

*Antony Kervazo-Canut*

# Terraform for Teenagers

**DU CODE POUR GÉRER  
L'INFRASTRUCTURE**



# SOMMAIRE

---

Introduction	3
Installation de Terraform	4
Configuration de Terraform	5
Démarrer avec Terraform	6
terraform init	7
terraform plan	8
terraform apply	9
terraform destroy	10
Gestion de l'état	11
Variables	12
Prevent Destroy	16
Modules	17
Conditions	20
Boucle for	21
Boucle for_each	22
Bloc dynamic	23
Terraform Registry	24
Vault	26
Consul	27

# Introduction



## Qu'est-ce que Terraform ?

Terraform est un outil d'Infrastructure as Code (IaC) développé par HashiCorp. Il permet aux utilisateurs de définir et de provisionner l'infrastructure des services cloud en utilisant un langage de configuration simple et déclaratif.

## Pourquoi utiliser Terraform ?

- Automatisation de l'infrastructure : Terraform automatise le déploiement et la gestion de l'infrastructure, réduisant le risque d'erreurs humaines.
- Gestion de l'état : Terraform maintient un état de l'infrastructure, permettant des modifications et des déploiements prévisibles.
- Support multi-cloud : Terraform peut gérer les ressources sur plusieurs plateformes cloud, offrant une grande flexibilité.
- Infrastructure en tant que code : Les configurations sont écrites dans des fichiers qui peuvent être versionnés, réutilisés et partagés.

# Installation de Terraform



```
## Linux
wget -O- https://apt.releases.hashicorp.com/gpg | sudo gpg --dearmor -o /usr/share/keyrings/hashicorp-archive-keyring.gpg
echo "deb [signed-by=/usr/share/keyrings/hashicorp-archive-keyring.gpg] https://apt.releases.hashicorp.com $(lsb_release -cs) main" | sudo tee /etc/apt/sources.list.d/hashicorp.list
sudo apt update && sudo apt install terraform

## Windows
wget -O- https://apt.releases.hashicorp.com/gpg | sudo gpg --dearmor -o /usr/share/keyrings/hashicorp-archive-keyring.gpg
echo "deb [signed-by=/usr/share/keyrings/hashicorp-archive-keyring.gpg] https://apt.releases.hashicorp.com $(lsb_release -cs) main" | sudo tee /etc/apt/sources.list.d/hashicorp.list
sudo apt update && sudo apt install terraform

## MacOS
brew tap hashicorp/tap
brew install hashicorp/tap/terraform
```

# Configuration de Terraform



Pour pouvoir utiliser Terraform, il faudra plusieurs prérequis en plus de Terraform lui même. Tout d'abord une clef SSH valide :

```
● ● ●  
# vérifier si une clef SSH est présente  
ls ~/.ssh/id_rsa.pub  
  
# Si ce n'est pas le cas, on la créer  
ssh-keygen
```

Pour Terraform avec Azure, il faudra également la présence d'azure CLI sur votre système :

<https://learn.microsoft.com/fr-fr/cli/azure/install-azure-cli>

Pour Terraform avec AWS, il faudra également la présence d'AWS CLI sur votre système :

<https://docs.aws.amazon.com/cli/latest/userguide/getting-started-install.html>

# Démarrer avec Terraform



```
● ● ●

provider "azurerm" {
    features {}
}

resource "azurerm_resource_group" "example" {
    name      = "example-rg"
    location  = "France Central"
}
```

Les deux fichiers écrits ici sont simple. Mais ils permettent d'annoncer quel est le fournisseur que vous allez utiliser. Au dessus : azure, en dessous aws. Par une préférence injuste, tout sera fait sur Azure.

```
● ● ●

provider "aws" {
    region = "eu-west-3" # Paris
}

resource "aws_vpc" "example" {
    cidr_block = "10.0.0.0/16"
    tags = [
        Name = "example-vpc"
    ]
}
```

