Go, Scala, Python, PHP, Ruby and Ballerina.

The OpenWhisk platform supports a programming model in which developers write functional logic (called Actions), in any supported programming language, that can be dynamically scheduled and run in response to associated events (via Triggers) from external sources (Feeds) or from HTTP requests. The project includes a REST API-based Command Line Interface (CLI) along with other tooling to support packaging, catalog services and many popular container deployment options.

Actions are stateless functions that run on the OpenWhisk platform. Actions encapsulate application logic to be executed in response to events. Actions can be invoked manually by the OpenWhisk REST API, OpenWhisk CLI, simple and user-created APIs or automated via Triggers.

Triggers and rules allow to automate OpenWhisk Actions in response to events. More exactly, triggers are named channels for classes or kinds of events sent from Event Sources and rules are used to associate one trigger with one action. After this kind of association is created, each time a trigger event is fired, the action is invoked.

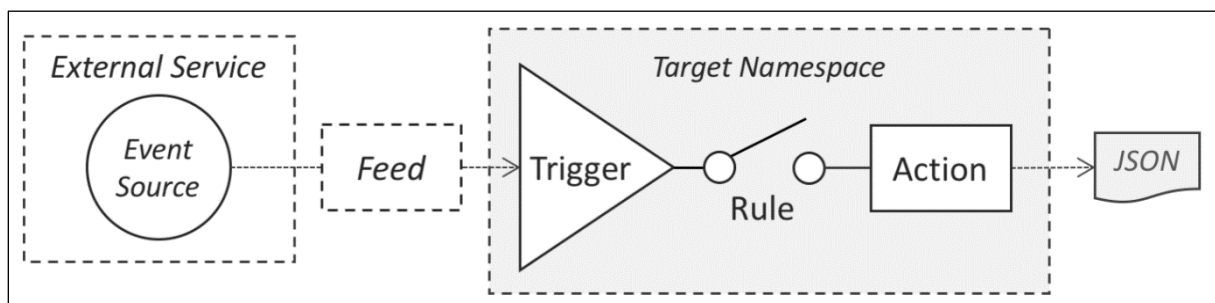


Figure 5 - OpenWhisk programming model

The event sources can be varied: a message arriving on message queues, a change in databases or in document stores, a website or web application interaction, etc.

To conclude, OpenWhisk provides a big amount of possibilities and that's why it is a powerful solution for implementing a FaaS environment.

