



Terraform pour le pilotage d'OpenStack, développer son infrastructure cloud

Mawaki PAKOUPETE



\$ whoami ?

- Mon nom est ***Mawaki PAKOUPETE***
- Ingénieur Cloud & DevOps
 - ◆ Débutant dans l'administration système : Unix/Linux, Virtualisation...
 - ◆ Ingénieur systèmes & DevOps (Docker, Kubernetes, Cloud AWS, GCP, Azure, Terraform...)
 - ◆ Ingénieur cloud travaillant pour une entreprise cloud qui utilise fortement les outils DevOps
- Certifications : Elasticsearch, CKS, CKA, AWS, RHCE, RHCSA, VMware LPIC-1, ITIL
- Passionné par la formation depuis 7 ans



Ice Breaker

- Prénom & Nom
- Entreprise actuelle & Rôle
- Attentes de la formation



\$ need --help

Comment demander de l'aide ?

- Cours qui se veut interactif
- Pour toute question, levez la main et posez là
- N'importe quel moment
- Vous pouvez utiliser le Chat également



Objectif de la formation

- Orchestrer des déploiements d'infrastructure avec Terraform
- Utiliser Terraform pour respecter les standards de l'Infrastructure as Code
- Structurer leurs projets pour les réutiliser efficacement
- Organiser leurs équipes pour travailler de concert autour de Terraform
- Reprendre en main une infrastructure AWS existante pour la faire évoluer avec Terraform
- Identifier les apports de solutions d'orchestration et d'automatisation

Prérequis :

- Connaissances Amazon Web Services de base.



Contenu du Cours

Introduction à Terraform

Infrastructure as Code
Qu'est-ce que Terraform ?
Pourquoi utiliser Terraform ?
Terraform VS Ansible
Architecture de Terraform
IaC avec Terraform
Introduction à HCL

Lab1 :

- Mise en place de votre environnement
- Découverte de la ligne de commande Terraform

Principes fondamentaux de Terraform

CLI de Terraform
Langage de configuration
Travailler avec des ressources
Variables d'entrée
Déclarer des variables de sortie
Modules

Sources des modules
Configuration du backend
Travailler avec des états
Gestion des espaces de travail

Terraform et OpenStack

Revue des Ressources OpenStack

Lab 2:

- Création d'une infrastructure simple sur OpenStack avec Terraform.

Terraform et AWS

Revue des Ressources AWS

Lab 3 :

- Création d'une infrastructure simple sur AWS avec Terraform
- Création d'une infrastructure web de production avec Terraform.
- Reprise d'infrastructure existante par import dans Terraform



Terraform - Introduction

- Infrastructure as Code
- Qu'est-ce que Terraform ?
- Pourquoi utiliser Terraform ?
- Terraform VS Ansible
- Architecture de Terraform
- IaC avec Terraform
- Introduction à HCL

Infrastructure as Code

Qu'est-ce que l'Infrastructure as Code (IaC) ?

- **Définition** : L'Infrastructure as Code (IaC) est une pratique clé du DevOps qui consiste à gérer et à provisionner l'infrastructure informatique à l'aide de fichiers de scripts lisibles par une machine, plutôt qu'à travers une configuration matérielle physique ou des outils de configuration interactifs.
- **Objectif** : Traiter l'infrastructure de la même manière que le code applicatif - versionné, testé et automatisé.



Infrastructure as Code

Avantages de l'IaC :

- **Agilité** : Provisionnement rapide et cohérent de l'infrastructure.
- **Reproductibilité** : Assurer la cohérence entre les environnements de développement, de test et de production.
- **Scalabilité** : Évoluer facilement l'infrastructure en fonction des besoins changeants.
- **Collaboration** : Facilite la collaboration entre les équipes de développement et d'exploitation.



Infrastructure as Code

Concepts Clés :

- **Déclaratif vs Impératif** : L'IaC déclaratif décrit l'état souhaité de l'infrastructure, tandis que l'IaC impératif définit les étapes pour atteindre cet état.
- **Idempotence** : Exécuter le même script IaC plusieurs fois produit le même résultat, indépendamment de l'état initial.



Infrastructure as Code

Quelques outils de l'Infrastructure as Code (IaC)



Infrastructure as Code

Gestion de configurations



Templating



Outil de Provisioning d'infrastructure



Qu'est-ce que Terraform ?

- HashiCorp Terraform est un outil d'Infrastructure as Code (IaC) qui vous permet de définir des ressources en Cloud et on-premise dans des fichiers de configuration lisibles par l'homme.
- **Caractéristiques du Code Terraform:**
 - ◆ Versioning
 - ◆ Réutilisation du code
 - ◆ Partage du code
 - ◆ Reproductibilité de l'infrastructure
 - ◆ Gestion de l'état de votre infrastructure
 - ◆ Utilisable dans un Pipeline DevOps pour approvisionner et gérer l'ensemble de votre infrastructure tout au long de son cycle de vie.
 - ◆ Gestion des composants Clouds : Instances, Stockage, Réseaux, DNS, Base de données...
- Créé en 2014



Pourquoi utiliser Terraform ?

Dans le Cloud - Problématique de gestion des ressources

- Cloud : Milliers de ressources, et de services
- Reproduire une architecture complexe dans plusieurs régions et sur différents cloud
- Gestion manuelle complexe
- Avoir des environnement identiques Dev, Prod
 - Environnement Multi-Cloud



