Terraform

Terraform

Les concepts Terraform

- **Providers**:
  Accès aux plateformes cloud (AWS, Azure, Docker…)
- **Resources** :
  Composants de l'infrastructure à provisionner (VMs, conteneurs, réseaux…)
- **State** :
  L'état connu de l'infrastructure

Le cycle de vie Terraform

- init — initialisation du projet
- plan — prévisualisation des changements
- apply — application des modifications
- destroy — suppression des ressources

Exemple

```
# main.tf

# Configuration du provider Docker
provider "docker" {  host = "unix:///var/run/docker.sock"}

# Déclaration d'une ressource conteneur
resource "docker_container" "mon_nginx" {
  name  = "mon-serveur-web“
  image = "nginx:latest“
  ports {
      internal = 80
      external = 8080
  }
}
```

Ligne de commande :

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

Exemple de ressources locales

- **local_file** : Crée des fichiers avec contenu dynamique via variables.

- **random_pet** : Génère des noms uniques automatiquement.

Variables et output

Variables

Elles permettent de paramétrer vos configurations, rendant votre code flexible et réutilisable.

Définissez des valeurs dynamiquement (ex: régions, noms d'environnement).

Par convention, on les définit dans un fichier **variables.tf**

Exemple

variables.tf

```
variable "region" {
  description = "Région AWS"
  type        = string
  default     = "eu-west-1"
}
variable "instance_type" {
  description = "Type d'instance EC2"
  type        = string
  default     = "t2.micro"
}
variable "instance_name" {
  description = "Nom de la machine"
  type        = string
  default     = "demo-ec2"
}
variable "ami_id" {
  description = "AMI à utiliser"
  type        = string
  default     = "ami-1234567890abcdef0"  # Exemple fictif
}
variable "subnet_id" {
  description = "Subnet où lancer l'instance"
  type        = string
  default     = "subnet-1234567890abcdef0" # Exemple fictif
}
variable "security_group_ids" {
  description = "Liste des security groups"
  type        = list(string)
  default     = ["sg-1234567890abcdef0"]  # Exemple fictif
}
```

main.tf

```
terraform {
  required_providers {
    aws = {
      source = "hashicorp/aws"
    }
  }
}

provider "aws" {
  region = var.region
}

resource "aws_instance" "vm" {
  ami                    = var.ami_id
  instance_type          = var.instance_type
  subnet_id              = var.subnet_id
  vpc_security_group_ids = var.security_group_ids

  tags = {
    Name = var.instance_name
  }
}
```

Positionnement des valeurs des variables

- **Valeur par défaut**

Définie dans la déclaration de la variable

- **Ligne de commande**

```
terraform apply -var="instance_type=t3.large"
```

- **Fichier tfvars** :

Fichier `terraform.tfvars` chargé automatiquement, ou différent passé sur la ligne de commande :

```
terraform apply -var-file="dev.tfvars"
```

Output

Récupérez des informations clés de votre infrastructure déployée. Les outputs exposent les attributs des ressources (ex: adresses IP, URL) pour une utilisation ultérieure.

```
resource "random_pet" "example" {
  length = 2
}

output "pet_name" {
  description = "Le nom généré"
  value       = random_pet.example.id
}
```

State

Introduction au tfstate

Fichier généré automatiquement par Terraform
Contient **l'état réel** de l'infrastructure connue par Terraform
Sert de **source de vérité** pour comparer :

- ce qui existe sur les plateformes
- ce qui est décrit dans les fichiers .tf

Nom par défaut : **terraform.tfstate**

Terraform doit savoir :

- quelles ressources existent déjà
- quels attributs elles ont (ID, IP, relations, metadata…)
- ce qu'il doit créer, modifier ou supprimer

Sans tfstate, Terraform devrait interroger toutes les ressources chaque fois = impossible ou trop lent.

Relation entre configuration et tfstate

Terraform compare **desired** vs **known** vs **actual** pour déterminer les actions nécessaires.

Fichiers .tf (desired state)	↔	tfstate (known state)	↔	Ressources réelles (actual state)

Cycle de vie du tfstate

1. `terraform init`
    - Prépare l'environnement, pas de changement au state
2. `terraform plan`
    - Compare fichiers .tf avec le state
    - Ne modifie **pas** le state
3. `terraform apply`
    - Crée/Modifie/Supprime
    - Met **à jour** le state après chaque action
4. `terraform destroy`
    - Supprime les ressources
    - Met le state à jour (ou le supprime si vide)

Opérations avancées (terraform state)

Terraform propose des commandes pour gérer manuellement le state :

```
terraform state list
terraform state show <ressource>
terraform state mv <src> <dst>
terraform state rm <ressource>
```

Utilisation :

- refactoriser votre code
- renommer une ressource sans destruction
- forcer la suppression d’une entrée incohérente du state

Où stocker le tfstate

Local (par défaut)

- `terraform.tfstate` dans le dossier courant
- Simple mais pas adapté au travail à plusieurs

Remote backend

- Amazon S3
- Terraform Cloud
- Azure Storage
- Google Cloud Storage
- Consul
- etc.

Avantages :

- verrouillage (lock)
- partage entre équipes
- versioning automatique
- sécurité

Sensibilité et sécurité du tfstate

Le tfstate peut contenir :

- mots de passe
- secrets
- IP
- identifiants cloud
- données personnelles

Recommandations :

- ne jamais le mettre dans Git
- utiliser le chiffrement (S3, GCS, Terraform Cloud)
- gérer les permissions avec soin

Docker avec Terraform

Provisionner une infrastructure Docker avec Terraform

Terraform peut gérer **Docker comme un provider**

Permet de créer :

- images
- containers
- réseaux
- volumes

Idéal pour des TP sans compte cloud : tout tourne en local

Image Docker

Image :

- Terraform télécharge l'image s'il le faut
- Peut aussi construire depuis un Dockerfile

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

Container :

- Expose le port 8080 sur la machine locale
- Démarre automatiquement après l'application du plan

```
resource "docker_container" "web" {
  name  = "webserver"
  image = docker_image.nginx.image_id

  ports {
    internal = 80
    external = 8080
  }
}
```

Gestion des réseaux

Réseau:

- Créer le réseau d'abord

```
resource "docker_network" "demo" {
  name = "demo-network"
}
```

Utilisation:

- Affecter un container au réseau nouvellement créé

```
resource "docker_container" "web" {
  name  = "webserver"
  image = docker_image.nginx.image_id
  networks_advanced {
    name = docker_network.demo.name
  }
}
```