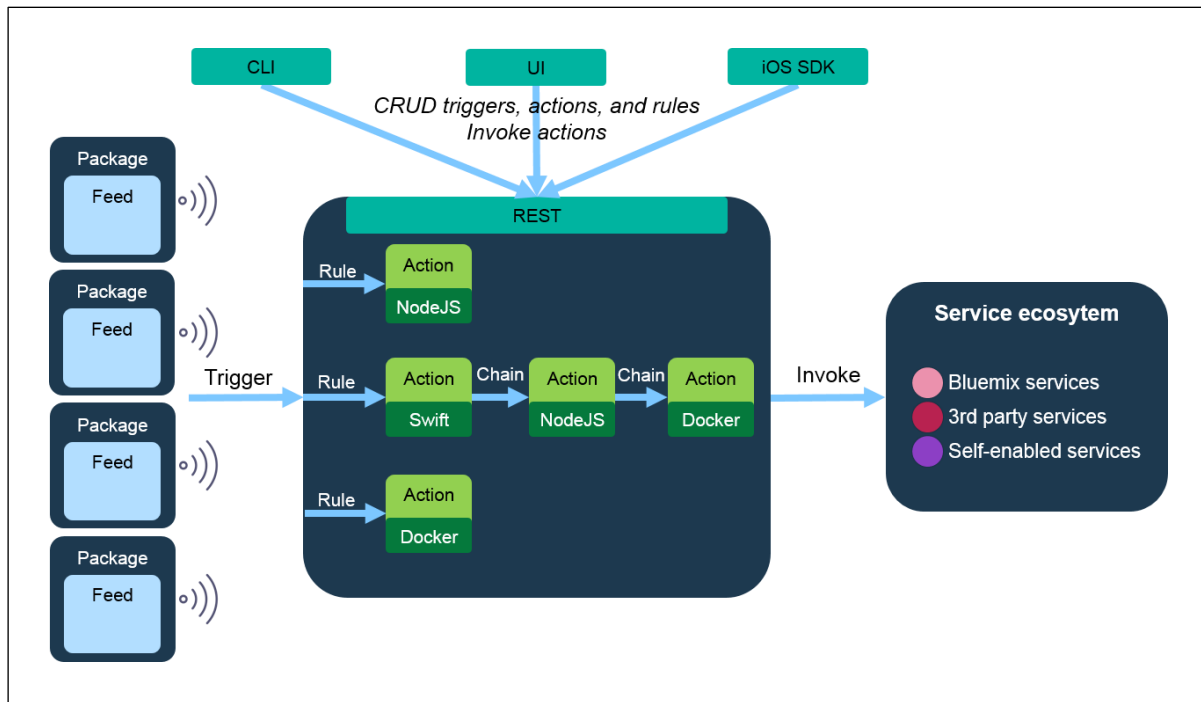


Figure 6 - OpenWhisk system overview

Being an open source project, OpenWhisk uses different external components, including Nginx, Kafka, Docker and CouchDB. Firstly, Nginx is the entry point to the system. It is used as an HTTP and reverse proxy server. Then, Nginx forwards entering HTTP requests to the Controller. This Controller translates HTTP requests to understand what the user is trying to do. After that, the Controller identifies the user and verifies he has the privilege to do what he wants to do using a CouchDB instance. If this step succeeds, the Controller loads for example the action the user wants to run from a database in CouchDB. Next, the Controller chooses an available Invoker to invoke the action requested. The Controller communicates with different Invokers through messages by Kafka, a high-throughput and distributed publish-subscribe messaging system. Then, when an Invoker must invoke an action, it spawns a Docker container where the action code is executed. Therefore, the result is retrieved by the Invoker and the Docker container is destroyed. Finally, the Invoker stores the result in a specific database in CouchDB.

