

# Types of IAC Tools

Configuration Management



SALTSTACK

Server Templating



Provisioning Tools



- **Configuration Management** : Use général pour installer et configurer des applications sur des infrastructures existantes (serveurs, BDD et autres...). Ils utilisent une structure de code standard qui optimise et facilite la maintenance et la configuration de ces applications. Ils sont idempotents c'ad l'on peut exécuter le code plusieurs fois et que à chaque fois l'on ne modifiera que les données qui ont variées dans le code.
- **Server Templating** : Utilisés pour créer une image d'une machine spécifique ou d'un container, ces images contiendront au préalables toutes les applications et dépendances nécessaires. Ce sont des infra immuables
- **Provisioning tools** : utiliser pour provisionner une infrastructure ou ses composant en utilisant des manifests ou code déclaratif (simple declarative code)

Terraform use HCL (Hashicorp Configuration Language) to deploy resources.

The code in Terraform is declarative

Terraform à un cycle de vie que l'on peut résumer en 03 phases :

- **Init** : durant la phase d'initialisation, Terraform initialise le projet et identifie et télécharge les provisionner à utiliser pour déployer les ressources demandées.
- **Plan** : Pendant cette phase, Terraform construit le plan pour atteindre l'environnement souhaité
- **Apply** : ici, Terraform applique les changements nécessaires sur l'environnement cible afin qu'il cadre à l'environnement souhaité.

NB : **terraform show** permet d'afficher une liste détaillée des ressources existantes dans notre fichier de configuration

## 1) Les bases du HCL (Hashicorp configuration language)

Le HCL est un langage constitué de blocs contenant à leur tour des paramètres de type clé/valeur. Les blocs sont définis dans les parenthèses {} contenant des arguments de type clé/valeur représentant la configuration des données.

```
<block> <parameters> {  
    key1 = value1  
    key2 = value2  
}
```

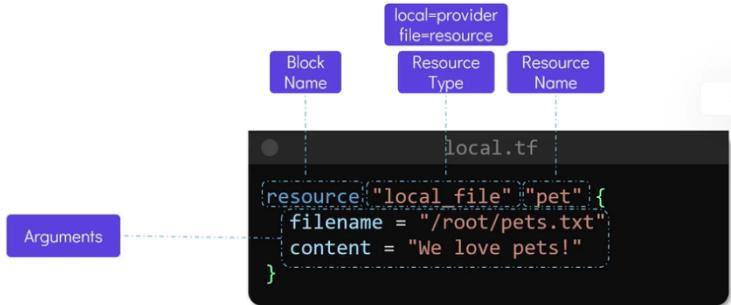
Ici dans cet exemple on a un bloc de type ressource identifié par le mot-clé « resource » au début du bloc, après le nom du bloc, on déclare le type de ressource que l'on veut créer, le type de ressource dépend du provider utilisé et après le on spécifie le nom de la ressource (le nom que terraform utilisera pour identifier la ressource)

**NB :** le nom du type de ressource fournit 02 informations :

**resource\_type= <nom du provider>\_<type de resource>**

Après avoir défini ces 03 paramètres, on entre dans le bloc

et on lui fournit des arguments ou paramètres obligatoires. Toute ressource dans Terraform a des arguments obligatoires qui devront toujours être défini lors de la création de ce type de ressource là.



## 2) Terraform Providers

Il existe 03 grands type de providers chez Terraform :

- **Official provider** : Ces providers sont fournis et maintenus par Hashicorp et comprennent les plugins pour les cloud providers tels que AWS, AZURE, GCP, ALICLOUD et LOCAL.
- **Verified provider** : ces providers ci sont fournis et maintenus par des entreprises tierces, mais sont vérifiés et validé par Hashicorp dans un process de partenariat avec ces entreprises-là. Exemple Bigip of F5, Heroku, digitalocean...
- **Community** : ces providers ou plugins sont publics et maintenus par des contributeurs individuels de Hashicorp

Lorsqu'on exécute la commande « Terraform Init », terraform initialise le projet et télécharge les fichiers nécessaires dans le répertoire plugins se trouvant dans le répertoire « .terraform » du projet

« <project\_file>/.terraform/plugins ». Le dossier du projet encore appelé « Configuration Directory » peut contenir plusieurs fichier \*.tf (configuration file), l'ensemble des fichiers de configuration \*.tf du dossier projet ou dossier de configuration utilisent le même fichier local d'état des ressources. Donc ce fichier local contiendra toutes les ressources déclarées des différents fichiers de configuration

## 3) Multiple providers

main.tf

```

resource "local_file" "pet" {
  filename = "/root/pets.txt"
  content = "We love pets!"
}

resource "random_pet" "my-pet" {
  prefix = "Mrs"
  separator = "."
  length = "1"
}
  
```

Il est possible d'utiliser plusieurs provider dans un seul fichier de configuration Terraform.

Dans notre exemple ci-contre, on utilise 02 providers différentes

- Le provider « local » identifié dans le nom de la ressource et
  - Le provider « random » qui est également un autre provider officiel de Hashicorp.
- Afin d'exécuter ce fichier il faudrait au préalable refaire un Terraform Init afin de télécharger les plugins nécessaires pour le bon fonctionnement du plugin random.

## 4) Les variables dans Terraform

main.tf

```

resource "local_file" "pet" {
  filename = var.filename
  content = var.content
}

resource "random_pet" "my-pet" {
  prefix = var.prefix
  separator = var.separator
  length = var.length
}
  
```

variables.tf

```

variable "filename" {
  default = "/root/pets.txt"
}
variable "content" {
  default = "My favorite pet is Mrs. Whiskers."
}
variable "prefix" {
  default = "Mrs"
}
variable "separator" {
  default = "."
}
variable "length" {
  default = "2"
}
  
```

Les blocs de variables peuvent supporter 03 arguments qui sont :

- Le Type : (string, number, bool, list..)
- La description
- la valeur par défaut « default »

Type	Example
string	"/root/pets.txt"
number	1
bool	true/false
any	Default Value
list	["cat", "dog"]
map	pet1 = cat pet2 = dog
object	Complex Data Structure
tuple	Complex Data Structure

```
variable "filename" {
  default = "/root/pets.txt"
  type = string
  description = "the path of local file"
}
```

```
variables.tf
variable "prefix" {
  default = ["Mr", "Mrs", "Sir"]
  type = list(string)
}
```

```
variables.tf
variable "prefix" {
  default = ["Mr", "Mrs", "Sir"]
  type = list(string)
}
```

```
variables.tf
variable file-content {
  type = map
  default = {
    "statement1" = "We love pets!"
    "statement2" = "We love animals!"
  }
}
```

```
maint.tf
resource "random_pet" "my-pet" {
  prefix = var.prefix[0]
}
```

Il est possible lors de la déclaration des variables de type liste, de forcer le type des éléments de la liste.

```
maint.tf
resource local_file my-pet {
  filename = "/root/pets.txt"
  content = var.file-content["statement2"]
}
```

```
variable "cats" {
  default = {
    "color" = "brown"
    "name" = "bella"
  }
  type = map(string)
}
```

```
variable "pet_count" {
  default = {
    "dogs" = "3"
    "cats" = "1"
    "goldfish" = "2"
  }
  type = map(number)
}
```

## Set

```
variables.tf
variable "prefix" {
  default = ["Mr", "Mrs", "Sir"]
  type = set(string)
}
```

```
variables.tf
variable "prefix" {
  default = ["Mr", "Mrs", "Sir", "Sir"]
  type = set(string)
}
```

```
variables.tf
variable "fruit" {
  default = ["apple", "banana"]
  type = set(string)
}
```

```
variables.tf
variable "fruit" {
  default = ["apple", "banana", "banana"]
  type = set(string)
}
```

```
variables.tf
variable "age" {
  default = ["10", "12", "15"]
  type = set(number)
}
```

```
variables.tf
variable "age" {
  default = ["10", "12", "15", "10"]
  type = set(number)
}
```

Les variables de type set sont identiques aux variables « list », à la seule différence que dans un set il n'est pas possible d'avoir plus d'une fois le même élément



```
variables.tf

variable "bella" {
  type = object({
    name = string
    color = string
    age = number
    food = list(string)
    favorite_pet = bool
  })
}

default = {
  name = "bella"
  color = "brown"
  age = 7
  food = ["fish", "chicken", "turkey"]
  favorite_pet = true
}
```

## Les Objets

Les variables de type objets permettent de créer des variables complexes capables de combiner tous les précédents types de variables abordés ci-dessus.

