Projet : Déploiement Automatisé d'un Environnement de Test avec Docker, Vagrant, Linux et GitHub

Objectif du Projet

L'objectif est de mettre en place un environnement de test local automatisé pour une application simple, permettant de simuler un environnement de production allégé. Ce projet t'aidera à maîtriser l'orchestration de conteneurs avec Docker, la gestion des machines virtuelles avec Vagrant, et la collaboration et le versionnement de code avec GitHub.

Description du Projet

Ce projet consiste à créer une application web avec Python déployée dans un environnement Docker. Le déploiement de l'environnement de test se fera dans une machine virtuelle provisionnée par Vagrant, en automatisant la configuration du serveur sous Linux.

Étapes du Projet

1. **Initialisation du Dépôt GitHub :**
    - Créer un dépôt GitHub pour stocker tout le code et les configurations.
    - Versionner les scripts de configuration et le `Dockerfile` de l'application.
    - Rédiger un README expliquant les étapes d'installation et de déploiement du projet.
2. **Configuration de la Machine Virtuelle avec Vagrant :**
    - Utiliser Vagrant pour créer une VM Linux (par exemple, Ubuntu) qui servira d'environnement de test.
    - Configurer le fichier `Vagrantfile` pour définir la mémoire, le processeur et le réseau de la VM.
    - Automatiser l'installation de Docker dans la VM avec un script de provisionnement (bash script dans le `Vagrantfile`).
3. **Déploiement de l'Application avec Docker :**
    - Écrire un `Dockerfile` pour créer une image Docker de l'application (par exemple, un petit service web en Python).
    - Configurer un `docker-compose.yml` pour orchestrer le service principal et, si possible, ajouter un second service pour simuler une base de données (ex. Redis ou MySQL).
    - Automatiser le démarrage des conteneurs via `docker-compose` sur la VM.
4. **Documentation et Automatisation sur GitHub :**
    - Créer un workflow GitHub Actions (ou une simple documentation) pour automatiser le test de l'image Docker (par exemple, vérification que l'image est construite avec succès).
    - Documenter chaque étape dans le README pour expliquer comment lancer l'environnement et accéder au service dans la VM.

Livrables Attendues

1. **Dépôt GitHub :**
    - `Vagrantfile` pour la configuration de la VM.

    - `Dockerfile` et `docker-compose.yml` pour la création de l'environnement Docker.
    - Script de provisionnement pour automatiser l'installation de Docker dans la VM.
    - Documentation détaillant le déploiement, les tests et les accès.
2. **Présentation :**
    - Présentation de 10 minutes montrant le fonctionnement de l'environnement de test.
    - Explication des choix de configuration et démonstration du déploiement de l'application.

Compétences Développées

- Gestion de version et documentation avec GitHub.
- Configuration d'environnements virtuels avec Vagrant.
- Conteneurisation de services avec Docker et orchestration de base avec Docker Compose.
- Gestion d'un environnement Linux et automatisation des tâches de base.
- Stocker l'image de ton code dans un repo local sécurisé.

Difficultés supplémentaire à faire à la fin du Bootcamp :

Une fois que tu auras maîtrisé un certain nombre de concepts, tu pourras à la fin du bootcamp

- Rajouter un reverse prox treafik pour accéder à l'application en http (nécessite l'utilisation du conteneur Traefik

Voici les étapes détaillées :

1. **Ajouter un reverse proxy avec Traefik** :
    - Utilisez Traefik comme conteneur de reverse proxy pour rediriger les requêtes HTTP vers l'application Flask.
    - Configurez un fichier `docker-compose.override.yml` pour inclure le conteneur Traefik et définir des règles de routage vers le conteneur de l'application.
    - Exemple de configuration pour Traefik :

```
services:

  traefik:

    image: traefik:v2.5
```

```
    command:
      - "--api.insecure=true"
      - "--providers.docker"
      - "--entrypoints.web.address=:80"
    ports:
      - "80:80"
      - "8080:8080"
    volumes:
      - "/var/run/docker.sock:/var/run/docker.sock:ro"
```

- Sécuriser le conteneur de l'application pyhton (limiter l'accès aux ressources CPU, RAM)

```
services:
  flask-app:
    deploy:
      resources:
        limits:
          cpus: "0.5"    # Limiter à 50% du CPU
          memory: "512M" # Limiter à 512 MB de RAM
```

- Faire un build multi stage pour réduire la taille du conteneur.

```
# Étape de build
FROM python:3.9-slim AS builder
WORKDIR /app
COPY requirements.txt .
```

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


# Étape finale

FROM python:3.9-slim

WORKDIR /app

COPY --from=builder /usr/local/lib/python3.9/site-packages
/usr/local/lib/python3.9/site-packages

COPY . .

CMD ["python", "app.py"]
```

Mettre en place un pipeline de sécurité pour l'image du conteneur :

- Utilisez un outil de scan de sécurité, comme Trivy, pour vérifier les vulnérabilités de l'image Docker.
- Créez un workflow GitHub Actions pour automatiser ce scan à chaque changement de code. Exemple de configuration dans `.github/workflows/security-scan.yml` :

name: Security Scan

```
pipeline {

    agent any


    environment {

        DOCKER_IMAGE = "portfolio-app"

        TAG = "latest"

    }


    stages {

        stage('Checkout Code') {

            steps {
```

```
                // Cloner le dépôt Git
                git url:
'https://github.com/utilisateur/portfolio-app.git', branch:
'main'
            }
        }

        stage('Build Multi-Stage Docker Image') {
            steps {
                script {
                    // Construire l'image Docker multi-stage
                    sh "docker build -t ${DOCKER_IMAGE}:${TAG}
."
                }
            }
        }

        stage('Scan Docker Image with Trivy') {
            steps {
                script {
                    // Scanner l'image Docker pour détecter
les vulnérabilités
                    sh "trivy image --severity HIGH,CRITICAL
${DOCKER_IMAGE}:${TAG}"
                }
            }
        }
```

```
        stage('Push to Docker Registry') {
            steps {
                script {
                    // Pousser l'image vers un registre
(Docker Hub ou autre)

withCredentials([usernamePassword(credentialsId:
'docker-hub-credentials', usernameVariable: 'DOCKER_USER',
passwordVariable: 'DOCKER_PASS')]) {
                        sh "docker login -u ${DOCKER_USER} -p
${DOCKER_PASS}"
                        sh "docker tag ${DOCKER_IMAGE}:${TAG}
${DOCKER_USER}/${DOCKER_IMAGE}:${TAG}"
                        sh "docker push
${DOCKER_USER}/${DOCKER_IMAGE}:${TAG}"
                    }
                }
            }
        }

        stage('Deploy with Docker Compose') {
            steps {
                script {
                    // Déployer l'application avec Docker
Compose
                    sh "docker-compose up -d"
                }
            }
        }
```

```
    }


    post {

        always {

            // Nettoyer les images après le pipeline pour
économiser l'espace

            sh "docker rmi ${DOCKER_IMAGE}:${TAG} || true"

        }

    }

}
```