Guide Complet

Terraform

Les Fondamentaux de
l'Infrastructure as Code

Terraform Hands-On

Contenu du guide :

- Introduction et Installation
- Concepts Fondamentaux
- Variables, Modules et Fonctions
- Gestion du State
- Sécurité et Bonnes Pratiques
- CI/CD et Automatisation
- Exemples Complets

Basé sur :

HashiCorp Certified Terraform
Associate 2025

Version 1.0 - By Mohammed DAHIRY

Table des matières

0.1 Introduction à Terraform

0.1.1 Qu'est-ce que Terraform ?

Terraform est un outil d'Infrastructure as Code (IaC) développé par HashiCorp qui permet de définir, provisionner et gérer l'infrastructure cloud de manière déclarative.

Définition **Infrastructure as Code (IaC)** : Approche de gestion et de provisionnement de l'infrastructure via du code plutôt que par des processus manuels.

0.1.2 Pourquoi utiliser Terraform ?

— **Automatisation** : Réduit les erreurs humaines et accélère le déploiement
— **Réutilisabilité** : Créez du code réutilisable pour des déploiements identiques
— **Multi-cloud** : Support de milliers de providers (AWS, Azure, GCP, etc.)
— **Versioning** : Contrôle de version pour votre infrastructure
— **Collaboration** : Travail d'équipe facilité

0.1.3 Cas d'usage réel

Imaginons que vous devez implémenter 250+ pages de règles de sécurité AWS dans 99 comptes différents. Manuellement, cela prendrait des semaines. Avec Terraform, vous écrivez le code une fois et le déployez partout en quelques minutes.

0.2 Installation et Configuration

0.2.1 Installation de Terraform

Terraform est distribué sous forme d'un binaire unique. L'installation est simple :

1. Téléchargez le binaire depuis https ://www.terraform.io/downloads
2. Décompressez l'archive
3. Placez le binaire dans votre PATH système

Plateformes supportées Windows, macOS, Linux, FreeBSD, OpenBSD, Solaris

0.2.2 Vérification de l'installation

```
terraform version
# Output: Terraform v1.x.x
```

Listing 1 – Vérifier la version de Terraform

0.2.3 Configuration de l'éditeur

Il est recommandé d'utiliser Visual Studio Code avec l'extension HashiCorp Terraform pour :

- Coloration syntaxique
- Autocomplétion
- Validation en temps réel
- Formatage automatique

0.3 Concepts Fondamentaux

0.3.1 Providers

Les providers sont des plugins qui permettent à Terraform d'interagir avec des APIs externes (cloud providers, services SaaS, etc.).

Exemple : Provider AWS

```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 5.0"
    }
  }
}

provider "aws" {
  region = "eu-west-1"
}
```

Types de Providers

- **Official** : Maintenus par HashiCorp
- **Partner** : Maintenus par des partenaires HashiCorp
- **Community** : Maintenus par la communauté

0.3.2 Resources

Les resources sont les composants principaux de Terraform. Elles décrivent un ou plusieurs objets d'infrastructure.

Exemple : Créer une instance EC2

```
resource "aws_instance" "web_server" {
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"

  tags = {
    Name = "MonServeurWeb"
    Environment = "Production"
```

```
  }
}
```

Anatomie d'une resource :

— `resource` : Mot-clé

— `"aws_instance"` : Type de resource

— `"web_server"` : Nom local de la resource

— Bloc {} : Arguments et configuration

0.3.3 State File (Fichier d'état)

Le fichier d'état (`terraform.tfstate`) est crucial : il stocke l'état actuel de votre infrastructure.

Important Le fichier d'état peut contenir des informations sensibles. Ne jamais le commiter dans un dépôt public !