## Les Tuples

```
variables.tf

variable kitty {
  type      = tuple([string, number, bool])
  default   = ["cat", 7, true]
}
```

```
variables.tf

variable kitty {
  type      = tuple([string, number, bool])
  default   = ["cat", 7, true, "dog"]
}
```

Le dernier type de variable dans Terraform les « Tuples » fonctionne comme les listes, mais à la seule différence que dans un tuple les éléments peuvent être de type différents et dans la définition de la variable, il faudra préciser le type des différents éléments du tuples. Si jamais ces type d'éléments ne sont pas respectés lors de l'attribution des valeurs, alors lors de l'exécution, terraform renverra une erreur

Il est également possible dans terraform de lancer la commande « `terraform apply` » en passant en paramètres les valeurs des variables que l'on souhaite surcharger ou alors qui n'ont pas d'argument `default` ou de valeur par défaut définie.

```
$ terraform apply -var "filename=/root/pets.txt" -var "content=We love Pets!" -var "prefix=Mrs" -var "separator=." -var "length=2"
```

```
$ export TF_VAR_filename="/root/pets.txt"
$ export TF_VAR_content="We love pets!"
$ export TF_VAR_prefix="Mrs"
$ export TF_VAR_separator=". "
$ export TF_VAR_length="2"
$ terraform apply
```

Il est également possible de surcharger les variables dans terraform en les définissant dans les variables d'environnement précédé du préfixe « `TF_VAR_<Nom_Variable>` »

```
terraform.tfvars

filename = "/root/pets.txt"
content = "We love pets!"
prefix = "Mrs"
separator = ". "
length = "2"
```

On peut surcharger également en créant dans le répertoire de configuration ou répertoire du projet un fichier « `terraform.tfvars` » dans lequel on définira les valeurs des différentes variables qui seront utilisées. Il est également possible dans la même lancée d'utiliser un fichier avec extension « `*.auto.tfvars = variable.auto.tfvars` »

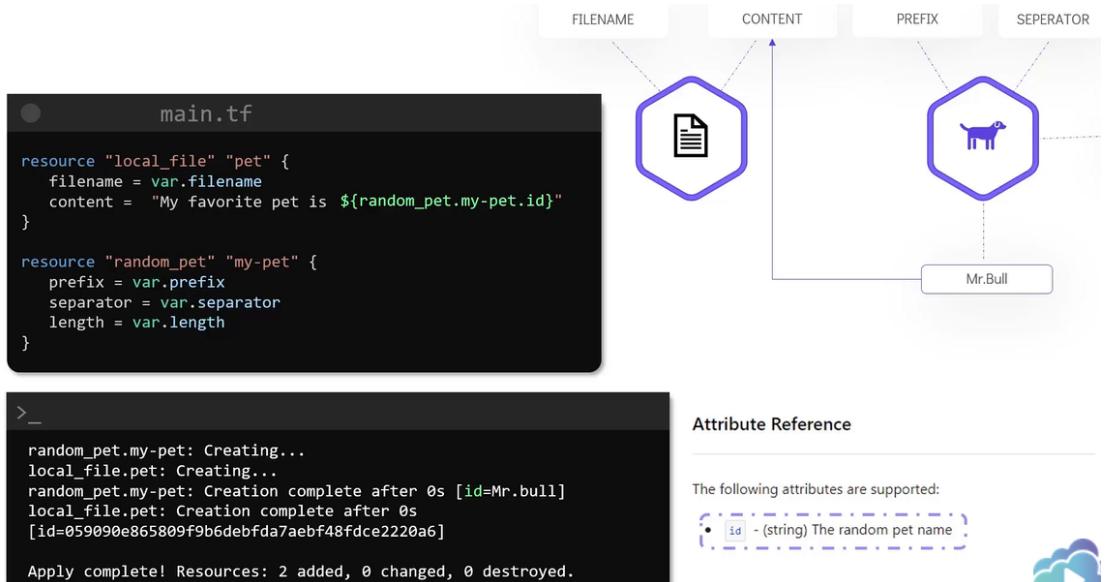
## Variable Definition Precedence

Order	Option
1	Environment Variables
2	<code>terraform.tfvars</code>
3	<code>*.auto.tfvars</code> (alphabetical order)
4	<code>-var</code> or <code>--var-file</code> (command-line flags)



Pour ce qui est de l'ordre de priorité lorsque diverses méthodes sont utilisées pour surcharger les. Ce sont les valeurs passées en paramètres dans la commandes `terraform apply` qui ont la plus grande priorité, ensuite celles du fichier `*.auto.tfvars`, après le fichier `terrafor.tfvars` et pour terminer les variables définie dans les variables d'environnement

## 5) Attributs de ressources



## 6) Les dépendances entre les ressources terraform

```
main.tf
resource "local_file" "pet" {
  filename = var.filename
  content = "My favorite pet is Mr.Cat"
  depends_on = [
    random_pet.my-pet
  ]
}

resource "random_pet" "my-pet" {
  prefix = var.prefix
  separator = var.separator
  length = var.length
}
```

Il existe 02 type de dépendances dans Terraform :

- **Les dépendances implicites** : il s'agit ici des dépendances constatées et créées par Terraform dans les cas où certains ressources auraient besoin des attributs d'autres ressources lors de leur création, alors dans ces cas terraform constate la dépendance et applique implicitement un ordre de dépendance qu'il utilisera pour créer les ressources. (Exemple précédent)
- **Les dépendances explicites** : dans ce cas, l'argument « depends\_on » doit être défini lors de la déclaration de la ressource et doit explicitement préciser le nom de la ressource dont elle dépend. L'argument depends\_on prend en paramètre une liste de ressource !!!

