



# 5DVOP

## Student List

### Résumé

Rendu du projet de 5DVOP sur la Dockerisation d'un projet existant

DEVIGNE Henri

---

## Table des matières

---

<b>Contexte du projet .....</b>	<b>3</b>
<b>Préparation des images de notre application .....</b>	<b>3</b>
<b>PROJET API :.....</b>	<b>3</b>
CONTRAINTES .....	3
CONSTRUCTION DE L'IMAGE.....	4
TEST DE NOTRE IMAGE .....	6
<b>Projet Front .....</b>	<b>7</b>

---

## Contexte du projet

---

L'objectif de ce projet est de moderniser l'infrastructure actuelle qui repose sur deux serveurs dont la maintenance devient compliquée (aussi bien pour le matériel, que logiciel).

POZOS souhaite utiliser les technologies Docker afin de rendre « portable », « scalable » et plus facile à déployer ses deux applications qui nécessitent beaucoup trop de temps actuellement d'autant plus que les technologies utilisées par cette application, sont marquées comme dépréciées et ne sont plus facilement installables sur un poste client ou un serveur. ( Python 2.7 ne sera plus maintenu dès Janvier 2020 )

Les deux applications que POZOS souhaite déployer sont :

- API
  - Le projet API consiste en l'exécution d'un script python (student\_age.py) qui est compatible uniquement Python 2.7 et utilise les librairies FLASK.
- FRONT
  - Le projet FRONT contient un script PHP nommé « index.php » et se basera sur l'image **php:apache**

Ce document contiendra l'intégralité des démarches afin de construire et rendre disponible notre application.

---

## Préparation des images de notre application

---

---

### PROJET API :

---

Comme indiqué et demandé par la société, nous utiliserons l'image Docker de **python2.7-stretch** comme base pour notre image Docker.

## CONTRAINTES

- Le projet utilise **python2.7** et fonctionne initialement sur **Debian**, nous privilégions l'image de base Debian par rapport à Alpine concernant les paquets et les dépendances

qui sont actuellement uniquement maitrisées par les développeurs ainsi que pour son grand nombre de paquet dont certains très spécifiques comme `libldap2-dev`.

- Les paquets suivants doivent être installés :
  - `python-dev`
  - `libssl-dev`
  - `libldap2-dev`
  - `libssl-dev`
- Le programme python utilise des dépendances fournies par le gestionnaire de paquet Python appelé « PIP »
  - `flask`
  - `flask_httpauth`
  - `flask_simpleldap`
  - `python-dotenv`

## CONSTRUCTION DE L'IMAGE

Commençons par créer notre Dockerfile (`simple_api/Dockerfile`)

La première ligne de notre **Dockerfile** sera l'image sur laquelle elle est basée, à savoir : **python2.7-stretch**.

```
1 FROM python:2.7-stretch
```

Henri Devigne sera responsable du maintien et des évolutions de l'image, nous allons donc le mentionner avec la fonction Docker: « **MAINTAINER** ».

```
3 MAINTAINER Henri Devigne <henri.devigne@supinfo.com>
```

L'information est bien prise en compte comme nous pouvons le voir en faisant un docker image inspect sur notre image :

```
"Author": "Henri Devigne <henri.devigne@supinfo.com>,"
```

Enfin, nous allons procéder à l'installation des différents paquets Debian nécessaires au bon fonctionnement de notre projet, mais tout d'abord, nous allons devoir procéder à la mise à jour des repositories car l'image **python2.7-stretch** a volontairement supprimé la liste des repositories afin d'alléger l'image de base comme nous pouvons le voir avec la commande : « `docker image history python2.7-stretch --no-trunc` »

```
/BIN/SH -C APT-GET UPDATE && APT-GET INSTALL -Y --NO-INSTALL-RECOMMENDS CA-  
CERTIFICATES CURL NETBASE WGET && RM -RF /VAR/LIB/APT/LISTS/*
```

Nous allons lancer donc la commande pour mettre à jours nos paquets, et les installer.

```

5  RUN apt-get update \
6  && apt-get install --yes \
7    python-dev \
8    libsasl2-dev \
9    libldap2-dev \
10   libssl-dev

```

Regardons désormais le poids de notre image après avoir lancé le build ( `docker build simple_api -t simple_api` ) avec `docker image history simple_api`

```

7fc1962f67ff      16 seconds ago      /bin/sh -c apt-get update && apt-get instal... 66.3MB

```

Optimisons désormais le poids de notre image en supprimant le cache APT ( Gestionnaire de paquet sous debian ) en ajoutant à la fin du **RUN** la suppression du dossier de cache (`/var/lib/apt/lists`)

```

5  RUN apt-get update \
6  && apt-get install --yes \
7    python-dev \
8    libsasl2-dev \
9    libldap2-dev \
10   libssl-dev \
11   && rm -rf /var/lib/apt/lists

```

Reconstruisons désormais notre image et regardons la différence.

```

ce45322e6856      23 seconds ago      /bin/sh -c apt-get update && apt-get instal... 49.8MB

```

Nous avons gagné approximativement 17MB sur notre image grâce à cette optimisation.