Cas du Cloud AWS : plus de 200 services



Amazon EC2



Amazon ECR



Amazon ECS



AWS Elastic
Beanstalk



AWS
Lambda



Auto Scaling



IAM



AWS KMS



Amazon
S3



Amazon
SES



Amazon
RDS



Amazon
Aurora



Amazon
DynamoDB



Amazon
ElastiCache



Amazon
SQS



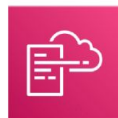
Amazon
SNS



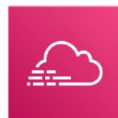
AWS Step Functions



Amazon
CloudWatch



AWS
CloudFormation



AWS
CloudTrail



Amazon API
Gateway



Elastic Load
Balancing



Amazon
CloudFront

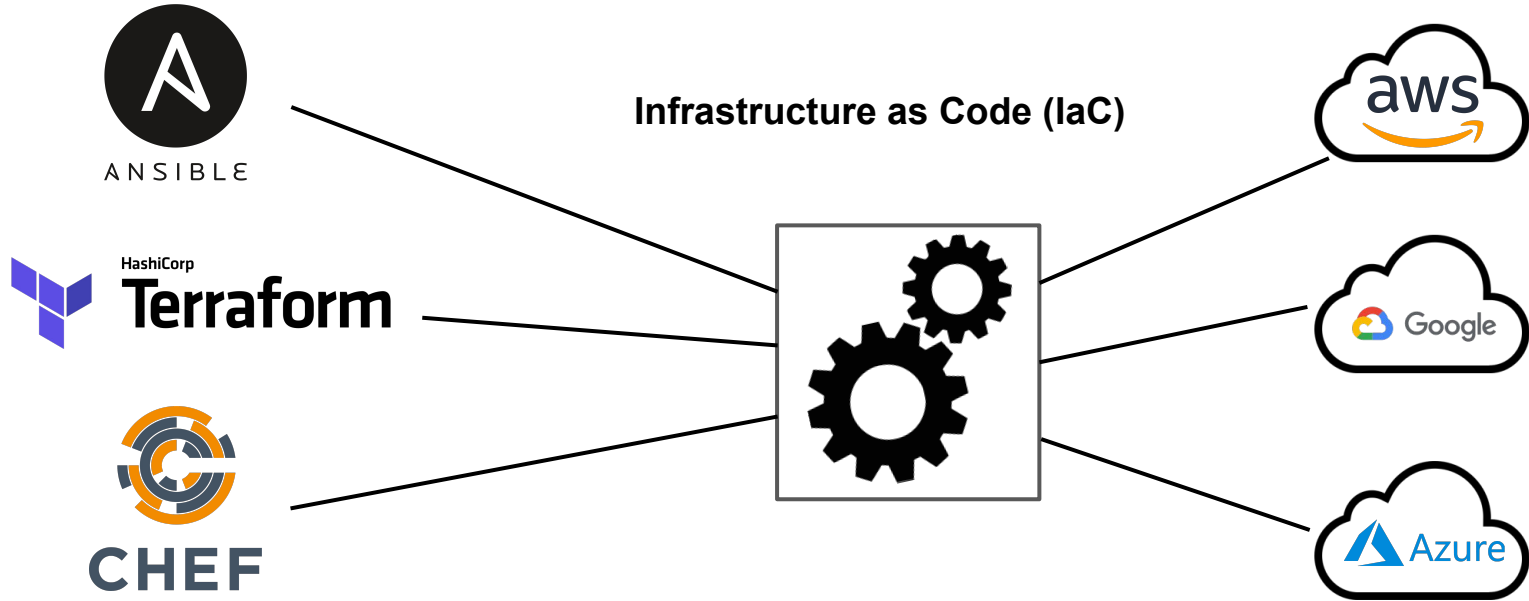


Amazon
Kinesis



Amazon
Route 53

Cloud - Gestion des ressources - Orchestrateurs



Terraform VS Ansible

	 Terraform	 Ansible
Type	Orchestration tool	Configuration management tool
Syntax	HCL	YAML
Language	Declarative	Procedural
Default approach	Mutable infrastructure	Immutable infrastructure
Lifecycle management	Does support	Doesn't support
Capabilities	Provisioning and configuring	Provisioning and configuring
Agentless	✓	✓
Masterless	✓	✓



Terraform VS Ansible

Caractéristiques	Terraform	Ansible
Type d'outil	Orchestration - Infrastructure as Code (IaC)	Automatisation et gestion de configuration
Langage de Description	HashiCorp Configuration Language (HCL)	YAML (Playbooks)
Approche	Déclaratif et orienté état	Agentless (pas d'agent requis sur les cibles)
Utilisation principale	Déployer et gérer des infrastructures	Automatiser des tâches, configurer des systèmes
Cible principale	Infrastructures et ressources cloud	Configuration système et applications
Dépendances	Crée et modifie les ressources indépendamment	Dépend de l'état actuel des cibles
Gestion de l'état	Gère l'état réel et le modifie pour correspondre à l'état désiré	Prévoit l'état désiré et applique les changements
Exemples de ressources	Machines virtuelles, réseaux, groupes de sécurité	Packages, fichiers, services, utilisateurs
Exécution en parallèle	Crée des ressources en parallèle, mais les dépendances peuvent limiter certaines actions	Peut exécuter des tâches en parallèle sur plusieurs cibles
Support des fournisseurs cloud	Fournit un large éventail de fournisseurs cloud. Également extensible par des modules personnalisés	Peut gérer différentes plates-formes avec des modules. Extensible par des modules personnalisés



Terraform VS CloudFormation





























Caractéristique	Terraform	CloudFormation
Langage de Déclaration	HCL (HashiCorp Configuration Language)	JSON ou YAML
Multi-Cloud	Oui, prend en charge plusieurs fournisseurs de cloud	Principalement centré sur AWS, mais prend également en charge d'autres clouds via des extensions ou des intégrations spécifiques.
Intégration d'Écosystème	Large écosystème , prend en charge de nombreux fournisseurs et services tiers	Intégré à l'écosystème AWS, offre une prise en charge directe des services AWS.
Documentation et Communauté	Documentation complète , communauté active et étendue	Documentation complète, grande communauté sur la plateforme AWS
Interopérabilité	Peut gérer des ressources de plusieurs fournisseurs de cloud dans un seul fichier de configuration	Principalement centré sur AWS, mais prend en charge certaines intégrations tierces via AWS Lambda-backed custom resources.
Langage de Script	Configurations déclaratives et impératives avec HCL	Configurations déclaratives avec JSON ou YAML
Gestion de l'État	Utilise des fichiers d'état locaux ou des backends distants , permettant une gestion flexible de l'état	Stockage de l'état géré par AWS, nécessitant un accès aux services AWS pour la gestion de l'état.