## 7) Les variables Outputs

```
resource "local_file" "pet" {
  filename = var.filename
  content = "My favorite pet is ${random_pet.my-pet.id}"
}

resource "random_pet" "my-pet" {
  prefix = var.prefix
  separator = var.separator
  length = var.length
}

output pet-name {
  value      = random_pet.my-pet.id
  description = "Record the value of pet ID generated by the random_pet resource"
}
```

```
>_
$ terraform output
pet-name = Mrs.gibbon

>_
$ terraform output pet-name
Mrs.gibbon
```

## 8) Terraform State

Le tfstate est un fichier « terraform.tfstate » que Terraform crée automatiquement lorsque des ressources sont créées qui contient la définition sous format Json de toutes les ressources créées dans le répertoire projet. Il enregistre toutes les ressources créées au fil de l'eau par Terraform. Chaque fois qu'une nouvelle commande « terraform plan ou terraform apply » est lancée, Terraform vérifie automatiquement le contenu du tfstate et le compare avec ce qui est

demandé afin de déterminer les actions à faire pour atteindre l'état souhaité. Lorsque terraform crée une ressources, il l'enregistre également de façon systématique dans le tfstate. Le terraform State n'est pas optionnel dans terraform, il est par défaut et obligatoire et de ce fait il y'a certaines considérations à prendre en compte :

- Le fichier terraform.tfstate contient des informations sensibles, il contient toutes les informations au détails près sur l'ensemble des ressources créées, de ce fait il est important de le stocker dans une safe zone
- Il n'est pas recommandé de stocker le fichier tfstate dans un outil de source control comme Git, GitLab
- Il est interdit de modifier manuellement le fichier tfstate étant donné que c'est un fichier JSON.

## 9) Terraform Commands

- **Terraform validate** : permet de vérifier et valider son code avant exécution, il vérifie tous les fichiers HCL (\*.tf) du répertoire de projet.
- **Terraform fmt** : formate votre code et le rend plus lisible
- **Terraform show** : permet d'afficher à l'écran l'ensemble des ressources créées et leurs attributs, il est possible d'ajouter « terraform show -json » afin d'avoir la liste des ressources au format JSON
- **Terraform providers** : permet d'avoir la liste de tous les providers utilisés dans le répertoire de configuration
- **Terraform providers mirrors /root/new\_terraform\_repo\_project** : permet de copier les plugins des providers du répertoire de configuration actuel vers un nouveau répertoire projet devant utiliser les mêmes providers.
- **Terraform output** : pour afficher toutes les variables output définies dans le fichier de configuration
- **Terraform refresh** : commande permettant de synchroniser terraform avec les ressources réelles déployées. Cette commande ne modifie pas les ressources, mais peut modifier le fichier terraform.tfstate, elle est lancée automatiquement par terraform lorsqu'on fait un terraform plan ou apply.
- **Terraform graph** : cette commande est utilisée pour afficher une représentation visuelle des différentes dépendances existantes dans votre répertoire de configuration

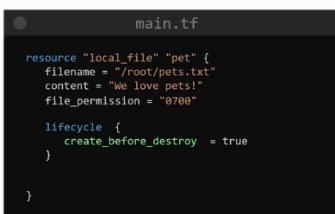
## 10) Terraform Mutable vs Immutable Infrastructure

Les infrastructures Mutable sont celles qui permettent des changements aux niveaux de certains de leurs ressources tandis que les infrastructures Immutables sont celles qui ne permettent pas les modifications sur leurs ressources, toutes modifications sur une ressource dans ce cas passe par la création d'une nouvelle ressource avec les nouveaux paramètres et la suppression de l'ancienne.

**NB** dans les infrastructures Immutables, il est fortement recommandé de ne procéder à la suppression de l'ancienne ressource si et seulement si la nouvelle ressource comportant les modifications a pu être déployé avec succès. Par défaut terraform supprime au préalable la ressource existante avant de créer la nouvelle, ce qui n'est pas bon car si jamais on rencontre des problèmes lors de la création de la nouvelle, on pourra plus faire de Rollback. Néanmoins il existe un moyen sur terraform permettant de forcer terraform à commencer par la création de la nouvelle ressource avant de supprimer l'ancienne. Pour cela, on utilise les règles de cycle de vie (Lifecycle Rules).

## 11) Lifecycle Rules

Il existe plusieurs règles de cycle de vie que l'on peut appliquer au niveau des ressources



```
main.tf
resource "local_file" "pet" {
  filename = "/root/pets.txt"
  content = "We love pets!"
  file_permission = "0700"
  lifecycle {
    create_before_destroy = true
  }
}
```

### 1) Create\_before\_destroy = true

Pour définir une règle de cycle de vie sur une ressource, on ajoute le bloc « lifecycle » dans son bloc « resource » et pour ce bloc « lifecycle », on définit l'argument « create\_before\_destroy » à True ; ce qui permettra de toujours créer la nouvelle ressource avant de supprimer l'ancienne.

```

main.tf

resource "local_file" "pet" {
  filename = "/root/pets.txt"
  content = "We love pets!"
  file_permission = "0700"

  lifecycle {
    prevent_destroy = true
  }
}

```

## 2) prevent\_destroy

cette règle permet d'empêcher toute modification entraînant la suppression de la ressource. Cela permet de sécuriser certaines ressources en empêchant leur modification ou suppression. Lorsqu'une modification entraînant la destruction de la ressource sera demandée, alors terraform refusera et renverra une erreur.

**NB :** La ressource pourra être détruite à partir de la commande « terraform destroy »

```

main.tf

resource "aws_instance" "webserver" {
  ami           = "ami-0edab43b6fa892279"
  instance_type = "t2.micro"
  tags = [
    Name = "ProjectA-Webserver"
  ]
  lifecycle {
    ignore_changes = [
      tags,ami
    ]
  }
}

```

## 3) ignore\_changes

Cette règle quant à elle permet d'empêcher la prise en compte des modifications sur un liste spécifique d'attributs de ladite ressource. Dans le cas où une modification est faite sur un des attributs de la liste, alors terraform ne prendra pas en compte cette modification.

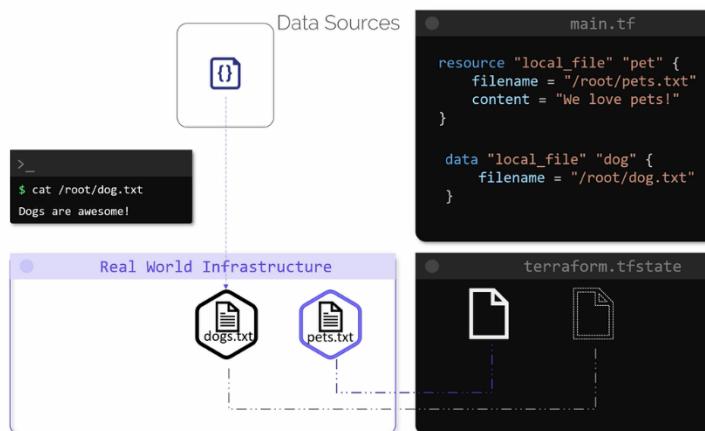
La valeur de l'argument « ignore\_changes =all » peut aussi être « all » dans ce cas terraform ne prendra en compte aucune modifications sur tous les attributs de ladite ressource

Order	Option	
1	create_before_destroy	Create the resource first and then destroy older
2	prevent_destroy	Prevents destroy of a resource
3	ignore_changes	Ignore Changes to Resource Attributes (specific/all)

## 12) Les Data Terraform

**sources**      **dans**

Les data sources permettent à Terraform de lire les attributs des ressources de notre environnement qui ont été créés manuellement, c'est-à-dire hors terraform. Grâce au data source, il est possible de récupérer et d'utiliser les attributs de ces ressources dans le but si nécessaire de créer un autre ressource à l'aide de Terraform.



