

# TERRAFORM



# Alves Lobo Michael

- <https://www.linkedin.com/in/michael-alves-lobo>
- <https://github.com/kairel-4a0057b3/>
- devops, dev, réseau et administration système
- En poste chez Click2buy
- [https://github.com/kairel/learning/blob/main/devops\\_terraform.pdf](https://github.com/kairel/learning/blob/main/devops_terraform.pdf)
- Automatisation
  - ansible
  - puppet
  - terraform
  - docker



# SOMMAIRE

## Terraform

- 1.Présentation de l'outil
- 2.Installation
- 3.Configuration
- 4.Providers
- 5.Resources
- 6.Architecture
- 7. Variables
- 8. Les autres concepts
- Liens utiles



# Terraform - Présentation

Comment on va procéder ?

- Pas bcp de théorie , bcp de pratique
- On va déployer une plateforme complète soit sur un provider en local(Docker)
- On va d'abord essayer de faire des choses qui fonctionnent , et on le modifiera pour suivre les bonnes pratiques et que ce soit réutilisable et compréhensible



# Terraform - Présentation

## 1.1 définition

Définition (wikipédia)

**Terraform** est un environnement logiciel d'« [infrastructure as code](#) » publié en [open-source](#) par la société [HashiCorp](#). Cet outil permet d'automatiser la construction des ressources d'une infrastructure de [centre de données](#) comme un [réseau](#), des [machines virtuelles](#), un groupe de sécurité ou une [base de données](#).

L'infrastructure est décrite sous forme du langage de configuration [Hashicorp Configuration Language](#) (HCL). Il est aussi possible d'utiliser le langage [JSON](#)<sup>2</sup>.

Terraform permet notamment de définir des topologies [cloud](#) pour les principaux fournisseurs d'infrastructure cloud, tels qu'[Amazon Web Services](#), [IBM Cloud](#) (anciennement [Bluemix](#)), [Google Cloud Platform](#), [Linode](#)<sup>3,4</sup>, [Microsoft Azure](#), [Oracle Cloud Infrastructure](#), [OVHcloud](#)<sup>5,6</sup> ou [VMware](#), [vSphere](#) ainsi que [OpenStack](#)<sup>7,8,9,10,11,12</sup>.

Les ressources décrites dans le code HCL Terraform sont dépendantes du fournisseur (« provider ») de l'infrastructure cloud. Par exemple, une ressource Terraform définie pour une topologie Amazon ne peut pas être réutilisée pour une topologie [OpenStack](#) ou [Microsoft Azure](#) puisqu'elle n'ont pas les mêmes propriétés.



# Terraform - Présentation:

## 1.2 Mot clés

Pourquoi Terraform ?

- IAC
- Agnostique
- Orchestration
- Déploiement
- Statefull



# Terraform - Présentation:

## 1.2 Mot clés

IAC

Infrastructure as a code

L'**infrastructure as code (IaC)** désigne le processus qui implique la mise à disposition d'environnements à l'aide d'un code contenant la configuration désirée d'une infrastructure.

Couplé à un système de gestion de code, comme git, plus un gestionnaire de configuration, comme Ansible, et vous avez tout pour faire un parfait **gitops**. Vous serez capable de reconstruire toute votre infrastructure à l'identique après une catastrophe, un changement de fournisseur, etc.



# Terraform - Présentation:

## 1.2 Mot clés

Agnostique

- Être indépendant des autres logiciels ou protocoles de communication.  
Terraform est indépendant de la plateforme visée, on peut déployer sur AWS, Azure, GCP mais aussi en local (virtualbox, proxmox etc ...)





# Terraform - Présentation:

## 1.2 Mot clés

Orchestration / déploiement

- Comme d'autres outils , terraform permet d'orchestrer le déploiement d'infrastructure complète , de la VM en passant par le réseau , le VPN , le stockage etc ....



# Terraform - Présentation:

## 1.2 Mot clés

Statefull

- Terraform fonctionne par état: il garde une trace de l'état(tfstate) en cours et applique un plan pour arriver à l'état voulu

Par exemple on a actuellement une infra avec 2 VMs front (l'état en cours) et on veut une infra avec 2 VMs front + 1 VM database sur un réseau à part (état voulu)



# Terraform - Présentation

## 1.3 Utilisation

A quoi ça sert, des exemples ?

