RAPPORT SAE 6.01
DEVCLOUD

Gérer le pipeline d’une application orientée cloud

PICHOT Owen
RT31 - Devcloud

1. Introduction

Rappel de l'objectif de la SAE :

En prenant comme base une API développée sous Python via le framework Tornado, déployer celle-ci sur un cluster de conteneurs par le biais d'un orchestrateur comme Kubernetes ou Docker Swarm.

Le Cluster sera hébergé à minima sur 3 VMs et une 4ème VM servant d'instance Gitlab devra également être mise en place et configurée.

Le tout sera contrôlé par le biais de l'hyperviseur ProxMox.

Pour finir, les procédures, la maintenance et les tests nécessaires au déploiement de l'API seront réalisés de manière automatisée par le biais d'un pipeline gitlab en utilisant l'environnement CI/CD que celui-ci propose.

Technologies choisies pour mon projet :

Pour la réalisation de la SAE, j'ai décidé d'utiliser l'orchestrateur Docker Swarm. En effet, celui-ci est plus simple à mettre en place pour des petites architectures et semble plus adapté que Kubernetes étant donné que nous n'hébergeront qu'une API répliquée sur de multiples instances.

Un réseau isolé sera mis en place pour les liens inter-vm au sein de l'hyperviseur ainsi que tous les services nécessaires à la gestion de celui-ci (DHCP, DNS, etc..).

2. Hyperviseur

a. Installation de ProxMox

i. Préparation du support d'installation

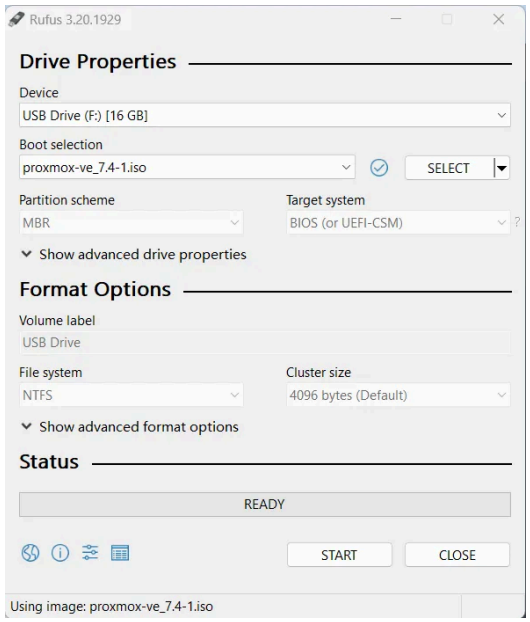


Fig.1 : Utilisation de Rufus pour la création d'une clef USB bootable

Commencer par télécharger la dernière image ISO stable de ProxMox sur leur site officiel.

Préparer ensuite une clé USB bootable via le logiciel RUFUS. Sur rufus, sélectionner l'iso de proxmox que nous avons téléchargé, choisir le mode de partition MBR et lancer la création de la clé bootable.

ii. Démarrage et installation de ProxMox

Une fois la clé prête, il suffit de l'insérer dans la machine sur laquelle on souhaite installer l'hyperviseur, l'allumer et accéder au BIOS/UEFI pour enclencher le boot sur USB.

On sélectionne notre clé USB dans le menu de démarrage.

Il suffit ensuite de lancer l'installation puis attendre la fin du processus.

Fig. 2 : Écran d'installation de l'hyperviseur ProxMox

iii. Configuration & Accès à l'interface WEB

Une fois l'installation terminée, retirer la clé USB et redémarrer la machine.

Un message devrait alors s'afficher sur l'écran présentant l'adresse IP de l'hyperviseur, il est ensuite possible d'accéder à l'interface d'administration depuis un navigateur web en entrant l'adresse IP de la machine suivie du port 8006.

Pour se connecter, il faut utiliser les crédentiels renseignés pendant l'installation de ProxMox.

iv. Etapes post-installation

Pour finir on peut éventuellement mettre à jour le système, ce qui est très recommandé notamment concernant les MàJ de sécurité.

```
pve@proxmox-server:~/$ apt update && apt full-upgrade -y
```

Fig.3 : Mise à jour de l'hyperviseur

Et enfin ajouter le dépôt de non-souscription à la liste des sources afin de supprimer le message d'erreur présent lors de l'accès à l'interface d'administration.

```
pve@proxmox-server:~/$ echo "deb http://download.proxmox.com/debian/pve bookworm pve-no-subscription" > /etc/apt/sources.list.d/pve-no-subscription.list

pve@proxmox-server:~/$ apt update
```

Fig.4 : Ajout du dépôt de non-souscription de proxmox

b. Stockage de l’hyperviseur

Le stockage de l’hyperviseur est divisé en 2 parties distinctes :

- *Local :*
  Stock les images Docker, Images ISO, Templates de VM et templates de conteneurs LXC.
- *Local-LVM :*
  Stock les données de chacune des VMs, c’est sur ce stockage ci que les disques des machines virtuelles sont montés.

c. Réseaux de l’hyperviseur

L'hyperviseur est équipé d'une seconde carte réseau, permettant d'isoler le réseau interne reliant les machines virtuelles (VMs) hébergées de l'infrastructure réseau de l'UHA.

Chaque machine virtuelle utilise une passerelle pointant vers l'hyperviseur, qui assure ensuite la redirection du trafic vers l'extérieur.

Concernant le trafic entrant, seules les connexions à destination de l'hyperviseur sont autorisées. L'accès aux services des VMs se fait via une redirection de ports spécifique vers la machine concernée (cf. annexe X).

Le schéma ci-dessous illustre l'architecture réseau de l'installation :

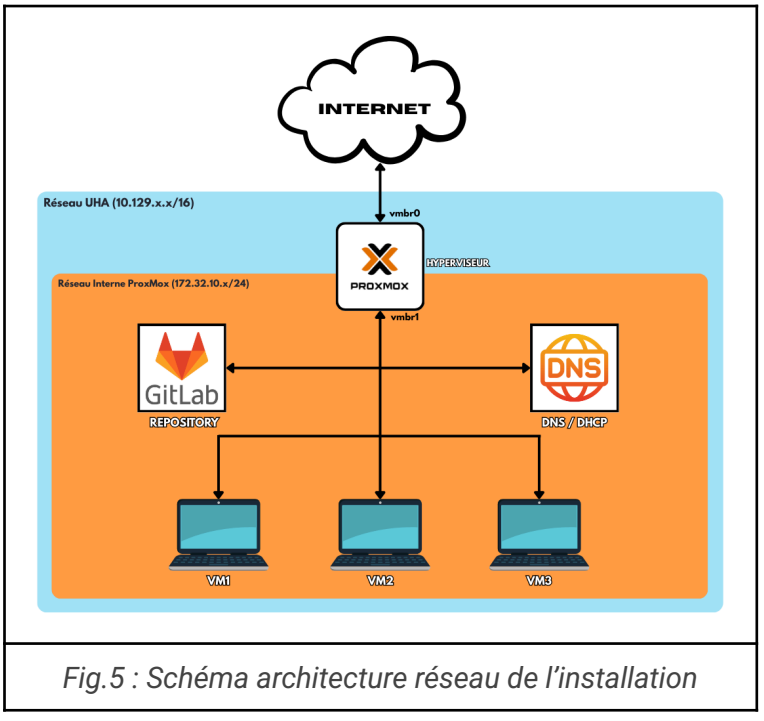


Fig.5 : Schéma architecture réseau de l’installation

d. Accès au VMs depuis l'extérieur

Comme mentionné ci-dessus, des règles NAT permettent l'accès aux services de chacunes des VMs depuis le réseau de l'UHA directement, l'alternative serait sinon de contacter toutes les machines depuis l'hyperviseur ce qui n'est pas pratique ni réaliste dans un environnement de production réel.

