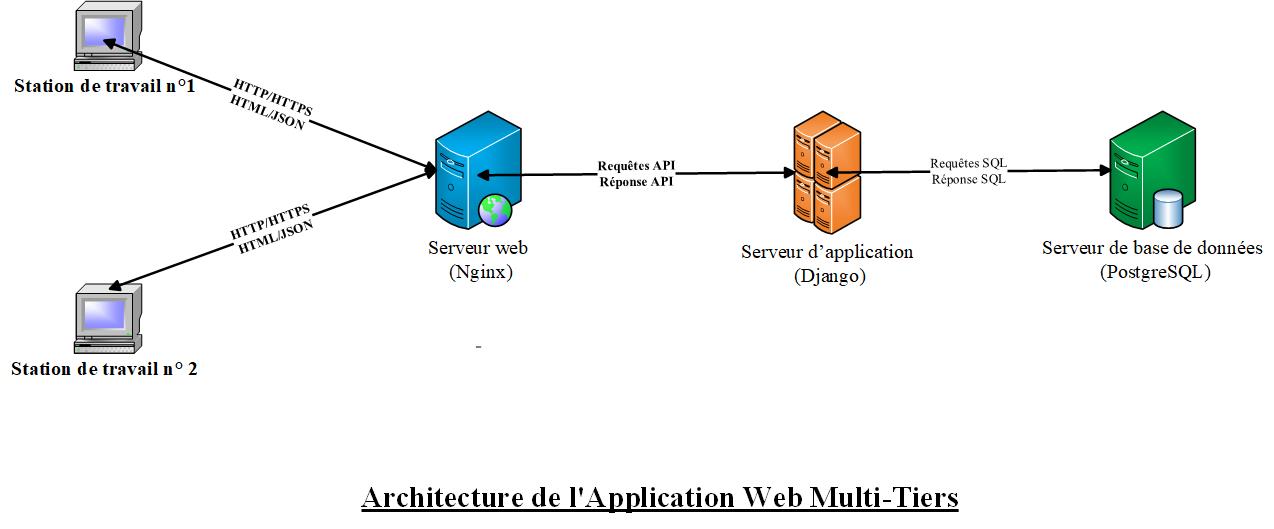
**Phase 1 : Préparation et Planification**

1. **Formation des groupes et distribution des rôles**

Pour la bonne marche de nos travaux de recherche, nous sommes constitués de 5 membres à savoir :

|  |  |  |
| --- | --- | --- |
| Nom & prénom (s) | Filière | Rôle spécifique |
| TRAORE Bafima Teli Martin Daouda | MISIE |  |
| KONATE Salim Adam | MISIE |  |
| KONATE Karim | MISI |  |
| BAZIE Alex Damien | MISI |  |
| NIKIEMA Guy Arthur | MISIE |  |

1. **Conception de l'architecture**



Nous allons faire une description de notre architecture :

* Description du Flux de Requêtes : les clients envoient des requêtes HTTP/HTTPS au serveur web Django, qui les traite et les transmet au serveur d'application pour le traitement des données.
* Interaction Serveur d'Application - Base de Données : le serveur d'application Django communique avec la base de données MySQL/PostgreSQL via des requêtes SQL pour récupérer, mettre à jour ou supprimer des données selon les demandes des utilisateurs.
* Retour des Résultats aux Clients : une fois que le serveur d'application Django a traité les données, il renvoie les réponses sous forme de pages HTML ou de données JSON aux clients via le serveur web, assurant ainsi une expérience utilisateur fluide.
* Sécurité et Gestion des Sessions : le serveur web Django gère la sécurité et les sessions des utilisateurs, assurant l'authentification et l'autorisation appropriées avant de permettre l'accès aux données sensibles stockées dans la base de données.
* Redondance et Scalabilité : pour garantir la redondance et la scalabilité, le déploiement utilise des configurations de serveur web et d'application Django réparties sur plusieurs instances, connectées à une base de données centralisée pour assurer une performance optimale et une disponibilité continue.

**3. Choix des technologies et outils**

L'architecture multi-tiers de l'application web présentée dans le diagramme utilise trois principaux composants : un serveur web Nginx, un serveur d'application Django, et une base de données PostgreSQL. Cette séparation des préoccupations permet une gestion et une maintenance plus faciles de chaque composant. Le serveur web Nginx gère les requêtes HTTP/HTTPS des clients, sert les contenus statiques et agit comme un reverse proxy pour le serveur d'application. Le serveur d'application Django contient la logique métier de l'application, traite les requêtes entrantes, exécute la logique de l'application et interagit avec la base de données. Le serveur de base de données PostgreSQL stocke les données de manière persistante et fournit des capacités de gestion de base de données relationnelle. Cette séparation permet une meilleure évolutivité et flexibilité, car chaque composant peut être mis à jour ou remplacé indépendamment des autres.

En utilisant Nginx comme serveur web et reverse proxy, l'application bénéficie de meilleures performances et de scalabilité. Nginx est réputé pour sa capacité à gérer un grand nombre de connexions simultanées avec une utilisation minimale des ressources. Il peut également équilibrer la charge entre plusieurs instances de serveur d'application Django, permettant ainsi de gérer une augmentation du nombre de requêtes sans compromettre les performances.

La séparation des différentes couches de l'application améliore également la sécurité globale. Le serveur web Nginx peut être configuré pour gérer les certificats SSL/TLS, offrant ainsi des communications sécurisées entre les clients et le serveur. Django inclut des fonctionnalités de sécurité intégrées, telles que la protection contre les attaques CSRF (Cross-Site Request Forgery) et XSS (Cross-Site Scripting). Les accès à la base de données PostgreSQL peuvent être restreints au serveur d'application uniquement, réduisant ainsi la surface d'attaque.

Enfin, en utilisant PostgreSQL comme base de données, l'application profite des fonctionnalités avancées de gestion des transactions et de la concurrence offertes par PostgreSQL. Cela assure l'intégrité des données même en cas d'accès simultané. Cette architecture multi-tiers utilisant Nginx, Django et PostgreSQL offre ainsi de nombreux avantages en termes de séparation des préoccupations, performances, scalabilité, sécurité et gestion des données. Elle permet de créer une application robuste, évolutive et maintenable, capable de répondre aux besoins des utilisateurs tout en offrant une base solide pour le développement futur.

* **Choisissez les outils** de provisionnement (par exemple, Ansible) et de gestion des versions (Git).

**Phase 2 : Configuration des machines virtuels**

1. **Création des vagrantfiles**

Pour notre projet, nous avons créé un dossier sur le bureau nommé « PROJET\_VAGRANT » où nous allons initialiser tous les éléments nécessaires pour assurer le bon fonctionnement de notre système. Mais avant, nous allons faire des installations par étape :

Etape 1 : installation des plugins nécessaire pour notre projet

Au cours de nos recherche nous avons énumérer un certain de plugins nécessaires pour notre projet. Ils sont :

* **vagrant-vbguest** : Ce plugin assure que les "Guest Additions" de VirtualBox sont à jour, ce qui est essentiel pour le bon fonctionnement des dossiers partagés et d'autres fonctionnalités.
* **vagrant-hostmanager** : Ce plugin gère le fichier /etc/hosts des hôtes et des invités, en ajoutant des entrées pour toutes vos machines définies dans Vagrant. Cela permet une résolution DNS plus facile entre vos machines.
* **vagrant-cachier** : Ce plugin met en cache les paquets et dépendances téléchargés pour réduire les temps de téléchargement et économiser de la bande passante.
* **vagrant-reload** : Ce plugin permet de recharger la machine après le provisionnement, ce qui peut être utile si vous devez redémarrer la machine pour appliquer certaines configurations.

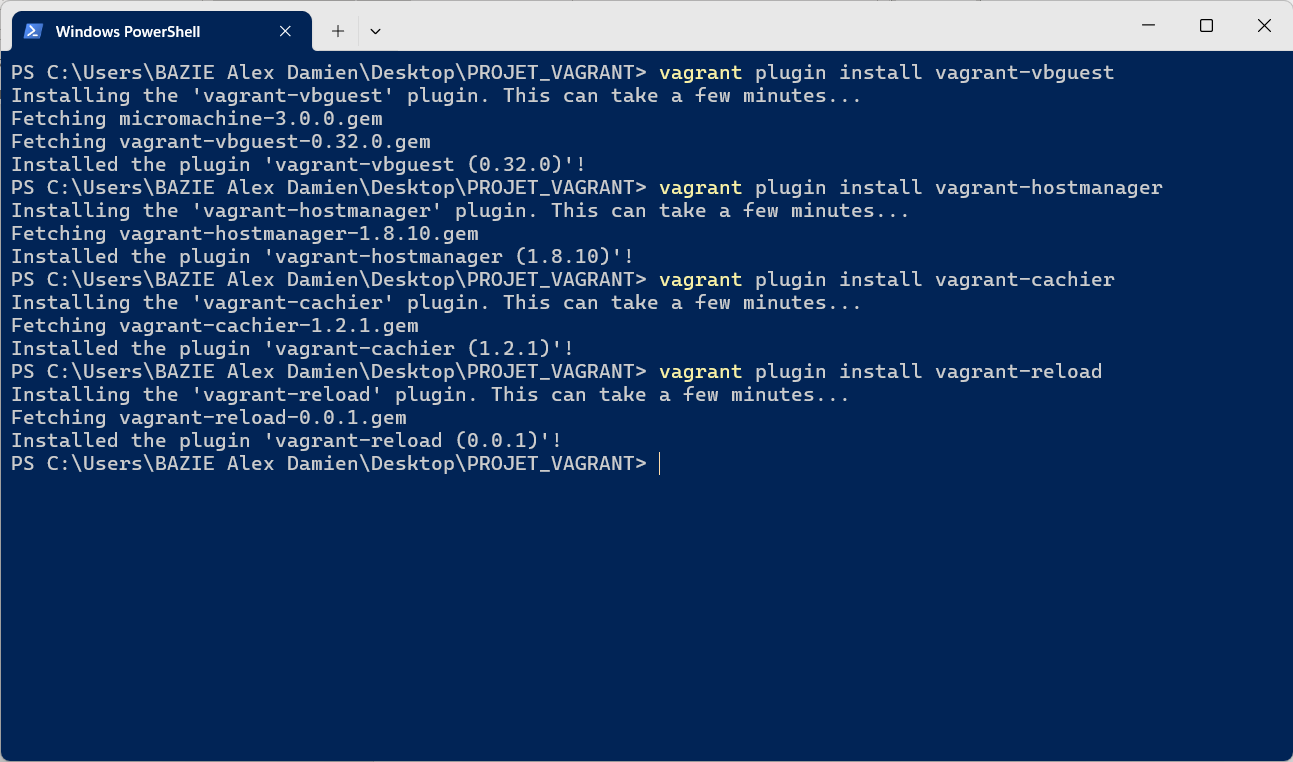
Procédons à l’installation de ses plugins en executant la commande « vagrant plugin install » suivie du nom du plugin :