Contenu du state file :

— Mapping entre les resources Terraform et les objets réels

— Métadonnées sur les resources

— Dépendances entre resources

0.4 Workflow Terraform de Base

0.4.1 Les 4 commandes essentielles

1. **terraform init** : Initialise le répertoire de travail
2. **terraform plan** : Prévisualise les changements
3. **terraform apply** : Applique les changements
4. **terraform destroy** : Détruit l'infrastructure

0.4.2 Exemple complet : Déployer une VM

Étape 1 : Créer le fichier main.tf

```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 5.0"
    }
  }
}

provider "aws" {
  region = "eu-west-1"
}

resource "aws_instance" "exemple" {
```

```
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"

  tags = {
    Name = "MonPremierServeur"
  }
}
```

Étape 2 : Commandes à exécuter

```
# Initialisation
terraform init

# Planification
terraform plan

# Application
terraform apply

# Destruction (quand n cessaire)
terraform destroy
```

0.4.3 Comprendre terraform plan

`terraform plan` compare l'état désiré (votre code) avec l'état actuel (state file + infrastructure réelle) et affiche :

- \+ Resources à créer
- ~ Resources à modifier
- \- Resources à détruire

0.5 Variables et Outputs

0.5.1 Variables d'entrée

Les variables permettent de paramétrer votre code Terraform.
Définition de variables

```
# variables.tf
variable "instance_type" {
  description = "Type d'instance EC2"
  type        = string
  default     = "t2.micro"
}

variable "environment" {
  description = "Environnement de deploiement"
  type        = string
}

variable "ports" {
  description = "Liste des ports a ouvrir"
```

```
  type        = list(number)
  default     = [80, 443]
}
```

0.5.2 Types de données

- **string** : Chaîne de caractères (`"hello"`)
- **number** : Nombre (`42`)
- **bool** : Booléen (`true` ou `false`)
- **list** : Liste ordonnée (`["a", "b", "c"]`)
- **map** : Collection clé-valeur (`{key = "value"}`)
- **set** : Collection de valeurs uniques

0.5.3 Utilisation des variables

Référencer une variable

```
resource "aws_instance" "app" {
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = var.instance_type

  tags = {
    Environment = var.environment
  }
}
```

0.5.4 Fichiers tfvars

Les fichiers `.tfvars` permettent de définir les valeurs des variables.
terraform.tfvars

```
instance_type = "t2.large"
environment   = "production"
ports         = [80, 443, 8080]
```

0.5.5 Priorité des variables

Ordre de priorité (du plus faible au plus fort) :

1. Valeur par défaut dans `variables.tf`
2. Variables d'environnement `TF_VAR_*`
3. Fichier `terraform.tfvars`
4. Fichiers `*.auto.tfvars`
5. Option `-var` en ligne de commande

0.5.6 Output Values

Les outputs permettent d'afficher des informations après l'application.

Définir des outputs

```
output "instance_ip" {
  description = "Adresse IP publique de l'instance"
  value       = aws_instance.app.public_ip
}

output "instance_id" {
  description = "ID de l'instance"
  value       = aws_instance.app.id
}
```

0.6 Data Sources

Les data sources permettent de récupérer des informations depuis l'infrastructure existante ou des sources externes.

Récupérer la dernière AMI Ubuntu

```
data "aws_ami" "ubuntu" {
  most_recent = true
  owners      = ["099720109477"] # Canonical

  filter {
    name   = "name"
    values = ["ubuntu/images/hvm-ssd/ubuntu-*-22.04-*"]
  }
}

resource "aws_instance" "web" {
  ami           = data.aws_ami.ubuntu.id
  instance_type = "t2.micro"
}
```

Différence Resource vs Data Source

- **Resource** : Crée/modifie/détruit des objets
- **Data Source** : Lit des informations existantes

0.7 Meta-Arguments

Les meta-arguments modifient le comportement des resources.

0.7.1 count

Crée plusieurs instances d'une resource.

Créer 3 serveurs

```
resource "aws_instance" "server" {
  count         = 3
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"

  tags = {
    Name = "Server-${count.index}"
  }
}
```

0.7.2 for_each

Itère sur une map ou un set pour créer des resources.
Créer des utilisateurs IAM

```
variable "users" {
  type    = set(string)
  default = ["alice", "bob", "charlie"]
}

resource "aws_iam_user" "users" {
  for_each = var.users
  name     = each.key
}
```

0.7.3 depends_on

Définit des dépendances explicites entre resources.
Dépendance explicite

```
resource "aws_s3_bucket" "data" {
  bucket = "mon-bucket-data"
}

resource "aws_instance" "app" {
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"

  depends_on = [aws_s3_bucket.data]
}
```

0.7.4 lifecycle

Contrôle le cycle de vie des resources.
Arguments lifecycle

```
resource "aws_instance" "app" {
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"

  lifecycle {
```

```
    # Creer avant de detruire
    create_before_destroy = true

    # Empecher la destruction
    prevent_destroy = false

    # Ignorer les changements sur certains attributs
    ignore_changes = [tags]
  }
}
```

0.8 Modules

Les modules permettent de réutiliser du code Terraform.