# terraform init



```
● ● ●  
  
# Connexion à Azure, importance d'avoir AzureCLI  
az login  
  
# Initialisation du fichier et du fournisseur  
terraform init
```

## Que fait terraform init ?

- **Installation des Plugins** : Terraform télécharge et installe les plugins nécessaires pour interagir avec les fournisseurs spécifiés dans votre configuration (par exemple, Azure, AWS).
- **Initialisation du Backend** : Si spécifié, configure le backend pour l'état de Terraform.
- **Préparation du Répertoire** : Terraform prépare le répertoire de travail pour l'exécution des autres commandes Terraform.

Un message de succès indiquera que l'initialisation s'est déroulée correctement. Vous êtes maintenant prêt à planifier et appliquer vos configurations Terraform.

# terraform plan



```
# Exécutez terraform plan dans votre répertoire de projet
pour voir quels changements seront appliqués à votre
infrastructure.
terraform plan
```

## Que fait terraform plan ?

- **Analyse de la Configuration** : Terraform analyse vos fichiers de configuration pour déterminer les ressources à créer, modifier ou détruire.
- **Affichage des Changements** : Terraform présente un plan d'exécution, montrant ce qui sera modifié dans l'infrastructure.
- **Prévention des Surprises** : Cela permet de vérifier les changements avant de les appliquer, réduisant le risque d'erreurs inattendues.

Le plan affiche des ajouts (+), des changements (~) et des suppressions (-). Il est crucial de le lire attentivement pour s'assurer que les changements correspondent à vos attentes.

```
Terraform used the selected providers to generate the following execution plan. Resource actions are indicated with the following symbols:
+ create

Terraform will perform the following actions:

# azurerm_resource_group.example will be created
+ resource "azurerm_resource_group" "example" {
    + id      = (known after apply)
    + location = "francecentral"
    + name    = "example-rg"
}

Plan: 1 to add, 0 to change, 0 to destroy.
```

# terraform apply



```
# Exécutez terraform apply pour appliquer les changements spécifiés dans votre plan d'exécution.  
terraform apply
```

## Que fait terraform apply ?

- Confirmation des Changements : Avant d'appliquer les changements, Terraform affiche le plan et demande une confirmation.
- Modification de l'Infrastructure : Si confirmé, Terraform applique les changements à l'infrastructure conformément à votre configuration.
- Mise à Jour de l'État : Après l'application des changements, Terraform met à jour le fichier d'état pour refléter l'état actuel de l'infrastructure.

L'état de Terraform est crucial pour maintenir l'alignement entre votre configuration et l'infrastructure réelle. Après chaque application, Terraform met à jour son fichier d'état.

# terraform destroy



```
# Utilisez terraform destroy pour supprimer toutes les
ressources gérées par votre configuration Terraform.
terraform destroy
```

## Que fait terraform destroy ?

- **Affichage des Ressources à Supprimer :** Terraform affiche la liste des ressources qu'il prévoit de supprimer.
- **Demande de Confirmation :** Avant de procéder, Terraform demande une confirmation pour s'assurer que vous souhaitez supprimer ces ressources.
- **Destruction des Ressources :** Si confirmé, Terraform supprime toutes les ressources listées, nettoyant ainsi l'infrastructure.

Cette commande est utile lors de la fin d'un projet, pour les environnements de test, ou pour reconstruire entièrement une infrastructure. Il est important de l'utiliser avec prudence, car elle supprime toutes les ressources gérées.

# Gestion de l'état



Terraform utilise un fichier d'état pour garder une trace de l'infrastructure et des configurations. Ce fichier d'état est crucial pour le fonctionnement de Terraform.