*vagrant plugin install vagrant-vbguest*

*vagrant plugin install vagrant-hostmanager*

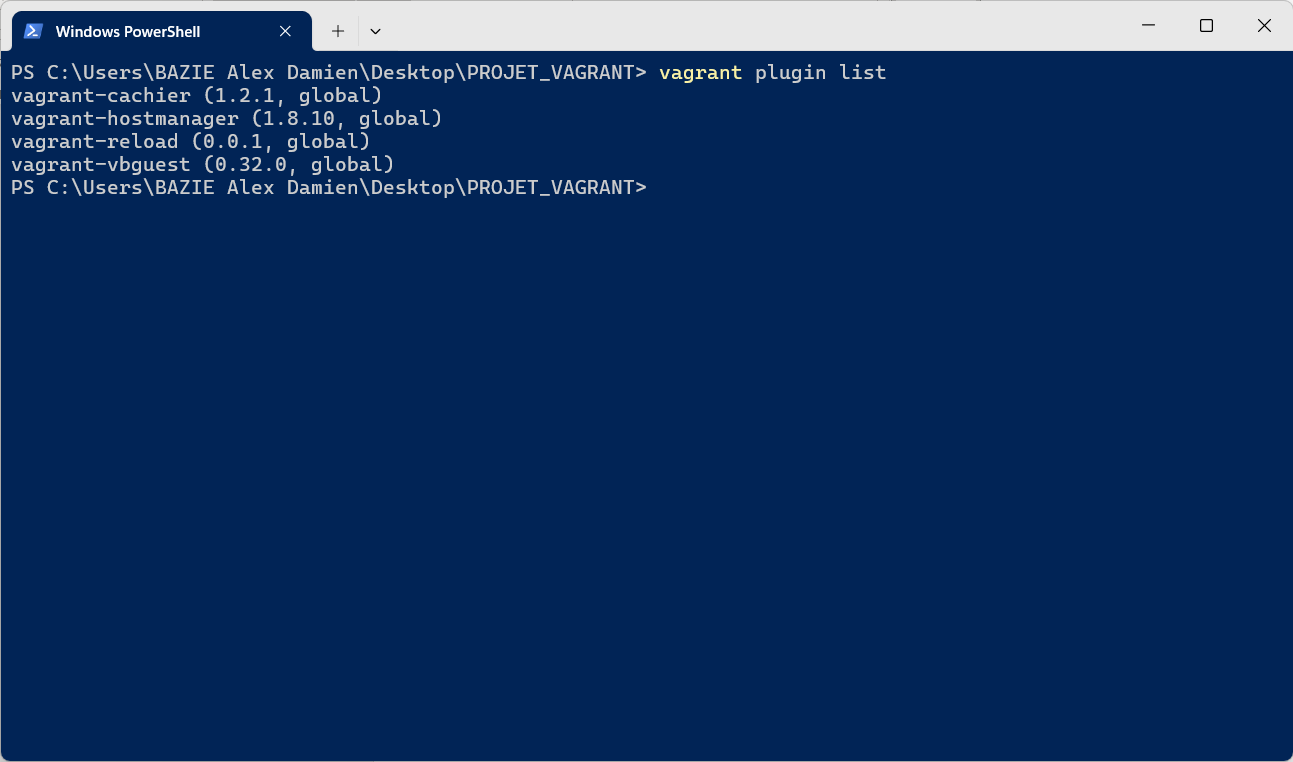
*vagrant plugin install vagrant-cachier*

*vagrant plugin install vagrant-reload*

**

Nous allons vérifier la liste des plugins pour être sur d’avoir installés les plugins nécessaires avec la commande :

*vagrant plugin list*

**

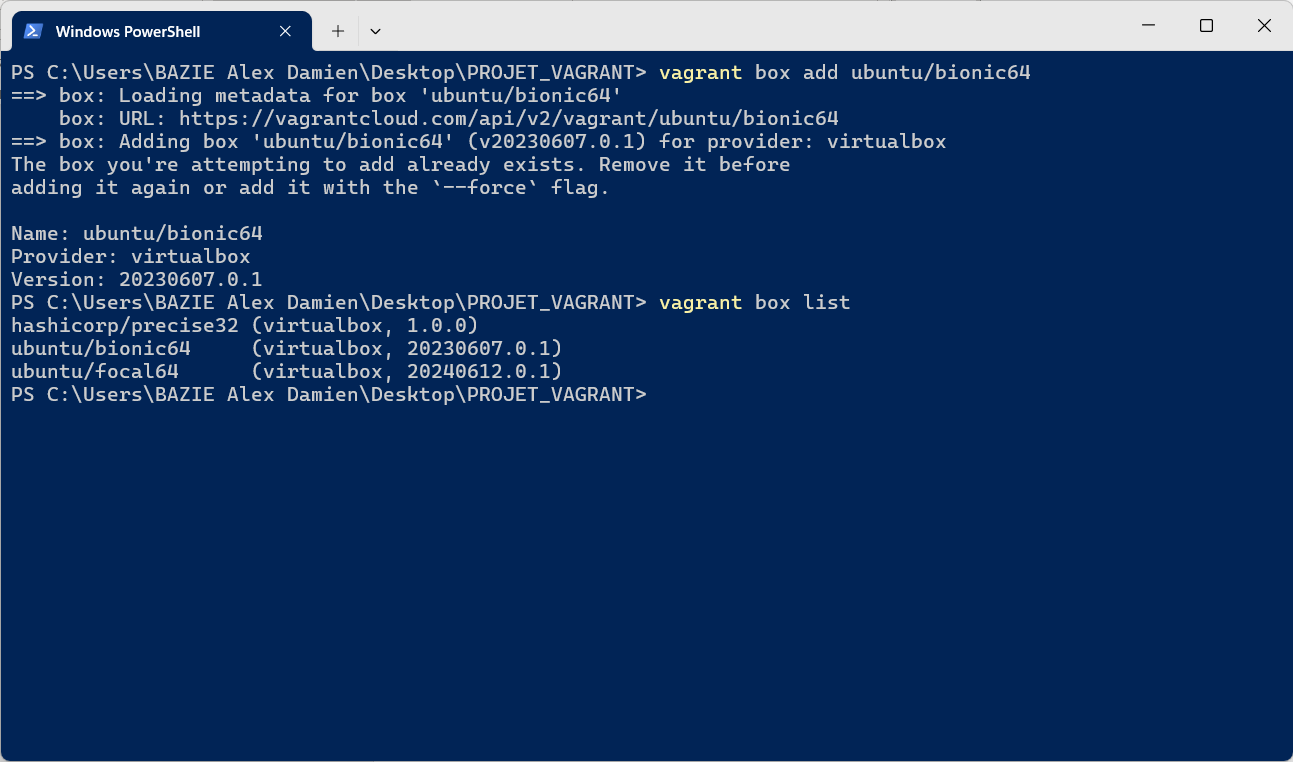
Nous constatons la présence effective de nos plugins.

Etape 2 : installation du box vagrant pour le projet

Dans le cadre de ce projet, le box vagrant qui a été recommandé est « ubuntu/bionic64 ». nous allons l’installer et vérifier que le box est bien n’installer avant le début des travaux a travers les commandes suivantes :

*vagrant box add ubuntu/bionic64*

*vagrant box list*

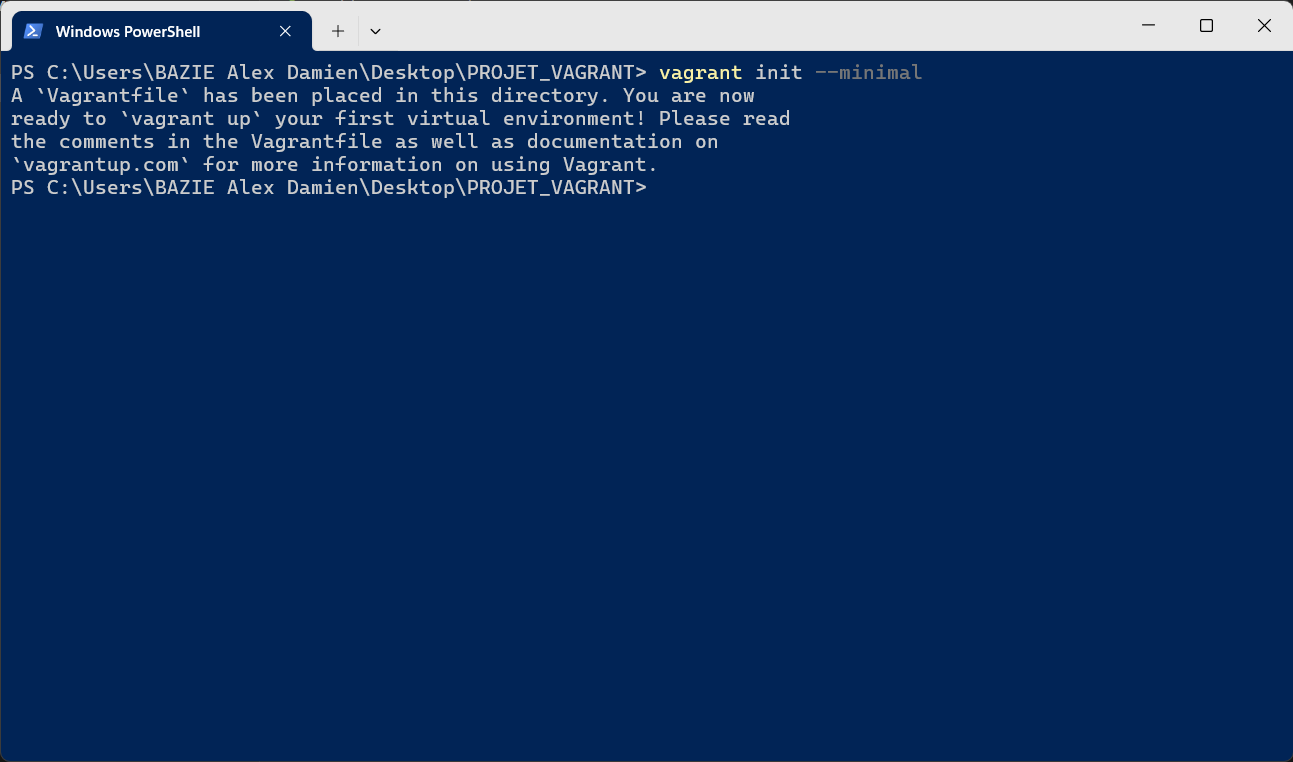


NB : le résultat de la commande pour l’installation nous notifie que nous avons le box installer et en vérifiant nous remarquons effectivement la présence du box dans la liste des box déjà installée.