0.8.1 Structure d'un module

```
modules/
 ec2-instance/
     main.tf
     variables.tf
     outputs.tf
     README.md
```

0.8.2 Créer un module simple

modules/ec2-instance/main.tf

```
resource "aws_instance" "this" {
  ami           = var.ami_id
  instance_type = var.instance_type

  tags = var.tags
}
```

modules/ec2-instance/variables.tf

```
variable "ami_id" {
  description = "AMI ID"
  type        = string
}

variable "instance_type" {
  description = "Type d'instance"
  type        = string
  default     = "t2.micro"
}

variable "tags" {
  description = "Tags"
  type        = map(string)
  default     = {}
```

```
}
```

0.8.3 Utiliser un module

Appel du module

```
module "web_server" {
  source = "./modules/ec2-instance"

  ami_id        = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.small"

  tags = {
    Name        = "WebServer"
    Environment = "Production"
  }
}
```

0.8.4 Sources de modules

- Chemin local : `./modules/mon-module`
- Terraform Registry : `terraform-aws-modules/vpc/aws`
- GitHub : `git::https://github.com/user/repo.git`
- S3, HTTP, etc.

0.9 Fonctions Terraform

Terraform inclut de nombreuses fonctions intégrées.

0.9.1 Fonctions numériques

```
max(5, 12, 9)      # Retourne 12
min(5, 12, 9)      # Retourne 5
ceil(4.3)          # Retourne 5
floor(4.9)         # Retourne 4
```

0.9.2 Fonctions sur les chaînes

```
upper("hello")              # "HELLO"
lower("WORLD")              # "world"
format("Hello, %s!", "Bob") # "Hello, Bob!"
join(", ", ["a", "b", "c"]) # "a, b, c"
split(",", "a,b,c")         # ["a", "b", "c"]
```

0.9.3 Fonctions sur les collections

```
length(["a", "b", "c"])          # 3
element(["a", "b", "c"], 1)      # "b"
lookup({a = "foo"}, "a", "def")  # "foo"
merge({a = 1}, {b = 2})          # {a = 1, b = 2}
```

0.9.4 Tester les fonctions

Utilisez `terraform console` pour tester les fonctions :

```
$ terraform console
> max(5, 12, 9)
12
> upper("hello")
"HELLO"
```

0.10 Expressions et Conditions

0.10.1 Expressions conditionnelles

Syntaxe : `condition ? true_val : false_val`
Instance selon l'environnement

```
variable "environment" {
  type = string
}

resource "aws_instance" "app" {
  ami = "ami-0c55b159cbfafe1f0"
  instance_type = var.environment == "prod" ? "m5.large" : "t2.micro"
}
```

0.10.2 Boucles for

Transformer une liste

```
variable "names" {
  default = ["alice", "bob", "charlie"]
}

output "upper_names" {
  value = [for name in var.names : upper(name)]
  # Resultat: ["ALICE", "BOB", "CHARLIE"]
}
```

0.10.3 Interpolation de chaînes

```
resource "aws_instance" "app" {
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"

  tags = {
    Name = "server-${var.environment}-${var.region}"
  }
}
```

0.11 Bonnes Pratiques

0.11.1 Structure de projet

```
projet-terraform/
 main.tf           # Resources principales
 variables.tf      # Definitions des variables
 outputs.tf        # Definitions des outputs
 terraform.tfvars  # Valeurs des variables
 providers.tf      # Configuration des providers
 modules/          # Modules locaux
```

0.11.2 Gestion des credentials

Sécurité **Ne jamais** hardcoder les credentials dans le code !
Approches recommandées :