Terraform VS CloudFormation

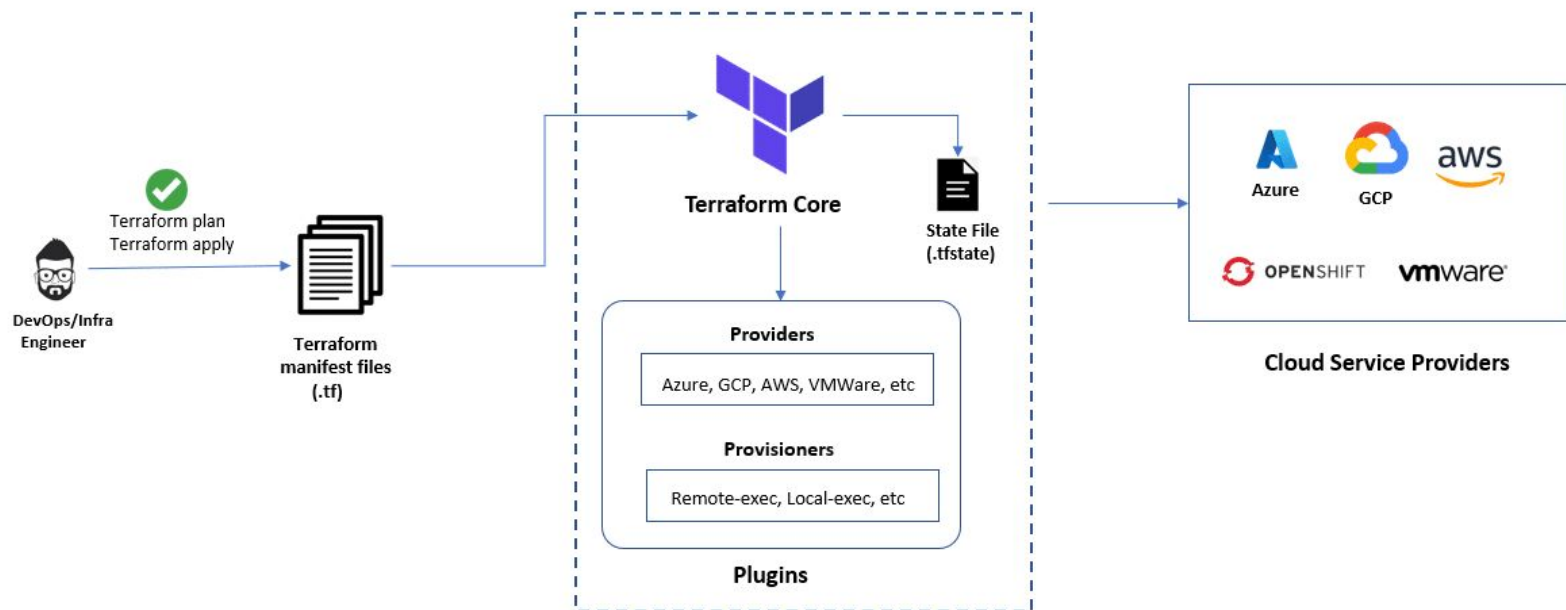
Caractéristique	Terraform	CloudFormation
Extensions et Modules	Modules permettant une modularité et une réutilisation élevées	Extensions pour certains cas, mais souvent moins modulaire que Terraform.
Rollbacks Automatiques	Oui, avec la possibilité de définir des règles de rollback	Oui, en cas d'échec de création d'une pile.
Gestion des Dépendances	Gestion automatique des dépendances entre ressources	Gestion manuelle des dépendances avec la propriété DependsOn
Visibilité des Opérations	Fournit des rapports détaillés après chaque exécution	Historique des événements dans le CloudFormation Console
Mise à l'Échelle des Opérations	Peut être utilisé pour gérer des infrastructures de petite à grande échelle	Capacités de gestion d'infrastructure de toutes tailles
Flexibilité et Extensibilité	Grande flexibilité avec des options avancées de personnalisation	Moins de flexibilité que Terraform, mais s'améliore avec le temps.

Positionnement de Terraform

 linux Public	Contributeurs : +14.473	 Watch 8.2k	 Fork 49.8k	 Star 157k
 kubernetes Public	Contributeurs : +3.497	 Watch 3.2k	 Fork 37.7k	 Star 102k
 ansible Public	Contributeurs : +5.505	 Watch 1.9k	 Fork 23.7k	 Star 58.5k
 terraform Public	Contributeurs : +1.758	 Watch 1.2k	 Fork 9k	 Star 38.8k
 salt Public	Contributeurs : +2.424	 Watch 538	 Fork 5.5k	 Star 13.5k
 chef Public	Contributeurs : +657	 Watch 371	 Fork 2.6k	 Star 7.3k
 puppet Public	Contributeurs : +566	 Watch 468	 Fork 2.3k	 Star 7.1k

Architecture de Terraform

Terraform Architecture



Installation de Terraform sur Linux

- **Ajouter la clé GPG de HashiCorp**
- **Ajouter le repo Linux de HashiCorp**
- **Mettre à jour et installer Terraform CLI**
- **Vérifier l'installation**
- **Activer l'achèvement des onglets**



LAB

- Mise en place de votre environnement
- Découverte de la ligne de commande Terraform

Terraform - Principes fondamentaux

- CLI de Terraform
- Langage de configuration
- Travailler avec des ressources
- Variables d'entrée
- Déclarer des variables de sortie
- Modules
- Sources des modules
- Configuration du backend
- Travailler avec des états
- Gestion des espaces de travail

Introduction à HCL

```
terraform version
```

→ Trouver la version de Terraform

```
terraform init
```

→ Initialiser le répertoire

```
terraform plan
```

→ Créer un plan d'exécution

```
terraform apply
```

→ Appliquer les changements

```
terraform destroy
```

→ Détruire l'infrastructure gérée



Introduction à HCL

```
terraform plan -out <plan_name>
```

→ Produire un plan de déploiement

```
terraform plan -destroy
```

→ Produire un plan de destruction

```
terraform apply <plan_name>
```

→ Appliquer un plan spécifique

```
terraform apply -target=<resource_name>
```

→ N'appliquer les modif. qu'à une ressource ciblée

```
terraform apply -var my_variable=<variable>
```