Vous pouvez retrouver ci-dessous un exemple de règle NAT appliqué à l'hyperviseur permettant d'accéder à la VM gitlab en SSH via le port 2222 qui sera alors redirigé sur le port 22 de la VM gitlab (172.32.10.50) :

```
pve@proxmox-server:~/$ iptables -t nat -A PREROUTING -p tcp --dport 2222 -j DNAT --to-destination 172.32.10.50:22
pve@proxmox-server:~/$ iptables -t nat -A POSTROUTING -p tcp -d 172.32.10.50 --dport 22 -j MASQUERADE
```

Fig.6 : Exemple d'ajout de règle NAT sur l'hyperviseur

Pour une vue complète des redirections disponibles depuis l'extérieur vers le réseau ProxMox, vous pouvez vous référer au tableau ci-dessous :

Machine	Adresse	Rôle	Port accès externe via IPTABLES	Description redirection
Proxmox	10.129.4.153 [external]	Hyperviseur	22	Accès SSH
	172.32.10.1 [Internal]	Gateway	x	x
Gtilab	172.32.10.50	Hébergement Gitlab & Gitlab-runners	2222	Accès SSH
			5555	Accès web gitlab
			8000	Accès instance de test pré-prod
			2485	Accès web coverage report pré-prod
			2478	Accès web coverage report prod VM1
			2479	Accès web coverage report prod VM2
			2480	Accès web coverage report prod VM3
DNS	172.32.10.254	Hébergement service DNS & service DHCP (interne)	2223	Accès SSH
VM1	172.32.10.20	VM de déploiement applicatif	2224	Accès SSH
			80	Accès web Application Prod
VM2	172.32.10.21	VM de déploiement applicatif	2225	Accès SSH
VM3	172.32.10.22	VM de déploiement applicatif	2226	Accès SSH

Tab.1 : Liste des machines disponibles, leurs adresses et leurs port disponibles.

e. VMs de l’hyperviseur & leurs utilité

Chacune des VMs présente sur l’hyperviseur est présente dans le cadre du bon déroulement du déploiement de l’API.

Ceci inclut une gestion de la base de code, des tests de l’appli, de l'exécution de pipeline, de machines dédiées à l’hébergement web et tous les services annexes permettant le bon fonctionnement d’une infrastructure réseau, notamment ses deux piliers principaux le DNS et le serveur DHCP.

Voyons ci-dessous le rôle en détail de chacune des machines hébergées sur l’hyperviseur (cf. Tab 1 & Fig 5, 7 en support) :

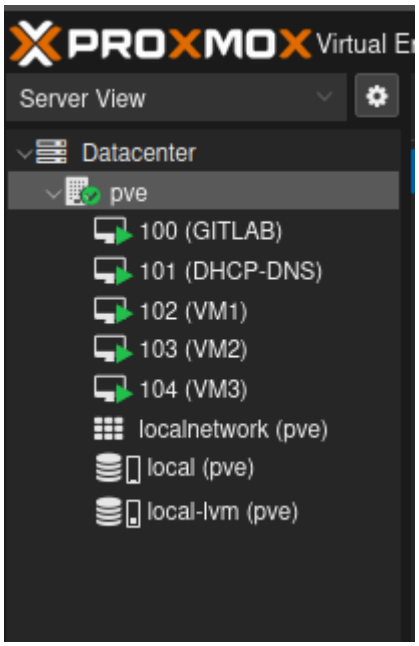


Fig.7 Liste des VMs proxmox

GITLAB : Cette machine héberge l’instance gitlab, ses runners pour l'exécution de pipelines, les instances de test de l’API, le registry d’images Docker et le repository.

DHCP-DNS : Cette machine héberge les services DNS & DHCP du réseau.

VM1, VM2, VM3 : Ces machines hébergent l’orchestrateur Docker Swarm et font office de Nodes pour le déploiement de l’API. Chacune d’entre elles peut héberger de multiples instances de l’API.

3. DNS & DHCP

a. Configuration du DNS

Le DNS est servi par le service BIND9 hébergé sur une VM basée sur un OS Debian 12.

Celui-ci est plutôt simple à mettre en place, il implique premièrement de déclarer une zone (nom de domaine du réseau), ajouter des hôtes au sein de ladite zone (entrées DNS) et enfin d'ajouter éventuellement des forwarders (serveurs DNS supplémentaires qui effectueront les résolutions si l'entrée demandée n'est pas présente sur le service contacté).

Vous pouvez retrouver ci-dessous chacun des fichiers de configurations cités ci-dessus :

```
 GNU nano 7.2        named.conf.options *
options {
        directory "/var/cache/bind";

        allow-query { any; };

        forwarders {
                8.8.8.8;
        };

        dnssec-validation auto;

        listen-on-v6 { any; };
};
```

Fig. 8 : Config DNS des forwarders

```
 GNU nano 7.2  named.conf.local *

zone "owenlab.com" {
        type master;
        file "/etc/bind/db.owenlab.com";
};
```

Fig. 9 : Config de la zone DNS

```
 GNU nano 7.2            db.owenlab.com
;
; BIND data file for local loopback interface
;
$TTL    604800
@       IN      SOA     ns.owenlab.com. root.owenlab.com. (
                              2         ; Serial
                         604800         ; Refresh
                          86400         ; Retry
                        2419200         ; Expire
                         604800 )       ; Negative Cache TTL
;
@       IN      NS      ns.owenlab.com.
ns      IN      A       172.32.10.254
proxmox IN      A       172.32.10.1
gitlab  IN      A       172.32.10.50
vm1     IN      A       172.32.10.20
vm2     IN      A       172.32.10.21
vm3     IN      A       172.32.10.22
```

Fig. 10 : Config des entrées de la zone DNS

b. Configuration du DHCP

Le DHCP est servi par le service isc-dhcp-server, celui-ci est hébergé sur la même machine qui héberge le DNS. Sa configuration est plutôt simple, outre le fait d'assigner une adresse IP statique à la machine hôte, il suffit ensuite de lui indiquer le réseau sur lequel elle doit se charger de distribuer des adresses.

Il est également possible de faire en sorte que certaines machines prennent toujours la même adresse en liant une certaine adresse à l'adresse MAC de la machine cible.

Vous pouvez retrouver ci-dessous la configuration DHCP de la machine :

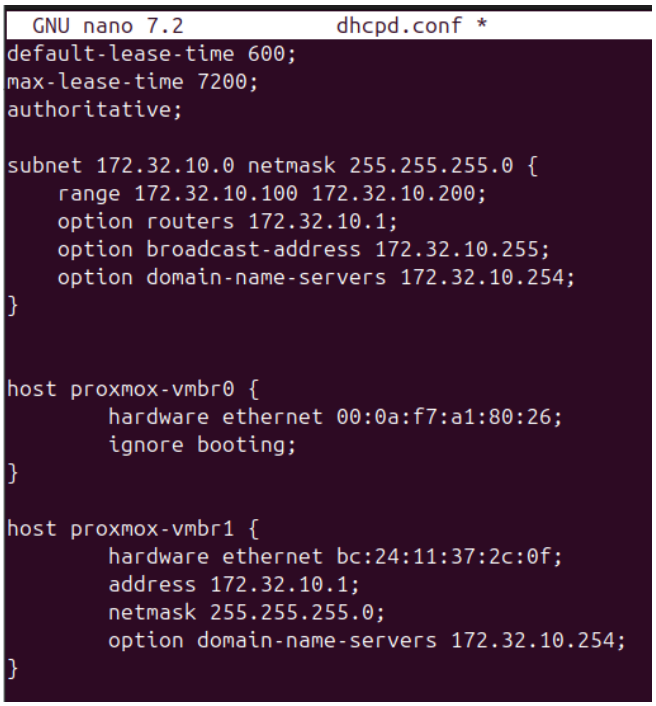


Fig. 11 : Fichier de configuration isc-dhcp-server (dhcpd.conf)

Il est important de noter dans la configuration les points suivants;

- Le sous-réseau sur lequel sont desservies les adresses est le sous réseau de l'interface vmbr1.