- Variables d'environnement
- Fichiers de configuration AWS/Azure CLI
- Secrets managers (Vault, AWS Secrets Manager)
- Profils IAM pour EC2

0.11.3 .gitignore pour Terraform

.gitignore recommandé

```
# State files
*.tfstate
*.tfstate.*
*.tfstate.backup

# Crash logs
crash.log

# Variables files (peuvent contenir des secrets)
*.tfvars
*.tfvars.json

# CLI configuration
.terraformrc
```

```
terraform.rc

# Plugin directory
.terraform/
.terraform.lock.hcl
```

0.11.4 Formatage du code

Utilisez `terraform fmt` pour formater automatiquement votre code :

```
terraform fmt          # Formate le repertoire courant
terraform fmt -recursive  # Formate recursivement
```

0.11.5 Validation du code

```
terraform validate     # Valide la syntaxe
```

0.12 Gestion du State

0.12.1 Backend local (par défaut)

Par défaut, Terraform stocke le state localement dans `terraform.tfstate`.
Limitation Le backend local n'est pas adapté au travail en équipe !

0.12.2 Backend distant (S3)

Pour le travail collaboratif, utilisez un backend distant.
Configuration S3 backend

```
terraform {
  backend "s3" {
    bucket         = "mon-bucket-tfstate"
    key            = "prod/terraform.tfstate"
    region         = "eu-west-1"
    encrypt        = true
    dynamodb_table = "terraform-locks"
  }
}
```

0.12.3 State Locking

Le locking empêche les modifications concurrentes du state.

Backends supportant le locking

- S3 (avec DynamoDB)
- Azure Blob Storage
- Google Cloud Storage
- Terraform Cloud
- Consul

0.12.4 Commandes de gestion du state

```
# Lister les resources
terraform state list

# Afficher une resource
terraform state show aws_instance.app

# Retirer une resource du state
terraform state rm aws_instance.app

# Deplacer une resource
terraform state mv aws_instance.old aws_instance.new
```

0.13 Workspaces

Les workspaces permettent de gérer plusieurs environnements avec le même code.

0.13.1 Commandes workspaces

```
# Lister les workspaces
terraform workspace list

# Creer un workspace
terraform workspace new dev

# Changer de workspace
terraform workspace select prod

# Afficher le workspace actuel
terraform workspace show
```

0.13.2 Utilisation dans le code

Adapter selon le workspace

```
resource "aws_instance" "app" {
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = terraform.workspace == "prod" ? "m5.large" : "t2.micro
   "
```

```
4
5   tags = {
6     Name        = "app-${terraform.workspace}"
7     Environment = terraform.workspace
8   }
9 }
```

0.14 Dépannage et Debugging

0.14.1 Logs détaillés

Activez les logs pour le debugging :

```
export TF_LOG=DEBUG
export TF_LOG_PATH=terraform.log
terraform apply
```

Niveaux de log :

- TRACE (le plus verbeux)
- DEBUG
- INFO
- WARN
- ERROR

0.14.2 Erreurs courantes

1. Provider not found

```
Solution: terraform init
```

2. State lock

```
Solution: terraform force-unlock <LOCK_ID>
```

3. Resource already exists

```
Solution: terraform import <resource> <id>
```

0.14.3 Refresh du state

```
# Mettre a jour le state sans appliquer
terraform plan -refresh-only
terraform apply -refresh-only
```

0.15 Exemple Complet : Application Web

Mettons tout ensemble pour déployer une application web complète.
Structure du projet

```
webapp/
 main.tf
 variables.tf
 outputs.tf
 terraform.tfvars
 modules/
     vpc/
     security-groups/
     compute/
```