Installons désormais les dépendances Flask avec l'instruction **RUN** :

```

13  RUN pip install flask flask_httpauth flask_simpleldap python-dotenv

```

Ajoutons notre fichier python « `student_age.py` » dans notre conteneur à la racine (`/`) comme demandé par les développeurs :

```

15  ADD student_age.py /student_age.py

```

Et enfin, définissons notre entrypoint, le port que peut exposer notre application, ainsi que son volume (`/data`)

```
17 CMD [ "python2.7", "/student_age.py" ]
18 VOLUME '/data'
19 EXPOSE '5000'
```

## TEST DE NOTRE IMAGE

Notre application est désormais prête à fonctionner, nous allons tester l'image avec un docker run

```
docker run -p 5000:5000 -v $PWD/simple_api/student_age.json:/data/student_age.json simple_api
* Serving Flask app "student_age" (lazy loading)
* Environment: production
  WARNING: This is a development server. Do not use it in a production deployment.
  Use a production WSGI server instead.
* Debug mode: on
* Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
* Restarting with stat
* Debugger is active!
* Debugger PIN: 506-434-369
```

Comme nous pouvons le voir l'application semble fonctionner correctement, mais nous devons malgré tout, tester son comportement avec un « curl »

```
curl -u toto:python -X GET http://127.0.0.1:5000/pozos/api/v1.0/get_student_ages
{
  "student_ages": {
    "alice": "12",
    "bob": "13"
  }
}
```

Nous avons donc notre « API » parfaitement fonctionnelle.

Le projet front se basera sur l'image **php:apache**.

## Contraintes

Voici les variables d'environnement nécessaire au bon fonctionnement du projet :

- USERNAME
- PASSWORD

Et le « website » doit être monté dans « /var/www/html ».

---

# Orchestration de nos conteneurs

---

Il est désormais temps d'orchestrer les conteneurs au préalable définis.

Créons de ce fait notre docker-compose.yml :

Commençons par indiquer la version utiliser ( ici, la version 3.7 ) :

```
1 version: '3.7'
```

Et définissons ensuite nos services.

Le premier service que nous allons définir est « API »

```
services:
  api:
    build: simple_api
    volumes:
      - ./simple_api/student_age.json:/data/student_age.json
```

Il n'est pas utile d'exposer le port de notre API dans la mesure où SEUL LE FRONT doit pouvoir accéder à l'API.

Toutefois, on l'exposera dans un premier dans pour une phase de test sur le port 5000

```
services:
  api:
    build: simple_api
    volumes:
      - ./simple_api/student_age.json:/data/student_age.json
    ports:
      - 5000:5000
```

Passons maintenant au service FRONT qui contient notre site internet.



```
front:
  image: php:apache
  volumes:
  - ./website:/var/www/html
  environment:
  - USERNAME=toto
  - PASSWORD=python
  ports:
  - 8000:80
  depends_on:
  - api
```

Comme expliqué précédemment, notre service est basé sur l'image php mode apache ( php:apache ) et est par défaut, configuré pour servir le dossier /var/www/html, ce pourquoi nous montons le dossier ./website dans le dossier /var/www/html.

Concernant les variables d'environnements, nous prenons soins de remplir les variables USERNAME, et PASSWORD comme indiqué dans la documentation des développeurs.

Nous précisons le port binding (à savoir ici : 8000 vers le port 80 du serveur web).

Et nous terminons par gérer la dépendances au service API avec le **depends\_on**, cela implique, que notre conteneur front démarrera uniquement une fois que le service API sera en ligne.

La dernière action qui nécessite une action, est celle de la configuration de l'adresse du serveur d'API :

```
$url = 'http://api:5000/pozos/api/v1.0/get_student_ages';
```

Nous utilisons ici : « api :5000 » étant donné qu'au sein d'un même réseau docker-compose, tout les conteneurs peuvent se joindre via leur nom de service déclaré dans le dockercompose.yml

---

*Automatisation des processus*

---

Afin de gagner un temps précieux et soulager les équipes INFRA, nous allons automatiser le processus de livraison en automatisant 2 processus :

- Construction des conteneurs
- Déploiement des conteneurs sur la production

Nous utiliserons Jenkins afin d'automatiser ces processus et nous déploierons notre propre Registry docker afin de stocker les images construites.

