

ECOLE SUPERIEURE D'INFORMATIQUE
SALAMA

République Démocratique Du Congo
Province du Haut-Katanga
Lubumbashi

www.esisalama.org

JUIN 2023



TRAVAIL PRATIQUE DU COURS DE DEVOPS

UTILISATION FLASK, DOCKERFILE, JENKINS

Par : KALALA MUKENDI John Frank
NGANDU MIKOMBE Steve

Dans un premier temps, j'ai commencé par étudier les exigences du projet. J'ai ainsi pu identifier les différentes tâches à accomplir et les ressources dont j'aurais besoin.

Comme spécifier dans le slide, le projet est structuré en 3 parties :

- Création d'une application Restful API avec Flask avec au minimum deux ressources de votre choix.
- Créer un fichier docker qui expose le service au port 9000.
- Créer un fichier Jenkinsfile pour générer le conteneur docker et l'exécuter.

Pour ce projet, j'ai préféré télécharger et installer Debian sur le sous-système linux WSL 2.

1. Création d'une application Restful API avec Flask avec au minimum deux ressources de votre choix.

Pour cette étape, j'ai commencé par créer un environnement virtuel Python et y ai installé Flask. Ensuite, j'ai créé un fichier Python nommé "myappjfk.py" qui contient le code de mon application Flask.

Dans ce fichier, j'ai importé la classe Flask et j'ai créé une instance de cette classe. Ensuite, j'ai créé deux fonctions pour les deux ressources que j'ai choisies. Ces fonctions renvoient des données dans un format JSON à l'utilisateur qui accède aux ressources.

Voici le code pour cette étape :

```
from flask import Flask, jsonify

myappjfk = Flask(__name__)

pcs = [
    {'id': 1, 'marque': 'HP'},
    {'id': 2, 'marque': 'SAMSUNG'}
]

users = [
    {'id': 1, 'name': 'KALALA'},
    {'id': 2, 'name': 'NGANDU'}
]

@myappjfk.route('/pcs', methods=['GET'])
def get_pcs():
    return jsonify({'pcs': pcs})

@myappjfk.route('/users', methods=['GET'])
def get_users():
    return jsonify({'users': users})

if __name__ == '__main__':
    myappjfk.run(host='0.0.0.0', port=9000, debug=True)
```

2. Création d'un fichier Docker qui expose le service au port 9000.

Pour cette étape, j'ai créé un fichier Docker nommé "Dockerfile" qui contient les instructions pour créer une image Docker de mon application Flask.

Dans ce fichier, j'ai d'abord spécifié l'image de base, puis j'ai copié le contenu de mon application Flask dans le conteneur. J'ai ensuite installé les dépendances nécessaires et exposé le port 9000.

```
FROM python:3.9-slim-buster
```

```
WORKDIR /mnt/c/myappjfk
```

```
COPY . /mnt/c/myappjfk
```

```
RUN pip install --no-cache-dir -r requirements.txt
```

```
EXPOSE 9000
```

```
CMD ["python3", "myappjfk.py"]
```

3. Création d'un fichier Jenkinsfile pour générer le conteneur Docker et l'exécuter.

Pour cette étape, j'ai créé un fichier Jenkinsfile qui contient les instructions pour générer le conteneur Docker et l'exécuter.

Dans ce fichier, j'ai d'abord spécifié l'image de base, puis j'ai créé une étape pour builder l'image Docker à partir du fichier "Dockerfile". J'ai ensuite créé une étape pour pousser l'image sur un registre Docker, et enfin une étape pour exécuter le conteneur Docker en utilisant l'image précédemment créée.