main.tf

```
terraform {
  required_version = ">= 1.5"
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 5.0"
    }
  }
}

provider "aws" {
  region = var.aws_region
}

# Security Group
resource "aws_security_group" "web" {
  name        = "${var.project_name}-web-sg"
  description = "Security group for web server"

  ingress {
    from_port   = 80
    to_port     = 80
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }

  ingress {
    from_port   = 443
    to_port     = 443
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }

  egress {
    from_port   = 0
    to_port     = 0
    protocol    = "-1"
```

```
38      cidr_blocks = ["0.0.0.0/0"]
39    }
40
41    tags = {
42      Name        = "${var.project_name}-web-sg"
43      Environment = var.environment
44    }
45  }
46
47  # Instance EC2
48  resource "aws_instance" "web" {
49    ami                    = data.aws_ami.ubuntu.id
50    instance_type          = var.instance_type
51    vpc_security_group_ids = [aws_security_group.web.id]
52
53    user_data = <<-EOF
54                #!/bin/bash
55                apt-get update
56                apt-get install -y nginx
57                systemctl start nginx
58                systemctl enable nginx
59                echo "<h1>${var.project_name} - ${var.environment}</h1>" >
      /var/www/html/index.html
60                EOF
61
62    tags = {
63      Name        = "${var.project_name}-web-server"
64      Environment = var.environment
65    }
66  }
67
68  # Elastic IP
69  resource "aws_eip" "web" {
70    instance = aws_instance.web.id
71    domain   = "vpc"
72
73    tags = {
74      Name = "${var.project_name}-eip"
75    }
76  }
77
78  # Data source pour l'AMI
79  data "aws_ami" "ubuntu" {
80    most_recent = true
81    owners      = ["099720109477"]
82
83    filter {
84      name   = "name"
85      values = ["ubuntu/images/hvm-ssd/ubuntu-jammy-22.04-amd64-server-*"]
86    }
87
88    filter {
89      name   = "virtualization-type"
90      values = ["hvm"]
91    }
92  }
```

variables.tf

```
variable "aws_region" {
  description = "Region AWS"
  type        = string
  default     = "eu-west-1"
}

variable "project_name" {
  description = "Nom du projet"
  type        = string
}

variable "environment" {
  description = "Environnement (dev, staging, prod)"
  type        = string

  validation {
    condition     = contains(["dev", "staging", "prod"], var.environment
   )
    error_message = "L'environnement doit etre dev, staging ou prod."
  }
}

variable "instance_type" {
  description = "Type d'instance EC2"
  type        = string
  default     = "t2.micro"
}
```

outputs.tf

```
output "instance_id" {
  description = "ID de l'instance EC2"
  value       = aws_instance.web.id
}

output "instance_public_ip" {
  description = "Adresse IP publique"
  value       = aws_eip.web.public_ip
}

output "instance_public_dns" {
  description = "DNS public"
  value       = aws_instance.web.public_dns
}

output "security_group_id" {
  description = "ID du security group"
  value       = aws_security_group.web.id
}

output "website_url" {
  description = "URL du site web"
  value       = "http://${aws_eip.web.public_ip}"
```

```
}
```

terraform.tfvars

```
aws_region    = "eu-west-1"
project_name  = "webapp"
environment   = "dev"
instance_type = "t2.micro"
```

0.15.1 Déploiement étape par étape

```
# 1. Initialisation
terraform init

# 2. Formatage du code
terraform fmt

# 3. Validation
terraform validate

# 4. Planification
terraform plan -out=tfplan

# 5. Application
terraform apply tfplan

# 6. Afficher les outputs
terraform output

# 7. Tester l'application
curl http://$(terraform output -raw instance_public_ip)

# 8. Destruction (quand termine)
terraform destroy
```

0.16 Provisioners

Les provisioners permettent d'exécuter des scripts sur les resources après leur création.

Attention Les provisioners sont un dernier recours ! Privilégiez user_data, cloud-init, ou des outils comme Ansible.

0.16.1 Types de provisioners

local-exec : Exécute localement

```
resource "aws_instance" "web" {
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"

  provisioner "local-exec" {
    command = "echo ${self.public_ip} >> ip_addresses.txt"
```

```
  }
}
```

remote-exec : Exécute sur la resource

```
resource "aws_instance" "web" {
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"
  key_name      = "ma-cle-ssh"

  connection {
    type        = "ssh"
    user        = "ubuntu"
    private_key = file("~/.ssh/id_rsa")
    host        = self.public_ip
  }

  provisioner "remote-exec" {
    inline = [
      "sudo apt-get update",
      "sudo apt-get install -y nginx",
      "sudo systemctl start nginx"
    ]
  }
}
```

0.16.2 Provisioner on failure

```
resource "aws_instance" "web" {
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"

  provisioner "local-exec" {
    command     = "echo 'Instance created!'"
    on_failure  = continue  # continue ou fail
  }
}
```

0.17 Import de Resources Existantes

Terraform peut importer des resources existantes dans son state.