- Déployer des serveurs dans des clouds
- Déployer des réseaux et déployer des serveurs dans ces réseaux
- Répartir les installations sur des tenants, datacenters différents



# Terraform - Présentation

## 1.3 Utilisation

- Hashicorp Configuration Language (HCL)

Le HCL permet d'être interprété par Terraform

D'une manière générale un HCL est un fichier qui contient un ensemble de ressources et de variables



# Terraform - Présentation

## 1.3 Utilisation

- Workflow

Terraform fonctionne par étape/workflow

1 - write: on écrit notre projet

2- init: on build notre projet (récupération des modules/lib)

3- plan: on audit et valide les changements

4- apply: on applique les changements sur la production

5- destroy: on détruit notre infra (à utiliser avec modération )



# Terraform - Présentation

## 1.3 Utilisation

Terraform n'est pas "partageur", il est conçu pour gérer tout seul les ressources d'un provider, il convient donc soit de l'utiliser pour toutes actions de l'infra chez le provider, soit de faire des mises à jour régulières de l'état des ressources, soit de faire un import complet des ressources, soit de l'utiliser uniquement sur une partie de l'infrastructure

Dans tous les cas la commande "terraform plan" est primordial pour voir ce qui va être fait sur votre infra



# Terraform - Installation

## 2.1 Installation (debian)

```
curl -fsSL https://apt.releases.hashicorp.com/gpg | sudo apt-key add -  
sudo apt-add-repository "deb [arch=$(dpkg --print-architecture)]  
https://apt.releases.hashicorp.com $(lsb_release -cs) main"
```

```
sudo apt install terraform
```



# Terraform - Installation

## 2.1 Installation (debian) autre méthode

```
TER VER=`curl -s https://api.github.com/repos/hashicorp/terraform/releases/latest  
grep tag name | cut -d: -f2 | tr -d \"\",\\v | awk '{$1=$1};1'
```

```
wget  
https://releases.hashicorp.com/terraform/${TER VER}/terraform ${TER VER} linux amd64  
zip
```

```
unzip terraform ${TER VER} linux amd64.zip
```

```
$ sudo mv terraform /usr/local/bin
```





# Terraform - Installation

## 2.1 Installation (redhat)

```
sudo yum install -y yum-utils
```

```
sudo yum-config-manager --add-repo https://rpm.releases.hashicorp.com/$release/hashicorp.repo
```

```
yum install terraform
```



# Terraform - Installation

## 2.1 Check de l'installation

Une fois l'installation terminée on va tester avec la commande

```
→ vagrant git:(main) terraform --version  
Terraform v1.1.9  
on darwin_amd64  
→ vagrant git:(main) □
```



# Terraform - Installation

A vous de jouer:

- Déployer une instance de debian 11
- Installer terraform avec ansible sur cette VM
- checker les infos



# Terraform - Configuration

## 3.1 Les actions principales

```
terraform init:
```

C'est la première commande qui permet d'initialiser un répertoire de travail

Elle sert donc à initialiser un projet avec sa configuration, on peut lancer cette commande plusieurs fois durant le cycle de vie de votre projet



# Terraform - Configuration

## 3.1 Les actions principales

```
terraform init
```

Il y a une multitude d'options permettant de configurer votre projet

- upgrade: permet de mettre à jour les modules et les plugins
- no-color: permet d'enlever les couleurs de l'output
- lock=false : désactive le lock des fichiers state pendant les opérations de synchronisations (à éviter car on peut vite avoir des fichiers corrompus)



# Terraform - Configuration

## 3.1 Les actions principales

```
terraform init:
```

Lorsque cette commande est appliquée , elle va chercher tous les modules et plugins à installer pour initialiser le projet



# Terraform - Configuration

## 3.1 Les actions principales

```
terraform plan
```

Cette commande permet de créer le plan d'exécution, il affiche donc le plan des changements que fera terraform sur votre infrastructure

Par défaut quand terraform crée un plan:

- il lit l'état de tous les remote objects pour s'assurer que l'état que terraform a localement est uptodate
- il compare la configuration courante actuelle à l'état précédent
- il propose et liste les changements qui vont être apportés sur l'infrastructure



# Terraform - Configuration

## 3.1 Les actions principales

```
terraform plan:
```

Cette commande n'applique aucun changement , elle ne fait qu'auditer les changements qui auront lieu si on les applique

Si il n'y a aucun changements , il n'y a rien en sortie de cette commande

on peut sauvegarder cette planification dans un fichier afin de la rejouer plus tard si nécessaire “-out=FILE”