- **Le fichier d'état de Terraform (terraform.tfstate) enregistre les ID et les propriétés des ressources créées par Terraform. Il est utilisé pour mapper les ressources réelles à votre configuration et pour garder une trace des métadonnées.**

```
● ● ●

# Liste toutes les ressources que Terraform gère
actuellement.
terraform state list

# Affiche l'état actuel de votre infrastructure telle que
connue par Terraform.
terraform show
```

# Variables



```
project/
└── globals.tfvars

└── environments/
    └── dev/
        ├── main.tf
        ├── variables.tf
        └── provider.tf
```

Quand un projet devient plus conséquent, il faut commencer à l'architecturer. On va alors commencer à séparer les différentes ressources, la définition du provider ou les variables.

Pour l'exemple suivant, nous allons reprendre le fichier de création de ressources sur azure en déplaçant le provider dans provider.tf.

```
provider "azurerm" {
  features {}
}
```

# Variables



● ● ●

```
resource "azurerm_resource_group" "example" {
    name      = "example-resources"
    location  = "France Central"
}
```

En déplaçant le resource group dans le fichier main.tf on obtient ceci. Mais en imaginant un monde avec plusieurs environnement, nous allons sortir les variables name et location pour les placer dans le fichier global.tfvars.

● ● ●

```
resource_group_name = "example-rg"
location          = "France Central"
```

# Variables



```
variable "resource_group_name" {
  description = "Nom global du groupe de ressources"
  type        = string
}

variable "location" {
  description = "Localisation globale pour les ressources"
  type        = string
}

variable "name_prefix" {
  description = "Préfixe pour le nom du groupe de ressources
dans l'environnement de développement"
  default     = "dev-"
  type        = string
}
```

On crée le fichier `variables.tf` (ci-dessus) et on modifie le `main.tf`. Comme notre architecture est découpé par environnement. On préfixe son nom par "dev-".

```
resource "azurerm_resource_group" "example" {
  name      = "${var.name_prefix}${var.resource_group_name}"
  location  = var.location
}
```

# Variables



```
● ● ●

# On change de répertoire
cd environments/dev

# Puisque ce n'est plus le même répertoire, il faut
recommencer l'init
terraform init
# Pour réussir correctement le plan, les variables vont être
necessaire
terraform plan -var-file=../../globals.tfvars
# Une fois que le résultat vous convient, vous pouvez
appliquer celui-ci
terraform apply -var-file=../../globals.tfvars
```

```
● ● ●

# azurerm_resource_group.example will be created
+ resource "azurerm_resource_group" "example" {
    + id      = (known after apply)
    + location = "francecentral"
    + name     = "dev-example-rg"
}

Plan: 1 to add, 0 to change, 0 to destroy.
```

# Prevent Destroy



Terraform permet de marquer une ressource pour empêcher sa destruction via l'attribut `lifecycle.prevent_destroy` dans le bloc de ressource. Cependant, cet attribut doit être utilisé avec précaution, car il rend la destruction de la ressource manuellement difficile sans modification du code Terraform.

```
● ● ●

resource "azurerm_resource_group" "example" {
    name      = var.resource_group_name
    location  = var.location

    lifecycle {
        prevent_destroy = true
    }
}
```

Cependant cette action va créer une erreur lors d'un `destroy`. Il faudra donc repenser l'architecture de votre projet Terraform pour ranger ensemble les ressources permanentes et mettre les ressources temporaires d'un autre côté.

# Modules



```
project/
  provider.tf
  main.tf
  modules/
    azure_rg_module/
      main.tf
      output.tf
      variables.tf
```

**Les modules permettent de réutiliser des morceaux de code dans différents projets ou parties d'un projet.**

**Organisation :** Ils aident à organiser et à structurer le code Terraform de manière plus lisible et maintenable.

**Encapsulation :** Les modules peuvent encapsuler une logique complexe, ne exposant que les paramètres nécessaires en tant qu'entrées et sorties.