0.17.1 Processus d'import

1. Écrire la configuration Terraform de la resource
2. Utiliser `terraform import` pour l'ajouter au state
3. Ajuster la configuration si nécessaire

Importer une instance EC2 existante

```
# 1. Ecrire la configuration dans main.tf
# resource "aws_instance" "imported" {
#   # Configuration minimale
# }

# 2. Importer l'instance
terraform import aws_instance.imported i-1234567890abcdef0

# 3. Verifier le state
terraform state show aws_instance.imported

# 4. Completer la configuration dans main.tf
# avec les attributs affiches
```

0.17.2 Import block (Terraform 1.5+)

Depuis Terraform 1.5, il est possible d'utiliser un bloc `import`.
Import déclaratif

```
import {
  to = aws_instance.example
  id = "i-1234567890abcdef0"
}

resource "aws_instance" "example" {
  # Configuration sera generee automatiquement
}
```

0.18 Moved Blocks

Les moved blocks permettent de renommer des resources sans les détruire.
Renommer une resource

```
# Ancien nom
# resource "aws_instance" "old_name" { ... }

# Nouveau nom
resource "aws_instance" "new_name" {
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"
}

# Bloc moved pour eviter la recreation
moved {
  from = aws_instance.old_name
  to   = aws_instance.new_name
}
```

0.19 Terraform Cloud et Enterprise

0.19.1 Terraform Cloud

Terraform Cloud est une plateforme SaaS de HashiCorp pour :

— Gestion du state distant
— Exécution des plans et applies
— Collaboration d'équipe
— Intégration CI/CD
— Policy as Code (Sentinel)

0.19.2 Configuration Terraform Cloud

backend.tf

```
terraform {
  cloud {
    organization = "mon-organisation"

    workspaces {
      name = "mon-workspace-prod"
    }
  }
}
```

0.19.3 Remote Execution

Avec Terraform Cloud, les commandes s'exécutent à distance :

```
# Se connecter
terraform login

# Les commandes s'executent dans Terraform Cloud
terraform plan
terraform apply
```

0.20 Testing

0.20.1 Validation de base

```
# Validation de la syntaxe
terraform validate

# Verification du formatage
terraform fmt -check

# Plan sans appliquer
terraform plan
```

0.20.2 Tests avec Terraform Test (Beta)

Terraform 1.6+ inclut un framework de test intégré.
tests/main.tftest.hcl

```
run "verify_instance_type" {
  command = plan

  assert {
    condition     = aws_instance.web.instance_type == "t2.micro"
    error_message = "Instance type doit etre t2.micro"
  }
}

run "verify_tags" {
  command = plan

  assert {
    condition     = aws_instance.web.tags["Environment"] != ""
    error_message = "Le tag Environment est requis"
  }
}
```

```
# Executer les tests
terraform test
```

0.21 Sécurité

0.21.1 Sensitive Data

Marquez les données sensibles pour qu'elles ne s'affichent pas dans les logs.
Variables sensibles

```
variable "db_password" {
  description = "Mot de passe de la base de donnees"
  type        = string
  sensitive   = true
}

output "db_endpoint" {
  value     = aws_db_instance.main.endpoint
  sensitive = true
}
```

0.21.2 Checklist Sécurité

State file stocké de manière sécurisée (S3 chiffré)

State locking activé

Pas de credentials en dur dans le code

Variables sensibles marquées `sensitive = true`

.gitignore correctement configuré

Utilisation de secrets managers

Review des permissions IAM/RBAC

Chiffrement activé sur les resources (EBS, S3, etc.)