Dans cet exemple, un fichier dogs.txt a été créé manuellement dans notre environnement réel, et dans ce cas pour utiliser certains de ses attributs, on le déclare dans un bloc de type « data » dans lequel on spécifie son argument « filename » afin que terraform sache où aller le chercher. Une fois déclarée à partir du bloc « data » on peut utiliser les attributs de cette ressource.

```

resource "local_file" "pet" {
  filename = "/root/pets.txt"
  content = data.local_file.dog.content
}

data "local_file" "dog" {
  filename = "/root/dog.txt"
}

```

**NB :** se rapprocher de la documentation officielle Hashicorp de chaque provider afin de se rassurer des réels arguments exposés par le bloc « datas » de chaque type de ressource.

Resource	Data Source
Keyword: <b>resource</b>	Keyword: <b>data</b>
Creates, Updates, Destroys Infrastructure	Only Reads Infrastructure
Also called <b>Managed Resources</b>	Also called <b>Data Resources</b>

ce tableau présente les différences significatives entre un bloc ressource et une data source

## 13) Meta Arguments dans Terraform

Les méta arguments sont des arguments spécifiques ou particuliers dans terraform pouvant être utilisés dans tout bloc de type « resource » dans un fichier de configuration Terraform afin de changer le comportement de ladite ressource lors de sa création. Il existe plusieurs type de méta Arguments parmi lesquelles celles déjà vues:

- **Depends\_on [ ]** : utilisés pour activer des dépendances explicites avec une liste de ressources
- **Lifecycle rules** : qui a des arguments permettant de définir comment les ressources seront créés, modifiés et supprimées.

ci-dessous nous parlerons de deux autres type de méta arguments de terraform utilisés pour les boucles et itérations dans Terraform

### a. Meta Argument : Count

Le méta argument « count » permet de créer de la manière la plus simple possible plusieurs ressources à la fois dans Terraform. Il suffit juste d'ajouter l'argument « count = valeur » dans le bloc de définition de la ressource afin créer autant de ressource de ce type fonction du nombre défini dans valeur.

Length Function

```
main.tf
resource "local_file" "pet" {
  filename = var.filename[count.index]
  count   = length(var.filename)
}

variables.tf
variable "filename" {
  default = [
    "/root/pets.txt",
    "/root/dogs.txt",
    "/root/cats.txt",
    "/root/cows.txt",
    "/root/ducks.txt"
  ]
}

>_
$ ls /root
pets.txt
dogs.txt
cats.txt
```

Avec count on crée plusieurs fois la ressource pet et chaque ressource est identifié sous forme d'une liste pet[0...2]. Afin de créer des ressources avec différents noms on peut utiliser le count.index et le coupler à la fonction **length** qui permet de retourner le nombre d'éléments d'une liste.

### b. Meta Argument : For-each

variable	function	value
fruits = [ "apple", "banana", "orange"]	length(fruits)	3
cars = [ "honda", "bmw", "nissan", "kia"]	length(cars)	4
colors = [ "red", "purple"]	length(colors)	2

Le meta argument for-each permet tout comme count de faire des itérations permettant également de créer plusieurs ressources dans un seul bloc de définition. Néanmoins l'argument « for-each » permet de résoudre les problèmes observés avec count car avec count, la ressource est créée sous forme de liste avec des index et cela peut causer des désagréments lors de la modification de ladite ressource. (La modification ou suppression d'un index avec count conduit à la suppression et au remplacement de cet index là par un nouveau, mais également au remplacement des autres car ils devront changer d'index dans la liste count). Avec for-each, ce problème est résolu car la modification d'un élément de sa liste n'impacte pas les autres éléments, c'est juste l'élément en question qui est modifié ou supprimé. Avec for-each, les ressources sont créées sous forme de map et non de liste, donc les index sont identifiés par la clé (clé/valeur) et non par un numéro devant toujours commencer par l'index 0.

L'attribut for-each ne peut prendre en paramètres que des variables de type set ou map donc si on veut lui passer en paramètres une liste, on doit utiliser la fonction « toset » qui permet de convertir une liste en set.

for\_each

```
main.tf
resource "local_file" "pet" {
  filename = each.value
  [for_each = toset(var.filename)]
}

variables.tf
variable "filename" {
  type = list(string)
  default = [
    "/root/pets.txt",
    "/root/dogs.txt",
    "/root/cats.txt"
  ]
}

>_
$ terraform plan
Terraform will perform the following actions:
* local_file "/root/cats.txt" will be created
+ resource "local_file" "pet" {
    + directory_permission = "0777"
    + file_permission = "0777"
    + filename = "/root/cats.txt"
    + id = "da39a3ee5e6b4b0d3255bfef95601890af80709"
  }
... (output trimmed)
```

```
$ terraform output
pets = {
  "/root/cats.txt" = {
    "directory_permission" = "0777"
    "file_permission" = "0777"
    "filename" = "/root/cats.txt"
    "id" = "da39a3ee5e6b4b0d3255bfef95601890af80709"
  }
  ...
  "/root/dogs.txt" = {
    "directory_permission" = "0777"
    "file_permission" = "0777"
    "filename" = "/root/dogs.txt"
    "id" = "da39a3ee5e6b4b0d3255bfef95601890af80709"
  }
}
```

## 14) Provider versions for version constraints

Il est important de définir la version du provider à utiliser lors de la définition de son fichier \*.tf, car il existe des différences drastiques entre chaque version pouvant créer des différences de comportements de vos ressources. Cela se fait dans la section provider du fichier à partir du champ « version »

```
main.tf
terraform {
  required_providers {
    local = {
      source = "hashicorp/local"
      version = "1.4.0"
    }
  }

  resource "local_file" "pet" {
    filename  = "/root/pet.txt"
    content   = "We love pets!"
  }
}
```

Overview Documentation USE PROVIDER

How to use this provider  
To install this provider, copy and paste this code into your Terraform configuration. Then, run terraform init.  
Terraform 0.13 [Latest]

```
terraform {
  required_providers {
    local = {
      source = "hashicorp/local"
      version = "1.4.0"
    }
  }
}
```

```
provider "aws" {
  region     = "us-west-2"
  version    = "2.7"
}
```

Version Number Arguments	Description
<code>&gt;=1.0</code>	Greater than equal to the version
<code>&lt;=1.0</code>	Less than equal to the version
<code>~&gt;2.0</code>	Any version in the 2.X range.
<code>&gt;=2.10,&lt;=2.30</code>	Any version between 2.10 and 2.30

Exemple : `version = ">=1.0"` # version à utiliser supérieure ou égale à 1.0.

## 15) Remote State

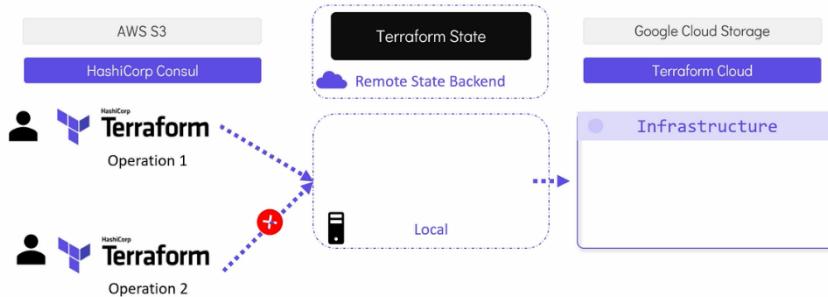
Le fichier terraform.tfstate offre plusieurs avantages qui sont définis dans l'image ci-contre.