→ Transmettre une variable via la ligne de commande

```
terraform providers
```

→ Obtenir les informations sur le fournisseur utilisées dans la configuration



HCL - langage de Configuration

```
resource "aws_vpc" "main" {  
  cidr_block = var.base_cidr_block  
}  
  
<BLOCK TYPE> "<BLOCK LABEL>" "<BLOCK LABEL>" {  
  # Block body  
  <IDENTIFIER> = <EXPRESSION> # Argument  
}
```

- **<BLOCK>** sont des conteneurs pour d'autres objets. Représentent généralement la configuration d'un certain type d'objet (Ex: une ressource)
 - ◆ Type de bloc,
 - ◆ 0 ou plusieurs étiquettes
 - ◆ Un corps qui contient un nombre quelconque d'arguments et de blocs imbriqués.
- **Arguments** : attribuent une valeur à un nom
- **Expressions**: représentent une valeur



HCL - langage de Configuration

```
resource "aws_instance" "web" {  
  instance_type = "t2.micro"  
  ami           = "ami-408c7f28"  
}
```

→ Exemple de configuration Terraform : lancer une Instance de type t2.micro dans AWS avec l'AMI ami-408c7f28

→ Fichier **override.tf**

```
resource "aws_instance" "web" {  
  ami = "ami-override"  
}
```



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

→ Terraform fusionne et prend en compte la valeur contenue dans **override.tf**

Syntaxe de configurations

```
image_id = "dummy"
```

→ Exemple d'argument

- **#** : Commentaire d'une ligne, se terminant à la fin d'une ligne
- **//** : Egalement un commentaire d'une ligne, comme alternative à #.
- **/* blah */** : sont des délimiteurs de début et de fin pour un commentaire qui peut s'étendre sur plusieurs lignes.

Syntaxe de configurations

```
variable "var_ami" {  
  default = "ami-perso"  
}  
  
resource "aws_instance" "example" {  
  instance_type = "t2.micro"  
  ami           = var_ami  
}
```

→ La déclaration de variables



JSON Config vs Terraform Config

```
resource "aws_instance" "example" {  
  lifecycle {  
    create_before_destroy = true  
  }  
}
```

→ Langage HCL

```
{  
  "resource": {  
    "aws_instance": {  
      "example": {  
        "lifecycle": {  
          "create_before_destroy": true  
        }  
      }  
    }  
  }  
}
```

→ La déclaration au format JSON



Travailler avec des ressources

```
resource "aws_instance" "web" {  
  instance_type = "t2.micro"  
  ami          = "ami-408c7f28"  
}
```

→ Les ressources constituent la partie la plus importante du langage Terraform. Les blocs de ressources décrivent des objets d'infrastructure tels que des réseaux virtuels, des instances de calcul, etc...

- **Providers**, qui sont des plugins pour Terraform offrant une collection de types de ressources.
- **Arguments**, qui sont spécifiques au type de ressource sélectionné.
- **Documentation**, que chaque fournisseur utilise pour décrire ses types de ressources et ses arguments.



Travailler avec des ressources

```
resource "aws_instance" "web" {  
  instance_type = "t2.micro"  
  ami          = "ami-408c7f28"  
}
```

→ Les ressources constituent la partie la plus importante du langage Terraform. Les blocs de ressources décrivent des objets d'infrastructure tels que des réseaux virtuels, des instances de calcul, etc...

- **Providers**, qui sont des plugins pour Terraform offrant une collection de types de ressources.
- **Arguments**, qui sont spécifiques au type de ressource sélectionné.
- **Documentation**, que chaque fournisseur utilise pour décrire ses types de ressources et ses arguments.



Meta-Arguments

`depends_on`

→ Spécifier les dépendances cachées.

`provider`

→ Sélectionnez une configuration de fournisseur autre que celle par défaut.

`count`

→ Créer plusieurs instances de ressources en fonction d'un décompte.

`for_each`

→ Créer plusieurs instances à partir d'une liste fournie

`lifecycle`

→ Définir les personnalisations du cycle de vie.



Meta-Arguments

Lorsque vous utilisez **for_each**, l'objet **each** devient disponible dans le bloc de ressources.

- **each.key** - La clé du map (l'objet en particulier) correspondant à cette instance.
- **each.value** - La valeur correspondante à cette instance.

Voici comment utiliser le méta-argument **for_each** :

```
resource "azurerm_resource_group" "rg" {  
  for_each = {  
    a_group = "eastus"  
    another_group = "westus2"  
  }  
  name      = each.key  
  location = each.value  
}
```

```
resource "aws_iam_user" "the-accounts" {  
  for_each = toset( ["Todd", "James", "Alice"] )  
  name     = each.key  
}
```

Travailler avec des ressources

```
resource "aws_db_instance" "example" {  
  # ...  
  timeouts {  
    create = "60m"  
    delete = "2h"  
  }  
}
```

→ Possible de modifier les Timeout par défaut (utile pour certaines ressources qui mettent du temps à être provisionnées)

→ [Timeouts par défaut](#)

→ **Formats de valeurs possibles:**

- ◆ "60m"
- ◆ "10s"
- ◆ "2h"



Travailler avec des ressources

→ Comment la configuration est-elle appliquée ?

- ◆ **Create** : Crée des ressources qui existent dans la configuration mais qui ne sont pas associées à un objet d'infrastructure réel dans l'état.
- ◆ **Destroy** : Détruit les ressources qui existent dans l'état mais qui n'existent plus dans la configuration
- ◆ **Update in-place** : Mettre à jour les ressources en place dont les arguments ont changé.
- ◆ **Destroy and re-create** : Détruit et recrée les ressources dont les arguments ont été modifiés, mais qui ne peuvent pas être mises à jour sur place en raison des limitations de l'API distante.