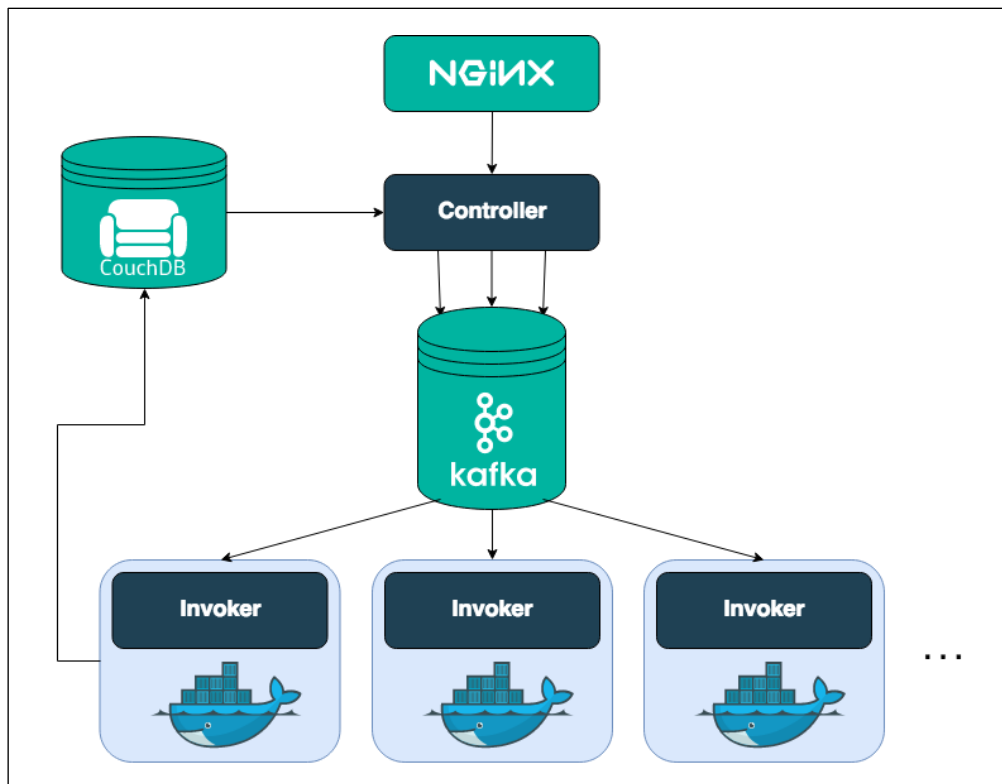


Figure 7 - Internal flow of processing

3. Specification and Design

To reach our goal, we had to create a FaaS environment using OpenWhisk. Logically, we created this environment locally on our machines. OpenWhisk offers a lot of deployment options. For local development, OpenWhisk recommends using Kubernetes. Kubernetes is a container orchestration platform that automates the deployment, scaling, and management of containerized applications. The first step is to create a Kubernetes cluster to support an OpenWhisk deployment. We opted for Minikube because it provides a single node Kubernetes cluster running inside a Virtual Machine (VM). For this VM, we opted for VirtualBox because some of us work on Windows machines. Then, for local development with Kubernetes, OpenWhisk can easily be deployed using Helm charts. So, we also opted for using Helm. Finally, OpenWhisk offers a Command Line Interface (CLI) to easily create, run and manage OpenWhisk entities. With this configuration, we should be able to create a FaaS environment and to manage it.

Then, our FaaS environment must support codes written in Java, Python, Node.js and PHP. So, we decided to create a simple function for each of these languages in order to test our environment. These functions will take one or two parameters and will return a simple json object. Moreover, external resources such as RabbitMQ and MariaDB must be supported by our FaaS environment. So, we hope being able to test these external resources, even if it seems more difficult to realize.

4. Implementation

As we saw in the specification part, the first step is to create a Kubernetes cluster. In our case, this Kubernetes cluster is provided by Minikube and Minikube will use a VirtualBox VM. So, we start by downloading VirtualBox, Minikube and Kubectl. Then, we can create a Minikube VM with settings recommended by OpenWhisk. Next, we put the docker network in promiscuous mode.

```

Microsoft Windows [version 10.0.17134.706]
(c) 2018 Microsoft Corporation. Tous droits réservés.

C:\Users\roule>minikube start --memory 4096 --cpus 2 --kubernetes-version v1.10.5
o minikube v1.0.1 on windows (amd64)
$ Downloading Kubernetes v1.10.5 images in the background ...
> Creating virtualbox VM (CPUs=2, Memory=4096MB, Disk=20000MB) ...
- "minikube" IP address is 192.168.99.100
- Configuring Docker as the container runtime ...
- Version of container runtime is 18.06.3-ce
: Waiting for image downloads to complete ...
- Preparing Kubernetes environment ...
- Pulling images required by Kubernetes v1.10.5 ...
X Unable to pull images, which may be OK: pull command is not supported by kubeadm v1.10.5
- Launching Kubernetes v1.10.5 using kubeadm ...
: Waiting for pods: apiserver proxy etcd scheduler controller dns
- Configuring cluster permissions ...
- Verifying component health .....
+ kubectl is now configured to use "minikube"
= Done! Thank you for using minikube!

C:\Users\roule>minikube ssh -- sudo ip link set docker0 promisc on

C:\Users\roule>
  