0.21.3 Scanning de sécurité

Utilisez des outils de scanning :

```
# tfsec - Scanner de securite
tfsec .

# Checkov - Policy as code
checkov -d .

# Terrascan
terrascan scan -t aws
```

0.22 Performance et Optimisation

0.22.1 Parallélisme

Terraform exécute les opérations en parallèle quand possible.

```
# Ajuster le parallelisme (defaut: 10)
terraform apply -parallelism=20
```

0.22.2 Plugin Cache

Évitez de télécharger les providers à chaque projet.

```
# Configurer le cache
export TF_PLUGIN_CACHE_DIR="$HOME/.terraform.d/plugin-cache"
mkdir -p $TF_PLUGIN_CACHE_DIR
```

0.22.3 Targets

Appliquez les changements sur des resources spécifiques.

```
# Appliquer seulement une resource
terraform apply -target=aws_instance.web

# Planifier pour plusieurs resources
terraform plan -target=aws_instance.web -target=aws_security_group.web
```

Attention L'utilisation de `-target` doit rester exceptionnelle !

0.23 CI/CD avec Terraform

0.23.1 Pipeline GitLab CI

.gitlab-ci.yml

```
image: hashicorp/terraform:latest

stages:
  - validate
  - plan
  - apply

variables:
  TF_ROOT: ${CI_PROJECT_DIR}
  TF_STATE_NAME: default

cache:
  paths:
    - ${TF_ROOT}/.terraform

before_script:
  - cd ${TF_ROOT}
  - terraform --version
  - terraform init

validate:
  stage: validate
  script:
    - terraform validate
    - terraform fmt -check

plan:
  stage: plan
  script:
    - terraform plan -out=tfplan
  artifacts:
    paths:
      - tfplan

apply:
  stage: apply
  script:
    - terraform apply -auto-approve tfplan
  when: manual
  only:
    - main
```

0.23.2 Pipeline GitHub Actions

.github/workflows/terraform.yml

```
name: Terraform

```

```yaml
on:
  push:
    branches: [ main ]
  pull_request:

jobs:
  terraform:
    runs-on: ubuntu-latest

    steps:
    - uses: actions/checkout@v3

    - name: Setup Terraform
      uses: hashicorp/setup-terraform@v2
      with:
        terraform_version: 1.5.0

    - name: Terraform Init
      run: terraform init

    - name: Terraform Format
      run: terraform fmt -check

    - name: Terraform Validate
      run: terraform validate

    - name: Terraform Plan
      run: terraform plan -no-color
      env:
        AWS_ACCESS_KEY_ID: ${{ secrets.AWS_ACCESS_KEY_ID }}
        AWS_SECRET_ACCESS_KEY: ${{ secrets.AWS_SECRET_ACCESS_KEY }}

    - name: Terraform Apply
      if: github.ref == 'refs/heads/main'
      run: terraform apply -auto-approve
      env:
        AWS_ACCESS_KEY_ID: ${{ secrets.AWS_ACCESS_KEY_ID }}
        AWS_SECRET_ACCESS_KEY: ${{ secrets.AWS_SECRET_ACCESS_KEY }}
```

0.24 Commandes Avancées

0.24.1 Terraform Graph

Générez un graphe de dépendances.

```bash
# Generer le graphe
terraform graph | dot -Tpng > graph.png

# Ou utiliser un outil en ligne
terraform graph > graph.dot
# Uploader sur http://www.webgraphviz.com/
```

0.24.2 Terraform Show

Affichez le state ou un plan de manière lisible.

```
# Afficher le state actuel
terraform show

# Afficher un plan sauvegarde
terraform show tfplan

# Format JSON
terraform show -json
```

0.24.3 Terraform Output

Récupérez les outputs facilement.

```
# Tous les outputs
terraform output

# Un output specifique
terraform output instance_ip

# Format raw (sans quotes)
terraform output -raw instance_ip

# Format JSON
terraform output -json
```

0.25 Ressources et Apprentissage

0.25.1 Documentation officielle

— https ://www.terraform.io/docs - Documentation complète
— https ://registry.terraform.io - Terraform Registry
— https ://learn.hashicorp.com/terraform - Tutoriels interactifs
— https ://developer.hashicorp.com - Documentation développeur

0.25.2 Certification

HashiCorp Certified : Terraform Associate

— Niveau : Débutant à intermédiaire
— Durée : 1 heure
— Questions : 57 questions
— Format : QCM
— Coût : 70 USD
— Validité : 2 ans