Travailler avec des ressources

- Exemple de Block **variable**
- Le nom d'une variable peut être n'importe quel **identifiant** valide, à l'exception de **source**, **version**, **providers**, **count**, **for_each**, **lifecycle**, **depends_on** et **locals**.

```
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"  
    }  
  ]  
}
```



Input Variables

→ **Arguments** facultatifs pour la déclaration de variables

- ◆ default
- ◆ type
- ◆ description
- ◆ validation
- ◆ sensitive

→ Contraintes sur les Type

- ◆ string
- ◆ number
- ◆ Bool

→ Types de constructeurs

- ◆ list(<type>)
- ◆ set(<type>)
- ◆ map(<type>)
- ◆ object({<attribute> = <type>, ...})
- ◆ tuple([<type>, ...])



Input Variables

→ Exemple de personnalisation des règles de validation des variables

```
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-\"."  
  }  
}
```

Input Variables

→ Utilisation des arguments sensibles (ex: mot de passe)

```
variable "user_information" {  
  type = object({  
    name    = string  
    address = string  
  })  
  sensitive = true  
}  
  
resource "some_resource" "a" {  
  name    = var.user_information.name  
  address = var.user_information.address  
}
```



→ Variables sont masquées

```
Terraform will perform the following actions:  
  
# some_resource.a will be created  
+ resource "some_resource" "a" {  
    + name = (sensitive)  
    + address = (sensitive)  
}  
  
Plan: 1 to add, 0 to change, 0 to destroy.
```

Utilisation des variables : `var.nom_variable`

Input Variables

- Comment assigner des valeurs aux variables ?
 - ◆ Dans un espace **Terraform Cloud workspace**
 - ◆ Individuellement, avec l'option de ligne de commande **-var**
 - ◆ Dans des fichiers de définition de variables comme **.tfvars** ou **.tfvars.json**
 - ◆ En tant que variables d'environnement

- En ligne de commande

```
$ terraform apply -var="image_id=ami-abc123"  
$ terraform apply -var='image_id_list=["ami-abc123","ami-def456"]' -var="instance_type=t2.micro"  
$ terraform apply -var='image_id_map={"us-east-1":"ami-abc123","us-east-2":"ami-def456"}'
```

Input Variables

- L'ordre dans lequel les variables sont chargées :
 - ◆ Variables d'environnement
 - ◆ `terraform.tfvars`
 - ◆ `terraform.tfvars.json`
 - ◆ `*.auto.tfvars` or `*.auto.tfvars.json`
 - ◆ Any command-line options like `-var` et `-var-file`
- En ligne de commande

```
$ terraform apply -var="image_id=ami-abc123"
$ terraform apply -var='image_id_list=["ami-abc123","ami-def456"]' -var="instance_type=t2.micro"
$ terraform apply -var='image_id_map={"us-east-1":"ami-abc123","us-east-2":"ami-def456"}'
```

Output Variables

- Les valeurs de sortie (Output Variables) rendent les informations sur votre infrastructure disponibles en ligne de commande, et peuvent exposer des informations que d'autres configurations Terraform peuvent utiliser.
- Exemple de déclaration

```
output "instance_ip_addr" {  
  value = aws_instance.server.private_ip  
  description = "The private IP address of the main server instance."  
}
```

- L'étiquette du **nom** après le mot-clé **output** doit être un identifiant valide
- L'argument **value** prend une expression dont le résultat sera renvoyé à l'utilisateur.
- La **description** doit expliquer clairement de quel **output** il s'agit et le type de valeur (**value**) attendue.



Modules

→ Types de Module

- ◆ Modules **racines** - Vous avez besoin d'au moins un module racine.
- ◆ Modules **enfants** - Modules appelés par le module racine.
- ◆ Modules **publiés** - Modules chargés à partir d'un registre privé ou public

→ Block **Module**

```
module "servers" {  
  source = "./app-cluster"  
  servers = 5  
}
```

- Un module **racine** qui inclut un bloc de module; ce dernier est appelé un **module enfant**. L'étiquette qui suit le mot-clé **module** est un nom local qui peut être utilisé pour faire référence au module.

Gestion des etats

- HashiCorp recommande aux débutants Terraform d'utiliser le backend **local** par défaut.
 - ◆ État stocké dans un fichier ***.tfstate**
- Si vous travaillez en équipe et gérez une grande infrastructure, HashiCorp vous recommande d'utiliser un backend **distant**.



Gestion des etats

- Terraform comprend une sélection intégrée de backends, et ce sont les seuls backends. Vous ne pouvez pas charger de backends supplémentaires en tant que plugins.
- La configuration du **backend** n'est utilisée que par Terraform **CLI**. Terraform Cloud et Enterprise utilisent toujours leur propre stockage d'état.
- Deux domaines du comportement de Terraform sont déterminés par le backend :
 - ◆ l'endroit où l'état est stocké ;
 - ◆ l'endroit où les opérations sont effectuées.



Gestion des etats

Component	Local Terraform	Terraform Cloud
Terraform Config	On disk	In VCS or uploaded via API/CLI
Variable Values	As .tfvars files, CLI arguments, or as shell environment	In workspaces
State	On disk or in remote backend	In workspaces
Credentials and Secrets	In shell environment or entered at prompts	In workspaces, stored as sensitive variables



Gestion des états - configuration du backend

```
terraform {  
  backend "remote" {  
    organization = "corp_example"  
  
    workspaces {  
      name = "ex-app-prod"  
    }  
  }  
}
```

- Une configuration ne peut fournir qu'un seul bloc **backend**.
- Lorsque le backend change dans une configuration, vous devez exécuter **terraform init**.
- Lorsque le backend change, Terraform vous donne la possibilité de migrer votre **état**.
- HashiCorp vous recommande de sauvegarder manuellement votre état. Copiez simplement le fichier **terraform.tfstate**.



Gestion des états - configuration du backend

→ Exemple de configuration du backend **Local**

```
terraform {  
  backend "local" {  
    path = "/chemin/vers/terraform.tfstate"  
  }  
}
```

- ◆ **path** - Chemin d'accès au fichier **tfstate**.
- ◆ **workspace_dir** - Chemin d'accès aux espaces de travail (workspaces) non définis par défaut.