# Terraform - Configuration

## 3.1 Les actions principales

```
terraform plan:
```

il y a plusieurs options à cette commande ,elle se divise en plusieurs catégories

- planning mode:
  - destroy mode: NE JAMAIS L'UTILISER: comme son nom l'indique vous permet de supprimer toutes les ressources qui existent
  - refresh -only mode: permet de réconcilier l'état de votre infra sur le serveur avec les fichiers d'états de terraform (l'exemple typique c'est des modifications manuelles sur l'infra ,on utilise dans ce cas une refresh pour mettre à jour l'état en local de votre projet terraform)



# Terraform - Configuration

## 3.1 Les actions principales

```
terraform plan:
```

il y a plusieurs options à cette commande ,elle se divise en plusieurs catégories

- planning options:
  - refresh=false: désactive la synchronisation de l'état de terraform avec les objets distants
  - var : permet de lancer le plan avec des variables (ligne de commande ou fichier)



# Terraform - Configuration

## 3.1 Les actions principales

```
terraform plan:
```

il y a plusieurs options à cette commande ,elle se divise en plusieurs catégories

- other options:
  - compact-warnings: compaction des warnings
  - detailed-exitcode: sorte de mode de debug
  - json: output en json



# Terraform - Configuration

## 3.1 Les actions principales

```
terraform apply:
```

Cette commande exécute les actions proposées dans le plan de terraform

Il est possible soit de lancer la commande et de confirmer les actions soit de lancer la commande avec un fichier sauvegardé , dans ce cas aucune confirmation ne sera demandée



# Terraform - Configuration

## 3.1 Les actions principales

```
terraform apply:
```

Il est possible de se passer du prompt de confirmation de l'exécution des actions, à n'utiliser que si ..... NE JAMAIS l'UTILISER

Les mêmes catégories d'options que pour le plan sont disponibles

- plan options
- etc ...



# Terraform - Configuration

## 3.2 Prise en main

A vous de jouer:

- On va faire notre premier projet terraform sans provider afin de prendre en main l'outil sans "rien casser"

on crée un répertoire vide

```
mkdir projet
```



# Terraform - Configuration

## 3.2 Prise en main

A vous de jouer:

Comme pour tout projet Terraform on commence par faire un terraform init

```
cd projet
```

```
terraform init
```



# Terraform - Configuration

## 2.2 Prise en main

A vous de jouer:

On va créer notre premier fichier avec le module output

```
nano main.tf
```

```
→ sample git:(main) ✕ cat  
output "my_first_var" {  
  value = "Hello world"  
}  
→ sample git:(main) ✕
```





# Terraform - Configuration

## 2.2 Prise en main

A vous de jouer:

On va lancer le plan d'exécution

```
terraform plan
```

```
→ sample git:(main) ✗ terraform plan
```

Changes to Outputs:

```
+ my_first_var = "Hello world"
```

You can apply this plan to save these new output values to the Terraform state, without changing any real infrastructure.

---

Note: You didn't use the -out option to save this plan, so Terraform can't guarantee to take exactly these actions if you run "terraform apply" now.



# Terraform - Configuration

## 3.2 Prise en main

A vous de jouer:

On va appliquer le plan d'exécution

```
terraform apply
```

```
→ sample git:(main) ✗ terraform apply
```

Changes to Outputs:

```
+ my_first_var = "Hello world"
```

You can apply this plan to save these new output values to the Terraform state, without changing any real infrastructure.

Do you want to perform these actions?

Terraform will perform the actions described above.

Only 'yes' will be accepted to approve.

Enter a value: yes

```
Apply complete! Resources: 0 added, 0 changed, 0 destroyed.
```

Outputs:

```
my_first_var = "Hello world"
```



# Terraform - Configuration

## 3.2 Prise en main

A vous de jouer:

Lorsque le plan a été exécuté on remarque quelque chose de nouveau  
qu'est-ce que c'est ?