- La carte vmbr0 est exclue du DHCP car celle-ci doit recevoir une adresse de l'UHA.

- La carte vmbr1 obtient une adresse statique afin de ne jamais changer (il s'agit de la gateway du réseau).

Cette configuration simple mais robuste permet d'ajouter à la volée de nouvelles machines (par exemple une machine VMX pour en faire un nouveau node) sans avoir à modifier leur fichier /etc/network/interfaces.

Elles recevront toutes une adresse entre 172.32.10.100 & 172.32.10.101.

4. Instance Gitlab

a. Création de la VM

La première étape a été de créer une nouvelle VM sur l'hyperviseur sur laquelle sera hébergée notre instance Gitlab. Celle-ci requiert un minimum de puissance de calcul car Gitlab, Gitlab-Runner et les processus exécutés par l'application peuvent rapidement être lourds.

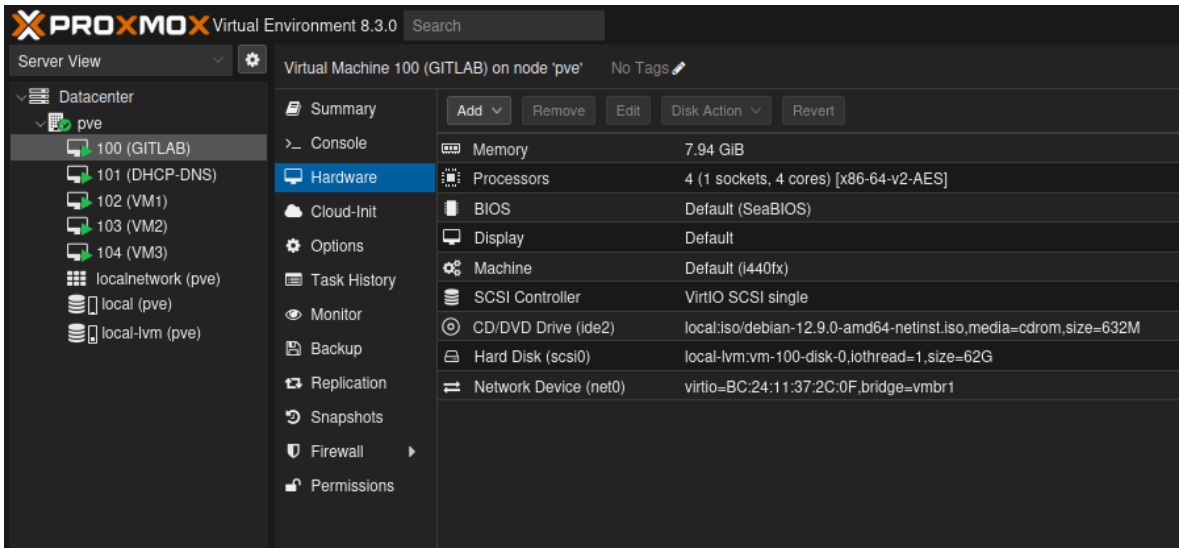


Fig. 12 Hardware de la VM Gitlab sur l'hyperviseur ProxMox

Pour avoir une puissance suffisante, la machine possède donc le matériel suivant :

- 8 Gb de RAM
- 4 Coeurs de processeur 64-bit
- 62 Go de stockage
- Une carte réseau bridgé sur l'interface vmbr1 de l'hyperviseur afin d'être inclue dans le réseau global de l'infrastructure.

Note : Le système d'exploitation de la VM choisi est Debian 12 étant donné qu'il s'agit d'une version stable, plutôt légère et que sans Gnome (environnement de bureau) le système d'exploitation ne dépasse pas les 1.5 Go d'espace. De plus, toutes les interfaces Gitlab sont accessibles via un navigateur sur n'importe quelle autre machine.

b. Installation de Gitlab

L'installation de Gitlab est très simple, il suffit d'ajouter le dépôt officiel Gitlab puis d'installer le paquet de la version communauté comme n'importe quel autre paquet apt.

```
toto@gitlab:~/$ curl https://packages.gitlab.com/install/repositories/gitlab/gitlab-ce/script.deb.sh | sudo bash
toto@gitlab:~/$ sudo EXTERNAL_URL="https://gitlab.owenlab.com" apt install gitlab-ce -y
```

Fig.13 : Commandes d'installation de Gitlab

Note : Le paramètre "EXTERNAL_URL" spécifié ici permet de déterminer sur quel adresse le service gitlab sera hébergé, grâce à la mise en place du DNS sur notre réseau, il n'est pas nécessaire de spécifier une adresse en dur mais son FQDN à la place, ceci évite de modifier la configuration du service si par la suite l'adresse venait à changer.

Une fois l'installation terminée, notre instance gitlab et son interface d'administration devrait d'ores et déjà être disponible sur un navigateur à l'adresse "https://gitlab.owenlab.com". Un message de sécurité sera cependant affiché tant que les clefs pour le https n'ont pas été créées (cf. étape suivante).

c. Configuration HTTPS

Pour sécuriser l'accès en HTTPS, nous utilisons un certificat auto-signé.

Ceux-ci sont générés puis déplacés dans un dossier spécifique aux certificats ssl de gitlab via les commandes suivantes :

```
toto@gitlab:~/$ openssl genpkey -out gitlab.owenlab.com.key -algorithm RSA -pkeyopt rsa_keygen_bits:2048
toto@gitlab:~/$ openssl req -new -key gitlab.owenlab.com.key -out gitlab.owenlab.com.csr -subj "/CN=gitlab.owenlab.com"
toto@gitlab:~/$ openssl x509 -signkey gitlab.owenlab.com.key -in gitlab.owenlab.com.csr -req -days 365 -out
gitlab.owenlab.com.crt
toto@gitlab:~/$ sudo mkdir -p /etc/gitlab/ssl
toto@gitlab:~/$ sudo mv gitlab.owenlab.com.key gitlab.owenlab.com.crt /etc/gitlab/ssl/
toto@gitlab:~/$ sudo chmod 600 /etc/gitlab/ssl/*
toto@gitlab:~/$ sudo gitlab-ctl reconfigure
```

Fig.14 : Commandes de création de certificat auto-signé pour Gitlab

Note: Dans les commandes ci-dessus, les trois premières lignes se chargent de générer les clés privées/publiques (.crt & .key), les lignes suivantes se

chargent de déplacer celles-ci dans le dossier ssl de gitlab et on finit par redémarrer le service gitlab via la commande Gitlab-ctl reconfigure.

d. Configuration de la registry Docker

La registry docker est ce qui nous permet de stocker nos images docker sur l'instance gitlab directement et d'y avoir accès par la suite dans notre CI/CD et sur nos pipelines.

