TD2 : Bases du provisionning

Terraform et Modularisation

- Comprendre l'intérêt des variables et des outputs
- Structurer un projet Terraform
- Créer et utiliser un module
- Manipuler Terraform de manière autonome

Contexte

Vous travaillez dans une équipe DevOps.

Votre équipe doit déployer plusieurs environnements de test pour différentes équipes :

- frontend
- backend
- data

Chaque équipe a besoin :

- d'un conteneur web
- accessible via un port différent

Problème actuel :

- duplication du code Terraform
- erreurs fréquentes (ports, noms…)
- difficile à maintenir

Objectif :

Créer une solution réutilisable permettant de déployer plusieurs conteneurs rapidement et sans erreur.

1. Structure du projet

```
mkdir -p terraform-td2/modules/nginx
cd terraform-td2
```

```
terraform-td2/
  main.tf
  modules/
    nginx/
      main.tf
      variables.tf
      outputs.tf
```

2. Création du module

Objectif : créer un composant réutilisable

2.1 Variables du module

Fichier : `modules/nginx/variables.tf`

```
variable "container_name" {
  description = "Nom du conteneur"
}

variable "external_port" {
  description = "Port exposé"
}
```

Les variables permettent de rendre un module réutilisable.

Sans variables :

- les valeurs seraient figées
- le module ne pourrait être utilisé que dans un seul contexte
- toute modification nécessiterait de dupliquer ou modifier le code

Avec des variables :

- le module devient générique
- les différences sont externalisées

2.2 Ressources

Fichier : `modules/nginx/main.tf`

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

resource "docker_container" "nginx" {
  name  = var.container_name
  image = docker_image.nginx.name

  ports {
    internal = 80
    external = var.external_port
  }
}
```

2.3 Outputs du module

Fichier : `modules/nginx/outputs.tf`

```
output "url" {
  value = "http://localhost:${var.external_port}"
}
```

Les outputs permettent d'exposer des informations produites par Terraform.

Ils sont souvent utilisés pour :

- chaîner plusieurs modules
- transmettre des informations à d'autres outils (CI/CD, Ansible, scripts)
- éviter de recalculer ou rechercher des valeurs déjà connues

3. Utilisation du module

Fichier : `main.tf`

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

provider "docker" {}

module "nginx1" {
  source = "./modules/nginx"

  container_name = "frontend"
  external_port  = 8080
}

module "nginx2" {
  source = "./modules/nginx"

  container_name = "backend"
  external_port  = 8081
}
```

4. Exécution

Dans le dossier `terraform-td2` :

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

Vérifier :

- http://localhost:8080
- http://localhost:8081

5. Problème volontaire

Fichier : `main.tf`

Modifier la configuration pour utiliser le même port :

```
external_port = 8080
```

Relancer :

```
terraform apply
```

Questions :

- Décrire précisément ce que Terraform tente de faire lors de l'apply
- Pourquoi cette opération échoue-t-elle au niveau du provider Docker ?
- Terraform aurait-il pu détecter ce problème lors du plan ? Pourquoi ?

6. Exploitation des outputs

```
terraform output
```

Question :

- Donner un exemple concret d'utilisation de cet output dans une pipeline CI/CD (tests, déploiement, monitoring…)

7. Compréhension

Un module Terraform fonctionne comme une fonction :

- il prend des entrées (variables)
- il produit des ressources
- il expose des sorties (outputs)

Le fichier `terraform.tfstate` contient l'état réel de l'infrastructure.

Terraform compare :

- la configuration (code)
- le state

pour déterminer les actions à effectuer (création, modification, suppression).

8. Extension

Fichier : `modules/nginx/variables.tf`

Rendre le module plus flexible :

```
variable "image_name" {
  description = "Image Docker"
  default     = "nginx:latest"
}
```

Fichier : `modules/nginx/main.tf`

Modifier :

```
resource "docker_image" "nginx" {
  name = var.image_name
}
```

Test :

- utiliser nginx:alpine

9. Challenge

Créer un troisième environnement :

- nom : data
- port : 8082

10. Bonus

Afficher toutes les URLs sous forme de map :

```
output "urls" {
  value = {
    nginx1 = module.nginx1.url
    nginx2 = module.nginx2.url
  }
}
```

Une map permet d'associer explicitement une valeur à une clé (ex : environnement → URL).

Avantages :

- accès direct par nom
- plus lisible
- moins d'erreurs qu'une liste indexée

Une liste ne garantit pas le lien entre la position et la signification.