# Terraform - Configuration

## 3.2 Les fichiers state

Après avoir appliqué notre premier “apply”, on s’aperçoit de la création d’un fichier state avec l’extension tfstate

```
→ sample git:(main) x ll
total 16
27780120 drwxr-xr-x  4 malveslobo  staff   128B 25 avr 22:48 .
25826129 drwxr-xr-x 12 malveslobo  staff   384B 25 avr 21:39 ..
27780701 -rw-r--r--   1 malveslobo  staff    50B 25 avr 21:38 main.tf
27799280 -rw-r--r--   1 malveslobo  staff   239B 25 avr 22:48 terraform.tfstate
```



# Terraform - Configuration

## 3.2 Les fichiers state

Ce fichier contient certaines informations comme l'output , les ressources utilisées , la version et une série de champs utilisés dans le mécanisme interne de terraform

```
→ sample git:(main) ✗ cat terraform.tfstate
{
  "version": 4,
  "terraform_version": "1.1.9",
  "serial": 1,
  "lineage": "3abd5288-0836-321f-c106-36e101756896",
  "outputs": {
    "my_first_var": {
      "value": "Hello world",
      "type": "string"
    }
  },
  "resources": []
}
```



# Terraform - Configuration

## 3.2 Les fichiers state

Pour effectuer des changements sur votre infrastructure terraform doit garder en mémoire l'état de l'infra

Cet état est stocké dans un fichier local terraform.tfstate, ce fichier peut être versionné et partagé avec d'autres personnes de votre équipe

Terraform utilise ce fichier pour apporter les modifications nécessaires à votre infrastructure



# Terraform -

## 4.1 Les providers

- Les providers sont des sortes de plugins qui interagissent avec les API , les fournisseurs de cloud etc ..
- Ils doivent être configurés pour être utilisés
- Ils permettent de manipuler les objects qui leurs sont propres (AWS-> image, Azure->network etc ...)



# Terraform -

## 4.1 Les providers

- Chaque provider possède des types de ressources des datasources
- Toutes les ressources sont implémenté par un provider, sinon terraform est incapable de manager l'infra
- Les providers sont disponibles via des registry , le principal et le plus utilisé et le registry de Terraform

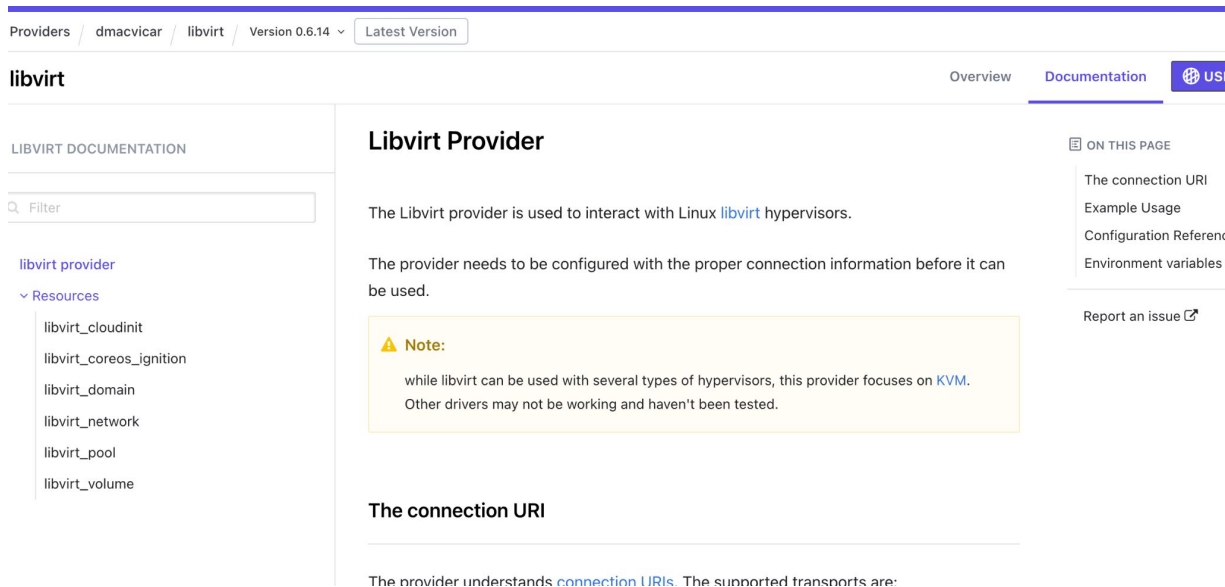




# Terraform -

## 4.1 Les providers

- Pour tous les providers les informations et la documentation sont fait de la même façon
  - des prérequis
  - la configuration à utiliser



The screenshot shows the Terraform documentation page for the **libvirt** provider. The page is titled "libvirt" and includes a navigation bar with "Providers", "dmacvilar", "libvirt", and "Version 0.6.14". The main content area is divided into two columns. The left column contains a search bar and a list of resources under the heading "libvirt provider". The right column contains the main documentation text, including a "Note" section and a "The connection URI" section. A sidebar on the right lists "ON THIS PAGE" with links to "The connection URI", "Example Usage", "Configuration Reference", and "Environment variables".