```

Figure 8 - Minikube VM creation and initialization

Now, our Minikube cluster is ready to deploy OpenWhisk. As we saw in the specification part, we opted for using Helm to simplify the deployment and management of applications on our Kubernetes cluster. So, we start by downloading Helm. When the Helm command line tool is installed, we can initialize Helm on our development machines.

```

C:\Users\roule>helm init
$HELM_HOME has been configured at C:\Users\roule\.helm.

Tiller (the Helm server-side component) has been installed into your Kubernetes Cluster.

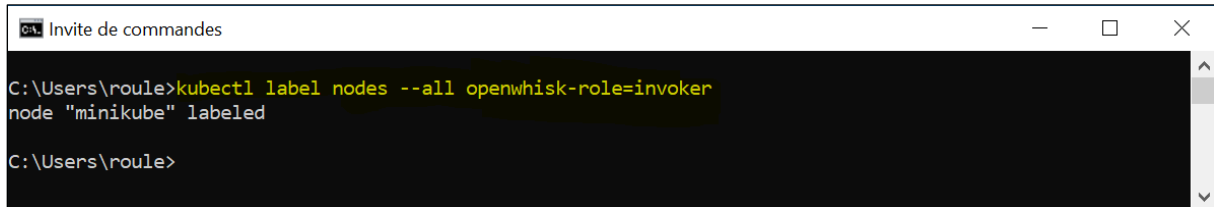
Please note: by default, Tiller is deployed with an insecure 'allow unauthenticated users' policy.
To prevent this, run `helm init` with the --tiller-tls-verify flag.
For more information on securing your installation see: https://docs.helm.sh/using_helm/#securing-your-helm-installation
Happy Helming!

C:\Users\roule>kubectl create clusterrolebinding tiller-cluster-admin --clusterrole=cluster-admin --serviceaccount=kube-system:default
clusterrolebinding.rbac.authorization.k8s.io "tiller-cluster-admin" created

C:\Users\roule>
  
```

Figure 9 - Helm initialization

Now that we have our Kubernetes cluster and have installed and initialized Helm, we are ready to deploy OpenWhisk. To prepare the deployment, we start by indicating the Kubernetes worker nodes that should be used for the execution of user containers by OpenWhisk's Invokers. Then, we must create a “mycluster.yaml” file to record key aspects of our Kubernetes cluster that are needed to configure the deployment of OpenWhisk into our cluster.

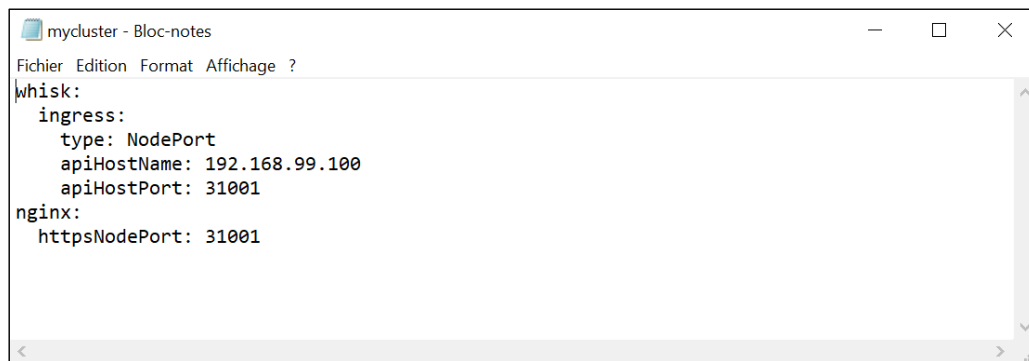


```

C:\Users\roule>kubectll label nodes --all openwhisk-role=invoker
node "minikube" labeled

C:\Users\roule>
  