Gestion des états - configuration du backend

→ Exemple de configuration du backend **Remote**

```
terraform {  
  backend "remote" {  
    hostname = "app.terraform.io"  
    organization = "company"  
  
    workspaces {  
      name = "my-app-prod"  
    }  
  }  
}
```

→ Utilisation du fichier backend avec **init**

```
terraform init -backend-config=backend.hcl
```

→ Utilisation du CLI Input

```
# main.tf  
terraform {  
  
  required_version = "~> 0.12.0"  
  
  backend "remote" {}  
}
```

→ Configuration du backend dans un fichier séparé

```
# backend.hcl  
workspaces { name = "workspace" }  
hostname = "app.terraform.io"  
organization = "company"
```



Gestion des états - configuration du backend avec S3

→ Exemple de configuration du backend AWS S3

```
terraform {  
  backend "s3" {  
    bucket = "mybucket"  
    key    = "path/to/my/key"  
    region = "us-east-1"  
  }  
}
```

→ Permissions pour ce Bucket S3

- ◆ s3:ListBucket
- ◆ s3:GetObject
- ◆ s3:PutObject

→ Configuration du Data Source

```
data "terraform_remote_state" "network" {  
  backend = "s3"  
  config = {  
    bucket = "terraform-state-prod"  
    key    = "network/terraform.tfstate"  
    region = "us-east-1"  
  }  
}
```

→ Permissions pour ce Bucket S3

- ◆ dynamodb:GetItem
- ◆ dynamodb:PutItem
- ◆ dynamodb:DeleteItem



Gestion des états - État distant

- L'état distant permet à Terraform d'écrire les données déclarées dans un datastore distant, qui peut être partagé par tous les membres de l'équipe.
 - ◆ L'état à distance vous permet de partager des valeurs de sortie avec d'autres configurations.
 - ◆ L'état à distance permet aux équipes de partager des ressources d'infrastructure en lecture seule.
 - ◆ L'état à distance peut être un mécanisme intégré pratique pour partager des données entre des configurations, mais vous pouvez préférer utiliser des datastore plus généraux pour transmettre des paramètres à d'autres configurations et à d'autres consommateurs.



Gestion des états

- Terraform doit suivre les métadonnées comme les dépendances des ressources. Pour garantir le fonctionnement, Terraform conserve une copie de l'ensemble le plus récent de dépendances dans le **State**.
- Terraform stocke un cache des valeurs d'attributs pour toutes les ressources dans le State. Cela s'avère pratique pour les grandes configurations, car vous n'avez pas besoin d'interroger toutes les ressources dans la configuration.
- Lorsque vous travaillez en équipe, il est recommandé d'utiliser le remote State. Il est important que tous les membres d'une équipe travaillent avec le même état. Le remote State garantit la synchronisation de l'état.



Gestion des états - Verrouiller l'état

- S'il est pris en charge par le backend de votre choix, Terraform verrouille votre état pour toutes les opérations.
- Le verrouillage de l'état se produit automatiquement pour toutes les opérations susceptibles d'écrire de l'état.
- Terraform affiche un message d'état si l'acquisition du verrou prend plus de temps que prévu.

```
terraform force-unlock [options] LOCK_ID [DIR]
```

- Soyez prudent avec cette commande ! Si vous déverrouillez l'état alors qu'il est en cours d'utilisation, vous risquez d'être à l'origine d'un grand nombre d'écriture simultanée.
- Pour éviter cela, la commande force-unlock requiert un identifiant de verrouillage unique.



Gestion des états - Verrouiller l'état

→ 10 **backends** qui supportent des espaces de travail (Workspaces) multiples

- ◆ AzureRM
- ◆ Consul
- ◆ COS
- ◆ GCS
- ◆ Kubernetes
- ◆ Local
- ◆ Manta
- ◆ Postgres
- ◆ Remote
- ◆ S3

→ <https://developer.hashicorp.com/terraform/language/settings/backends/pg>



Gestion des états

→ Syntaxe Terraform state

```
terraform state <subcommand> [option] [args]
```

→ Lister les états

```
terraform state list
```

→ Voir le state

```
terraform state show 'module.name.foo.worker'
```

→ Déplacer un état

```
terraform state mv packet.foo packet.bar
```

→ Supprimer un état

```
terraform state rm [option] [args]
```

→ Extraire un state

```
terraform state pull
```



Gestion des workspaces

→ Lister les Workspace

```
terraform workspace list
  default
* development
  jsmith-test
```

→ Selectionner un workspace

```
terraform workspace select default
Switched to workspace "default".
```

→ Voir un espace de travail

```
terraform workspace show development
```

→ Supprimer un Workspace

```
terraform workspace delete example
Deleted workspace "example".
```



Gestion des workspaces

→ Nouvel espace de travail

```
terraform workspace new example
```

```
Created and switched to workspace "example"!
```

You're now on a new, empty workspace. Workspaces isolate their state, so if you run "terraform plan" Terraform will not see any existing state for this configuration.

→ Nouvel espace de travail à partir d'un state

```
terraform workspace new -state=old.terraform.tfstate example
```

```
Created and switched to workspace "example"!
```

You're now on a new, empty workspace. Workspaces isolate their state, so if you run "terraform plan" Terraform will not see any existing state for this configuration.