Providers / dmacvilar / libvirt / Version 0.6.14 ▾ Latest Version

**libvirt** Overview Documentation US

LIBVIRT DOCUMENTATION

Filter

libvirt provider

Resources

- libvirt\_cloudinit
- libvirt\_coreos\_ignition
- libvirt\_domain
- libvirt\_network
- libvirt\_pool
- libvirt\_volume

### Libvirt Provider

The Libvirt provider is used to interact with Linux [libvirt](#) hypervisors.

The provider needs to be configured with the proper connection information before it can be used.

**Note:**

while libvirt can be used with several types of hypervisors, this provider focuses on [KVM](#). Other drivers may not be working and haven't been tested.

### The connection URI

The provider understands [connection URIs](#). The supported transports are:

ON THIS PAGE

- The connection URI
- Example Usage
- Configuration Reference
- Environment variables

Report an issue



# Terraform -

## 4.1 Les providers

- Les providers sont installés au moment du “terraform init”, vous pouvez rejouer le terraform init à tout moment pour installer un nouveau provider



# Terraform -

## 4.1 Les providers

- Des exemples de providers (<https://registry.terraform.io/browse/providers>)
  - cloud
    - AWS
    - Azure
    - Vsphere
  - autres
    - active directory
    - Kubernetes
    - FortiOS
    - Docker



# Terraform -

## 4.1 Les providers

- Exemple avec FortiOS



on the left to read more details about the available resources.

### Configuration for FortiGate

#### Example Usage

```
# Configure the FortiOS Provider for FortiGate
provider "fortios" {
  hostname     = "192.168.52.177"
  token        = "jn3t3Nw7qckQzt955Htkfj5hwQ6jdb"
  insecure     = "false"
  cabundlefile = "/path/yourCA.crt"
}

# Create a Static Route Item
resource "fortios_networking_route_static" "test1" {
  dst      = "110.2.2.122/32"
  gateway  = "2.2.2.2"
  # ...
}
```

If it is used for testing, you can set `insecure` to "true" and unset `cabundlefile` to quickly set the provider up, for example:

```
provider "fortios" {
  hostname = "192.168.52.177"
  token    = "jn3t3Nw7qckQzt955Htkfj5hwQ6jdb"
  insecure = "true"
}
```

# Terraform -

## 4.1 Les providers

- Pour utiliser un provider il faut le déclarer et l'instancier

Il n'est pas nécessaire de déclarer tous les providers , ils sont installés par défaut avec Terraform , fortios par exemple n'a pas besoin d'être déclaré contrairement à libvirt



# Terraform -

## 4.1 Les providers

Déclarer un provider non fourni par défaut

```
terraform {  
  required_providers {  
    libvirt = {  
      source = "powerdns"  
    }  
  }  
}
```



# Terraform -

## 4.1 Les providers

instancier un provider , les attributs dépendent du provider

```
# Configure the FortiOS Provider for FortiGate
provider "fortios" {
  hostname      = "192.168.52.177"
  token         = "jn3t3Nw7qckQzt955Htkfj5hwQ6jdb"
  insecure      = "false"
  cabundlefile  = "/path/yourCA.crt"
}
```

```
# Create a Static Route Item
```



# Terraform Provider

## 4.2 Un exemple

- A vous de jouer, on va entrer dans le vif du sujet on va utiliser le provider docker

[:https://registry.terraform.io/providers/kreuzwerker/docker/latest/docs](https://registry.terraform.io/providers/kreuzwerker/docker/latest/docs)

- on va donc installer docker sur votre debian et ensuite on va créer un nouveau projet terraform dans lequel on va créer un fichier main.tf qui instancie le provider docker





# Terraform -

## 5.1 Les ressources

Les ressources sont les éléments centraux de terraform

Chaque block de resource décrit des objets d'infrastructure comme les réseaux, les instances, les volumes , les polices de sécurité etc ....



