TD3 : Stack applicative 3 tiers avec Terraform et Ansible

Objectifs

- Provisionner une stack front / back / db avec Terraform
- Configurer chaque service avec Ansible
- Utiliser les groupes, variables et handlers Ansible
- Comprendre l'ordre d'execution entre services dependants

Contexte

Vous travaillez dans une equipe DevOps pour une startup.

L'equipe doit deployer un environnement de developpement compose de :

- un frontend NGINX (page statique)
- un backend Python Flask
- une base de donnees PostgreSQL

Probleme actuel :

- les developpeurs configurent les services a la main
- les environnements different d'une machine a l'autre
- le backend demarre avant la base de donnees et plante

Objectif :

Automatiser le deploiement et la configuration de la stack complete.

1. Preparation du projet

Creer la structure suivante :

```
mkdir td3-stack
cd td3-stack
mkdir terraform ansible
```

Structure attendue :

```
td3-stack/
  terraform/
    main.tf
    variables.tf
    outputs.tf
  ansible/
```

```
    inventory.ini
    playbook.yml
    group_vars/
      frontend.yml
      backend.yml
      database.yml
```

2. Provisionning Terraform

2.1 Fichier fourni a completer

Le fichier `terraform/main.tf` vous est fourni avec le frontend et le backend.

Fichier : `terraform/main.tf`

```
terraform {
  required_providers {
    docker = {
      source = "kreuzwerker/docker"
    }
  }
}

provider "docker" {}

# --- Images ---

resource "docker_image" "nginx" {
  name = "nginx:latest"
}

resource "docker_image" "flask" {
  name = "python:3.11-slim"
}

# A COMPLETER : image PostgreSQL
# resource "docker_image" "postgres" { ... }

# --- Reseau commun ---

resource "docker_network" "stack" {
  name = "td3-network"
}

# --- Frontend ---

resource "docker_container" "frontend" {
  name  = "frontend"
  image = docker_image.nginx.name

  networks_advanced {
    name = docker_network.stack.name
```

```
  }

  ports {
    internal = 80
    external = 8080
  }
}

# --- Backend ---

resource "docker_container" "backend" {
  name  = "backend"
  image = docker_image.flask.name

  networks_advanced {
    name = docker_network.stack.name
  }

  ports {
    internal = 5000
    external = 5000
  }

  command = ["sleep", "infinity"]
}

# A COMPLETER : conteneur PostgreSQL
# resource "docker_container" "database" { ... }
# port interne : 5432
# port externe : 5432
# variable d'environnement : POSTGRES_PASSWORD = "devpass"
```

Completer le fichier :

- ajouter l'image PostgreSQL
- ajouter le conteneur database avec les specifications ci-dessus
- connecter le conteneur au reseau td3-network

Question :

- Pourquoi tous les conteneurs sont-ils connectes au meme reseau Docker ?
- Que se passerait-il sans ce reseau commun ?

2.2 Variables

Fichier : `terraform/variables.tf`

```
variable "db_password" {
```

```
  description = "Mot de passe PostgreSQL"
  default     = "devpass"
}
```

Question :

- Ce fichier convient pour un environnement de developpement. Pourquoi est-il insuffisant en production ?

2.3 Outputs

Fichier : `terraform/outputs.tf`

```
output "frontend_url" {
  value = "http://localhost:8080"
}

output "backend_url" {
  value = "http://localhost:5000"
}

output "database_host" {
  value = "localhost"
}
```

2.4 Execution

```
cd terraform
terraform init
terraform plan
terraform apply
```

Verifier que les trois conteneurs sont actifs :

```
docker ps
```

3. Configuration Ansible

3.1 Inventaire

Fichier : `ansible/inventory.ini`

```
[frontend]
localhost ansible_connection=local

[backend]
localhost ansible_connection=local

[database]
localhost ansible_connection=local
```

Question :

- Cet inventaire utilise localhost pour tous les groupes. Dans un vrai projet, que contiendrait-il a la place ?

3.2 Variables par groupe

Fichier : `ansible/group_vars/frontend.yml`

```
container_name: frontend
service_port: 8080
env_label: "Developpement"
```

Fichier : `ansible/group_vars/backend.yml`

```
container_name: backend
service_port: 5000
db_host: database
db_port: 5432
```

Fichier : `ansible/group_vars/database.yml`

```
container_name: database
db_user: postgres
db_name: appdb
```

Question :

- Pourquoi utiliser des fichiers `group_vars` plutot que de mettre les variables directement

dans le playbook ?

