TD2 : Créer un module Terraform réutilisable

Objectifs

- 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 Comparaison

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 ?