Voici le code :

```
pipeline {  
  agent any  
  stages {  
    stage('Build') {  
      steps {  
        script {  
          docker.build("myimagejfk:latest", "-f Dockerfile .")  
        }  
      }  
    }  
    stage('Run') {  
      steps {  

```

```
script {
  docker.run("myimagejfk:latest", "-p 9000:9000")
}
```

RESULTAT

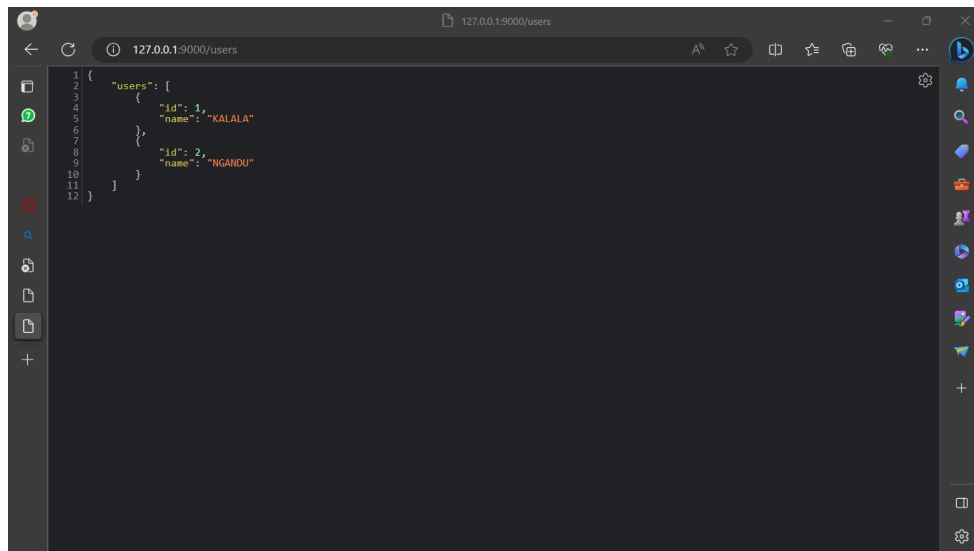
```

emutienz@JFK: /mnt/c
[+] Building 15.7s (10/10) FINISHED
=> [internal] load .dockerignore 1.2s
=> => transferring context: 2B 0.0s
=> [internal] load build definition from Dockerfile 1.7s
=> => transferring dockerfile: 213B 0.4s
=> [internal] load metadata for docker.io/library/python:3.9-slim- 9.7s
=> [auth] library/python:pull token for registry-1.docker.io 0.0s
=> [1/4] FROM docker.io/library/python:3.9-slim-buster@sha256:320a 0.0s
=> [internal] load build context 0.6s
=> => transferring context: 197B 0.2s
=> CACHED [2/4] WORKDIR /mnt/c/myappjfk 0.0s
=> CACHED [3/4] COPY . /mnt/c/myappjfk 0.0s
=> CACHED [4/4] RUN pip install --no-cache-dir -r requirements.txt 0.0s
=> exporting to image 0.5s
=> => exporting layers 0.0s
=> => writing image sha256:247f5bb9e7a55c14eea5b689c16e71cc74247f7 0.2s
=> => naming to docker.io/library/myimagejfk 0.3s

What's Next?
View summary of image vulnerabilities and recommendations → docker scout quickview
emutienz@JFK:/mnt/c$ docker run myimagejfk
* Serving Flask app 'myappjfk'
* Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use a p
roduction WSGI server instead.
* Running on all addresses (0.0.0.0)
* Running on http://127.0.0.1:9000
* Running on http://172.17.0.2:9000
Press CTRL+C to quit
* Restarting with stat
* Debugger is active!
* Debugger PIN: 115-760-431

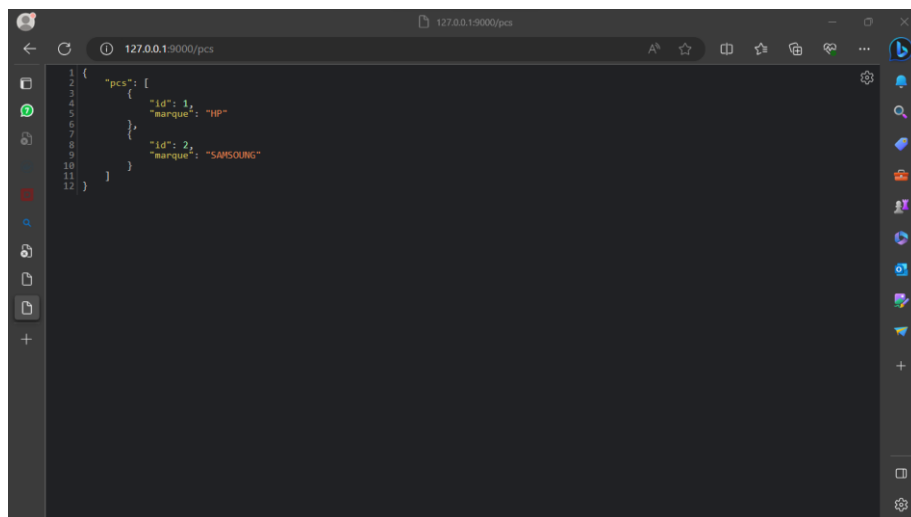
```