```

Figure 10 - Kubernetes nodes configuration

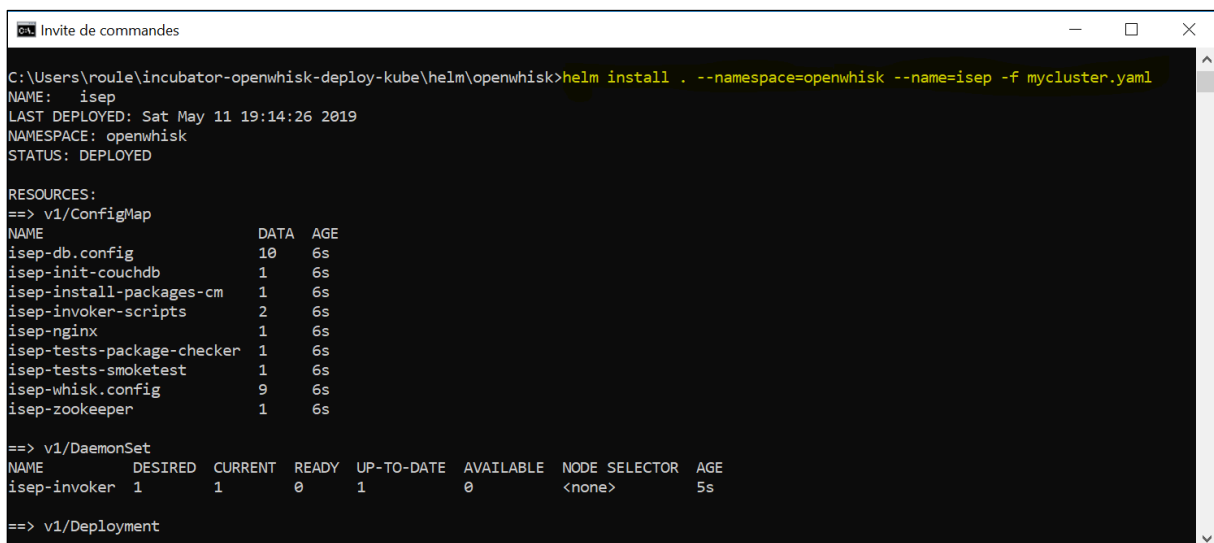


```

whisk:
  ingress:
    type: NodePort
    apiHostName: 192.168.99.100
    apiHostPort: 31001
  nginx:
    httpsNodePort: 31001
  
```

Figure 11 - mycluster.yaml file

Now, we can deploy OpenWhisk with Helm using the “mycluster.yaml” file.



```

C:\Users\roule\incubator-openwhisk-deploy-kube\helm\openwhisk>helm install . --namespace=openwhisk --name=isep -f mycluster.yaml
NAME: isep
LAST DEPLOYED: Sat May 11 19:14:26 2019
NAMESPACE: openwhisk
STATUS: DEPLOYED

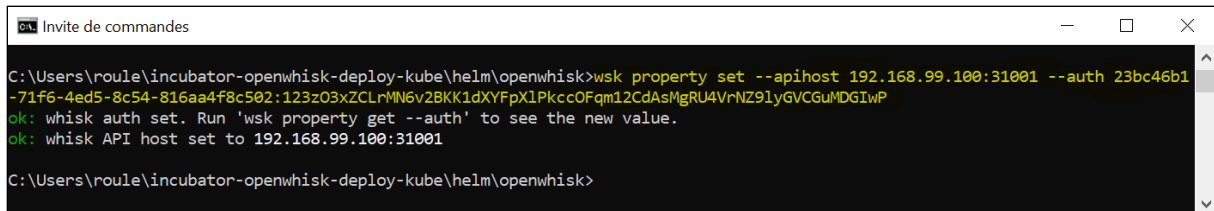
RESOURCES:
==> v1/ConfigMap
NAME          DATA  AGE
isep-db.config  10    6s
isep-init-couchdb  1    6s
isep-install-packages-cm  1    6s
isep-invoker-scripts  2    6s
isep-nginx     1    6s
isep-tests-package-checker  1    6s
isep-tests-smoketest  1    6s
isep-whisk.config  9    6s
isep-zookeeper  1    6s

==> v1/DaemonSet
NAME          DESIRED  CURRENT  READY  UP-TO-DATE  AVAILABLE  NODE SELECTOR  AGE
isep-invoker  1        1        0      1           0          <none>         5s

==> v1/Deployment
  
```

Figure 12 - OpenWhisk deployment with Helm

The Openwhisk deployment takes a few minutes. Actually, different pods must be installed and run. Once the isep-install-packages pod is in the Completed state, our OpenWhisk deployment is ready to be used. Then, we must configure our OpenWhisk CLI by setting the auth and apihost properties. For local development, the auth key property is available on GitHub.