0.25.3 Communauté

— **Forum** : https ://discuss.hashicorp.com
— **GitHub** : https ://github.com/hashicorp/terraform
— **Discord** : HashiCorp Community Discord
— **Twitter** : @HashiCorp, @TerraformIO

0.25.4 Livres recommandés

— *Terraform : Up & Running* par Yevgeniy Brikman
— *Getting Started with Terraform* par Kishore Kumar
— *Infrastructure as Code* par Kief Morris

0.26 Glossaire

Provider Plugin permettant à Terraform d'interagir avec une API externe
Resource Composant d'infrastructure (VM, réseau, etc.)
Data Source Récupération d'informations depuis l'infrastructure existante
Module Ensemble de resources réutilisables
State Fichier contenant l'état actuel de l'infrastructure
Backend Emplacement de stockage du state file
Workspace Environnement isolé avec son propre state
Plan Aperçu des changements avant application
Apply Application des changements sur l'infrastructure
Destroy Suppression de l'infrastructure gérée
HCL HashiCorp Configuration Language
IaC Infrastructure as Code
Meta-argument Arguments modifiant le comportement des resources (count, for_each, etc.)
Provisioner Script exécuté lors de la création/destruction d'une resource

0.27 Exercices Pratiques

0.27.1 Exercice 1 : Première Infrastructure

Objectif : Créer une instance EC2 avec un security group

1. Créez un fichier `main.tf`
2. Définissez un provider AWS
3. Créez un security group autorisant SSH (port 22)
4. Créez une instance EC2 t2.micro

5. Ajoutez des tags appropriés
6. Déployez avec `terraform apply`
7. Détruisez avec `terraform destroy`

0.27.2 Exercice 2 : Variables et Outputs

Objectif : Paramétrer votre infrastructure

1. Extrayez les valeurs en dur dans des variables
2. Créez un fichier `variables.tf`
3. Créez un fichier `terraform.tfvars`
4. Ajoutez des outputs pour l'IP publique
5. Testez avec différentes valeurs

0.27.3 Exercice 3 : Module

Objectif : Créer un module réutilisable

1. Créez un module pour déployer un serveur web
2. Le module doit accepter : instance_type, ami_id, tags
3. Le module doit retourner : instance_id, public_ip
4. Utilisez le module dans votre configuration principale
5. Déployez 2 serveurs web avec des configurations différentes

0.27.4 Exercice 4 : Multi-environnement

Objectif : Gérer dev et prod

1. Utilisez des workspaces pour dev et prod
2. Utilisez des conditionnelles pour adapter les tailles d'instances
3. Prod : t2.medium, Dev : t2.micro
4. Testez le déploiement dans les deux environnements

0.28 Conclusion

0.28.1 Ce que vous avez appris

Félicitations ! Vous maîtrisez maintenant :

— Les concepts fondamentaux de Terraform
— Le workflow de base (init, plan, apply, destroy)
— La gestion des variables et outputs
— L'utilisation des modules
— La gestion du state et des backends
— Les bonnes pratiques de sécurité
— L'intégration CI/CD

0.28.2 Prochaines étapes

Pour aller plus loin :

1. Pratiquez régulièrement avec des projets réels
2. Explorez le Terraform Registry pour découvrir des modules
3. Apprenez des patterns avancés (remote state, terragrunt)
4. Étudiez les providers de votre cloud favori
5. Passez la certification Terraform Associate
6. Contribuez à la communauté open source

0.28.3 Conseils finaux

Bonnes pratiques à retenir

- Toujours utiliser un backend distant pour le travail en équipe
- Ne jamais commiter de credentials ou de state files
- Utiliser des modules pour la réutilisabilité
- Toujours exécuter `terraform plan` avant `apply`
- Documenter votre code et vos modules
- Automatiser avec CI/CD
- Tester avant de déployer en production

0.28.4 Remerciements

Merci d'avoir suivi ce guide ! N'hésitez pas à :

- Partager ce guide avec vos collègues
- Contribuer avec vos retours et suggestions
- Rejoindre la communauté Terraform

Bonne chance avec Terraform !
« Infrastructure as Code is not just a technology, it's a culture. »