Terraform pour OpenStack

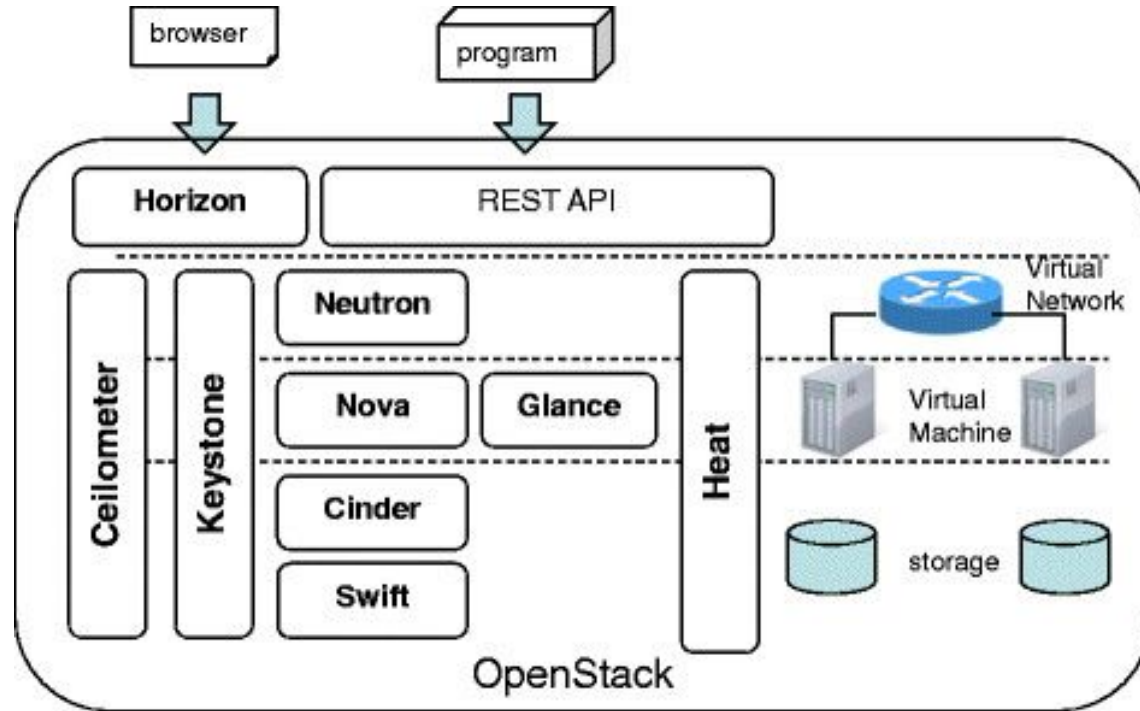
- Revue des ressources Terraform AWS
(Compute / Nova, Networking / Neutron...)
- Configuration des accès
- Construire son infrastructure

Qu'est ce que OpenStack ?


- OpenStack est une plateforme open-source de cloud computing qui fournit des services d'infrastructure en tant que service (IaaS) - Lancé en 2010
- **Composants OpenStack** : ensemble de services interconnectés, chacun fournissant une fonctionnalité spécifique
 - ◆ Nova (compute)
 - ◆ Swift (stockage d'objets)
 - ◆ Cinder (stockage en bloc)
 - ◆ Neutron (réseau)
 - ◆ Keystone (identité et authentification)
 - ◆ Glance (gestion des images).



Qu'est ce que OpenStack ?



Terraform - OpenStack provider

 Registry [Browse](#) [Publish](#) [Sign-in](#) [Use Terraform Cloud for free](#)

[Providers](#) / [terraform-provider-openstack](#) / [openstack](#) / Version 1.53.0 [Latest Version](#)

openstack

[Overview](#) [Documentation](#) [USE PROVIDER](#)

OPENSTACK DOCUMENTATION

- [openstack provider](#)
- [Block Storage / Cinder](#)
- [Compute / Nova](#)
- [Container Infra / Magnum](#)
- [DNS / Designate](#)
- [Databases / Trove](#)
- [FWaaS / Neutron](#)
- [Identity / Keystone](#)
- [Images / Glance](#)
- [Key Manager / Barbican](#)
- [Load Balancing as a Service / Octavia](#)
- [Networking / Neutron](#)
- [Object Storage / Swift](#)
- [Orchestration / Heat](#)
- [Shared Filesystem / Manila](#)
- [VPNaaS / Neutron](#)
- [Deprecated](#)

OpenStack Provider

The OpenStack provider is used to interact with the many resources supported by OpenStack. The provider needs to be configured with the proper credentials before it can be used.

Use the navigation to the left to read about the available resources.

Example Usage

```
# Define required providers
terraform {
  required_version = ">= 0.14.0"
  required_providers {
    openstack = {
      source = "terraform-provider-openstack/openstack"
      version = "~> 1.53.0"
    }
  }
}

# Configure the OpenStack Provider
provider "openstack" {
  user_name = "admin"
  tenant_name = "admin"
  password = "pwd"
  auth_url = "http://myauthurl:5000/v2.0"
  region = "RegionOne"
}

# Create a web server
resource "openstack_compute_instance_v2" "test-server" {
  # ...
}
```

[Copy](#)

ON THIS PAGE

- [Example Usage](#)
- [Configuration Reference](#)
- [Overriding Service API Endpoints](#)
- [Additional Logging](#)
- [OpenStack Releases and Versions](#)
- [Testing and Development](#)

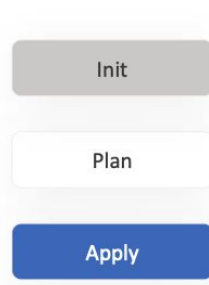
[Report an issue](#)

Fourni et maintenu par la communauté et non Hashicorp

<https://registry.terraform.io/providers/terraform-provider-openstack/openstack/latest/docs>

Terraform - OpenStack provider

```
resource "openstack_compute_instance_v2" "instance_1" {  
  name          = "instance_1"  
  image_id      = "ad091b52-742f-469e-8f3c-fd81cadf0743"  
  flavor_id     = 3  
  key_pair      = "my_key_pair_name"  
  security_groups = ["default"]  
}  
  
resource "openstack_networking_floatingip_v2" "fip_1" {  
  pool = "my_pool"  
}  
  
resource "openstack_compute_floatingip_associate_v2" "fip_1" {  
  floating_ip = openstack_networking_floatingip_v2.fip_1.address  
  instance_id = openstack_compute_instance_v2.instance_1.id  
}
```



Adresse IP



Instance

Terraform pour AWS

- Revue des ressources Terraform AWS (Compute / EC2, Networking / VPC...)
- AWS CLI and Configuration des accès
- Construire son infrastructure

