🎓 Travaux Pratiques : Infrastructure as Code (IaC) et Pipelines CI/CD

🚀 Présentation du Sujet : Du Code à l'Infrastructure Automatisée

Ce TP vise à solidifier la compréhension de l'**Infrastructure as Code (IaC)** en utilisant **Terraform** pour orchestrer des conteneurs **Docker** (simulant des services cloud) et en intégrant le tout dans un **pipeline CI/CD**.

L'objectif principal est de maîtriser le **Cycle de Vie du Déploiement (DLC)** complet et automatisé, sans nécessiter de crédits sur des plateformes cloud majeures (Azure, AWS).

Fiche Technique	Détails
Public Cible	Ingénieur Génie Logiciel / DevOps
Outils Clés	Docker, Terraform (CLI), Git, GitHub Actions/GitLab CI
Durée Estimée	6 heures
Dépendance Cloud	**Aucune (Infrastructure Locale Simulé)**

💻 Prérequis Techniques

Pour réaliser ce TP, vous devez avoir installé et configuré les outils suivants sur votre machine :

1. **Docker Desktop** (ou Docker Engine)
2. **Terraform CLI** (version ≥ 1.0)
3. **Git**
4. Un compte **GitHub** ou **GitLab** pour la partie CI/CD.

Partie I : L'Infrastructure as Code (IaC) Locale

L'objectif est de définir l'infrastructure cible — composée d'une base de données PostgreSQL et d'une application web simple — sous forme de code Terraform.

1. Structure du Projet

Créez le répertoire et la structure de fichiers initiale.

tp-iac-local/

├── main.tf # Définition des ressources et du provider Docker

├── variables.tf # Paramètres configurables (ports, credentials)

├── outputs.tf # Informations de sortie après déploiement

├── Dockerfile_app # Blueprint de l'image de l'application web

└── .gitignore

2. Conteneurisation de l'Application (Dockerfile_app)

Le fichier Dockerfile_app doit construire une image légère (ex : basée sur Alpine) contenant un serveur web minimal (Nginx). L'objectif est de prouver que l'application est bien accessible après le déploiement IaC.

Consigne : Créez le Dockerfile_app pour :

1. Utiliser une image de base alpine:latest.
2. Installer Nginx.
3. Créer un fichier index.html affichant : "**Application Deployed via Terraform IaC!**".
4. Exposer le port **80**.

3. Définition Terraform (main.tf, variables.tf, outputs.tf)

Rédigez le code Terraform pour orchestrer les deux conteneurs :

- **Provider** : Déclarez le provider **Docker** (kreuzwerker/docker).
- **Base de Données** : Définissez les ressources pour l'image et le conteneur **PostgreSQL** (postgres:latest), en exposant le port **5432** et en configurant les variables d'environnement (utilisateur, mot de passe, nom de la DB) via variables.tf.
- **Application Web** : Définissez les ressources pour construire l'image à partir du Dockerfile_app local, puis créez le conteneur, en mappant le port interne **80** au port externe **8080**.

4. Application du Cycle de Vie du Déploiement (DLC)

Exécutez les étapes du DLC dans l'ordre, en documentant les résultats :

1. **Initialisation** : terraform init (Téléchargement des providers)
2. **Planification** : terraform plan (Simulation du déploiement)
3. **Application** : terraform apply -auto-approve (Création des conteneurs)
4. **Validation** : Vérifiez l'accès à l'application via http://localhost:8080/.
5. **Destruction** : terraform destroy -auto-approve (Nettoyage complet)

Partie II : L'Automatisation DevOps (Pipeline CI/CD)

L'objectif est d'intégrer les étapes de Terraform dans un pipeline afin d'automatiser le déploiement de l'infrastructure dès qu'une modification est poussée sur la branche principale.

1. Configuration du Pipeline

1. Poussez votre projet sur un dépôt Git (GitHub ou GitLab).
2. Créez le fichier de configuration de pipeline (ex : .github/workflows/main.yml pour GitHub Actions).

2. Étapes du Pipeline et Rôle DLC

Votre pipeline doit comporter les étapes suivantes, s'exécutant automatiquement sur chaque *push* vers main :

Étape Pipeline	Commande Exécutée	Rôle dans le DLC
Setup Terraform	Configuration de l'environnement	Préparation
Init	terraform init	Initialisation de l'état
Plan	terraform plan	**Validation & Revue** (Détection des changements)
Apply	terraform apply -auto-approve	**Déploiement Continu** (Application des changements)

Attention : Le Apply ne doit s'exécuter que si le Plan précédent a réussi, garantissant l'intégrité du processus.

3. Validation de l'Automatisation

1. Modifiez une variable dans variables.tf (ex : le mot de passe de la DB).
2. Commitez et poussez la modification.
3. Observez le pipeline s'exécuter jusqu'à l'étape **Apply** et confirmez que la ressource est mise à jour.