# Terraform -

## 5.1 Les resources

La déclaration des resource se fait par défaut dans un fichier main.tf de votre projet c'est lui qui aura toutes les resources de votre infrastructure

la syntax d'une resource est simple par défaut

described later in this page.

```
resource "aws_instance" "web" {  
    ami          = "ami-a1b2c3d4"  
    instance_type = "t2.micro"  
}
```

A `resource` block declares a resource of a given type ("aws\_instance")



# Terraform - Les ressources

## 5.2 La syntaxe

- le prefix resource pour déclarer une resource
  - le type qui doit exister ici "aws\_instance"
  - le nom à donner à la resource: ce nom est utilisé pour accéder à la resource partout dans le fichier terraform
- described later in this page.

```
resource "aws_instance" "web" {  
  ami           = "ami-a1b2c3d4"  
  instance_type = "t2.micro"  
}
```

A `resource` block declares a resource of a given type ("aws\_instance")



# Terraform - Les ressources

## 5.2 La syntaxe

- Ce qui se trouve entre les accolades sont les configurations des arguments de cette resource
- Les arguments dépendent de la resource (ami & instance\_type)

described later in this page.

```
resource "aws_instance" "web" {  
  ami           = "ami-a1b2c3d4"  
  instance_type = "t2.micro"  
}
```

A `resource` block declares a resource of a given type ("aws\_instance")



# Terraform - Les ressources

## 5.3 Les types

- Chaque resource définit des types , il faut aller dans la documentation de chaque provider pour voir les types définis et disponible

```
resource "fortios_user_fsso" "trname1" {  
  name      = "fssos1"  
  port      = 32381  
  port2     = 8000  
  port3     = 8000  
  port4     = 8000  
  port5     = 8000  
  server    = "1.1.1.1"  
  source_ip = "0.0.0.0"  
  source_ip6 = "::"  
}  
  
resource "fortios_user_adgrp" "trname" {  
  name      = "user_adgrp1"  
  server_name = fortios_user_fsso.trname1.name  
}
```



# Terraform - Les ressources

## 5.4 Les Behavior

- Les behavior sont littéralement les comportements des resource lors de la planification
- lorsque l'on applique une configuration on fait des opération de CRUD (create, update , destroy), les behavior sont donc les règles qui définissent l'action qui va être choisie par terraform parmi les actions CRUD



# Terraform - Les ressources

## 5.4 Les Behavior

- Create:

Lorsque une resource existe dans la configuration mais n'est pas associé à un objet réel de l'infra existant

Dans l'exemple suivant si la resource "trname" n'est pas trouvée dans le fichier state elle est créée dans l'infra, ce qui veut dire qu'il est très important de systématiquement avoir son state à jour

Configure user groups.

### Example Usage

```
resource "fortios_user_group" "trname" {  
  company      = "optional"  
  email        = "enable"  
  expire       = 14400  
  expire_type  = "immediately"  
  group_type   = "firewall"  
  max_accounts = 0  
  mobile_phone = "disable"  
  multiple_guest_add = "disable"  
  name         = "s1"  
  
  member {  
    name = "guest"  
  }  
}
```



# Terraform - Les ressources

## 5.4 Les Behavior

- Destroy:

Lorsque une resource n'existe pas dans la configuration mais qu'elle est présente dans le state si par exemple ici on a une resource "trname1" dans le state mais pas dans le fichier de conf à droite, elle sera supprimée, je vous laisse imaginer à quel point il faut prendre des précautions

Configure user groups.

### Example Usage

```
resource "fortios_user_group" "trname" {  
  company      = "optional"  
  email        = "enable"  
  expire       = 14400  
  expire_type  = "immediately"  
  group_type   = "firewall"  
  max_accounts = 0  
  mobile_phone = "disable"  
  multiple_guest_add = "disable"  
  name         = "s1"  
  
  member {  
    name = "guest"  
  }  
}
```





# Terraform - Les ressources

## 5.4 Les Behavior

- Update:

Lorsque une resource existe des 2 côtés

et qu'une valeur d'argument à changée , on update

sur l'infrastructure

Configure user groups.

### Example Usage

```
resource "fortios_user_group" "trname" {  
  company      = "optional"  
  email        = "enable"  
  expire       = 14400  
  expire_type  = "immediately"  
  group_type   = "firewall"  
  max_accounts = 0  
  mobile_phone = "disable"  
  multiple_guest_add = "disable"  
  name         = "s1"  
  
  member {  
    name = "guest"  
  }  
}
```



# Terraform - Les ressources

## 5.4 Les Behavior

- Recreate:

Lorsque une resource doit être supprimée

parce qu'elle ne peut pas être modifiée (

si l'api du provider ne permet pas de le faire)



Configure user groups.