# Modules

## azure\_rg\_module



```
# main.tf
resource "azurerm_resource_group" "example" {
    name      = var.resource_group_name
    location  = var.location
}
```

```
# output.tf
output "resource_group_id" {
    value = azurerm_resource_group.example.id
    description = "L'ID du groupe de ressources créé"
}
```

```
# variables.tf
variable "resource_group_name" {
    description = "Nom global du groupe de ressources"
    type        = string
}

variable "location" {
    description = "Localisation globale pour les ressources"
    type        = string
}
```

# Modules



A la racine, dans le main.tf, on déclare le module que l'on a créé en lui passant directement les variables en paramètre.

L'output permet de logguer la création du resource group.

```
● ● ●

module "azure_rg_module" {
  source    = "./modules/azure_rg_module"
  resource_group_name      = "example-rg"
  location = "France Central"
}

# Pour afficher l'output
output "premier_groupe_resource_id" {
  value = module.azure_rg_module.resource_group_id
}
```

```
● ● ●

# On initialise pour ajouter le module
terraform init

# On vérifie le plan et on exécute
terraform plan
terraform apply
```

# Conditions



Terraform permet d'utiliser des conditions logiques dans les configurations en utilisant une combinaison de variables et de l'opérateur ternaire. Cela permet d'ajuster dynamiquement la configuration en fonction de certaines conditions.

```
## Condition ternaire sur boolean
variable "create_resource" {
    description = "Un drapeau pour créer ou non une ressource"
    type        = bool
    default     = false
}

resource "azurerm_resource_group" "example" {
    count      = var.create_resource ? 1 : 0  # Utilisation d'une
condition
    name       = "example-resource-group"
    location   = "West Europe"
}
```

# Boucle for



Créez une ressource, par exemple un groupe de sécurité réseau, et utilisez une expression for pour générer des tags.

```
variable "instance_names" {
  type    = list(string)
  default = ["instance1", "instance2", "instance3"]
}

resource "azurerm_network_security_group" "example" {
  name          = "nsg-example"
  location      = "West Europe"
  resource_group_name = azurerm_resource_group.example.name

  tags = { for name in var.instance_names : name => "${name}-security" }
}
```

Dans cet exemple, pour chaque nom d'instance dans var.instance\_names, un tag est créé où la clé est le nom de l'instance et la valeur est une combinaison du nom de l'instance et d'un suffixe (par exemple, "instance1" devient "instance1-security").

# Boucle for\_each



`for_each` est utilisé pour itérer sur chaque élément d'un map ou d'une liste, créant une instance de la ressource pour chaque élément.

```
variable "instance_tags" {
  type = map(string)
  default = {
    "instance1" = "tag1"
    "instance2" = "tag2"
  }
}

resource "azurerm_virtual_machine" "example" {
  for_each          = var.instance_tags
  name              = each.key
  tags              = { "Name" = each.value}
  location          = "West Europe"
  resource_group_name = azurerm_resource_group.example.name
  # Autres configurations ...
}
```

Ici, `for_each` crée une VM pour chaque élément dans `var.instance_tags`, utilisant la clé pour le nom et la valeur pour le tag.

# Bloc dynamic



Considérons la création de plusieurs règles dans une politique de sécurité réseau, où le nombre de règles est variable.

```
variable "nsg_rules" {
  description = "Liste des règles de sécurité à appliquer au NSG."
  type = list(object({
    name          = string
    priority      = number
    direction     = string
    access         = string
    protocol      = string
    source_port_range = string
    destination_port_range = string
    source_address_prefix = string
    destination_address_prefix = string
  }))
  default = [
    {
      name          = "allow-http"
      priority      = 200
      direction     = "Inbound"
      access         = "Allow"
      protocol      = "Tcp"
      source_port_range = "*"
      destination_port_range = "80"
      source_address_prefix = "*"
      destination_address_prefix = "*"
    },
    {
      name          = "deny-outbound"
      priority      = 300
      direction     = "Outbound"
      access         = "Deny"
      protocol      = "*"
      source_port_range = "*"
      destination_port_range = "*"
      source_address_prefix = "*"
      destination_address_prefix = "*"
    }
  ]
}
```