La première étape consiste à ajouter dans notre fichier de configuration gitlab (/etc/gitlab/gitlab.rb) la ligne suivante pour lui indiquer à quelle adresse servir la registry :

> registry_external_url "https://gitlab.owenlab.com:5050"

Il faut ensuite configurer docker présent sur la machine Gitlab pour que celui-ci utilise le même certificat que Gitlab :

```
toto@gitlab:~/$ sudo mkdir -p /etc/docker/certs.d/gitlab.owenlab.com:5050
toto@gitlab:~/$ sudo cp /etc/gitlab/ssl/gitlab.owenlab.com.crt /etc/docker/certs.d/gitlab.owenlab.com:5050/ca.crt
toto@gitlab:~/$ sudo systemctl restart docker
toto@gitlab:~/$ sudo gitlab-ctl reconfigure
```

Fig.15 : Commandes de création de certificat auto-signé pour Gitlab

La procédure est assez simple, on peut voir qu'il suffit de copier notre certificat Gitlab déjà présent dans le dossier que Docker consulte pour trouver un certificat spécifique (/etc/docker/certs.d).

On redémarre ensuite respectivement Gitlab puis Docker pour que les deux services prennent en compte les changements.

e. Installation & Configuration de Gitlab-Runner

Le Runner Gitlab est impératif pour l'exécution de nos pipelines CI/CD.
En effet un Runner est simplement un programme qui se chargera de générer un ou plusieurs conteneur docker dans lesquels seront exécutés chacune des étapes d'un pipeline.

L'installation du Runner se fait en 2 temps, la déclaration d'un runner sur gitlab, puis l'enregistrement de celui-ci sur la VM en CLI.

Commencer par se rendre sur l'interface gitlab dans la section "<Projet>/ CI/CD Settings / New Runner", entrer les tags "shared, docker" afin de permettre au runner de lancer les pipelines d'autres projets et d'utiliser docker comme moteur.

Ceci nous donne un Token qu'il faudra entrer dans l'étape finale ci-dessous.

Installation de Gitlab-runner (Similaire à l'installation Gitlab) :

```
toto@gitlab:~/$ curl -L https://packages.gitlab.com/install/repositories/runner/gitlab-runner/script.deb.sh | sudo bash
toto@gitlab:~/$ sudo apt install gitlab-runner -y
```

Fig.16 : Commandes d'installation de Gitlab-Runner

Configuration & enregistrement du runner :

```
toto@gitlab:~/$ sudo mkdir -p /etc/gitlab-runner/certs
toto@gitlab:~/$ sudo cp /etc/gitlab/ssl/gitlab.owenlab.com.crt /etc/gitlab-runner/certs/
toto@gitlab:~/$ gitlab-runner register \
  --url "https://gitlab.owenlab.com" \
  --registration-token "GLRT-oicneincosi" \
  --executor "docker" \
  --docker-image "alpine:latest" \
  --docker-network-mode "host" \
  --docker-extra-hosts ["gitlab.owenlab.com:172.32.10.50"] \
  --tag-list "shared,docker" \
  --name "runner"
```

Fig.17 : Commandes d'enregistrement d'un runner

5. VMs hôte du Docker Swarm

a. Configuration

La configuration des VMs nécessaires à la mise en place du swarm est plutôt basique, en effet il suffit de configurer une seule des trois machines avec les paramètres matériels suivants sur notre hyperviseur :

- 2Gb de RAM
- 1 Processeur 2 Coeurs 64-bits
- 32Gb de Stockage
- Debian 12 Stable en tant que système d'exploitation

Étant donné que ces machines ne servent que de serveurs web et de nodes au sein du cluster docker, ils ne nécessitent pas énormément de puissance de calcul. De plus, il n'est pas nécessaire de créer à la main les trois, le plus simple est de créer la première vm (Master), puis d'en générer X copies via ProxMox afin de gagner en temps. J'en ai profité également pour en créer un template général réutilisable par la suite pour recréer à la volée par code des VMs supplémentaires.

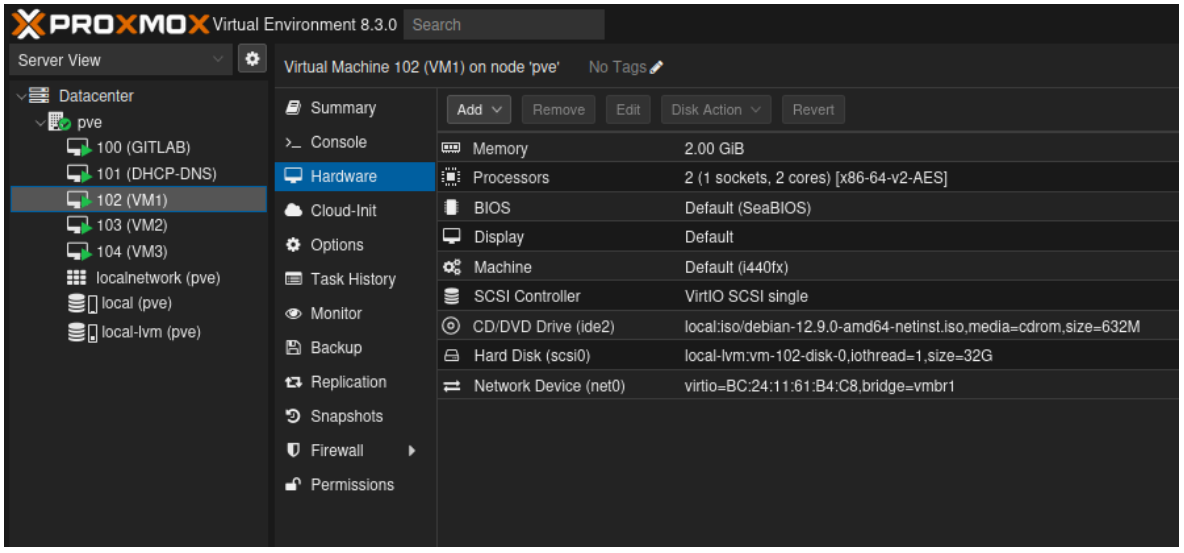


Fig. 18 : Hardware de la VM Master sur l'hyperviseur

Note: Avant la création du template basé sur VM1, Docker à été installé sur la machine pour ne pas reproduire le processus d'installation sur les deux autres machines.

b. Docker Swarm

La procédure pour initialiser Docker Swarm sur 2 ou plusieurs VMs est très simple, il suffit de réaliser les étapes suivantes :

- Initier le Swarm sur la VM Master choisie
- Noter le token généré par la machine
- Ajouter X workers au Swarm en contactant la machine master

Voici ces étapes présentées ci-dessous (le token généré est factice, il s'agit d'ici d'une démonstration).

```
toto@VM1:~/$ docker swarm init --advertise-addr 172.32.10.100

Swarm initialized: current node (abcdefgh1234) is now a manager.

To add a worker to this swarm, run the following command:

    docker swarm join --token SWMTKN-1-23fake456token789abc123 172.32.10.100:2377
```

Fig.19 : Commande d'initialisation du Swarm sur VM Master

```
toto@VM2:~/$ docker swarm join --token SWMTKN-1-fake1234567890abcdef-0987654321abcdef
172.32.10.100:2377
```

Fig.20 : Commande d'initialisation du/des Workers.