### Example Usage

```
resource "fortios_user_group" "trname" {  
  company      = "optional"  
  email        = "enable"  
  expire       = 14400  
  expire_type  = "immediately"  
  group_type   = "firewall"  
  max_accounts = 0  
  mobile_phone = "disable"  
  multiple_guest_add = "disable"  
  name         = "s1"  
  
  member {  
    name = "guest"  
  }  
}
```

# Terraform - Les ressources

## 5.4 Autres

- Les ressources peuvent être dépendantes les unes des autres, terraform s'arrange alors pour retrouver de lui-même les dépendances et de les jouer dans le bon ordre
- il est possible d'utiliser des providers locaux qui permettent d'effectuer des opérations directement sur la machine où se trouve terraform comme générer des clés ssh



# Terraform - La hiérarchie des fichiers

## 6.1 description

- Comme on l'a vu il y a différents objets dans terraform
- terraform se charge de lire les infos de tous les fichiers .tf (dont le main) présent dans le répertoire courant
- il est possible de tout mettre dans le fichier main.tf ou de splitter pour réutilisation du code
- on ne verra pas dans ce cours toutes les notions de terraform mais l'architecture ci-dessous reprend certaines bonnes pratiques



# Terraform - La hiérarchie des fichiers

## 6.1 Un exemple

```
.
├── README.md
├── main.tf
├── variables.tf
├── outputs.tf
├── resources.tf
├── provider.tf
├── terraform.tfvars
├── modules/
│   └── module1/
│       ├── README.md
│       ├── variables.tf
│       ├── main.tf
│       └── outputs.tf
```



# Terraform - La hiérarchie des fichiers

## 6.1 Un exemple

- le fichier readme pour la documentation
- le main.tf qui est le fichier principal
- variables.tf pour les variables (on verra plus tard)
- outputs.tf( même combat que pour les variables)
- provider.tf pour les providers
- ressources pour les gros projets avec plein de ressources
- modules (on abordera pas ce point)



# Terraform - La hiérarchie des fichiers

## 6.1 Un exemple

- A vous de jouer
- on va reprendre le projet de tout à l'heure et on va donc le modifier pour qu'il crée un container ubuntu sur docker



# Terraform - Les variables

## 7.1 Définitions

- Comme dans tous les langages de programmation les variables permettent d'ajouter du contenu dynamique , les variables dans ansible n'échappent pas à la règle
- Il y a plusieurs types de variables
  - input :comparable a des arguments de fonctions
  - output:comparable aux valeurs de retour d'une fonction
  - local: comparable aux variables à portée local(à l'intérieur d'une fonction )





# Terraform - Les variables

## 7.2 Déclarations

A l'intérieur d'un block variable

```
variable "image_id" {  
  type = string  
}  
  
variable "availability_zone_names" {  
  type    = list(string)  
  default = ["us-west-1a"]  
}  
  
variable "docker_ports" {  
  type = list(object({  
    internal = number  
    external = number  
    protocol = string  
  }))  
  default = [  
    {  
      internal = 8300  
      external = 8300  
      protocol = "tcp"  
    }  
  ]  
}
```



# Terraform - Les variables

## 7.2 Déclarations

Après le mot clés variable on a le nom de la variable, ces noms doivent être unique dans le même module

on a ensuite une liste d'attributs:

- le type
- la valeur par défaut
- la description
- la validation
- la nullité (spécifie si une variable peut être nulle)



# Terraform - Les variables

## 7.2 Déclarations

- Les types de variables:
  - string
  - number
  - bool
  - list
  - object




# Terraform - Les variables

## 7.2 Déclarations

- La description:

```
variable "image_id" {  
  type      = string  
  description = "The id of the machine image (AMI) to use for the server."  
}
```

Copy 



# Terraform - Les variables

## 7.2 Déclarations

- La validation:

```
variable "image_id" {  
  type      = string  
  description = "The id of the machine image (AMI) to use for the server."  
  
  validation {  
    condition     = length(var.image_id) > 4 && substr(var.image_id, 0, 4) == "ami-"  
    error_message = "The image_id value must be a valid AMI id, starting with \"ami-\"."  
  }  
}
```

Copy 




# Terraform - Les variables

## 7.2 Utilisation

- La variable une fois déclarée est accessible via le mot clé “var.<name>”:

```
resource "aws_instance" "example" {  
  instance_type = "t2.micro"  
  ami           = var.image_id  
}
```