📝 Évaluation et Questions de Révision

Critères de Notation

Votre travail sera évalué sur :

Critère	Description	Pondération
1. Maîtrise IaC	Clarté et exactitude des fichiers .tf et du Dockerfile_app.	40%
2. Exécution DLC	Documentation des 5 étapes (Init $\rightarrow$ Destroy) et validation.	30%
3. Pipeline CI/CD	Fichier de pipeline correct, exécution réussie et compréhension des étapes.	30%

Questions d'Approfondissement (À joindre au rapport)

1. **Définition d'État :** Quel est l'impact réel de la commande terraform destroy sur le fichier **terraform.tfstate** ? Expliquez son rôle dans la réconciliation entre l'infrastructure réelle et le code.
2. **Immuabilité :** Dans le contexte de l'IaC, expliquez ce que signifie l'**Immutabilité de l'Infrastructure**.

3. **Planification :** Pourquoi l'étape Plan est-elle considérée comme la porte d'entrée (ou *gate*) essentielle du DLC dans un contexte de production avant l'étape Apply ?
4. **Alternative :** Si vous deviez créer un **réseau Docker** et y connecter vos deux conteneurs, quelle nouvelle ressource Terraform devriez-vous utiliser ?

Eléments de Réponse

🗝️ Solution du TP : Fichiers IaC et Docker

1. Dockerfile_app

Ce fichier est le blueprint pour construire l'image de l'application web. Il utilise Nginx sur Alpine pour la légèreté.

Dockerfile

```
# Utilise une image de base très légère (Alpine)
FROM alpine:latest

# Mise à jour et installation du serveur web Nginx
RUN apk update && \
    apk add nginx && \
    rm -rf /var/cache/apk/*

# Création d'une page HTML simple comme preuve de déploiement
RUN echo "<h1>Application Deployed via Terraform IaC!</h1>" > /var/www/localhost/index.html

# Expose le port par défaut de Nginx
EXPOSE 80

# Commande pour démarrer Nginx en mode non-démonisé
CMD ["nginx", "-g", "daemon off;"]
```

---

2. variables.tf

Ce fichier contient les paramètres configurables de l'infrastructure, rendant le code plus flexible et prêt pour la Partie II (injection par pipeline).

Terraform

```
# --- Variables de Base de Données (PostgreSQL) ---
variable "db_name" {
  description = "Nom de la base de données PostgreSQL."
  type        = string
  default     = "devops_db"
}

variable "db_user" {
  description = "Nom d'utilisateur PostgreSQL."
  type        = string
  default     = "devops_user"
}

variable "db_password" {
  description = "Mot de passe PostgreSQL (ATTENTION: Simulé, ne pas utiliser en Prod !)"
  type        = string
  default     = "strongpassword123"
}

# --- Variables d'Application ---
variable "app_port_external" {
  description = "Port externe pour accéder à l'application web (mappé au 80 interne)."
  type        = number
  default     = 8080
}
```

---

3. main.tf

C'est le fichier central qui définit les ressources (*resources*) de l'infrastructure en utilisant le **Provider Docker**.

Terraform

```
# --- Configuration Terraform et Provider ---
```

```
terraform {
  required_providers {
    docker = {
      source  = "kreuzwerker/docker"
      version = "~> 3.0.1" # Spécifie le provider Docker
    }
  }
}

provider "docker" {}

# --- 1. Ressource : Base de Données PostgreSQL ---

# Télécharge l'image PostgreSQL depuis Docker Hub
resource "docker_image" "postgres_image" {
  name         = "postgres:latest"
  keep_locally = true
}

# Crée et configure le conteneur PostgreSQL
resource "docker_container" "db_container" {
  name  = "tp-db-postgres"
  image = docker_image.postgres_image.latest

  ports {
    internal = 5432
    external = 5432 # Mappage au port par défaut 5432
  }

  # Configuration de la DB via les variables d'environnement
  env = [
    "POSTGRES_USER=${var.db_user}",
```

```
    "POSTGRES_PASSWORD=${var.db_password}",
    "POSTGRES_DB=${var.db_name}",
  ]
}

# --- 2. Ressource : Application Web Nginx ---

# Construit l'image de l'application à partir du Dockerfile_app local
resource "docker_image" "app_image" {
  name = "tp-web-app:latest"
  build {
    context    = "."          # Répertoire courant
    dockerfile = "Dockerfile_app"
  }
}

# Crée le conteneur de l'application web
resource "docker_container" "app_container" {
  name  = "tp-app-web"
  image = docker_image.app_image.latest

  # Dépendance explicite : la DB doit être prête avant l'Application
  depends_on = [
    docker_container.db_container
  ]

  # Mappage du port 80 interne au port externe défini dans variables.tf (par défaut 8080)
  ports {
    internal = 80
    external = var.app_port_external
  }
}
```

---

4. outputs.tf

Ce fichier affiche les informations clés après l'exécution de terraform apply, permettant aux étudiants de valider rapidement leur déploiement.

Terraform

```
output "db_container_name" {
  description = "Nom du conteneur de la base de données."
  value       = docker_container.db_container.name
}


output "app_access_url" {
  description = "URL d'accès à l'application web."
  value       = "http://localhost:${docker_container.app_container.ports[0].external}"
}
```

Ces fichiers, combinés à la mise en place du pipeline CI/CD (Partie II), offrent une solution complète et pédagogique aux objectifs du TP.

Souhaitez-vous les éléments de solution pour le pipeline CI/CD (Partie II - le fichier .github/workflows/main.yml) ?