Une fois ceci fait il est possible de vérifier le statut du swarm en utilisant la commande suivante :

> docker node ls

Ceci retourne la liste des nodes disponibles au sein du swarm :

```
toto@VM1:~$ docker node ls
ID                            HOSTNAME   STATUS    AVAILABILITY   MANAGER STATUS   ENGINE VERSION
c2rkq1p64m61nyuy4os173seo *   VM1        Ready     Active         Leader           28.0.1
jy2mmmcyufn7jnwxxqlzov3i6     VM2        Ready     Active                          28.0.1
pq21r239gddv96azzeg7sm3ir     VM3        Ready     Active                          28.0.1
toto@VM1:~$
```

Fig. 21 : Résultat de la commande docker node ls sur la VM Master

N.B : Il n'est pas nécessaire d'exécuter la commande depuis le node Master, celle-ci peut être exécutée sur n'importe quel node au sein du swarm.

6. Pipeline & CI/CD Gitlab

a. Déroulement général & fonctionnalitées

Le Pipeline à été créé de manière à avoir deux environnements dédiés :

Un environnement simple de déploiement de l'application sans réplique et sans swarm, celui-ci permet de tester lors de chaque push sur une branche autre que la branche 'main' du repository que tout fonctionne, l'application se lance en local à même la machine Gitlab et des tests sont réalisés (détails ci-dessous).

Un second environnement, celui de production sera créé lors du push du code sur la branche main, l'utilisateur peut alors soit décider de d'abord lancer l'application sur un conteneur simple et effectuer des tests automatisés dessus avant de déployer le swarm, soit de déployer directement le swarm. Il est important de noter qu'une fois l'application déployée, des tests seront exécutés sur chacune des instances pour s'assurer de la bonne exécution de l'API.

Vous pouvez retrouver ci-dessous une vue globale du Pipeline lorsque celui-ci déploie une instance de test de l'API sur un conteneur simple et réalise les tests sur celui-ci.

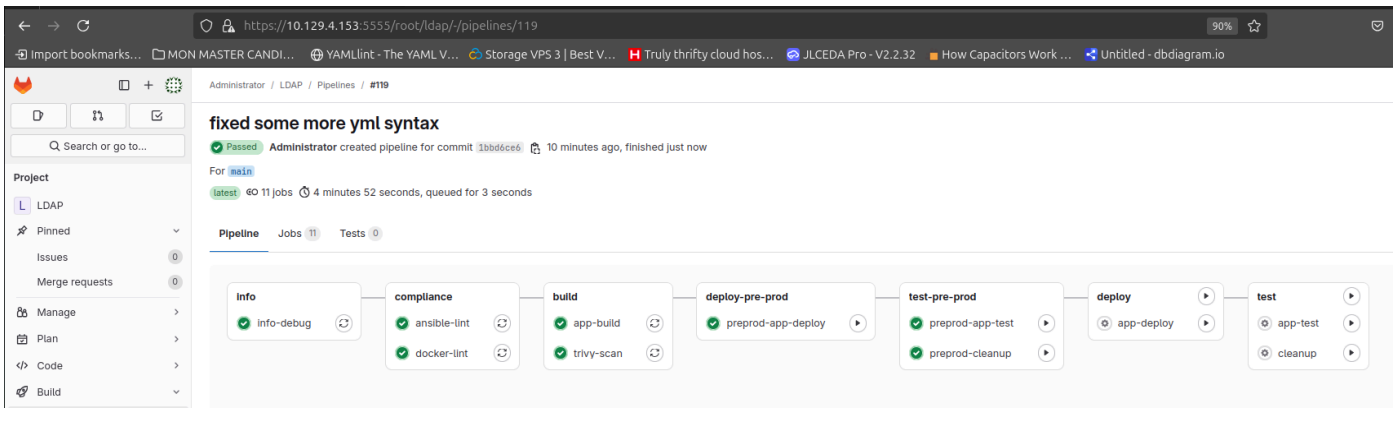


Fig. 22 : Pipeline Gitlab exécuté sur une branche 'feat', le déploiement swarm est bloqué.

⚠️ L'entièreté des playbooks, fichiers Gitlab CI et résultat d'exécution des tasks du pipeline sont trouvable en annexes à la fin de ce document ⚠️

b. Info / Débug

i. Rôle

Cette étape du Pipeline est présente à des fins d'analyse lors de son exécution.

Elle se charge d'imprimer toutes les variables CI/CD qui seront utilisées durant le pipeline pour vérifier si quelque chose cloche, par exemple une variable écrasée ou manquante.

Cette étape est non-bloquante, si elle échoue le pipeline continue son exécution. Il y aura cependant une notification d'erreur afin de laisser savoir au développeur qu'il est nécessaire d'analyser les résultats du Job avant de continuer le déploiement sans quoi l'application déployée pourrait échouer continuellement.

ii. Exécution

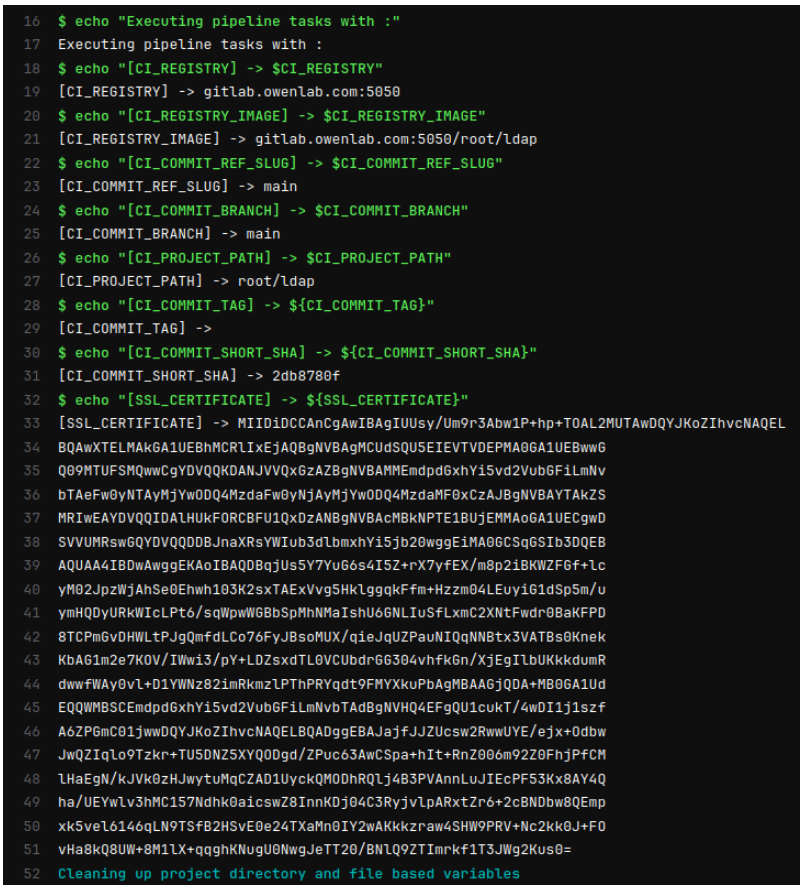


Fig.23 : Exécution de la tâche info/debug du pipeline

c. Conformité

i. Rôle

Docker Lint :

Cette étape du pipeline vérifie la conformité du fichier Dockerfile utilisé pour construire l'image de l'application. Elle utilise l'outil Hadolint, qui analyse le fichier `APP/Dockerfile` et signale les éventuelles erreurs ou mauvaises pratiques. Cette étape est non bloquante : en cas d'échec, une notification est envoyée, mais le pipeline continue son exécution.