Toutefois afin de permettre la collaboration, il est important que le fichier terraform.tfstate soit stocké sur un espace sécurisé partagé et non en local sur une machine. Pour permettre le partage d'un fichier terraform.tfstate entre les membres d'un équipe, on utilise le bloc « backend » dans le bloc « terraform »



```
>_
$ ls
main.tf variables.tf terraform.tfstate
```

### State Locking

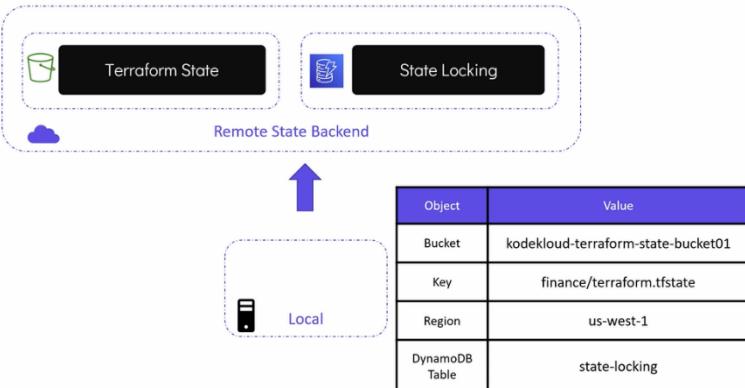


Automatically Load and Upload State File

Many Backends Support State Locking

Security

### Remote Backend



```
terraform {
  backend "s3" {
    bucket      = "kodekloud-terraform-state-bucket01"
    key         = "finance/terraform.tfstate"
    region      = "us-west-1"
    dynamodb_table = "state-locking"
  }
}
```

Lorsqu'un terraform plan ou apply est réalisé par un des membres de l'équipe, alors le fichier tfstate passe en mode bloqué et ne permet plus l'accès à d'autres users. Une fois les ressources appliqués et le tfstate modifié, le fichier tfstate se débloque

## 16) Terraform state commands

```
>_  
  
$ vi terraform.tfstate  
$ terraform state show aws_s3_bucket.finance  
# terraform state <subcommand> [options] [args]
```

Sub-command
list
mv
pull
rm
show

**1) Terraform List :** permet de lister toutes les ressources créées (cette commande affiche juste l'adresse de chaque ressource et aucun autre attribut)

```
# terraform state list [options] [address]  
$ terraform state list  
aws_dynamodb_table.cars  
aws_s3_bucket.finance-2020922  
  
$ terraform state list aws_s3_bucket.finance-2020922  
aws_s3_bucket.finance-2020922
```

```
# terraform state show [options] [address]  
$ terraform state show aws_s3_bucket.finance-2020922  
resource "aws_s3_bucket" "terraform-state" {  
    acl           = "private"  
    arn           = "arn:aws:s3::: finance-2020922 "  
    bucket        = "finance-2020922"  
    bucket_domain_name = "finance-2020922.s3.amazonaws.com"  
    bucketRegionalDomainName = " finance-2020922.s3.us-west-1.amazonaws.com"  
    force_destroy = false
```

**2) Terraform show :** cette commande permet d'avoir le détail sur les différents attributs des ressources créées.

**3) Terraform mv :** cette commande est utilisée pour déplacer des objets dans le fichier terraform.tfstate. les objets

ne peuvent être déplacées que vers le même répertoire projet, ce qui revient en réalité à juste modifier l'objet par exemple changer de nom...

Dans cet exemple, on modifie le nom de la ressource « state-locking » à « state-locking-db » en passant par le tfstate file.

NB : après cela il faudrait également renommer la ressource dans son fichier de configuration afin que Terraform ne détecte aucune modifications à appliquer lors du prochain terraform apply.



```
>_  
  
# terraform state mv [options] SOURCE DESTINATION  
$ terraform state mv aws_dynamodb_table.state-locking aws_dynamodb_table.state-locking-db  
Move "aws_dynamodb_table.state-locking" to "aws_dynamodb_table.state-locking-db"  
Successfully moved 1 object(s).
```

**4) Terraform pull :** utilisée pour récupérer le fichier tfstate qui peut être situé sur un remote backend afin de le sauvegarder en JSON dans un fichier ou l'afficher à l'écran.

```
main.tf provider.tf  
# terraform state pull [options] SOURCE DESTINATION  
$ terraform state pull  
{  
  "version": 4,  
  "terraform_version": "0.13.0",
```

```
# terraform state rm ADDRESS  
$ terraform state rm aws_s3_bucket.finance-2020922  
Acquiring state lock. This may take a few moments...  
Removed aws_s3_bucket.finance-2020922  
Successfully removed 1 resource instance(s).  
Releasing state lock. This may take a few moments...
```

**5) Terraform rm :** utilisée pour supprimer un objet du fichier tfstate. Bien entendu, il faudrait également supprimer sa déclaration dans le fichier de configuration afin que terraform ne détecte aucune modification à appliquer au prochain terraform plan ou apply.

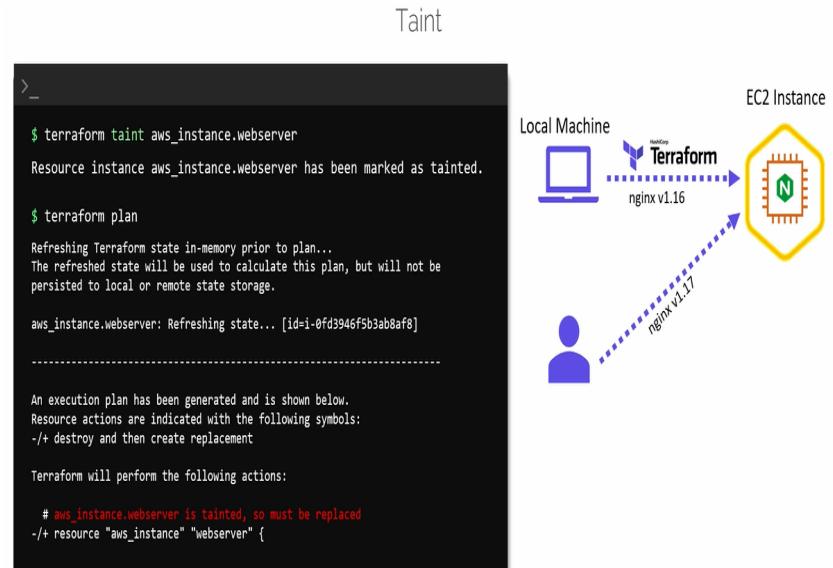
**NB :** le fait de supprimer une ressource du tfstate ne la supprime pas de votre infrastructure réelle, il la retire juste de la gestion de Terraform.

## 17) Terraform Import, Tainting resources and debugging

### a) Terraform taint

Lorsque vous essayez de créer une ressource avec terraform et que cela ne fonctionne pas car lors de sa création une erreur survient empêchant la création de ladite ressource, alors terraform marque ou teinte ladite ressource afin de la recréer entièrement la prochaine fois qu'on fera un terraform apply. Vous pouvez le constater si vous refaites un terraform plan à nouveau. Il est possible de marquer manuellement une ressource afin que cette dernière soit recréer au prochain terraform apply et ce peu importe son état elle sera recréée et remplacée par terraform. On utilise la commande :

« `terraform taint <nom_de_la_ressource>` »



```
$ terraform untaint aws_instance.webserver
Resource instance aws_instance.webserver has been successfully
untainted.
```

Pour retirer la teinte, on utilise la commande :  
« `terraform untaint <nom_de_la_ressource>` »

### b) Terraform Debugging

Cette partie concerne le debugging dans terraform. Quand les ressources ne fonctionnent pas comme souhaité lors d'un terraform apply, il est possible d'aller regarder dans les fichier logs de terraform afin de rechercher la cause de ce dysfonctionnement. Pour cela, il faut à partir des variables d'environnements, définir le niveau de logs que vous souhaitez avoir à travers la variable TF\_LOG. Terraform offre 05 niveaux de logs qui sont :



En fonction du niveau de logs demandés au travers de la variable d'environnement TF\_LOG, vous aurez plus ou moins de logs affichés à l'écran chaque fois que vous lancerez une commande terraform. TRACE, est le niveau de logs le plus détaillé qui affiche une multitude de logs lors de l'exécution des commandes TERRAFORM.