Nous déployons donc la registry docker conformément aux documentations de Docker

```
services:
  registry:
    image: registry:2
    restart: always
    volumes:
      - ./config.yml:/etc/docker/registry/config.yml
      - ./registry_data:/var/lib/registry
    ports:
      - 8000:80
    depends_on:
      - redis

  redis:
    restart: always
    image: redis:6.2.1-alpine
    command: "redis-server --requirepass superPassword"
```

Je précise que je déploie un redis, pour optimiser les Performances de la registry.

La société souhaitait utiliser un Centos 7. Ce que nous avons déployé.

Un jenkins est déployé avec l'image recommandée sur laquelle nous avons ajouté la commande docker pour permettre le build des images.

Pour se connecter au Jenkins, nous allons récupérer le mot de passe grâce à :  
docker-compose exec jenkins cat

/var/jenkins\_home/secrets/initialAdminPassword

Ce qui nous donne: efb92da2c1df4c1fa7860ce483b17379

## Configuration du projet Jenkins

### Gestion de code source

☐ Aucune

☒ Git

Repositories

Repository URL

Credentials

### Ce qui déclenche le build

☐ Déclencher les builds à distance (Par exemple, à partir de scripts)

☐ Construire après le build sur d'autres projets

☐ Construire périodiquement

☒ Scrutation de l'outil de gestion de version

Planning

⚠ Vouslez-vous vraiment dire "chaque minute" avec l'expression "\* \* \* \* \*" ? Peut-être voulez-vous dire "H \* \* \* \* \*"

Aurait été lancé à Monday, June 28, 2021 at 8:41:15 AM Coordinated Universal Time; prochaine exécution à Monday, June 28, 2021 at 8:41:15 AM Coordinated Universal Time.

☐ Ignore post-commit hooks

## Voici les jobs

### Exécuter un script shell

Commande

```
scp -r * "xxxxx:/var/www/5dvop/project"
ssh xxxxx -p 2121 "cd /var/www/5dvop/project| && docker-compose down && docker-compose up -d"
```

[Voir la liste des variables d'environnement disponibles](#)

Afin de pouvoir se connecter à la machine distante.

## On effectue un ssh-keygen depuis le contenu de la machine

```
jenkins@2e679588a5be:/ $ ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (/var/jenkins_home/.ssh/id_rsa):
Created directory '/var/jenkins_home/.ssh'.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /var/jenkins_home/.ssh/id_rsa.
Your public key has been saved in /var/jenkins_home/.ssh/id_rsa.pub.
The key fingerprint is:
SHA256:OPcEYmPxjz/B9h1j2povFmbaUTU47an+vJaMyzYfw jenkins@2e679588a5be
The key's randomart image is:
+-----[RSA 2048]-----+
|
|  oo. . |
| . . . o |
| . o o + + . |
| o o . + o = . |
| o =oooS. * |
| . +.o+ . o.o |
| o .+o . o+ . |
| o oo + .mo. |
| .ooo .o+E. |
+-----[SHA256]-----+
jenkins@2e679588a5be:/ $ cat ~/.ssh/id_rsa.pub
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAQCD7N24M/zHS+Q1c+j4S1hi6cUcVvV1MzerVtSzsn7OpKEDbhjx3CLA6Ji1u/7vdi21xghwsP8pCPXuJj0y1KzlaG1HzABavuJHT0cPnVENOMQBCr47PLo10B6d1XR1tG169mZ6yF1NU42jJn/eLJQMvV6bLGoNlbcS3EvqeoqmgXx3bSbhS
pi080/gs+7SHAyF3Qhcgva0LpSMOIRpnFmHPWzgm+EQq1B1TzRz0AWAHBz6Kmpu6id6JUL/RZm2MjVqNS7aepgV5s868ZSKUJduIzKAzs1Wsgx9ULUk+9qwZBiotWSf8pun4S/G7P+Jf8kD/Vrz0Q9eG1R81 jenkins@2e679588a5be
```

ET on autorise la clé sur le serveur de déploiement.

Dernière chose à faire.

On s'assure que notre registry est autorisé pour communiquer en http puisque le client ne nous a pas fourni de certificat

Voici la syntaxe générale que l'on place dans /etc/docker/daemon.json

```
{  
  "insecure-registries" : ["myregistrydomain.com:5000"]  
}
```

Dans notre cas, le registry est public sur la machine. Donc on pourra y saisir l'hostname complet sans soucis.

Une fois terminé, il suffira de lancer le job.

ET on a un résultat fonctionnel.