Copy 



# Terraform - Les variables

## 7.2 Utilisation

- Les variables peuvent être définies à plusieurs endroits , on retiendra qu'on peut les mettre soit dans le fichier main.tf soit dans un fichier .tfvars avec la même syntax qu'un fichier tf

```
image_id = "ami-abc123"  
availability_zone_names = [  
    "us-east-1a",  
    "us-west-1c",  
]
```

Copy 



# Terraform - Les variables

## 7.2 Utilisation

- Il y a un ordre de surcharge des variables, le dernier endroit où une variable est définie est celui qui surcharge toutes les autres, dans l'ordre (similaire à ansible):
  - environnement variables
  - .tfvars
  - -var -var-file dans la ligne de commande terraform





# Terraform - Les autres concepts

## 8.1 Datasources

- Permet d'étendre les fonctionnalités de terraform afin qu'il puisse accéder à des informations en dehors de son scope
- Chaque provider peut définir des data sources



# Terraform - Les autres concepts

## 8.1 Datasources

- La syntaxe
  - mot clé data suivi du nom et des arguments

```
data "aws_ami" "example" {  
  most_recent = true  
  
  owners = ["self"]  
  tags = {  
    Name     = "app-server"  
    Tested   = "true"  
  }  
}
```

Copy 



# Terraform - Les autres concepts

## 8.1 Datasources

- Un exemple avec les ami aws

```
# Find the latest available AMI that is tagged with Component = web
data "aws_ami" "web" {
  filter {
    name   = "state"
    values = ["available"]
  }

  filter {
    name   = "tag:Component"
    values = ["web"]
  }

  most_recent = true
}
```

Copy




# Terraform - Les autres concepts

## 8.1 Datasources

- on filtre ici les images ami par tag “web” et qui sont available
- on utilise le résultat dans la définition de la resource

```
resource "aws_instance" "web" {  
  ami          = data.aws_ami.web.id  
  instance_type = "t1.micro"  
}
```

Copy 



# Terraform - Les autres concepts

## 8.2 Outputs

- Les outputs sont des types de variable un peu particulier, il permet d'afficher des informations sur le terraform apply qui vient d'être appliqué
- Elle peuvent donc être utilisées par un module enfant , parent



# Terraform - Les autres concepts

## 8.3 Provisioners

- Un provisioners est une façon d'étendre les fonctionnalités de terraform
- en effet terraform part du principe qu'il n'est pas et qu'il ne sera jamais parfait et qu'il ne peut pas gérer toutes les étapes de déploiement d'une infrastructure complète , pour cela il utilise des provisioners qui comble ce manque et s'appuie sur des logiciels tierce comme chef, puppet etc ...



# Terraform - Les autres concepts

## 8.3 Provisioners

- un exemple avec le provisioner local-exec pour récupérer l'adresse ip d'un serveur déployé , on pourrait même imaginer lancer un playbook ansible qui fait des actions de post-install sur les VMs

```
resource "aws_instance" "web" {  
  # ...  
  
  provisioner "local-exec" {  
    command = "echo ${self.private_ip} >> private_ips.txt"  
  }  
}
```

Copy 

local-exec



# Terraform - Les autres concepts

## 8.4 L'import

- Terraform est censé fonctionner pour gérer entièrement une infrastructure complète de manière indépendante , cependant il est possible d'importer une infrastructure existante avec donc de ressources qui ont été créés par d'autres biaux que terraform
- C'est généralement une bonne solution pour une transition vers terraform depuis une infra existante
- Attention terraform importe uniquement dans son fichier d'état (state) , il faut également écrire la ressource dans le module terraform pour qu'elle ne soit pas supprimée, il est donc conseillé de d'abord écrire la description des ressources dans le module , de faire l'import et de lancer un terraform plan pour vérifier que tout est ok





# Terraform - Les autres concepts

- A vous de jouer

nouveau projet on va toujours déployer sur kvm 2 serveurs ubuntu minimal un front-1 et un db-1 , tous les deux sur un même réseau privé (a vous de choisir le réseau , soit c'est une 2ème interface soit l'interface principal) accessible via une clé ssh définie en avance

vous lancerez le playbook depuis un terraform apply du cours précédent (ou un nouveau) qui viendra installer sur le front apache2 et sur db-1 postgresql