Pour sauvegarder les logs dans un fichier, on utilise la variable d'environnement TF\_LOG\_PATH.

```
$ export TF_LOG_PATH=/tmp/terraform.log
```

NB : pour exporter les logs on doit en plus de configurer le TF\_LOG\_PATH configurer également le TF\_LOG afin de sélectionner un niveau de logs

### c) Terraform Import

Permet d'importer une infrastructure existante (hors terraform) dans un fichier de configuration terraform (pour que ces ressources existantes puissent dorénavant être gérées par Terraform ) à l'aide des commandes terraform import.

```
# terraform import <resource_type>.<resource_name> <attribute>
$ terraform import aws_instance.webserver-2 i-026e13be10d5326f7
Error: resource address "aws_instance.webserver-2" does not exist
in the configuration.

Before importing this resource, please create its configuration in
the root module. For example:

resource "aws_instance" "webserver-2" {
    # (resource arguments)
}
```

pour ce faire on utilise la commande « terraform import » dans laquelle on spécifie le type de ressource avec un de ses attributs uniques tels que son adresse ou son id afin de correctement l'identifier lors de l'import.

NB : cette commande ne crée pas la ressource dans le fichier de configuration à votre place, elle modifie juste le tfstate en incluant cette ressource. Donc vous devez par la suite manuellement modifier votre \*.tf afin de l'y inclure.

## 18) fonctions Terraform

Pour l'instant, nous avons déjà vu les fonctions suivantes :

- **file (file\_path)** : qui permet de lire le contenu d'un fichier passé en paramètres et le récupérer afin de l'affecter à un autre attribut.

```
policy = file("admin-policy.json")
```

- **Length** : qui permet d'avoir le nombre d'éléments d'une variable de type liste par exemple

- **Toset** : qui permet de convertir une variable de type liste en une variable de type set.

Terraform possède également une console interactive permettant de tester ses différentes fonctions avant leur utilisation, pour entrer dans cette console, on tape la commande « *terraform console* ». cette console charge par défaut l'état des ressources et des variables du répertoire projet, permettant ainsi de pouvoir interagir avec ces dernières dans l'environnement de la console.

```
$ terraform console
>file("/root/terraform-projects/main.tf")
resource "aws_instance" "development" {
  ami           = "ami-0edab43b6fa892279"
  instance_type = "t2.micro"
}
> length(var.region)
3
> toset(var.region)
[
  "ca-central-1",
  "us-east-1",
]
]
```

Il existe plusieurs catégories de fonctions disponibles dans terraform parmi lesquelles les plus utilisées qui sont :

**1) Numeric functions** : La catégorie de fonctions numériques possède les fonctions utilisées pour manipuler et transformer les nombres. Exemple.

- **max** et **min** qui sont utilisés pour déterminer le plus grand ou plus petit nombre d'une liste
- **ceil** : cette fonction prend en paramètre un nombre décimal et retourne l'entier le plus proche et supérieur à ce nombre. (arrondi par excès)
- **floor** : pareil que ceil mais à la place retourne plutôt le nombre le plus proche et inférieur (arrondi par défaut)

```
$ terraform console
> max (-1, 2, -10, 200, -250)
200
> min (-1, 2, -10, 200, -250)
-250
> max(var.num...)
250
> ceil(10.1)
11
> ceil(10.9)
11
> floor(10.1)
10
> floor(10.9)
10
```

**2) String functions** : fonctions utilisées pour manipuler et transformer le texte

- **split** : qui prend en paramètre un séparateur et un texte par la suite sépare le texte en utilisant ce séparateur afin de créer une liste avec comme éléments les mots du texte séparés.
- **lower(text)** : permet de mettre le texte passé en paramètre en minuscules
- **upper(text)** : le texte passé en paramètres est mis en majuscules
- **title(text)** : permet de mettre en majuscule juste la première lettre de chaque mot
- **susbstr(text, i, n)** : permet d'extraire un mot d'une phrase passée en paramètre en précisant l'index de démarrage et le nombre de caractère à extraire à partir de cet index.
- **join (",", list(str))** : c'est le contraire de split, il prend en paramètre un séparateur et une liste de string et les relie pour ne former qu'une phrase.

```
$ terraform console
> split(",","ami-xyz,AMI-ABC,ami-efg")
[ "ami-xyz", "AMI-ABC", "ami-efg" ]
> split(",", var.ami)
[ "ami-xyz", "AMI-ABC", "ami-efg" ]
> lower(var.ami)
ami-xyz,ami-abc,ami-efg
> upper(var.ami)
AMI-XYZ,AMI-ABC,AMI-EFG
> title(var.ami)
Ami-xyz,AmI-ABC,Ami-efg
> substr(var.ami, 0, 7)
ami-xyz
> substr(var.ami, 8, 7)
AMI-ABC
> substr(var.ami, 16, 7)
ami-efg
```

**3) Collection functions** : fonctions utilisées pour collecter les type données sur les éléments tel que les set, les listes et les map.

- **length(list[])** : renvoi le nombre d'élément de la liste passée en paramètre
- **index(liste, « mot »)** : permet de renvoyer l'index du mot passé en paramètre dans le liste également passée en paramètres
- **element(list[], index)** : renvoi la valeur de l'élément situé à l'index passé en paramètre dans la liste.
- **Contains(list,mot)** : vérifie si l'élément ou le mot passé en paramètre est présent dans la liste et renvoie un booléen (true or false)

```
$ terraform console
> length(var.ami)
3
> index(var.ami, "AMI-ABC")
1
> element(var.ami,2)
ami-efg
> contains(var.ami, "AMI-ABC")
true
> contains(var.ami, "AMI-XYZ")
false
```

**4) Map functions** : fonctions utilisées sur les variables de type maps

- **Keys(map)** : permet de convertir les clés d'une map en liste ( juste les clés sont mises dans une liste)
- **Values(map)** : permet à son tour de convertir juste les valeurs de la map passée en paramètre en tant que List.
- **Lookup(map, key)** : permet d'afficher la valeur de la clé passée en paramètre dans cette map.

> lookup (var.ami, "us-west-2", "ami-pqr") On peut passer un 3<sup>e</sup> argument à cette commande afin qu'il soit retourné si jamais la clé passé en paramètre n'existe pas dans cette map

```
$ terraform console
> keys(var.ami)
[
  "ap-south-1",
  "ca-central-1",
  "us-east-1",
]
> values(var.ami)
[
  "ami-ABC",
  "ami-efg",
  "ami-xyz",
]
> lookup(var.ami, "ca-central-1")
ami-efg
```

## 19) Expression conditionnelles

Il est également possible d'utiliser la console interactive terraform pour tester le fonctionnement des expressions conditionnelles avant de les utiliser dans nos différents fichiers de configuration. Terraform supporte tous les opérateurs numériques basiques existants (A+B ; A-B ; A\*B ; A/B ) et également des opérateurs d'égalité et de

comparaison qui sont ( $A==B$  ;  $A != B$  ;  $A < B$  ;  $A > B$  ;  $A \leq B$  ;  $A \geq B$ ), bien entendu les opérations de comparaison ou d'égalité produisent des booléens comme résultat. On distingue également des opérateurs logiques ( $A \& \& B$  ;  $A | | B$  ;  $!(NOT)$ ).