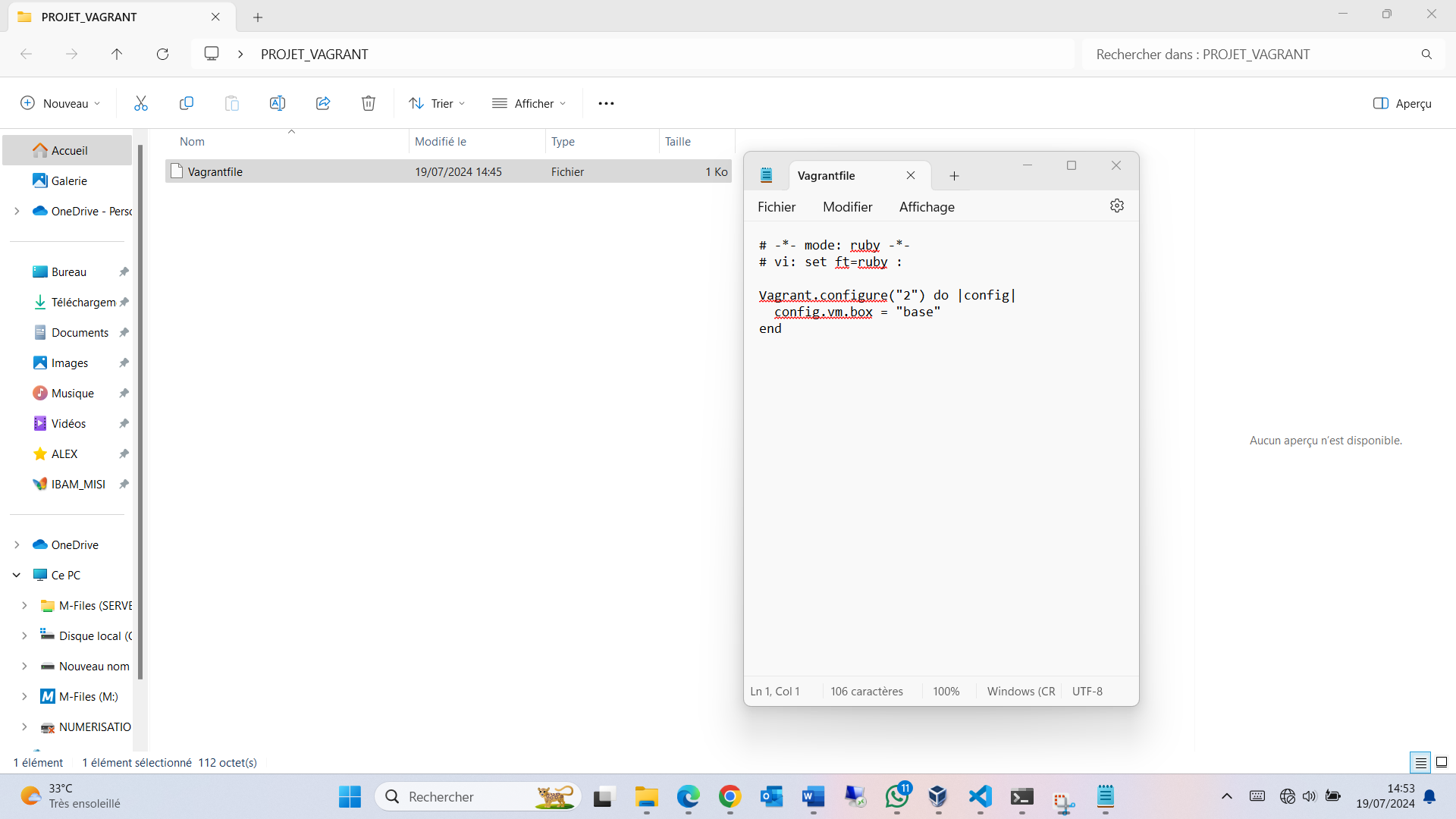
Etape 2 : initialisation du fichier vagrantfiles

Nous allons initialiser le fichier vagrantfiles de façon minimal a travers la commande suivante :

*vagrant init --minimal*



Après initialisation, nous constatons la présence effective de notre fichier « vagrantfiles » avec le contenu minimal :



Etape 3 : Ecriture des scripts pour la créer et configurer (réseaux et partage de fichiers) les machines virtuelles nécessaires pour chaque composant de l’architecture :

Composant 1 : le serveur web nommé « web\_server »

Pour le serveur web, nous aurons le script suivant :

Vagrant.configure("2") do |config|

  # Configuration du serveur web

  config.vm.define "web\_server" do |web|

    web.vm.box = "ubuntu/bionic64"

    web.vm.network "private\_network", ip: "192.168.33.12"

    web.vm.synced\_folder ".", "/vagrant", disabled: true

    web.vm.synced\_folder "./projetAlumni22/AgenceLuxe/static", "/home/vagrant/projetAlumni22/static"

    web.vm.provider "virtualbox" do |vb|

      vb.memory = "1024"

    end

    web.vm.provision "shell", path: "provision\_web\_server.sh"

  end

end

Ce script vagrant configure une machine virtuelle pour un serveur web sous Ubuntu 18.04. Il définit un serveur appelé « web\_server » avec une adresse IP privée de "192.168.33.12". La synchronisation des dossiers est activée pour rendre le répertoire de travail actuel disponible dans le dossier « /vagrant » de la machine virtuelle, et un dossier spécifique « ./projetAlumni22/AgenceLuxe/static » est synchronisé avec « /home/vagrant/projetAlumni22/static ». La machine virtuelle utilise VirtualBox avec 1024 Mo de mémoire. Enfin, un script de provisionnement (provision\_web\_server.sh) est exécuté pour configurer le serveur web une fois la machine virtuelle démarrée.

Composant 2 : le serveur d’application nommé « app\_server »

Pour le serveur d’application, nous aurons le script suivant :

Vagrant.configure("2") do |config|

  # Configuration du serveur d'application

  config.vm.define "app1\_server" do |app|

    app.vm.box = "ubuntu/bionic64"

    app.vm.network "private\_network", ip: "192.168.33.10"

    app.vm.synced\_folder ".", "/vagrant", disabled: true

    app.vm.provider "virtualbox" do |vb|

      vb.memory = "2048"

    end

    app.vm.provision "shell", path: "provision\_app\_server.sh"

    app.vm.synced\_folder "./projetAlumni22", "/home/vagrant/projetAlumni22"

  end

end

Ce script vagrant configure une machine virtuelle pour un serveur d'application sous Ubuntu 18.04. Il définit un serveur appelé « app1\_server » avec une adresse IP privée de "192.168.33.10". La synchronisation des dossiers est activée pour rendre le répertoire local « ./projetAlumni22 » disponible dans le dossier « /home/vagrant/projetAlumni22 » de la machine virtuelle, tandis que la synchronisation du dossier par défaut est désactivée. La machine virtuelle utilise VirtualBox avec 2048 Mo de mémoire. Enfin, des scripts de provisionnement (provision\_app\_server.sh et provision\_lb.sh) est exécuté pour configurer le serveur d'application une fois la machine virtuelle démarrée.

Composant 3 : le serveur de base de données nommé « db\_server »

Pour le serveur de base de données, nous aurons le script suivant :

Vagrant.configure("2") do |config|

# Configuration du serveur de base de données

  config.vm.define "db\_server" do |db|

    db.vm.box = "ubuntu/bionic64"

    db.vm.network "private\_network", ip: "192.168.33.11"

    db.vm.synced\_folder ".", "/vagrant", disabled: true

    db.vm.provider "virtualbox" do |vb|

      vb.memory = "4096"

    end

    db.vm.provision "shell", path: "provision\_db\_server.sh"

  end

end

Ce script vagrant configure une machine virtuelle pour un serveur de base de données sous Ubuntu 18.04. Il définit un serveur appelé « db\_server » avec une adresse IP privée de "192.168.33.11". La synchronisation du dossier par défaut est désactivée. La machine virtuelle utilise VirtualBox avec 4096 Mo de mémoire. Enfin, un script de provisionnement (provision\_db\_server.sh) est exécuté pour configurer le serveur de base de données une fois la machine virtuelle démarrée.