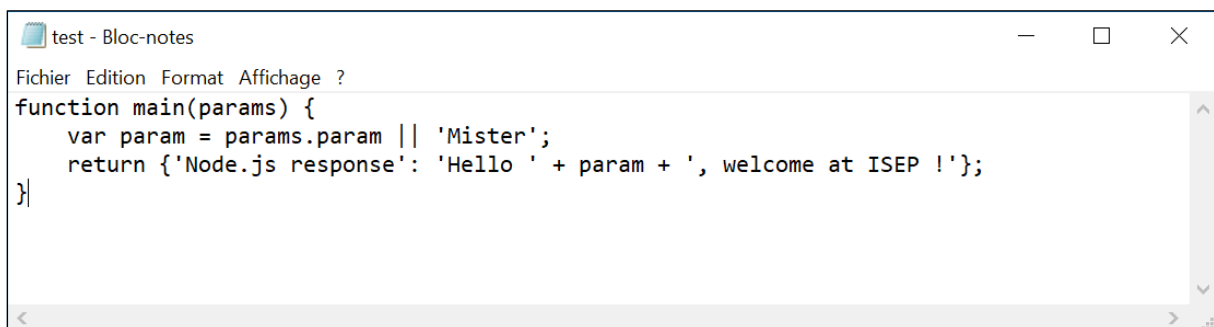
```

C:\Users\roule\incubator-openwhisk-deploy-kube\helm\openwhisk>wsk property set --apihost 192.168.99.100:31001 --auth 23bc46b1-71f6-4ed5-8c54-816aa4f8c502:123z03xZCLrMN6v2BKK1dXYFpX1PkccOFqm12CdAsMgRU4VrNZ9lyGVCguMDGIwP
ok: whisk auth set. Run 'wsk property get --auth' to see the new value.
ok: whisk API host set to 192.168.99.100:31001
C:\Users\roule\incubator-openwhisk-deploy-kube\helm\openwhisk>

```

Figure 13 - OpenWhisk CLI configuration

As we saw in the specification part, we planned to write simple functions to test our FaaS environment later. So, we write a similar function in Java, Python, Node.js and PHP which takes one parameter and returns a simple json object.



```

function main(params) {
  var param = params.param || 'Mister';
  return {'Node.js response': 'Hello ' + param + ', welcome at ISEP !'};
}

```

Figure 14 - Node.js function

To conclude, our OpenWhisk FaaS environment is normally created and operational. Moreover, we have simple functions to test and see if this environment supports codes written in the different required languages.

5. Results and Evaluation

After a few minutes, we can check if our OpenWhisk FaaS environment is correctly deployed. For this, we check that all pods have been run, and more particularly the isep-install-packages pod. Moreover, we can do a test using Helm to run a basic verification test.

```

C:\Users\roule\incubator-openwhisk-deploy-kube\helm\openwhisk>kubect1 get pods -n openwhisk
NAME                                READY    STATUS    RESTARTS   AGE
isep-alarmprovider-9798d489c-rsv7c 1/1      Running   0           44m
isep-apigateway-fb6ff866c-ngcg7     1/1      Running   0           44m
isep-cloudantprovider-cb5f47cb5-qtznx 1/1      Running   0           44m
isep-controller-0                   1/1      Running   13          44m
isep-couchdb-686b6d979b-hmrc2       1/1      Running   0           44m
isep-init-couchdb-vf4cb              0/1      Completed 0           44m
isep-install-packages-zk6rp         0/1      Completed 0           44m
isep-invoker-t7w6f                  1/1      Running   0           44m
isep-kafka-0                         1/1      Running   0           44m
isep-kafkaprovider-6585cbd88-95qtd   1/1      Running   0           44m
isep-nginx-6b45ccd6df-6z7ph         1/1      Running   0           44m
isep-redis-5f89f4c8cc-dgrvt         1/1      Running   0           44m
isep-wskadmin                       1/1      Running   0           44m
isep-zookeeper-0                    1/1      Running   0           44m
C:\Users\roule\incubator-openwhisk-deploy-kube\helm\openwhisk>