L'attribution d'une valeur à une variable de façon conditionnelle se fait de la manière suivante dans terraform :

```
resource "random_password" "password-generator" {
    length = var.length < 8 ? 8 : var.length
}

resource "aws_instance" "dev" {
    ami = "ami-0083662ba17882949"
    instance_type = "t2.micro"
    count = var.istest == true ? 3 : 0
}

resource "aws_instance" "prod" {
    ami = "ami-0083662ba17882949"
    instance_type = "t2.large"
    count = var.istest == false ? 1 : 0
}

variable "istest" {}
```

condition ? true\_val : false\_val

Dans cet exemple, on utilise la variable « istest » qui est une variable booléenne qui nous permettra de dynamiser notre déploiement.

Ici on voudra créer des ressources aws instances différentes fonction de si l'on se trouve en environnement de test ou de prod. En environnement de test, la valeur de count prendra la valeur 3 si « istest=true » et 0 sinon donc dans le cas où on n'est pas en environnement de test, la ressource « dev » ne sera pas créée car istest=false.

Idem pour le test lors de la création de la ressource « prod »  
*Count = var.istest==false ? 1 : 0 # la condition (var.istest==false) détermine la valeur de count.*

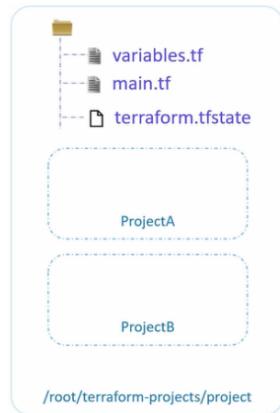
## 20) Terraform Workspaces (OSS)

Les Workspace permettent de créer des environnements virtuels dans terraform dans lequel il est possible de créer des ressources avec des attributs différents d'un environnement à un autre. De ce fait chaque Workspace a son fichier tfstate ressortant l'état des ressources dans ce Workspace-là. Ce concept de Workspace de terraform permet de créer des ressources différentes pour plusieurs projets à partir du même répertoire projet, cela permet de simplifier le code et ne demande pas à créer autant de répertoire projet que d'environnement. Exemple : si vous avez un environnement test et un environnement projet qui doivent tous les deux utiliser les mêmes ressources mais avec des caractéristiques différentes, plutôt que de créer deux répertoire projet pour chacun de vos environnements, terraform propose d'utiliser le concept de Workspace dans lequel vous n'aurez qu'un seul et même répertoire projet mais dans ce répertoire projet vous pouvez créer des Workspace vous permettant de créer des ressources avec des attributs différents par environnement en variabilisant les valeurs de ces attributs fonction du Workspace dans lequel vous vous trouvez.

Pour manipuler les Workspace dans terraform, on utilise les commandes suivantes :

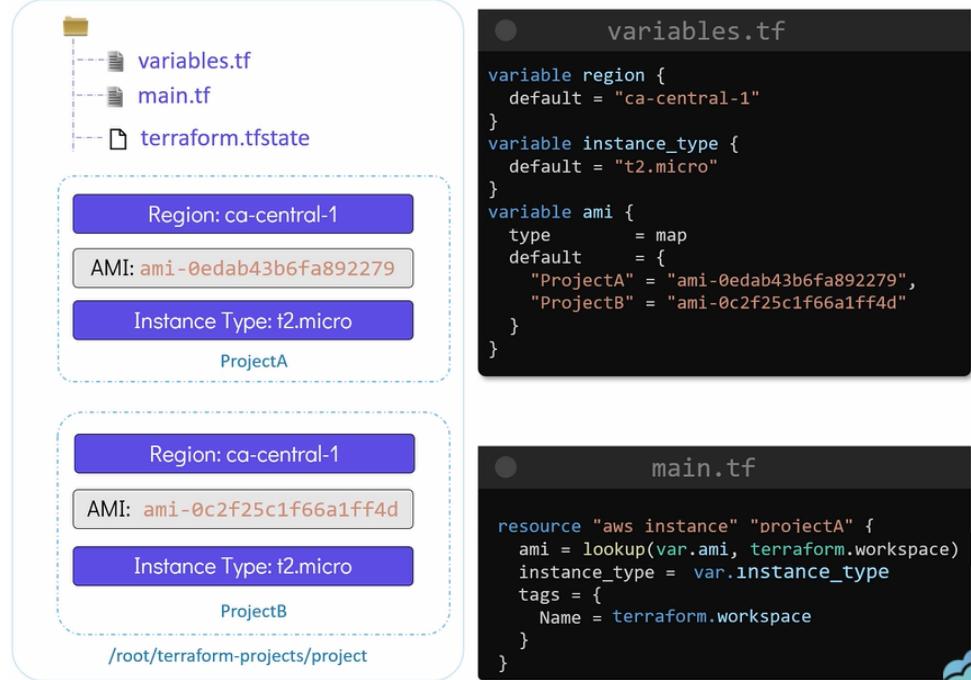
- **Terraform Workspace new <nom\_du\_nouveau\_workspace>** : permet de créer un nouveau Workspace et de switcher immédiatement vers ce Workspace.
- **Terraform Workspace list** : lister les workspace existants (avec un « \* » sur le Workspace utilisé)
- **Terraform Workspace select <nom\_du\_wp>** : permet de switcher vers le Workspace passé en paramètres.

Ci-dessous la méthode de variabilisation des attributs des ressources afin d'avoir des valeurs différentes fonction du Workspace dans lequel on se situe. Par défaut, il existe la variable « **terraform.workspace** » qui renvoi le nom du Workspace en cours dans



```
$ terraform console
> terraform.workspace
ProjectA
> lookup(var.ami, terraform.workspace)
ami-0edab43b6fa892279
```

lequel on se situe, cette variable est utilisée pour changer le comportements des ressources selon le Workspace dans lequel on se situe.



Dans cet exemple, on veut créer deux instances Ec2 dans 02 environnements différents, pour cela au lieu d'avoir 02 répertoires projets terraform différents, on utilise un seul et même répertoire couplé à la notion de Workspace. La différence au niveau des EC2 réside dans l'image qu'elles utiliseront.

On utilise une variable de type map dans laquelle on crée des clés correspondants aux noms des environnements avec pour valeurs respectives leur AMI respectifs et à l'aide de la fonction « `lookup` » et `terraform.workspace`, on récupère l'AMI exact fonction du Workspace dans lequel on se trouve. Et pour variabiliser le tag `Name` de chaque instance fonction de l'environnement, on utilise la variable `terraform.workspace`

Concernant le fichier `tfstate` lorsqu'on travail dans un environnement à plusieurs Workspace, terraform crée un nouveau répertoire appelé « `terraform.tfstate.d` » dans lequel il crée plusieurs autres répertoires distincts avec le nom de chaque Workspace existant et dans ces répertoires il met le fichier `terraform.tfstate` de chaque Workspace.