Ansible Lint :

Cette étape garantit la qualité et la conformité des scripts Ansible en analysant les fichiers de l'infrastructure situés dans le dossier `ANSIBLE/*`. Elle installe les dépendances nécessaires, notamment Ansible Lint et la collection community.docker, avant d'exécuter les vérifications. Comme pour le Docker Lint, une erreur à cette étape n'empêche pas la poursuite du pipeline, mais une notification est générée pour signaler les problèmes.

ii. Exécution

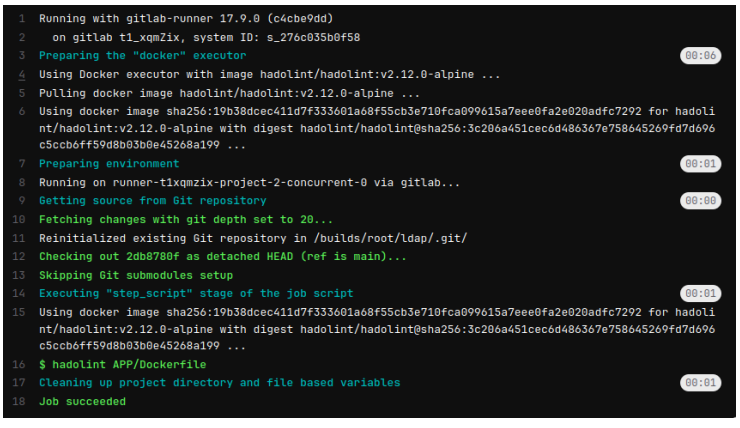


Fig.24 : Exécution de la tâche Docker Lint

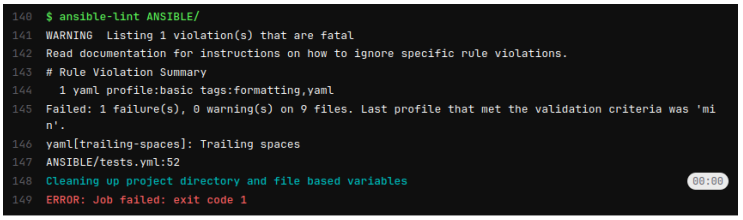


Fig.25 : Exécution de la tâche Ansible Lint

On peut voir ci-dessus fig.25 que la tâche Ansible Lint renvoie une erreur car un espace superflu est présent dans le playbook ansible 'tests.yml' à la ligne 52. Ceci ne prévient pas l'exécution du reste du pipeline cependant car il ne s'agit pas d'une erreur de syntaxe mais une erreur de bonne pratique.

d. Construction

i. Rôle

App Build :

Cette étape est responsable de la construction de l'image Docker de l'application. Elle utilise Kaniko, un outil permettant de construire des images sans nécessiter un démon Docker. L'image résultante est ensuite poussée vers le GitLab Container Registry avec un tag correspondant au hash court du commit (`CI_COMMIT_SHORT_SHA`). Cette étape est **essentielle** : un échec entraînerait l'arrêt du pipeline.

Trivy Scan :

Cette étape effectue une analyse de sécurité sur l'image Docker générée afin de détecter d'éventuelles vulnérabilités. L'outil Trivy scanne l'image et signale toute faille de sécurité identifiée. Une connexion au GitLab Container Registry est nécessaire pour récupérer l'image. Cette étape est **bloquante** : toute vulnérabilité critique empêchera la poursuite du pipeline.

ii. Exécution

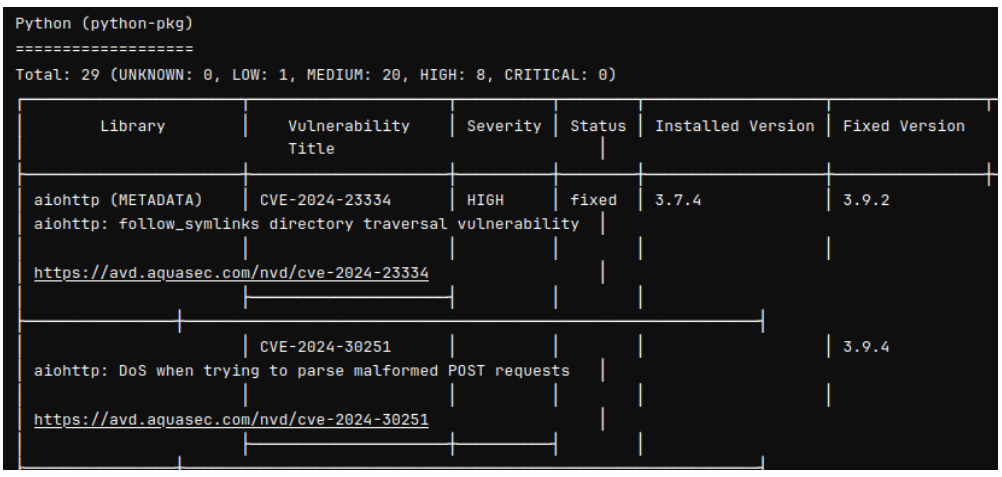


Fig.26 : Exécution de la tâche Trivy Scan

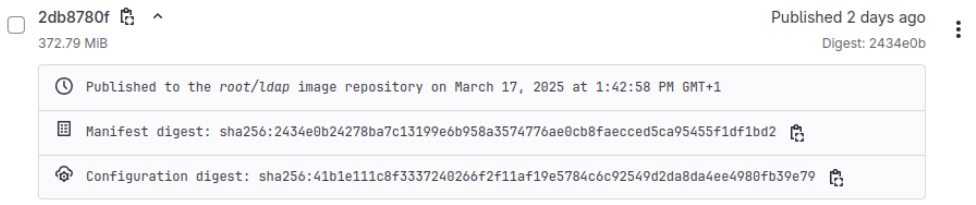


Fig.27 : Image Docker générée dans la registry Gitlab suite au Build.

e. Déploiement pré-production

i. Rôle

Cette étape déploie l'application sur l'environnement de préproduction en utilisant un playbook Ansible. Elle s'assure que le déploiement se fait sur l'infrastructure définie dans inventory-feat.ini et exécute deploy.yml. Ce déploiement se fait manuellement, nécessitant une validation explicite avant d'être exécuté.

Le serveur hébergeant l'API sera alors contactable via le port 8000 sur l'adresse de la machine ProxMox (cf. tab.1).

Il est important de noter que inventory-feat.ini ne contient qu'une seule machine (voir Annexes), la machine Gitlab en l'occurrence car le déploiement de pré-production n'est réalisé que sur la machine en local avant un déploiement global en swarm.