```

Figure 15 - Pods status

```

C:\Users\roule\incubator-openwhisk-deploy-kube\helm\openwhisk>helm test isep
RUNNING: isep-tests-smoketest
PASSED: isep-tests-smoketest
RUNNING: isep-tests-package-checker
PASSED: isep-tests-package-checker
C:\Users\roule\incubator-openwhisk-deploy-kube\helm\openwhisk>

```

Figure 16 - Helm test

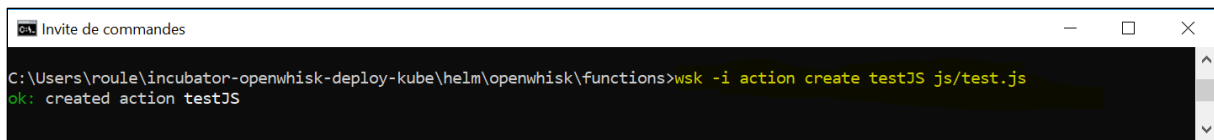
After these tests, our OpenWhisk FaaS environment seems correctly deployed. Now, we can test our environment. For this, we start by the configuration of our OpenWhisk CLI by setting the namespace property. For local development, the namespace property value is “guest”. Then, we can create actions using the simple functions written in the implementation part. Next, we can check if these actions have been created. Then, we can invoke these actions to test our FaaS environment.

```

C:\Users\roule\incubator-openwhisk-deploy-kube\helm\openwhisk>wsk -i property get
client cert
Client key
whisk auth                23bc46b1-71f6-4ed5-8c54-816aa4f8c502:123z03xZCLrMn6v2BKK1dXYFpX1PkccOFqm12CdAsMgRU4VrNZ91yGVCGuMDGIwP
whisk API host            192.168.99.100:31001
whisk API version         v1
whisk namespace           guest
whisk CLI version         2019-03-20T20:17:38.064+0000
whisk API build           2019-04-05-18:03:02Z
whisk API build number    20190405a
C:\Users\roule\incubator-openwhisk-deploy-kube\helm\openwhisk>

```

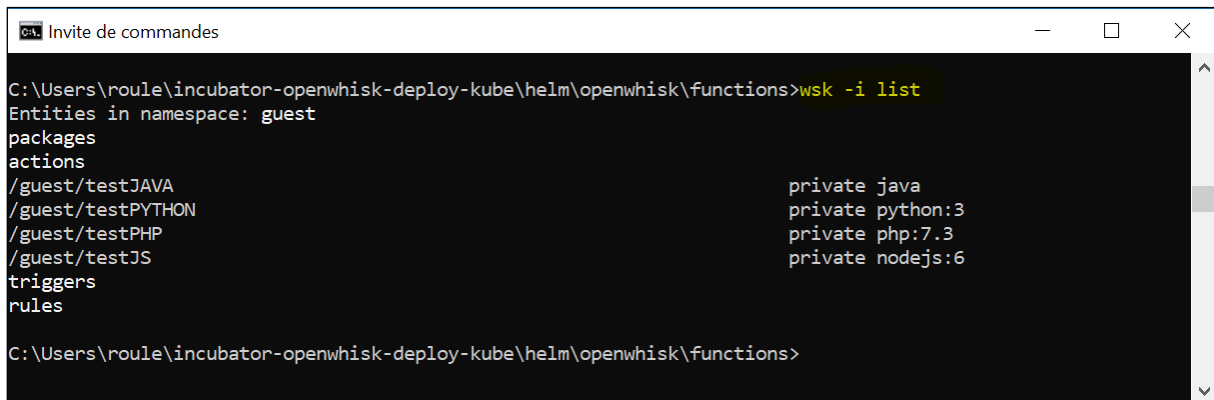
Figure 17 - OpenWhisk CLI properties



```

C:\Users\roule\incubator-openwhisk-deploy-kube\helm\openwhisk\functions>wsk -i action create testJS js/test.js
ok: created action testJS
  