NB : lorsque l'on lance terraform, il crée le Workspace par défaut appelé « `default` » et c'est à l'intérieur qu'il crée les ressources car aucun autre Workspace n'existe à ce moment-là.

```
$ ls
main.tf provider.tf terraform.tfstate.d variables.tf

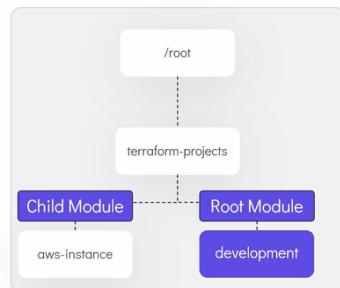
$ tree terraform.tfstate.d/
terraform.tfstate.d/
|-- ProjectA
|   '-- terraform.tfstate
|-- ProjectB
|   '-- terraform.tfstate
2 directories, 2 files
```

## 21) Terraform modules

Le concept de module a été créé pour résoudre les problèmes de limites de réutilisabilité du code ; en effet pour créer une ressource, on a besoin de la déclarer dans un bloc dans lequel on créera cette ressource et ceci pour le nombre de ressource que l'on souhaite déployer. Maintenant que se passe-t-il lorsqu'on souhaite réutiliser ce code dans un autre projet, sans le concept de module, il fallait aller copier les lignes de déclaration de la ressource et coller cela dans le fichier de configuration du nouveau projet, on se rend très vite compte que ça fait perdre beaucoup de temps et aussi qu'il y'a d'énormes risques d'erreurs qui peuvent survenir. C'est pour répondre à ce problème que Terraform a créé le concept de module.

```
>_
$ mkdir /root/terraform-projects/development
main.tf

main.tf
module "dev-webserver" {
  source = "../aws-instance"
}
```



Dans Ansible, le root module correspond en effet au répertoire projet dans lequel on se place pour exécuter les commandes terraform. Et si jamais dans ce root module on appelle ou utilise le bloc module pour appeler un autre répertoire de configuration terraform, alors ce répertoire distant ou tiercé est appelé child module. Le bloc module permettra d'exécuter tous les fichiers de configuration `*.tf` qui se trouvent dans le child module, créant ainsi toutes les ressources y étant déclarées.

**NB :** lors de l'appel d'un module à partir du bloc module, on peut surcharger les variables définies dans ce module-là.

Complex Configuration Files

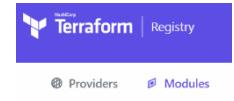
Duplicate Code

Increased Risk

Limits Reusability

### a) Modules from terraform registry

Le registry terraform en plus d'être utilisé pour le téléchargement des plugins des différents providers, possède également des modules pouvant être utilisés automatiquement dans nos fichiers de configuration.



Les modules dans le registre terraform sont regroupé fonction des différents providers de ces modules et sont utilisables par tous. Il sont regroupés en 02 grands types :

- **Les verified modules** : sont testés et maintenus par Hashicorp et ont un grand T (terraform) bleu.
- **Les community modules** : sont des modules publiés par des membres de la communauté mais qui ne sont pas vérifiés et validés par Terraform

Pour utiliser le module, il faut aller dans le registry terraform et effectuer la recherche du module fonction du nom de la ressource que vous souhaitez créer à partir d'un module, terraform vous affichera la liste des modules vérifiés et communautaires répondant à votre besoin. Chaque module à une aide montrant comment l'utiliser dans un fichier de configuration terraform.

The screenshot shows a code editor with a dark theme containing a Terraform configuration file named 'main.tf'. The code defines a module 'security-group\_ssh' with a source of 'terraform-aws-modules/security-group/aws/modules/ssh', version '3.16.0', and various configuration parameters like 'vpc\_id', 'ingress\_cidr\_blocks', and 'name'. To the right, a 'Provision Instructions' panel is displayed, showing the module definition again with a note to copy and paste it into a Terraform configuration, insert variables, and run 'terraform init'.

## 22) Terraform multiregion

Ce concept de multiregion permet de créer à l'aide d'un seul répertoire de configuration des ressources dans plusieurs régions (cloud providers regions) à la fois. Pour ce faire, on définit les différents blocs providers en spécifiant pour chaque bloc la région utilisée ; on définit également l'argument alias qui permet de différencier les providers les uns des autres et plus tard lors de la définition du bloc ressource, on précisera l'alias du provider à utiliser à travers l'argument « provider » dans le bloc « resource ».

The screenshot shows a code editor with a dark theme containing a Terraform configuration file. It includes two 'resource' blocks for 'aws\_eip'. The first block uses the default provider ('aws'), while the second block specifies a provider alias ('aws.aws02'). Both blocks have the 'vpc' argument set to 'true'.

Dans cet exemple, on définit deux configurations pour le provider aws, l'un avec l'argument alias pour les différencier. Et lors de la déclaration des ressources, lorsque l'argument « provider » n'est pas fourni, Terraform utilisera le provider par défaut (celui où on n'a pas défini l'alias) et si on veut utiliser un provider particulier, on le spécifie à travers de l'argument provider en fournissant l'adresse de l'alias (<provider>.<alias> = aws.aws02).

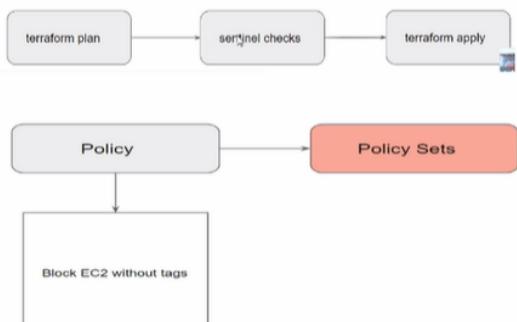
## 23) Terraform Cloud

Il s'agit d'un terraform entièrement managé directement par Hashicorp. Terraform cloud en plus de la partie Terraform managé possède également plusieurs autres fonctionnalités :

- **L'accès control** : possibilité de filtrer l'accès à notre plateforme terraform cloud en définissant des utilisateurs avec des droits
- **Private registry** : terraform cloud possède également un registre privé vous permettant de stocker vos propres modules privés
- **Policy control** : permet de définir des politiques de contrôle sur vos fichiers de configuration afin d'assurer un minimum de conformité (exemple de policy : Empêcher toute ressource déclarée sans tag d'être déployée)
- **Evaluation de coûts** : il permet d'évaluer les couts de vos ressources sur les plateformes cloud

## A) Sentinel

Il s'agit d'un composant de terraform cloud qui permet de garantir un certain niveau de conformité de notre infrastructure. Il est mis en place afin de s'assurer de la qualité et du respect des normes en vigueur dans notre entreprise des ressources à déployer.



Lorsqu'on configure un projet sur terraform Cloud et que l'on définit des Policy à partir de sentinel, ce workflow pour le déploiement est le suivant :

Après chaque Terraform plan, l'étape suivante est celle des vérification sentinel quant à la cohérence des ressources à dévoyée par rapport aux règles ou Policy de déploiement mis en place. Et une fois et seulement si ces règles sont Ok, les ressources sont déployées à partir de terraform apply.

La sentinel est constituée effectivement de règles de sécurité appelées policies et un ensemble de policies correspond à un Policy sets. Donc lors de la définition d'une sentinel, on devra la rattacher à un Policy sets. Donc l'ordre est le suivant :

- On crée la Policy
- On met la Policy dans la Policy sets
- Et on crée une sentinel à laquelle on rattache la précédente Policy sets

NB : La sentinel c'est une option payante dans Terraform Cloud

Tips à lire entièrement avant l'examen

<https://medium.com/bb-tutorials-and-thoughts/250-practice-questions-for-terraform-associate-certification-7a3ccebe6a1a>