1. **Provisionnement automatique**

Composant 1 : le fichier de provisionnement nommé « provision\_web\_server.sh »

Pour ce fichier, nous aurons le script suivant :

#!/bin/bash

# Mettre à jour les paquets et installer nginx

sudo apt-get update

sudo apt-get install -y nginx

# Configurer NGINX pour l'équilibrage de charge

sudo bash -c 'cat > /etc/nginx/sites-available/default <<EOF

upstream django\_servers {

    server 192.168.33.10:8000;

}

server {

    listen 80;

    location / {

        proxy\_pass http://django\_servers;

        proxy\_set\_header Host \$host;

        proxy\_set\_header X-Real-IP \$remote\_addr;

        proxy\_set\_header X-Forwarded-For \$proxy\_add\_x\_forwarded\_for;

        proxy\_set\_header X-Forwarded-Proto \$scheme;

    }

}

EOF'

# Configurer Nginx

sudo rm /etc/nginx/sites-enabled/default

cat <<EOL | sudo tee /etc/nginx/sites-available/projetAlumni22

server {

    listen 80;

    server\_name 192.168.33.12;

    location / {

        proxy\_pass http://192.168.33.10:8000;  # Adresse et port de votre application Django

        proxy\_set\_header Host \$host;

        proxy\_set\_header X-Real-IP \$remote\_addr;

        proxy\_set\_header X-Forwarded-For \$proxy\_add\_x\_forwarded\_for;

        proxy\_set\_header X-Forwarded-Proto \$scheme;

    }

    location /static/ {

        alias /home/vagrant/projetAlumni22/static/;  # Chemin vers vos fichiers statiques Django

    }

    location /media/ {

        alias /home/vagrant/projetAlumni22/media/;  # Chemin vers vos fichiers media Django

    }

}

EOL

sudo ln -s /etc/nginx/sites-available/projetAlumni22 /etc/nginx/sites-enabled/projetAlumni22

sudo systemctl restart nginx

Ce script Bash met à jour les paquets du système et installe NGINX. Ensuite, il configure NGINX pour équilibrer la charge entre les serveurs d'application. Le fichier de configuration par défaut d'NGINX est remplacé par une nouvelle configuration qui définit un pool de serveurs d'application en amont. Le script configure également un nouveau fichier de site pour le projet "projetAlumni22", définissant les règles de proxy pour acheminer les requêtes vers un serveur Django fonctionnant à l'adresse 192.168.33.10:8000. Il configure aussi des alias pour les fichiers statiques et media de Django. Enfin, le script active la nouvelle configuration du site et redémarre NGINX pour appliquer les modifications.

Composant 2 : le fichier de provisionnement nommé « provision\_app\_server.sh »

Pour ce fichier, nous aurons le script suivant :

#!/bin/bash

# Mettre à jour la liste des paquets

sudo apt-get update

# Installer pip3 s'il n'est pas déjà installé

sudo apt-get install -y python3-pip

# Installer les dépendances nécessaires

sudo apt-get install -y python3-pip libpq-dev

# Désinstaller Django

sudo pip3 uninstall -y Django

# Installer les paquets Python requis

sudo pip3 install django==2.2.28

sudo pip3 install djangorestframework

sudo pip3 install django-phonenumber-field

sudo pip3 install psycopg2

Ce script Bash met à jour la liste des paquets du système et installe pip3 (le gestionnaire de paquets pour Python 3) s'il n'est pas déjà installé. Il installe également les dépendances nécessaires, notamment « libpq-dev » pour l'interaction avec PostgreSQL. Ensuite, il désinstalle Django s'il est déjà installé, puis installe une version spécifique de Django (2.2.28) ainsi que d'autres paquets Python requis pour le projet : « djangorestframework » pour les API REST, « django-phonenumber-field » pour la gestion des numéros de téléphone, et « psycopg2 » pour la connexion à une base de données.

Composant 3 : le fichier de provisionnement nommé « provision\_db\_server.sh »

Pour ce fichier, nous aurons le script suivant :

#!/usr/bin/env bash

# Mettre à jour les paquets et installer PostgreSQL

sudo apt-get update

sudo apt-get install -y postgresql postgresql-contrib

# Configurer PostgreSQL

sudo -u postgres psql -c "CREATE USER vagrant WITH PASSWORD 'password';"

sudo -u postgres psql -c "ALTER ROLE vagrant SET client\_encoding TO 'utf8';"

sudo -u postgres psql -c "ALTER ROLE vagrant SET default\_transaction\_isolation TO 'read committed';"

sudo -u postgres psql -c "ALTER ROLE vagrant SET timezone TO 'UTC';"

sudo -u postgres psql -c "CREATE DATABASE myproject;"

sudo -u postgres psql -c "GRANT ALL PRIVILEGES ON DATABASE myproject TO vagrant;"

# Modifier pg\_hba.conf pour ajouter une nouvelle ligne

echo "host    all             all             192.168.33.10/32        md5" | sudo tee -a /etc/postgresql/10/main/pg\_hba.conf

# Modifier postgresql.conf pour ajouter listen\_addresses = '\*'

sudo sed -i "/^#listen\_addresses = 'localhost'/c\listen\_addresses = '\*'" /etc/postgresql/10/main/postgresql.conf

# Redémarrer PostgreSQL pour appliquer les modifications

sudo systemctl restart postgresql

Ce script Bash met à jour la liste des paquets du système et installe PostgreSQL ainsi que ses utilitaires supplémentaires. Il configure ensuite PostgreSQL en créant un utilisateur nommé « vagrant » avec le mot de passe « password ». Le script configure également ce rôle pour utiliser l'encodage UTF-8, une isolation de transaction par défaut en lecture commise, et le fuseau horaire UTC. Ensuite, il crée une base de données nommée « myproject » et accorde tous les privilèges sur cette base de données à l'utilisateur « vagrant ».

Pour permettre les connexions réseau à PostgreSQL, le script ajoute une ligne au fichier « pg\_hba.conf » pour autoriser les connexions à partir de l'adresse IP « 192.168.33.10 » en utilisant l'authentification MD5. Il modifie également le fichier « postgresql.conf » pour permettre à PostgreSQL d'écouter sur toutes les adresses réseau en remplaçant « #listen\_addresses = 'localhost' » par « listen\_addresses = '\*' ». Enfin, le script redémarre PostgreSQL pour appliquer ces modifications.

1. **Exécution de notre fichier vagrantfilles**

Après la création des scripts nous allons rassembler les fichiers de création des machines virtuelles dans un fichier unique vagrantfile :

# -\*- mode: ruby -\*-

# vi: set ft=ruby :

Vagrant.configure("2") do |config|

    # Configuration du serveur web

    config.vm.define "web\_server" do |web|

      web.vm.box = "ubuntu/bionic64"

      web.vm.network "private\_network", ip: "192.168.33.12"

      web.vm.synced\_folder ".", "/vagrant", disabled: true

      web.vm.synced\_folder "./projetAlumni22/AgenceLuxe/static", "/home/vagrant/projetAlumni22/static"

      web.vm.provider "virtualbox" do |vb|

        vb.memory = "1024"

      end

      web.vm.provision "shell", path: "provision\_web\_server.sh"

    end

  end

  # Configuration du serveur d'application

  config.vm.define "app\_server" do |app|

    app.vm.box = "ubuntu/bionic64"

    app.vm.network "private\_network", ip: "192.168.33.10"

    app.vm.synced\_folder ".", "/vagrant", disabled: true

    app.vm.provider "virtualbox" do |vb|

      vb.memory = "2048"

    end

    app.vm.provision "shell", path: "provision\_app\_server.sh"

    app.vm.provision "shell", path: "provision\_db\_server.sh"

    app.vm.synced\_folder "./projetAlumni22", "/home/vagrant/projetAlumni22"

  end

# Configuration du serveur de base de données

  config.vm.define "db\_server" do |db|

    db.vm.box = "ubuntu/bionic64"

    db.vm.network "private\_network", ip: "192.168.33.11"

    db.vm.synced\_folder ".", "/vagrant", disabled: true

    db.vm.provider "virtualbox" do |vb|

      vb.memory = "4096"

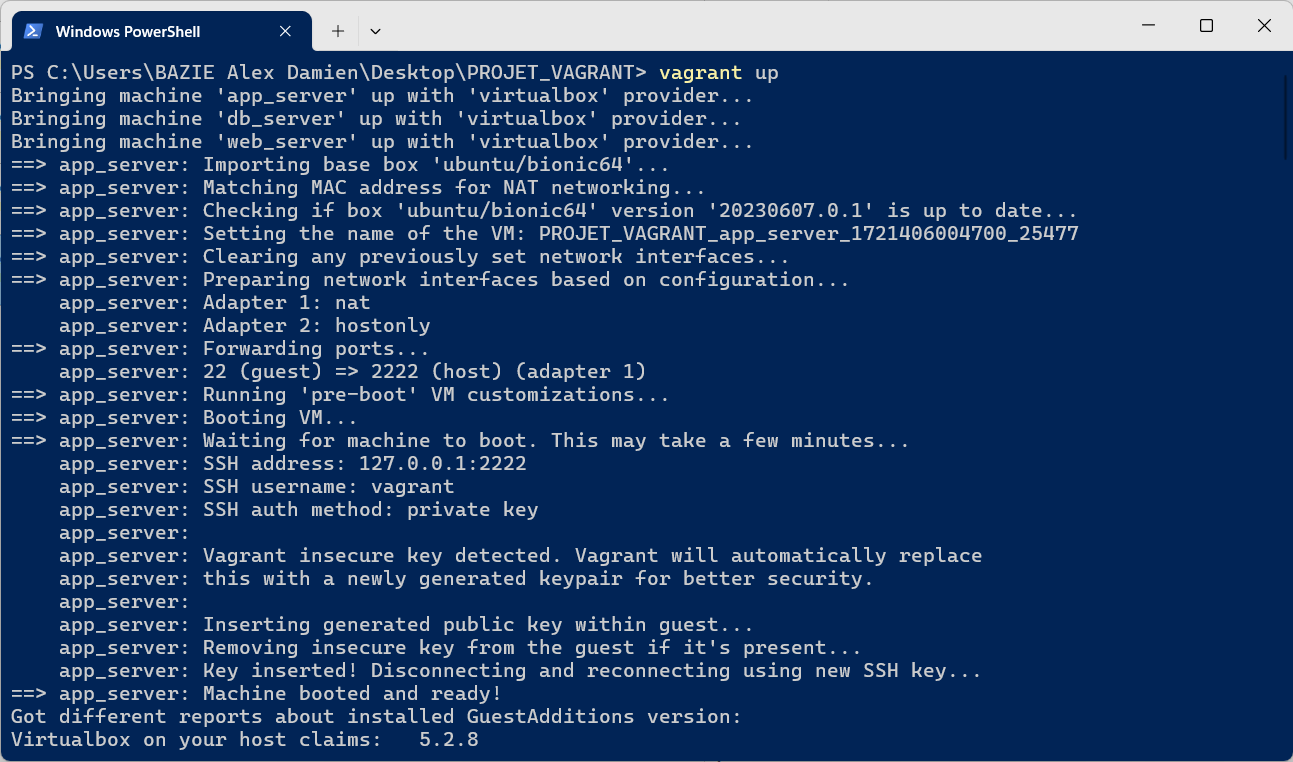
    end

    db.vm.provision "shell", path: "provision\_db\_server.sh"