ii. Exécution

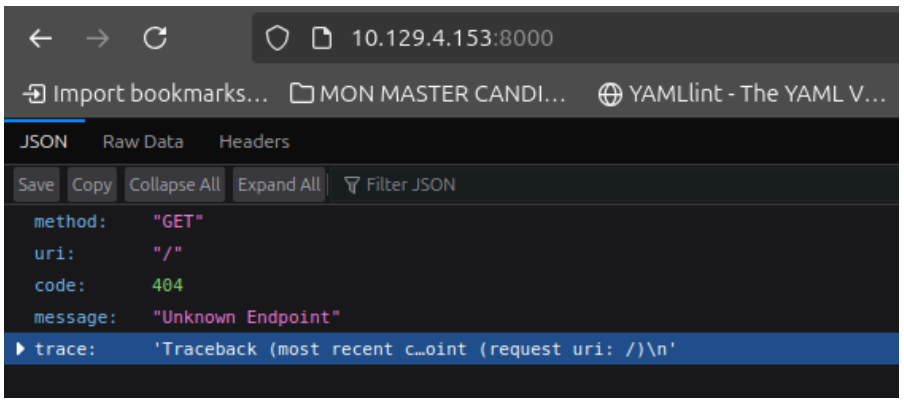


Fig.28 : Accès à l'application pré-prod sur le port 8000

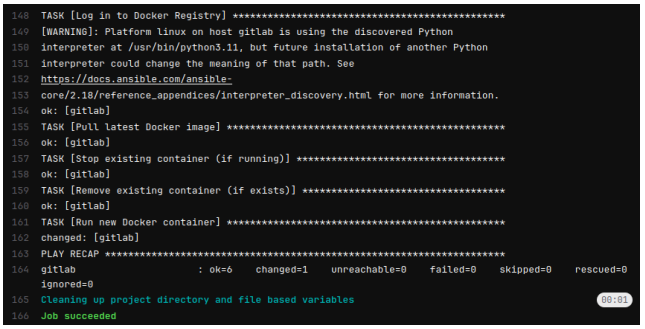


Fig.29 : Exécution de la task déploiement préprod au sein du pipeline

f. Tests en pré-production

i. Rôle

Cette étape du pipeline à pour but de réaliser une série de tests sur l'API afin de déterminer son bon fonctionnement avant de déployer celle-ci en swarm.

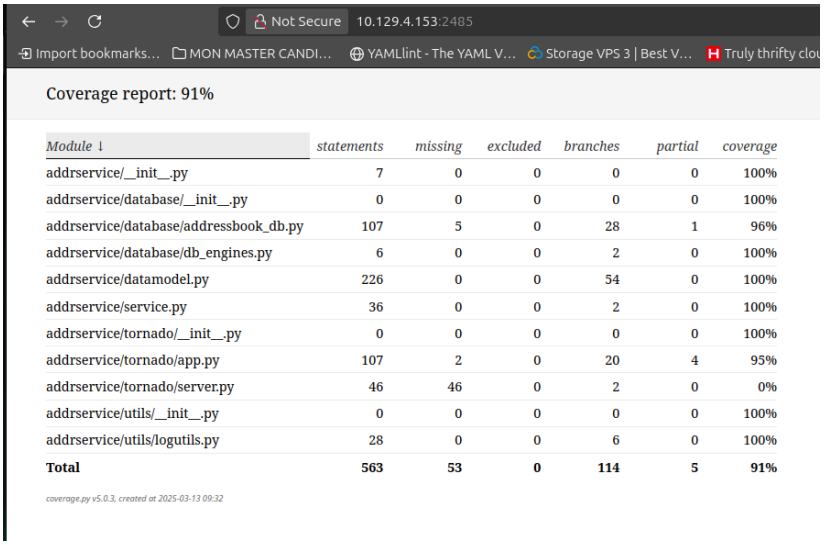
Les tests sont appelés grâce à un playbook 'tests.yml', celui-ci réalise alors tout une batterie de tests, notamment :

- Type check (Fourni par le framework Tornado)
- Lint Tests
- Unit tests
- Coverage tests
- Curl tests du CRUD (post, get, delete, update) via un script bash.

Lors de la réalisation du coverage test qui vise à analyser l'entièreté du code de l'application pour d'éventuelles erreurs de logique ou de syntax, un rapport html est alors généré, ce rapport est ensuite récupéré depuis le conteneur jusqu'à la machine gitlab ou un conteneur nginx web est lancé pour présenter le rapport du test sur le port 2485. Ce rapport est intéractif et permet de voir le statut de chaque fichier de l'application ligne par ligne.

ii. Exécution

Fig.30 : Coverage report d'un fichier



Module ↓	statements	missing	excluded	branches	partial	coverage
addrservice/__init__.py	7	0	0	0	0	100%
addrservice/database/__init__.py	0	0	0	0	0	100%
addrservice/database/addressbook_db.py	107	5	0	28	1	96%
addrservice/database/db_engines.py	6	0	0	2	0	100%
addrservice/datamodel.py	226	0	0	54	0	100%
addrservice/service.py	36	0	0	2	0	100%
addrservice/tornado/__init__.py	0	0	0	0	0	100%
addrservice/tornado/app.py	107	2	0	20	4	95%
addrservice/tornado/server.py	46	46	0	2	0	0%
addrservice/utils/__init__.py	0	0	0	0	0	100%
addrservice/utils/logutils.py	28	0	0	6	0	100%
Total	**563**	**53**	**0**	**114**	**5**	**91%**

coverage.py v5.0.3, created at 2025-03-13 09:32

Fig.31 : Page d'accueil coverage report de l'API

g. Déploiement Production

i. Rôle

Comme son nom l'indique, cette étape est la plus importante du pipeline, elle gère le déploiement final de l'API sur le cluster Docker Swarm ce qui rend alors l'API disponible au public via le port 80 sur l'adresse de notre hyperviseur.

Un load balancer est inclus de base via docker swarm sur la VM Master ce qui fait que tout le trafic peut être redirigé sur celle-ci.

Pour déployer l'API en production, le gitlab CI fait appel au playbook 'deploy-swarm.yml'.

Ce playbook est exécuté sur l'inventaire 'inventory.ini' qui regroupe les trois machines virtuelles d'hébergement de l'API.

ii. Exécution

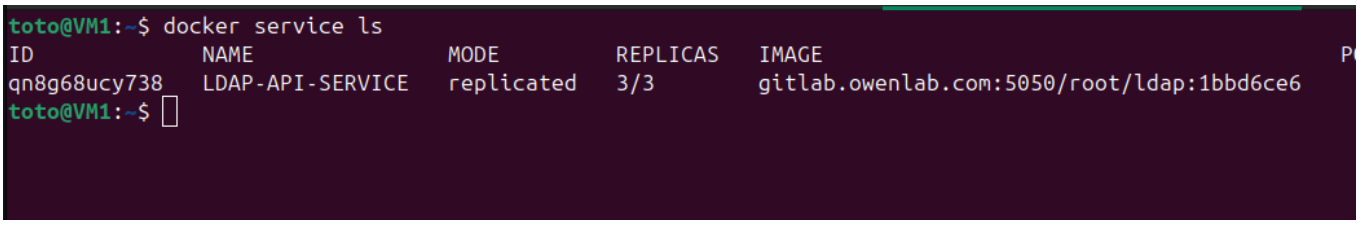


Fig.32 : Résultat de la commande docker service ls pour le statut de déploiement de l'API