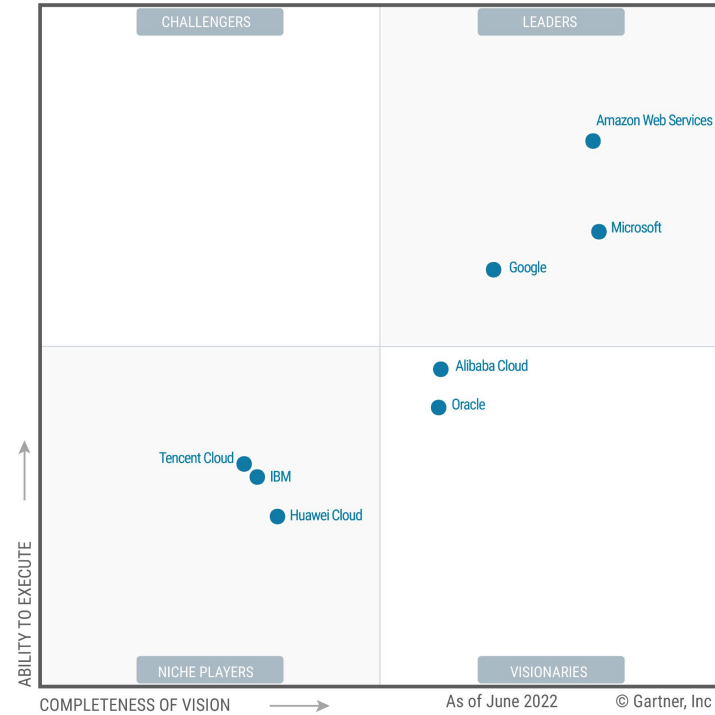
Qu'est ce que AWS ?

- AWS = Amazon Web Services
- Division du géant du commerce en ligne **Amazon**
- Spécialisée dans les services de Cloud Computing Public à la demande pour
- Fournisseur de services Cloud Public (IaaS, PaaS, SaaS)
 - ◆ Calcul
 - ◆ Stockage
 - ◆ Réseaux
 - ◆ ...
 - ◆ A la demande, scalable et élastique
 - ◆ Pour les entreprises et particuliers
- 20 ans depuis son lancement



AWS aujourd'hui

Figure 1: Magic Quadrant for Cloud Infrastructure and Platform Services



+200 services AWS



Amazon EC2



Amazon ECR



Amazon ECS



AWS Elastic
Beanstalk



AWS
Lambda



Auto Scaling



IAM



AWS KMS



Amazon
S3



Amazon
SES



Amazon
RDS



Amazon
Aurora



Amazon
DynamoDB



Amazon
ElastiCache



Amazon
SQS



Amazon
SNS



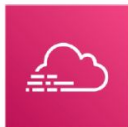
AWS Step Functions



Amazon
CloudWatch



AWS
CloudFormation



AWS
CloudTrail



Amazon API
Gateway



Elastic Load
Balancing



Amazon
CloudFront



Amazon
Kinesis



Amazon
Route 53



+200 services AWS



Terraform pour la gestion des ressources



Terraform - AWS provider

Providers

hashicorp

aws

Version 5.29.0

Latest Version

aws

Overview

Documentation

USE PROVIDER

AWS DOCUMENTATION

Filter

aws provider

Guides

ACM (Certificate Manager)

ACM PCA (Certificate Manager Private Certificate Authority)

AMP (Managed Prometheus)

API Gateway

API Gateway V2

Account Management

Amazon Bedrock

Amplify

App Mesh

App Runner

AppConfig

AppFlow

AppIntegrations

AppStream 2.0

AppSync

Application Auto Scaling

Athena

Audit Manager

Auto Scaling

Auto Scaling Plans

Backup

Batch

CE (Cost Explorer)

Chime

AWS Provider

Use the Amazon Web Services (AWS) provider to interact with the many resources supported by AWS. You must configure the provider with the proper credentials before you can use it.

Use the navigation to the left to read about the available resources. There are currently 1289 resources and 528 data sources available in the provider.

To learn the basics of Terraform using this provider, follow the hands-on [get started tutorials](#). Interact with AWS services, including Lambda, RDS, and IAM by following the [AWS services tutorials](#).

Example Usage

Terraform 0.13 and later:

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

# Configure the AWS Provider
provider "aws" {
  region = "us-east-1"
}

# Create a VPC
resource "aws_vpc" "example" {
  cidr_block = "10.0.0.0/16"
}
```

New

Multi-language provider docs

Terraform

The Registry now supports multi-language docs powered by CDK for Terraform. [Learn more](#)

ON THIS PAGE

Example Usage

Authentication and Configuration

AWS Configuration Reference

Custom User-Agent Information

Argument Reference

Getting the Account ID

Report an issue

<https://registry.terraform.io/providers/hashicorp/aws>

Ressources Terraform AWS

```
main.tf

resource "aws_instance" "weberver" {
  # configuration here
}

resource "aws_key_pair" "key" {
  # configuration here
}

resource "aws_security_group" "ssh-access" {
  # configuration here
}

resource "aws_s3_bucket" "data-bucket" {
  # configuration here
}

resource "aws_dynamodb_table" "user-data" {
  # configuration here
}

resource "aws_instance" "web-server-2" {
  # configuration here
}
```



aws_instance



aws_key_pair



aws_iam_policy



aws_s3_bucket



aws_dynamodb_table



aws_instance

Ressources Terraform AWS

```
main.tf

resource "aws_instance" "webserver" {
  ami          = "ami-0edab43b6fa892279"
  instance_type = "t2.micro"
}

resource "aws_s3_bucket" "finance" {
  bucket = "finanace-21092020"
  tags = {
    Description = "Finance and Payroll"
  }
}

resource "aws_iam_user" "admin-user" {
  name = "lucy"
  tags = {
    Description = "Team Leader"
  }
}
```

Init

Plan

Apply

Infrastructure déployée dans AWS



EC2



S3



User



LAB

Terraform pour AWS

- Création d'une infrastructure simple sur AWS avec Terraform
- Création d'une infrastructure web de production avec Terraform.
- Reprise d'infrastructure existante par import dans Terraform