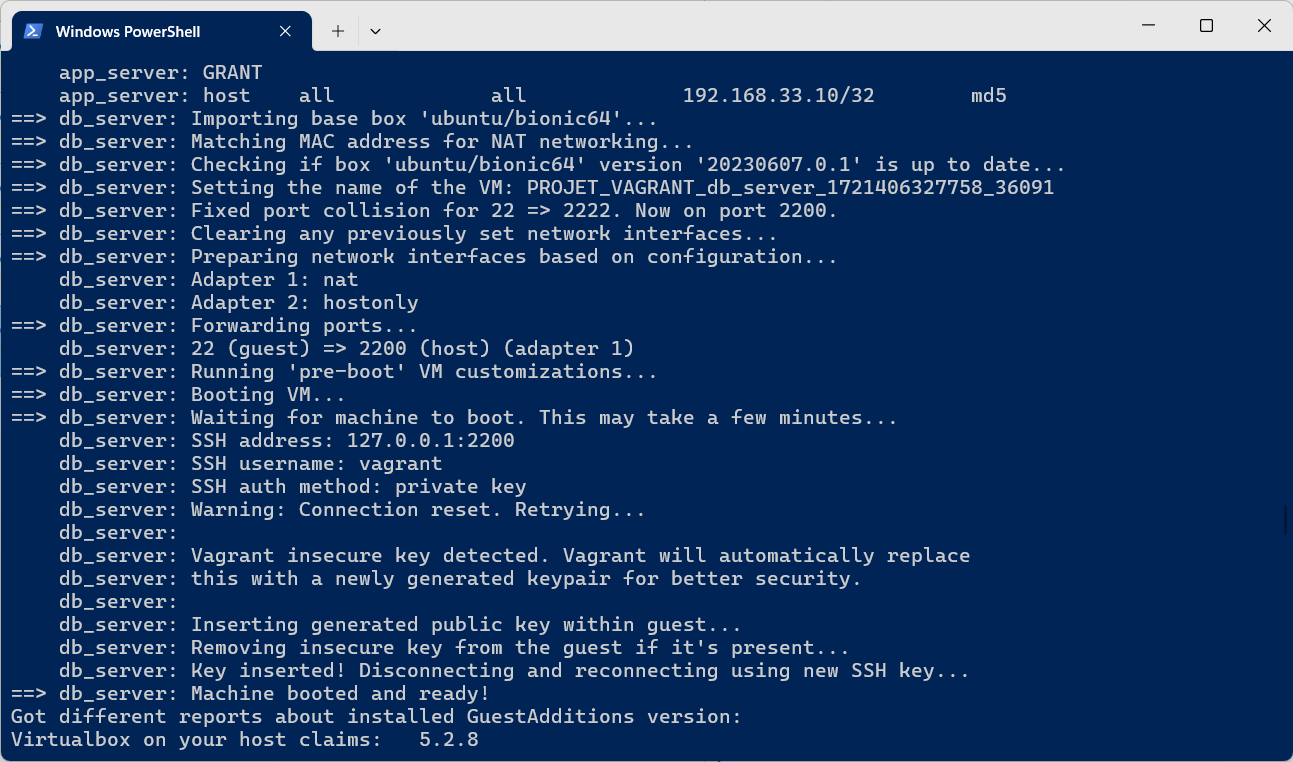
  end

Après la constitution du fichier unique vagrantfile, nous allons l’exécuter a travers la commande suivante :

*vagrant up*



Nous remarquons dans la capture ci-dessous l’exécution sans erreur de notre fichier vagrantfile :



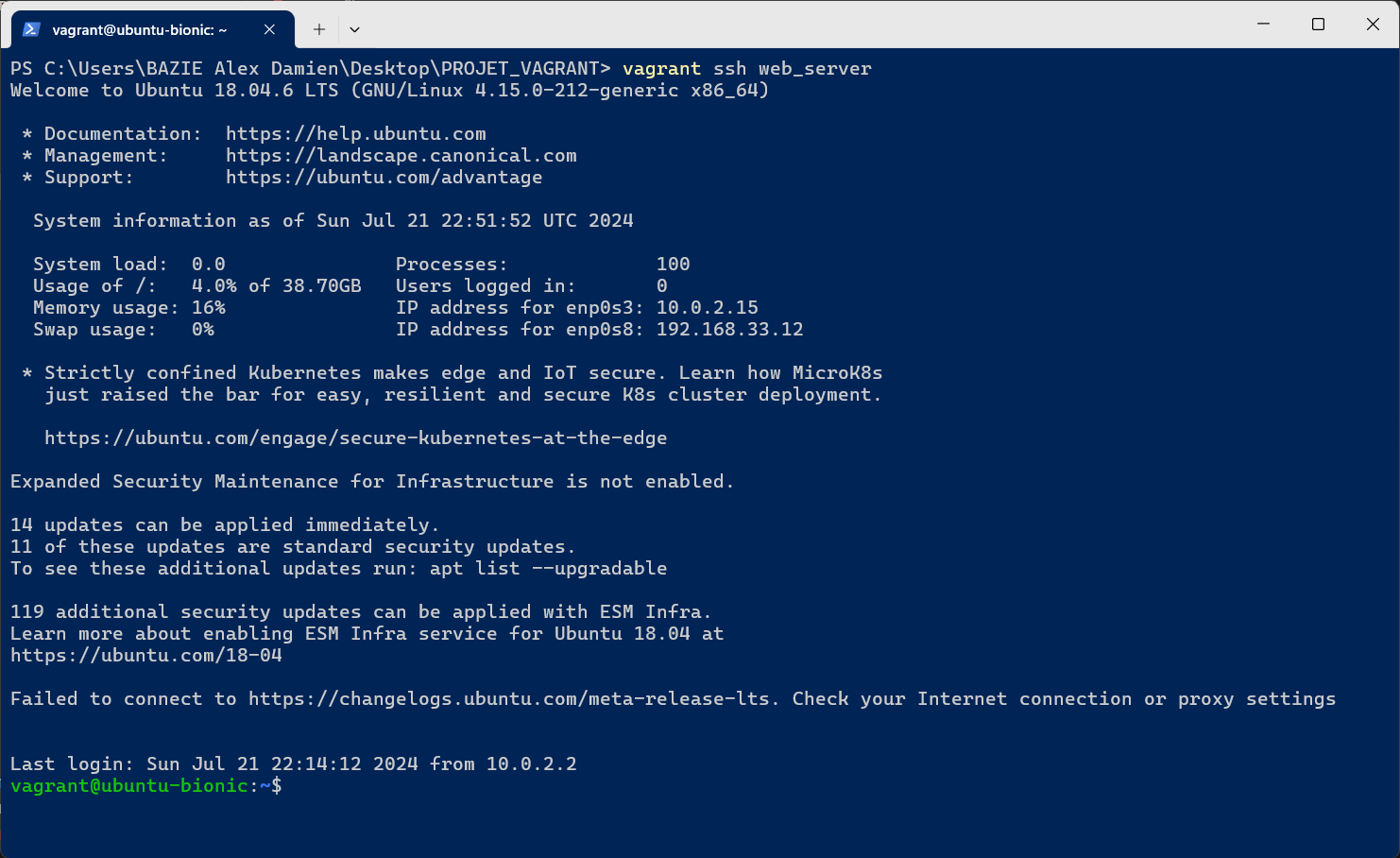
1. **Connexions aux machines virtuelles et vérifications des instances**

Nous allons procéder à la connexion au machine virtuelle et vérifier les instances installées :