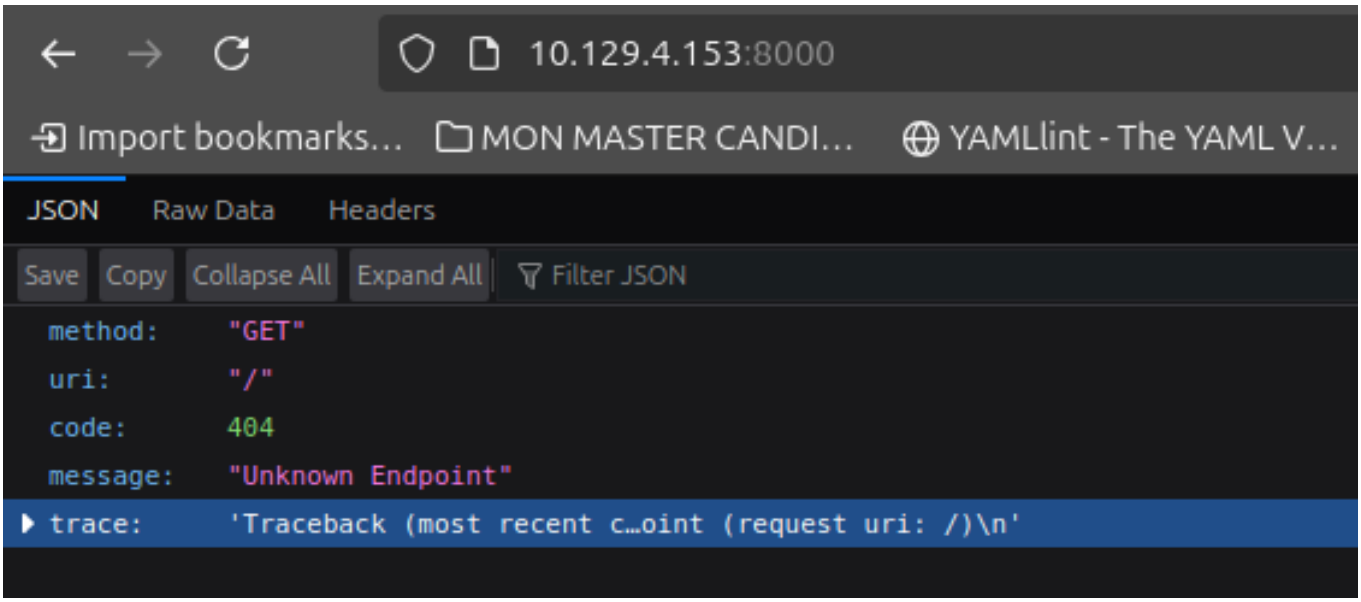


Fig.33 : API ouverte dans le navigateur après déploiement en production

h. Tests en production

i. Rôle

De manière générale, les tests effectués sur les instances servant l'API sont les mêmes que ceux réalisés en pré-production sur une seule instance, la différence étant que ceux-ci seront réalisés sur chacune des instances lancées par Docker Swarm.

Un coverage report sera également déployé pour chacune des instances. Même si ces coverages report devrait techniquement être identiques les uns avec les autres, ceci nous permet de monitorer le cas potentiel ou l'une des instances d'une version précédente n'aurait pas été correctement arrêtée ce qui donnerait alors un coverage report différent des autres.

Le meilleur moyen de remédier à cela si ça venait à arriver serait de réaliser un cleanup (voir section suivante) puis de redéployer de nouveau l'API en production.

Les coverages report de chaque instance sont disponibles au port 2478 et s'incrémente de 1 pour chaque instance, ex: si VM1 à son coverage report disponible au port 2478, alors VM4 aura son rapport disponible sur le port 2481.

ii. Exécution

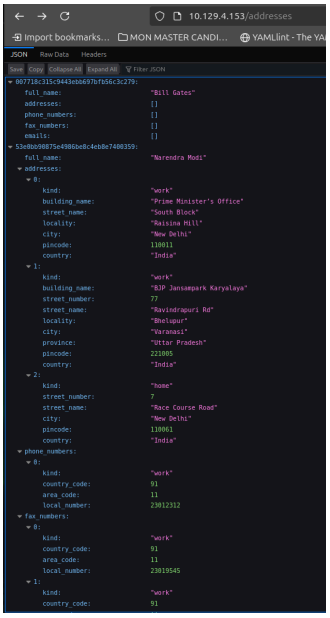


Fig. 34 : API en production

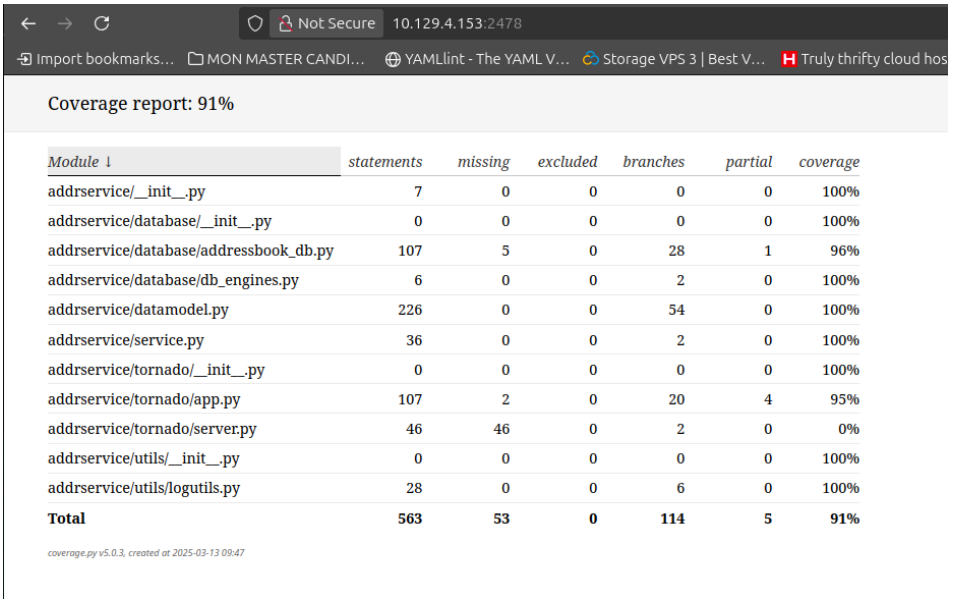


Fig.35 : Coverage Report de VM1 sur le port 2478

iii. Ressenti personnel & Conclusion

Ce projet a été une expérience particulièrement enrichissante, tant sur le plan technique que méthodologique. Travailler sur le déploiement d'une application orientée cloud avec Docker Swarm et GitLab CI/CD m'a permis de consolider mes compétences en infrastructure réseau, virtualisation et automatisation des déploiements. La mise en place du pipeline a nécessité une réflexion approfondie sur l'organisation des tâches, la gestion des environnements de préproduction et production, ainsi que l'optimisation des tests pour garantir la fiabilité de l'application.

La SAE en soi n'aura pas été compliquée mais c'est une bonne chose étant donné que nous arrivons désormais à la fin du BUT, si certaines des tâches m'avait parue comme étant insurmontables cela m'aurait fait me questionner quand à ma place dans une spécialisation DevOps.

Cependant, cela ne veut pas dire que ce projet n'aura pas eu ces défis, notamment la gestion des permissions Ansible, l'intégration des tests automatisés et la mise en réseau des différentes VMs sous ProxMox, ont été autant d'occasions d'approfondir ma compréhension des bonnes pratiques DevOps. La mise en place d'un cluster Docker Swarm m'a également permis de mieux appréhender le fonctionnement des orchestrateurs et l'équilibrage de charge entre plusieurs instances d'un service.

En conclusion, cette SAE m'a apporté une vision concrète des enjeux du déploiement cloud et de l'automatisation des workflows de développement. L'expérience acquise sera précieuse pour de futurs projets nécessitant un déploiement fiable, scalable et maintenable. Avec du recul, certaines améliorations pourraient être apportées, comme l'optimisation des tests ou l'ajout de monitoring avancé sur les instances déployées. Cependant, le pipeline mis en place permet déjà une gestion efficace du cycle de vie de l'application, offrant ainsi un socle solide pour des évolutions futures.

Merci pour votre attention.

7. Annexes

- Code source de la SAE: , inclu :
    - Code source de l'API
    - Dockerfile utilisé pour build l'image de déploiement
    - Fichier Gitlab CI pour le pipeline
    - Fichiers de test y compris tests customisés
    - Fichiers d'inventaire Ansible
    - Playbooks Ansibles

> *https://gitlab.wewenito.ddns.net:8085/owen/ldap-sae-6.01/*