La ressource utilisateurs



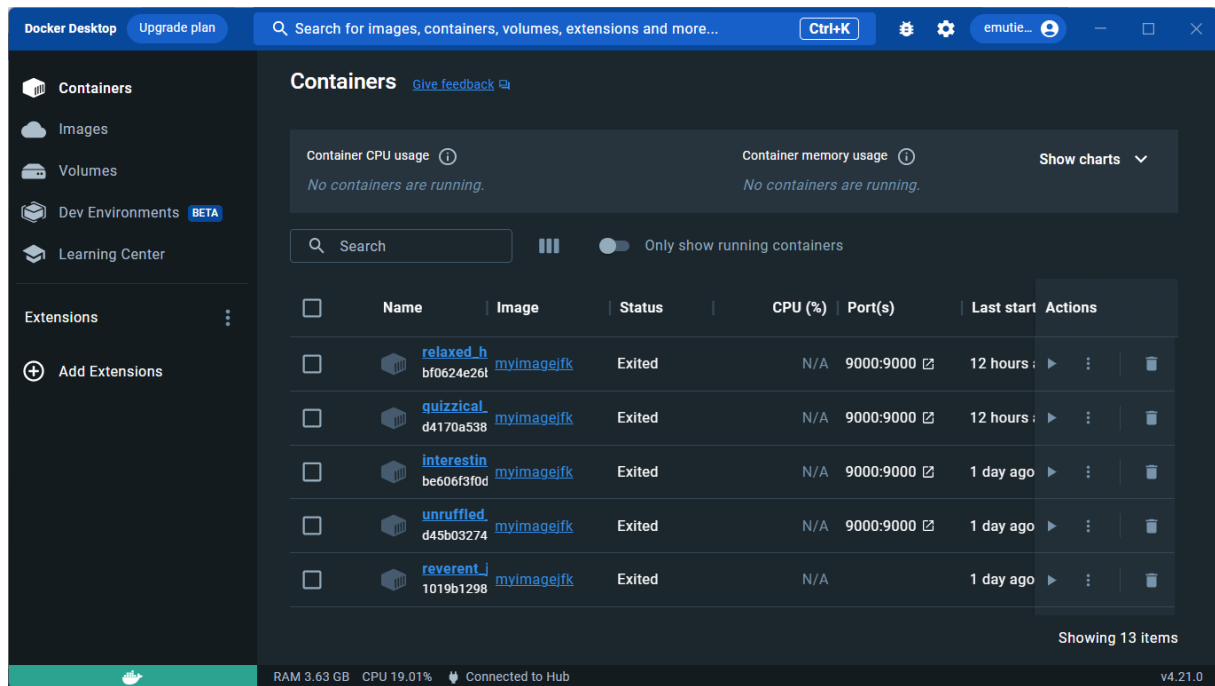
```
1 {
2   "users": [
3     {
4       "id": 1,
5       "name": "KALALA"
6     },
7     {
8       "id": 2,
9       "name": "NGANDU"
10    }
11  ]
12 }
```

La ressource PCs



```
1 {
2   "pcs": [
3     {
4       "id": 1,
5       "marque": "HP"
6     },
7     {
8       "id": 2,
9       "marque": "SAMSUNG"
10    }
11  ]
12 }
```

Docker desktop



En conclusion, ces différentes étapes ont permis de créer une application Restful API avec Flask, de la mettre en conteneur Docker et de l'exécuter grâce à un fichier Jenkinsfile. Le code utilisé pour chaque étape a permis de réaliser ce projet.