* Serveur web

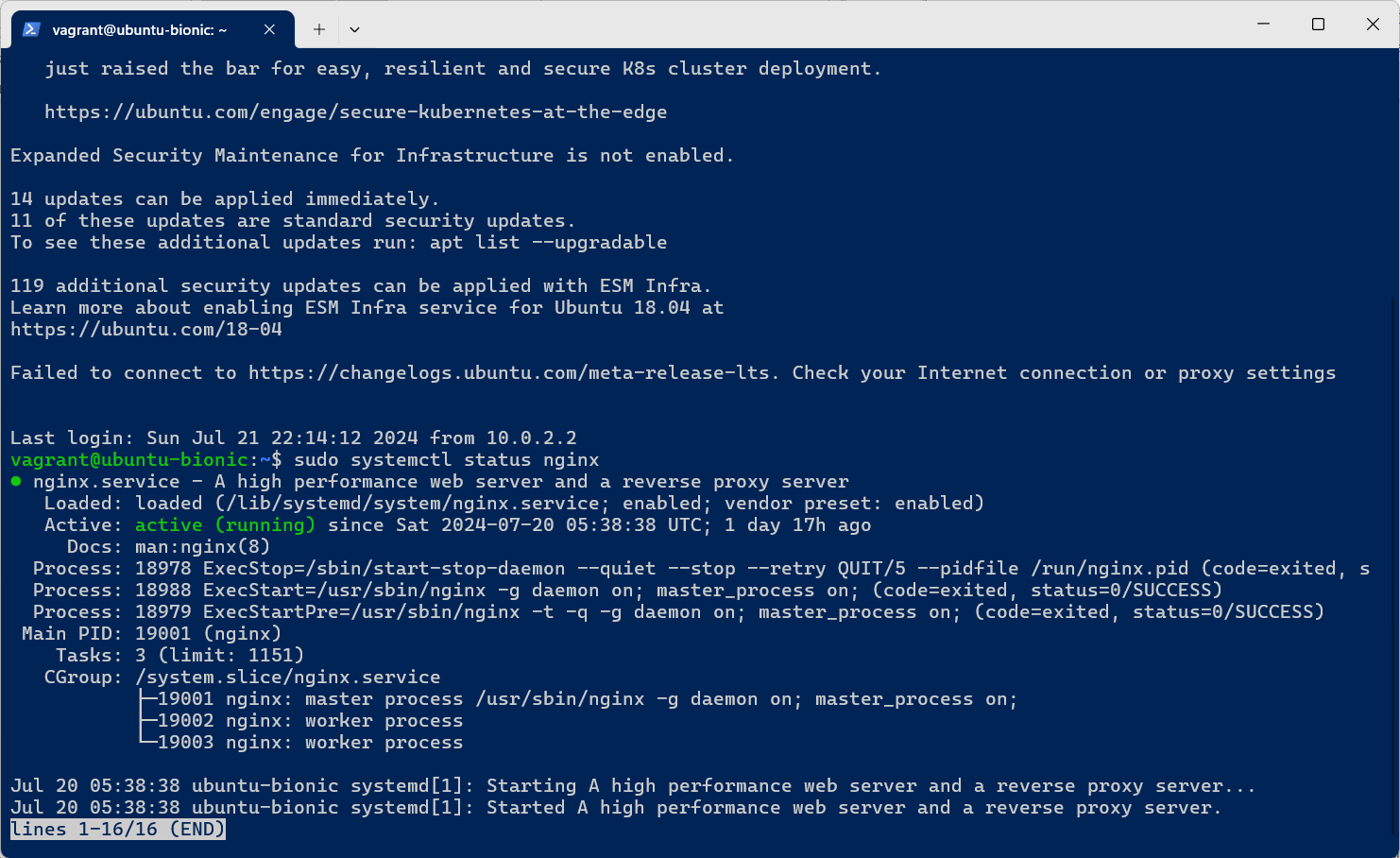
Pour se connecter à cette machine, nous allons exécuter la commande :

*vagrant ssh web\_server*

**

Nous allons vérifier les services du serveur nginx sur notre serveur web a travers la commande suivante :

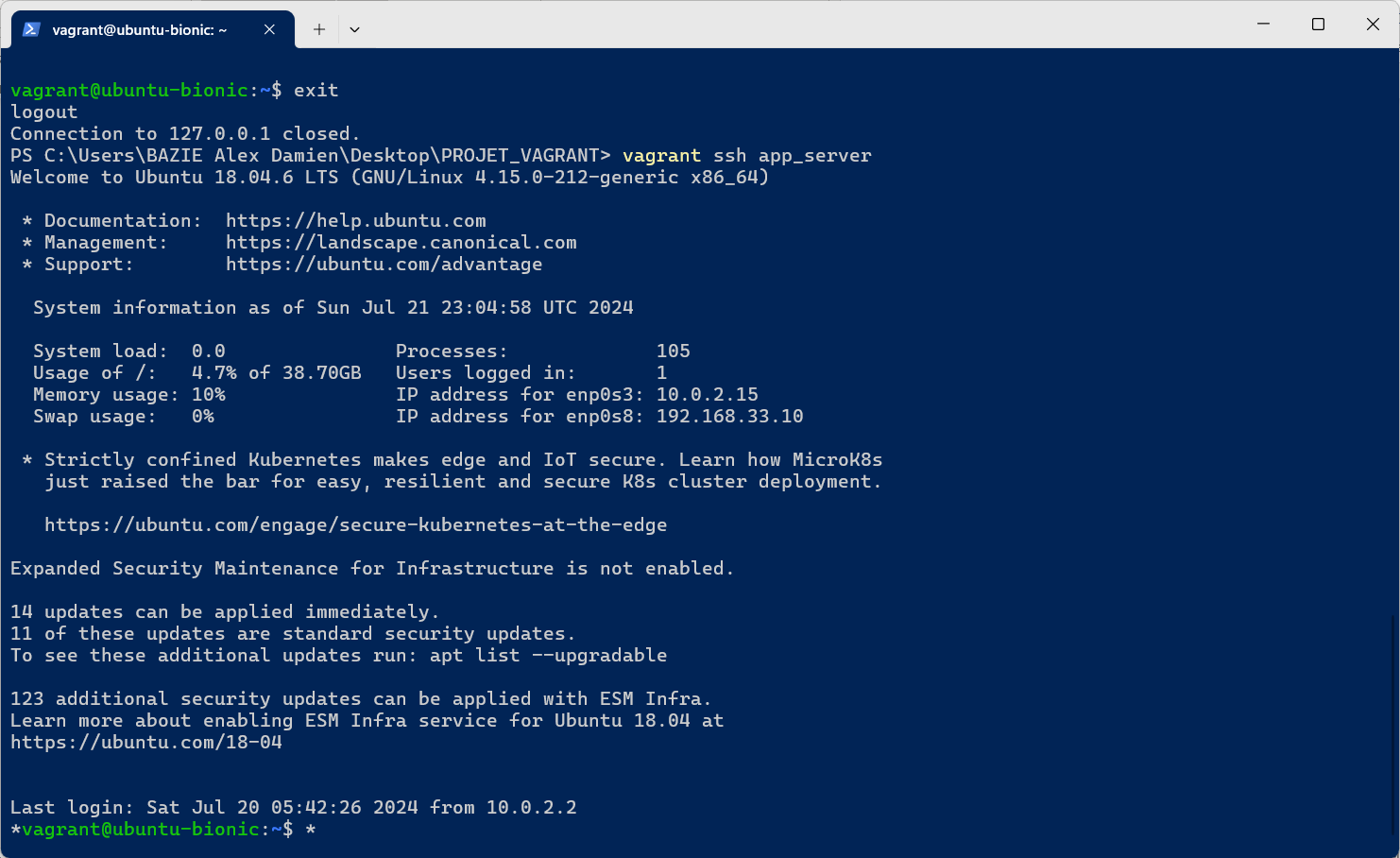
*sudo systemctl status nginx*



* Serveur d’application

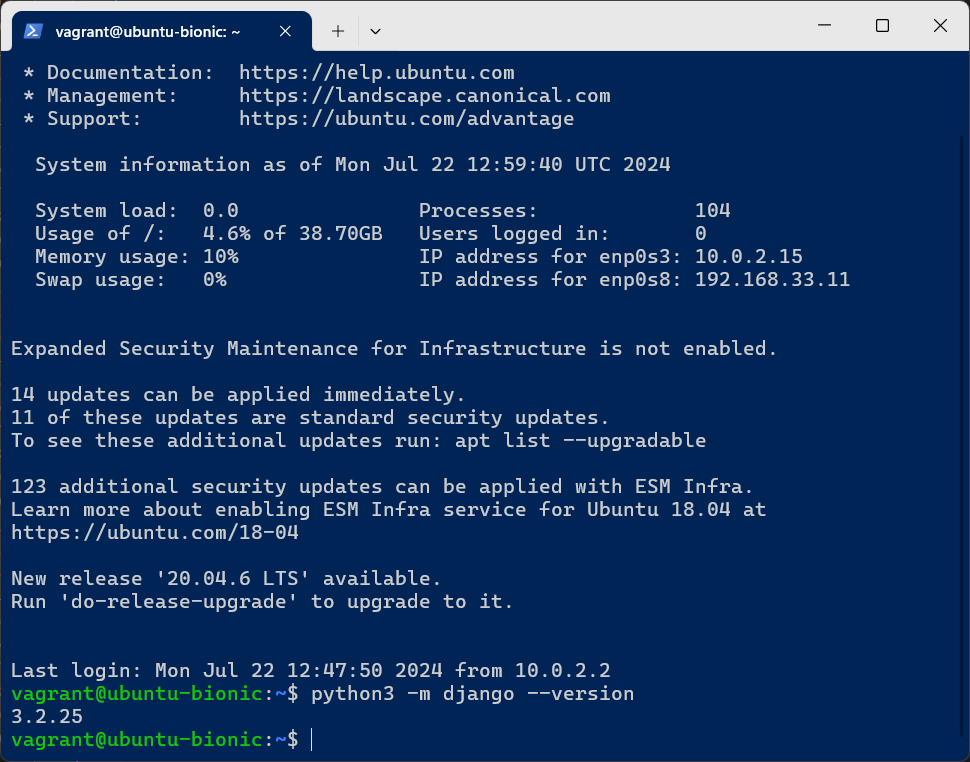
Pour se connecter à cette machine, nous allons exécuter la commande :

*vagrant ssh app\_server*

**

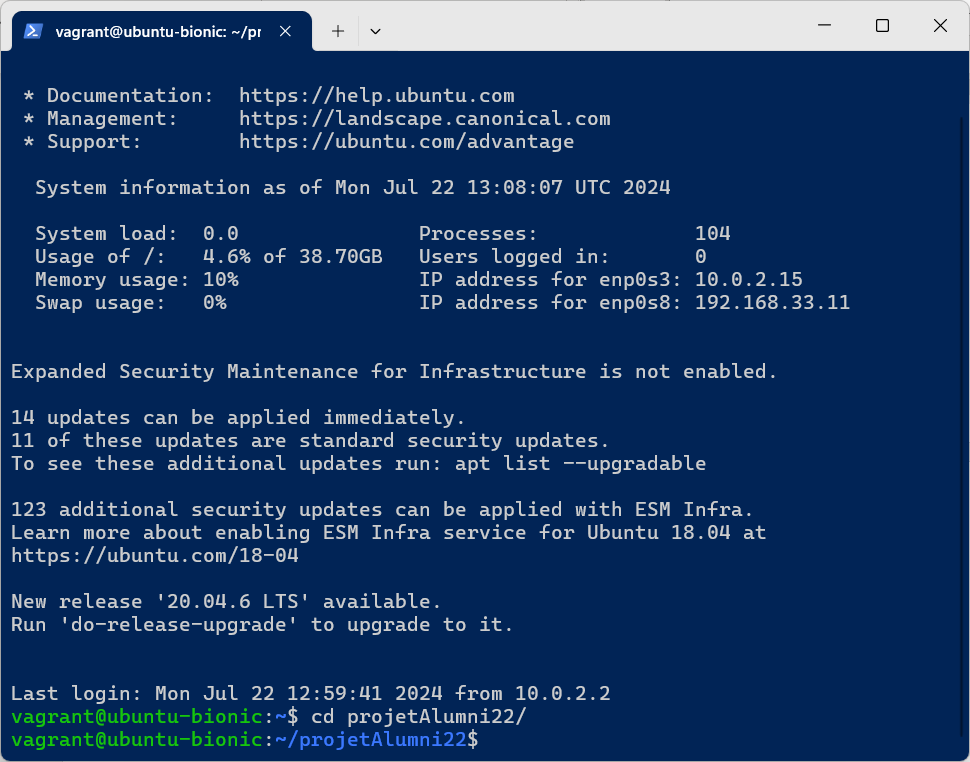
Nous allons vérifier si le framework django est installer sur notre serveur d’application à travers la commande suivante :