3.3 Playbook principal

Fichier : `ansible/playbook.yml`

```
---
- name: Configuration du frontend
  hosts: frontend
  gather_facts: false

  tasks:
    - name: Copier la page HTML du frontend
      community.docker.docker_container_exec:
        container: "{{ container_name }}"
        command: >
          bash -c "echo '<h1>{{ env_label }}</h1>
          <p>Frontend actif sur le port {{ service_port }}</p>'
          > /usr/share/nginx/html/index.html"
      notify: Recharger nginx

  handlers:
    - name: Recharger nginx
      community.docker.docker_container_exec:
        container: "{{ container_name }}"
        command: nginx -s reload

- name: Configuration du backend
  hosts: backend
  gather_facts: false

  tasks:
    - name: Installer Flask
      community.docker.docker_container_exec:
        container: "{{ container_name }}"
        command: pip install flask psycopg2-binary

    - name: Copier l'application Flask
      community.docker.docker_container_exec:
        container: "{{ container_name }}"
        command: >
          bash -c "cat > /app.py << 'EOF'
          from flask import Flask
          app = Flask(__name__)

          @app.route('/')
          def index():
              return 'Backend actif. DB: {{ db_host }}:{{ db_port }}'

          if __name__ == '__main__':
              app.run(host='0.0.0.0', port={{ service_port }})
          EOF"
```

```
    - name: Demarrer Flask
      community.docker.docker_container_exec:
        container: "{{ container_name }}"
        command: bash -c "nohup python /app.py &"

- name: Configuration de la base de donnees
  hosts: database
  gather_facts: false

  tasks:
    - name: Creer la base de donnees applicative
      community.docker.docker_container_exec:
        container: "{{ container_name }}"
        command: >
          psql -U {{ db_user }} -c "CREATE DATABASE {{ db_name }};"
```

Question :

- Le playbook configure le frontend, puis le backend, puis la base de donnees.
- Quel probleme cela pose-t-il dans un contexte reel ?
- Dans quel ordre devrait-on configurer ces services ?

4. Probleme volontaire : ordre de demarrage

Executer le playbook tel quel :

```
cd ansible
ansible-playbook -i inventory.ini playbook.yml
```

La tache de creation de base de donnees peut echouer si PostgreSQL n'est pas encore pret.

Questions :

- Quelle tache echoue en premier ?
- Pourquoi PostgreSQL n'est-il pas immediatement disponible apres le demarrage du conteneur ?
- Comment Ansible permet-il de gerer ce type de dependance temporelle ?

4.1 Correction : attendre que la base soit prete

Modifier le play database dans `ansible/playbook.yml` :

Fichier : `ansible/playbook.yml` (section database uniquement)

```
- name: Configuration de la base de donnees
  hosts: database
  gather_facts: false

  tasks:
    - name: Attendre que PostgreSQL soit pret
      community.docker.docker_container_exec:
        container: "{{ container_name }}"
        command: pg_isready -U postgres
      register: pg_ready
      until: pg_ready.rc == 0
      retries: 10
      delay: 3

    - name: Creer la base de donnees applicative
      community.docker.docker_container_exec:
        container: "{{ container_name }}"
        command: >
          psql -U {{ db_user }} -c "CREATE DATABASE {{ db_name }};"
```

Relancer le playbook et verifier que la tache reussit.

Question :

- Que font les parametres `retries` et `delay` ?
- Que se passe-t-il si PostgreSQL ne repond toujours pas apres 10 tentatives ?

5. Verification

Verifier chaque service :

```
curl http://localhost:8080
curl http://localhost:5000
```

Verifier les conteneurs :

```
docker ps
docker network inspect td3-network
```

6. Handlers : configuration conditionnelle

Un handler ne se declenche que si la tache associee a effectue un changement.

Questions :

- Relancer le playbook une deuxieme fois sans modifier les fichiers.
- Le handler “Recharger nginx” se declenche-t-il ?
- Pourquoi ce comportement est-il utile en production ?

7. Challenge

Ajouter un quatrieme service : un reverse proxy.

Specifications :

- image : nginx:latest
- nom du conteneur : proxy
- port externe : 80
- connecte au reseau td3-network
- configuration nginx : rediriger / vers le frontend (port 8080) et /api vers le backend (port 5000)

Etapes :

- ajouter le conteneur dans Terraform
- ajouter un groupe `proxy` dans l'inventaire
- ajouter un fichier `group_vars/proxy.yml`
- ajouter un play de configuration dans le playbook

Questions :

- Pourquoi un reverse proxy est-il utile dans cette architecture ?
- Que faudrait-il changer si le backend ecoutait sur un port different ?

8. Bonus

Generer l'inventaire Ansible automatiquement depuis les outputs Terraform.

Etapes :

- exporter les outputs Terraform au format JSON

```
terraform output -json > ../ansible/tf_outputs.json
```

- ecrire un script Python ou bash qui lit ce fichier et genere `inventory.ini`
- l'inventaire genere doit contenir les memes groupes et variables que celui ecrit a la main

Question :

- Quelle est la difference entre un inventaire statique et un inventaire dynamique ?
- Dans quel cas l'inventaire dynamique devient-il indispensable ?