11. Passage à l'échelle avec map et for_each

Objectif

- Éviter la duplication de code
- Générer dynamiquement plusieurs ressources
- Manipuler des structures de données (map)

Contexte

Jusqu'à présent, vous avez défini plusieurs modules manuellement :

- nginx1
- nginx2
- nginx3

Ce fonctionnement pose un problème :

- le code est dupliqué
- difficile à maintenir
- peu scalable (que faire avec 10 ou 20 environnements ?)

Objectif :

Factoriser la configuration pour décrire les environnements sous forme de données, et laisser Terraform générer les ressources.

11.1 Définir les environnements

Fichier : `main.tf`

```
locals {
  environments = {
    frontend = {
      port = 8080
    }
    backend = {
      port = 8081
    }
    data = {
      port = 8082
    }
  }
}
```

Cette structure est une map d'objets :

- clé = nom de l'environnement
- valeur = configuration associée

Elle permet de décrire l'infrastructure sous forme de données, plutôt que de multiplier les blocs Terraform.

11.2 Générer les modules automatiquement

Supprimer les blocs :

```
module "nginx1" { ... }
module "nginx2" { ... }
```

Remplacer par :

```
module "nginx" {
  for_each = local.environments

  source = "./modules/nginx"
```

```
  container_name = each.key
  external_port  = each.value.port
}
```

`for_each` permet de créer plusieurs instances d'un même bloc à partir d'une collection.

Pour chaque élément :

- `each.key` correspond au nom (ex : frontend)
- `each.value` correspond à la configuration associée

Terraform crée une instance par entrée dans la map.

Question :

- Pourquoi for_each est-il plus adapté que count dans ce cas ?

11.3 Adapter les outputs

Fichier : `main.tf`

```
output "urls" {
  value = {
    for env, mod in module.nginx :
    env => mod.url
  }
}
```

Cet output reconstruit une map en reprenant :

- les clés des environnements
- les URLs produites par chaque module

Cela permet de conserver une structure cohérente entre les entrées (environments) et les sorties (urls).

11.4 Vérification

```
terraform apply
terraform output
```

Vérifier :

- http://localhost:8080
- http://localhost:8081
- http://localhost:8082

11.5 Analyse

- Si vous supprimez un environnement de la map, que va faire Terraform ?
- Que se passe-t-il si vous renommez une clé ?
- Quel impact sur le state ?

11.6 Limites

Question :

- Votre équipe utilise cette approche avec `map + for_each`.

Un développeur renomme la clé `frontend` en `front`.

- Que va faire Terraform lors du prochain apply ?
- Pourquoi ce comportement peut-il être dangereux en production ?
- Comment éviter ce problème dans un projet réel ?

12. Extension — Terraform → Ansible

Objectifs

- Réutiliser les outputs Terraform
- Introduire Ansible comme outil de configuration
- Comprendre l'intégration entre outils
- Comparer deux approches d'architecture

12.1 Contexte

Vous avez déployé plusieurs conteneurs nginx avec Terraform.

Chaque conteneur est accessible via une URL différente :

- frontend → http://localhost:8080
- backend → http://localhost:8081
- data → http://localhost:8082

Problème :

- toutes les pages nginx sont identiques
- impossible d'identifier facilement l'environnement

Objectif :

Configurer dynamiquement chaque conteneur avec Ansible.

12.2 Mise en place d'Ansible

Créer un dossier :

```
mkdir ansible
cd ansible
```

Créer un fichier `playbook.yml` :

```
- name: Configuration des conteneurs nginx
  hosts: all
  gather_facts: false

  tasks:
    - name: Copier une page HTML personnalisée
      copy:
        content: |
          <h1>Environnement : {{ inventory_hostname }}</h1>
          <p>URL : http://localhost:{{ ansible_port }}</p>
        dest: /usr/share/nginx/html/index.html
```

Chaque conteneur aura une page différente en fonction de son nom.

Vous utilisez ici les variables Ansible :

- inventory_hostname → nom de l'hôte
- ansible_port → port utilisé

12.3 Problème à résoudre

Ansible a besoin d'un **inventory** pour savoir :

- quelles machines cibler
- comment s'y connecter

Or, ces informations sont connues par Terraform.

Comment faire le lien entre les deux ?

12.4 Approche A — Génération via Terraform

Modifier Terraform pour générer un fichier `inventory.ini`.