*python3 -m django --version*



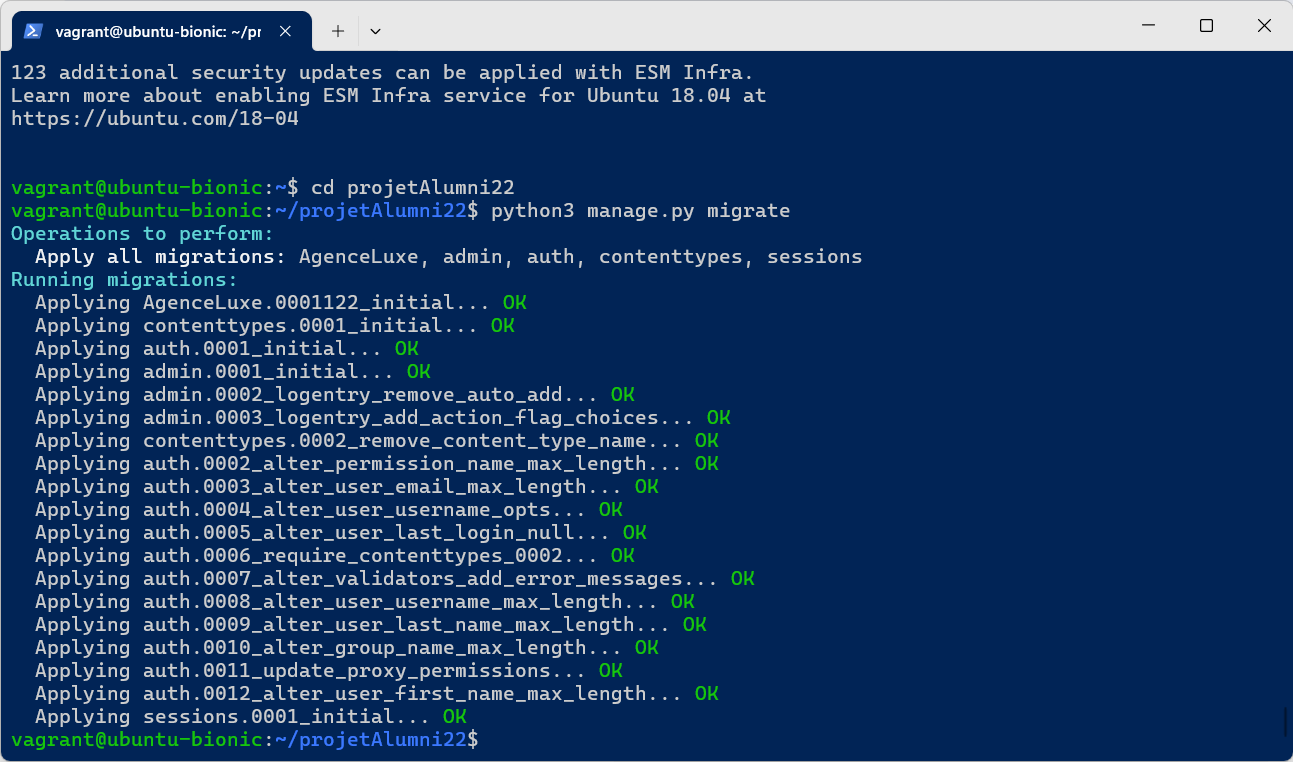
D’abord, nous allons nous mettre sur notre dossier partagé avec la commande :

*cd projetAlumni22*

**

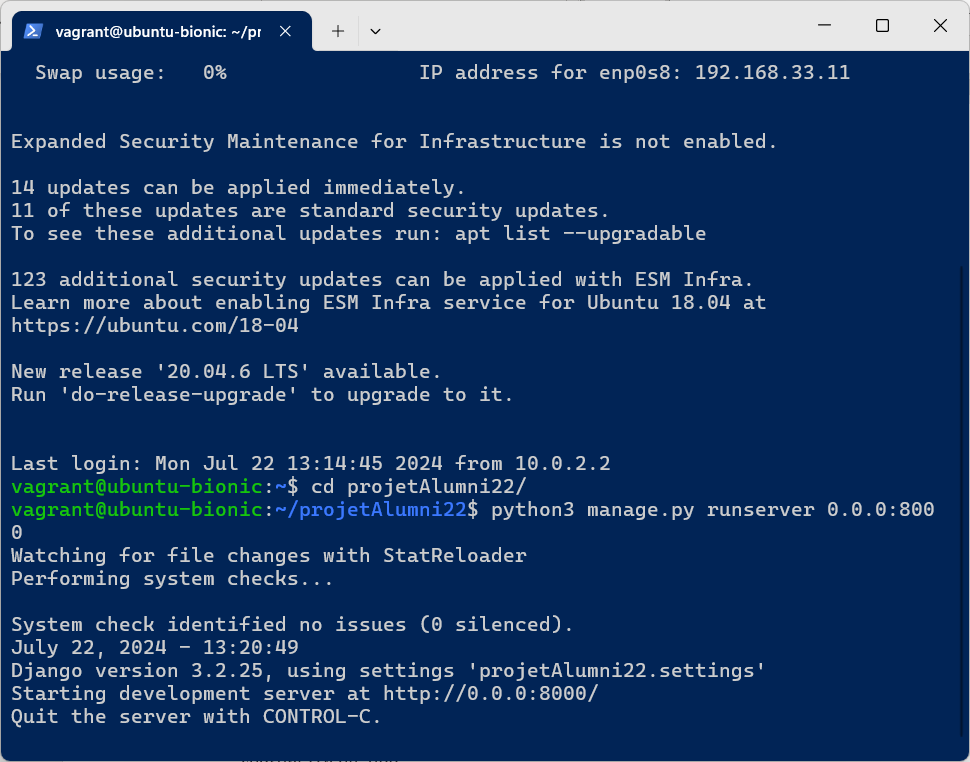
Ensuitenous allons procéder à la migration de la base de données sur le serveur de base données avec la commande :

*python3 manage.py migrate*



Enfin, nous allons lancer notre application avec la commande suivante :

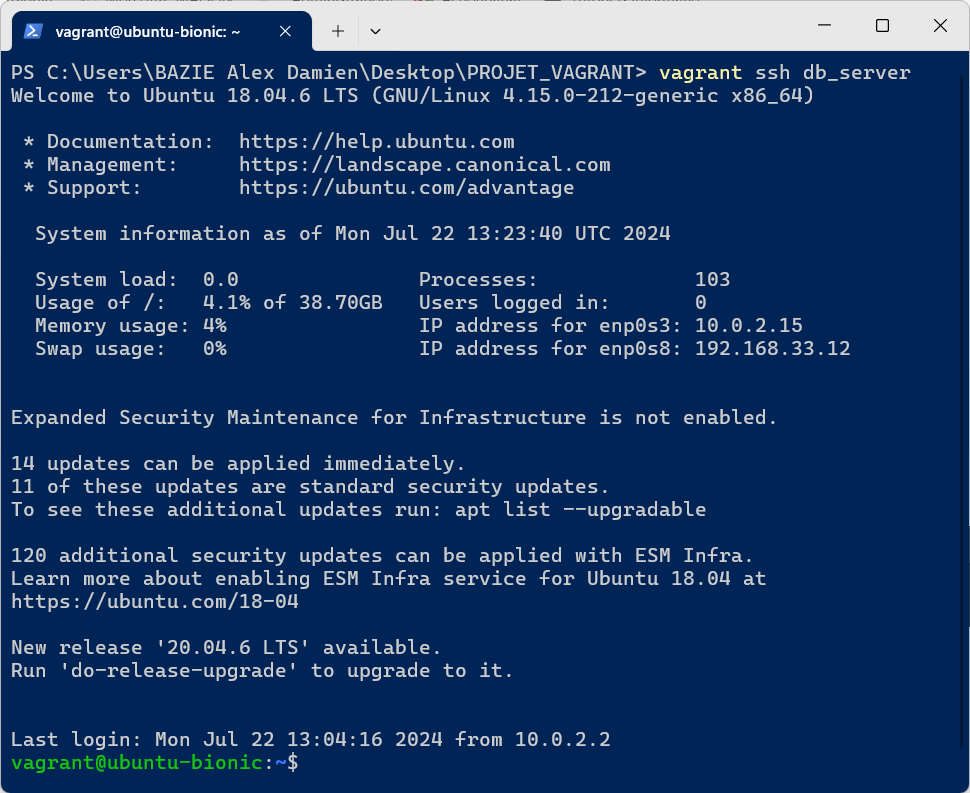
*python3 manage.py runserver 0.0.0:8000*