# Bloc dynamic



Dans cet exemple, `nsg_rules` est une variable qui contient une liste d'objets, chaque objet représentant une règle de sécurité réseau spécifique pour le groupe de sécurité Azure à l'aide de blocs dynamiques dans Terraform, offrant ainsi une méthode flexible et puissante pour gérer les configurations de sécurité réseau dans Azure.

```
resource "azurerm_network_security_group" "example" {
    name          = "example-nsg"
    location      = var.location
    resource_group_name = azurerm_resource_group.example.name

    dynamic "security_rule" {
        for_each = var.nsg_rules

        content {
            name          = security_rule.value.name
            priority      = security_rule.value.priority
            direction     = security_rule.value.direction
            access         = security_rule.value.access
            protocol       = security_rule.value.protocol
            source_port_range =
                security_rule.value.source_port_range
            destination_port_range =
                security_rule.value.destination_port_range
            source_address_prefix =
                security_rule.value.source_address_prefix
            destination_address_prefix =
                security_rule.value.destination_address_prefix
        }
    }
}
```

# Terraform Registry



Terraform propose une registry publique disponible ici :  
<https://registry.terraform.io/>

Leur utilisation est relativement simple et un simple "terraform init" après avoir écrit les avoir configuré dans votre projet permet de les utiliser.

L'exemple suivant fait exactement la même chose que dans nos exemples précédents depuis ce module :  
<https://registry.terraform.io/modules/getindata/resource-group/azurerm/latest>

```
module "resource-group" {
  source  = "getindata/resource-group/azurerm"
  version = "1.2.1"
  location = "France Central"
  name      = "example-rg-from-module"
}
```

# Vault



```
provider "vault" {  
    # Assurez-vous que ces variables d'environnement sont  
    définies :  
    VAULT_ADDR : L'adresse de votre serveur Vault  
    VAULT_TOKEN : Un token avec accès aux secrets nécessaires  
}  
  
data "vault" "password" {  
    path = "secret/data/azure/db"  
}  
  
resource "azurerm_sql_server" "example" {  
    name                  = "example-sqlserver"  
    resource_group_name   =  
    azurerm_resource_group.example.name  
    location              =  
    azurerm_resource_group.example.location  
    version               = "12.0"  
    admini_login          = "sqladmin"  
    admini_password        = data.vault.password.data["password"]  
}
```

Vault peut être utilisé avec Terraform pour gérer de manière sécurisée les secrets ou les données sensibles nécessaires dans vos configurations Terraform.

Terraform dispose d'un provider Vault qui permet de lire des informations de Vault et de les utiliser dans vos configurations Terraform.



# Consul

```
● ● ●

provider "consul" {
  address      = "consul.mycompany.com"
  datacenter   = "dc1"
}

data "consul_keys" "app_config" {
  key {
    path = "config/myapp/port"
  }
}

resource "azurerm_virtual_machine" "example" {
  name          = "example-vm"
  location      =
  azurerm_resource_group.example.location
  resource_group_name = azurerm_resource_group.example.name
  network_interface_ids =
  [azurerm_network_interface.example.id]
  vm_size        = "Standard_DS1_v2"

  os_profile {
    computer_name  = "hostname"
    admin_username = "adminuser"
  }

  tags = {
    "AppPort" = data.consul_keys.app_config.var.port
  }
}
```

Consul peut être utilisé avec Terraform pour stocker des configurations ou pour découvrir des services et des adresses IP au sein de votre infrastructure.

# Dans la même collection

## Orchestration et Gestion de Conteneurs



## Infrastructure as Code



## Sécurité & Gestion des secrets



## Développement & CI/CD



↓ FOLLOW ME ↓



[ANTONYCANUT](#)



[ANTONY KERVAZO-CANUT](#)