Indice :

- utiliser `templatefile`
- utiliser `local_file`

Exemple attendu :

```
[web]
frontend ansible_host=localhost ansible_port=8080
backend ansible_host=localhost ansible_port=8081
data ansible_host=localhost ansible_port=8082
```

12.5 Approche B — Approche découplée

Utiliser les outputs Terraform pour générer l'inventory avec un script externe.

```
terraform output -json > outputs.json
```

Transformer ce fichier en `inventory.ini`.

Langage au choix :

- bash
- python
- jq

12.6 Exécution

Lancer Ansible :

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

Vérifier :

- chaque URL affiche un contenu différent

12.7 Analyse

Questions :

- Quelle approche est la plus simple à mettre en place ?
- Quelle approche est la plus maintenable ?
- Que se passe-t-il si vous ajoutez un nouvel environnement ?
- Quelle solution limite le couplage entre Terraform et Ansible ?
- Dans un projet réel, laquelle choisiriez-vous ? Pourquoi ?

12.8 Bonus

- Ajouter une variable Terraform `env_type` (dev, prod…)
- L'afficher dans la page HTML via Ansible

Objectif :

Comprendre comment une information définie dans Terraform peut être utilisée dans Ansible.

C'est un cas réel fréquent en DevOps.

13. Extension : Alternative avec Pulumi (approche orientée développeur)

Objectifs

- Découvrir une alternative à Terraform
- Comprendre la différence déclaratif vs programmation
- Réécrire une logique Terraform avec un langage classique
- Identifier avantages et limites

13.1 Principe

Contrairement à Terraform :

- Terraform → langage déclaratif (HCL)
- Pulumi → langage de programmation (TypeScript ici)

Un module Terraform devient :

- une fonction
- ou une classe

Un `for_each` devient :

- une boucle classique

13.2 Initialisation du projet

```
mkdir pulumi-td2
cd pulumi-td2

pulumi new typescript
```

Choisir :

- project name : pulumi-td2
- stack : dev

Installer le provider Docker :

```
npm install @pulumi/docker
```

13.3 Création d'un composant réutilisable

Créer un fichier `nginx.ts` :

```
import * as docker from "@pulumi/docker";

export function createNginx(name: string, port: number) {

  const image = new docker.RemoteImage(name, {
    name: "nginx:latest",
  });

  const container = new docker.Container(name, {
    image: image.imageId,
    ports: [
      {
        internal: 80,
        external: port,
      },
    ],
  });

  return {
    url: `http://localhost:${port}`,
  };
}
```

En Pulumi, on remplace les modules Terraform par du code réutilisable.

13.4 Utilisation (équivalent main.tf)

Modifier `index.ts` :

```
import { createNginx } from "./nginx";

const frontend = createNginx("frontend", 8080);
const backend = createNginx("backend", 8081);

export const frontendUrl = frontend.url;
```

```
export const backendUrl = backend.url;
```

13.5 Exécution

```
pulumi up
```

13.6 Version dynamique

Equivalent du `for_each` Terraform :

```
import { createNginx } from "./nginx";

const environments: Record<string, number> = {
  frontend: 8080,
  backend: 8081,
  data: 8082,
};

const urls: Record<string, string> = {};

for (const [name, port] of Object.entries(environments)) {
  const nginx = createNginx(name, port);
  urls[name] = nginx.url;
}

export { urls };
```

13.7 Validation des erreurs

```
const ports = Object.values(environments);

if (new Set(ports).size !== ports.length) {
  throw new Error("Ports dupliqués détectés !");
}
```

Contrairement à Terraform, Pulumi permet d’ajouter des validations personnalisées avant même le déploiement.

13.8 Comparaison

Questions :

- Qu’est-ce qui est plus simple à écrire ?

- Qu’est-ce qui est plus lisible ?
- Où Pulumi est-il plus puissant ?
- Où Terraform est-il plus prévisible ?

13.9 Analyse

Pulumi permet :

- d’utiliser des structures natives (boucles, fonctions)
- de factoriser facilement le code
- d’ajouter des validations complexes

Mais :

- nécessite des compétences en développement
- peut introduire de la complexité inutile

13.10 Conclusion

Terraform et Pulumi répondent au même besoin avec deux approches :

- Terraform → simplicité, standardisation, lisibilité
- Pulumi → flexibilité, puissance, approche développeur

Dans un projet réel :

- Terraform est souvent utilisé côté Ops
- Pulumi est pertinent pour des équipes orientées développement