**

* Serveur de base de données

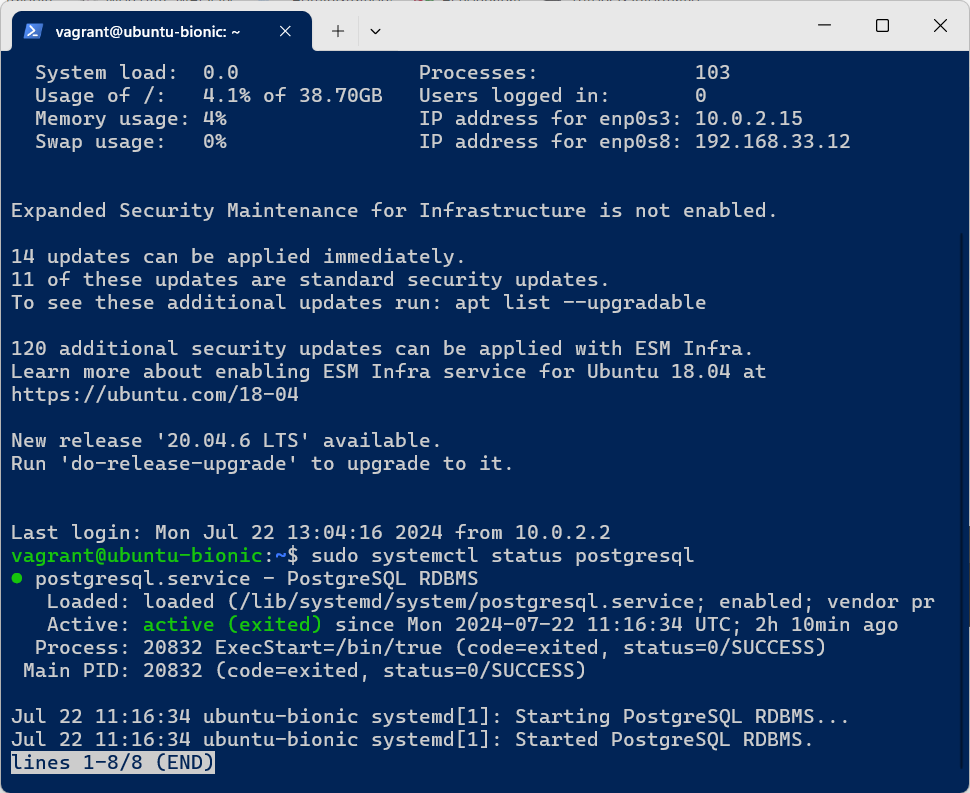
Pour se connecter à cette machine, nous allons exécuter la commande :

*vagrant ssh db\_server*



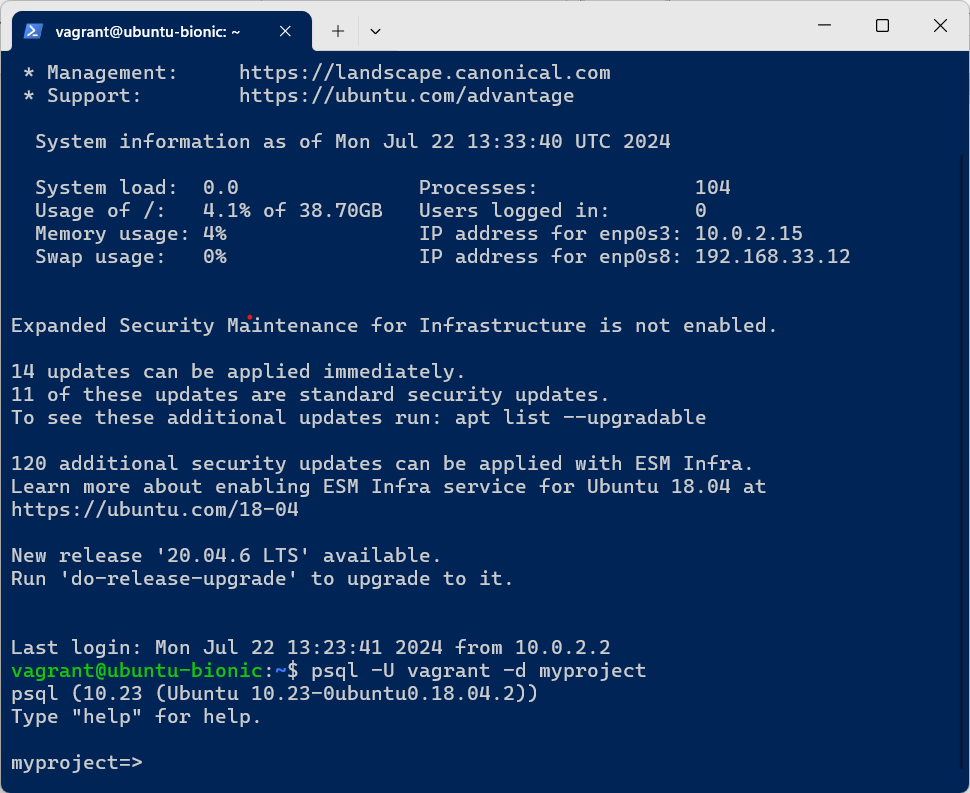
Après connexion, nous allons tester le service de prostgresql avec la commande suivante :

*sudo systemctl status postgresql*



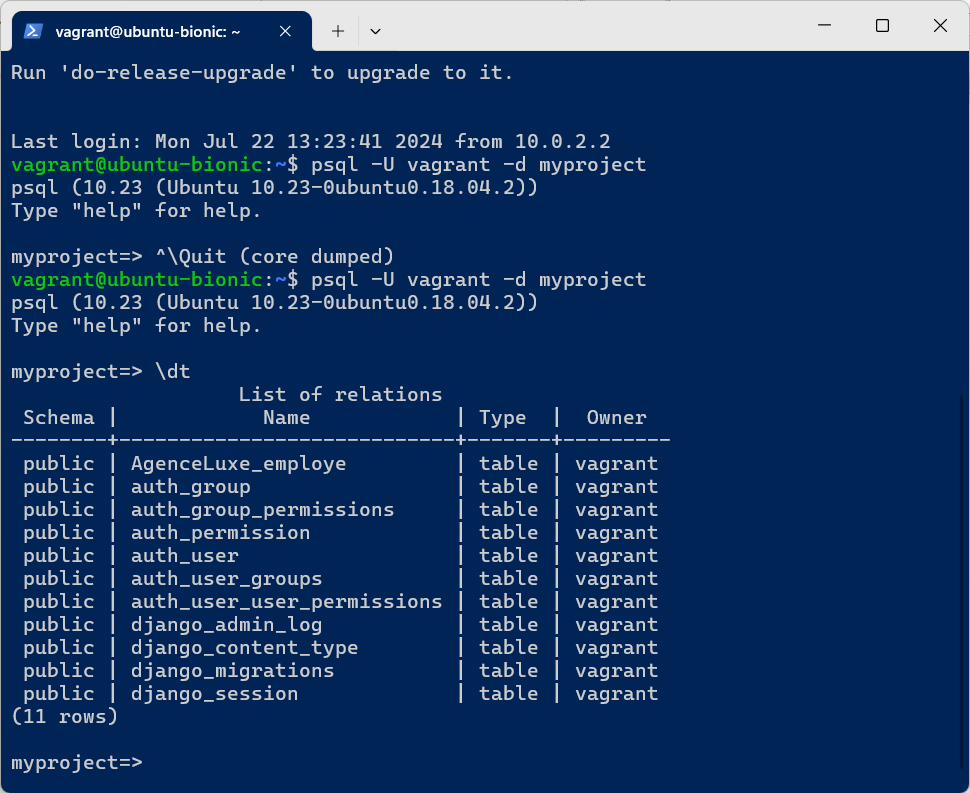
Nous allons nous connecter à notre base de données a travers la commande suivante :

*psql -U vagrant -d myproject*



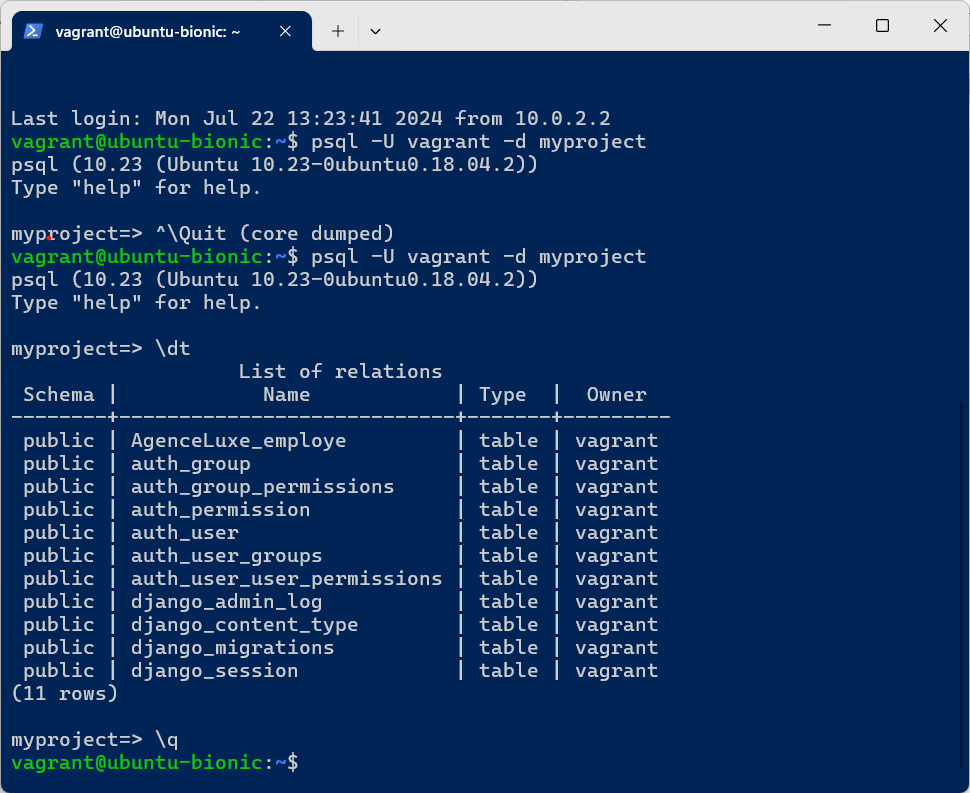
Nous venons de nous connecter à la base de données du projet, nous allons maintenant vérifier la liste des tables :

*\dt*

**

Pour quitter la base de données nous allons exécuter la commande suivante :

*\q*

**

1. **Connexions à l’application à travers le client**

**Notre machine sera utiliser comme client pour nous connecter à notre application sur le lien**