```

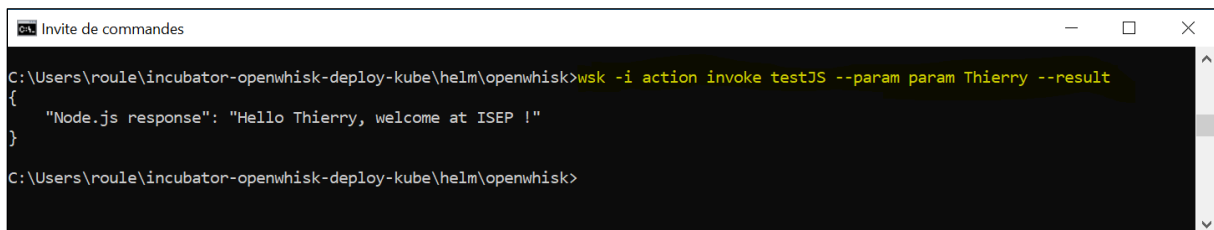
Figure 18 – Creation of the Node.js action



```

C:\Users\roule\incubator-openwhisk-deploy-kube\helm\openwhisk\functions>wsk -i list
Entities in namespace: guest
packages
actions
/guest/testJAVA                                private java
/guest/testPYTHON                             private python:3
/guest/testPHP                                private php:7.3
/guest/testJS                                 private nodejs:6
triggers
rules
C:\Users\roule\incubator-openwhisk-deploy-kube\helm\openwhisk\functions>
  
```

Figure 19 - OpenWhisk entities list in guest namespace



```

C:\Users\roule\incubator-openwhisk-deploy-kube\helm\openwhisk>wsk -i action invoke testJS --param param Thierry --result
{
  "Node.js response": "Hello Thierry, welcome at ISEP !"
}
C:\Users\roule\incubator-openwhisk-deploy-kube\helm\openwhisk>
  
```

Figure 20 - Invocation of the Node.js action

To sum up, our actions return the expected results. So, we can assure that our OpenWhisk FaaS environment is correctly deployed and works. We also can affirm that codes written in Java, Node.js, Python and PHP are supported by our FaaS environment. Logically, our environment should also support external resources such as RabbitMQ and MariaDB. However, we have not managed to test these external resources. Actually, such tests require the use of other packages which are way harder to set up.

6. Conclusions

As a reminder, our project goal was to create a FaaS environment using Apache OpenWhisk. This environment had to support codes written in Java, Node.js, Python and PHP and external resources such as RabbitMQ and MariaDB. Thanks to our good research work, we understood the global functioning of OpenWhisk and so we opted for a good configuration to achieve our FaaS environment creation. Therefore, we managed to create our FaaS environment. Then, as we saw in the background part, OpenWhisk is a powerful open source solution. That's why our environment supports codes written in the different required languages. We realized tests to assure this part. Next, our environment also supports all expected external resources. However, we have not managed to test this part because it was much more difficult. Moreover, OpenWhisk offers a large amount of possibilities and so we could have done other tests, but we ran out of time.

7. Reflections on Learning

This project allowed us to understand cloud computing better and more particularly its different models and how it works. So, we improved our global understanding of cloud computing thanks to this project. Then, we also improved our technical skills in developing, deploying and testing our FaaS environment with OpenWhisk. More particularly, we discovered some interesting components as Minikube or Helm. We may be required to work with these components again in our future projects. To conclude, this project was very interesting and constructive.

8. References

- <https://openwhisk.apache.org/>
- <https://helm.sh/>
- <https://kubernetes.io/>
- <https://www.virtualbox.org/>
- <https://github.com/apache/incubator-openwhisk-deploy-kube/>
- <https://medium.com/openwhisk/deploying-openwhisk-on-kubernetes-3f55f781fbab>
- <https://www.lebigdata.fr/faas-definition>
- <https://www.lemagit.fr/definition/Function-as-a-service-FaaS>
- https://en.wikipedia.org/wiki/Function_as_a_service