Terraform

bstorm TRAINING

Table des matières

Introduction à l'Infrastructure as Code (IaC)

bstorm TRAINING

Introduction à l'IAC

Infrastructure as Code (IaC) : gestion de l'infrastructure via du code au lieu de processus manuels
Principes clés :

- Automatisation
- Reproductibilité
- Versionnement (Git, CI/CD)
- Cohérence des environnements

Avantages :

- Réduction des erreurs humaines
- Déploiements rapides et fiables
- Collaboration facilitée entre équipes
- Meilleure traçabilité et gouvernance

Présentation de Terraform

Historique :

Terraform est un outil d'IaC (Infrastructure as Code) créé par HashiCorp en 2014, il est open-source et multi-cloud.

Concepts clés :

- Langage déclaratif (HCL)
- Providers (AWS, Azure, GCP, Kubernetes, etc.)
- State (fichier qui décrit l'infrastructure actuelle)
- Modules pour la réutilisation du code

Présentation de Terraform

Avantages :

- Multi-cloud et agnostique
- Large écosystème de providers
- Open source, communauté active
- Intégration facile aux CI/CD

Cas d'usage :

- Provisioning cloud (VM, réseaux, bases de données)
- Gestion Kubernetes
- Infrastructure hybride
- Scalabilité et DRP (Disaster Recovery Plans)

Comparaison avec d'autres outils IaC

Terraform :

- Multi-cloud, langage déclaratif HCL
- Écosystème riche, état géré par fichier ou backend

AWS CloudFormation :

- Outil AWS natif
- Fortement intégré à AWS mais limité au cloud AWS

Azure Resource Manager (ARM Templates) :

- Dédié à Azure
- JSON verbeux, moins lisible que HCL

Ansible (partiellement IaC) :

- Plutôt pour configuration que provisionning
- Exécute des scripts idempotents

Installation et Configuration

bstorm TRAINING

Installation

Pour l'installation il faut suivre la documentation, nous conseillons l'installation via un package manager (Homebrew sur MacOs, Chocolatery sur Windows ou apt/yum/packman/... sur Linux).

https://developer.hashicorp.com/terraform/tutorials/aws-get-started/install-cli

Vous devrez aussi installer les outils pour gérer le cloud depuis votre machine :

- AWS CLI : https://docs.aws.amazon.com/cli/latest/userguide/getting-started-install.html
- Azure CLI : https://learn.microsoft.com/en-us/cli/azure/?view=azure-cli-latest
- GCloud CLI : https://cloud.google.com/sdk/docs/install
- OCI CLI : https://docs.oracle.com/en-us/iaas/Content/API/SDKDocs/cliinstall.htm

Installation

Pour ce cours nous utiliserons *Amazon AWS* qui peut être facilement simulé en local via *LocalStack* :

- LocalStack CLI : https://docs.localstack.cloud/aws/getting-started/installation/
- Docker-Compose (Recommandé) : https://docs.localstack.cloud/aws/getting-started/installation/#docker-compose
- awslocal : https://docs.localstack.cloud/aws/integrations/aws-native-tools/aws-cli/#localstack-aws-cli-awslocal

Mise en place de l'environnement (Docker compose) :

```yaml
version: "3.8"

services:
 localstack:
   container_name: "localstack-main"
   image: localstack/localstack
   ports:
     - "127.0.0.1:4566:4566"            # LocalStack Gateway
     - "127.0.0.1:4510-4559:4510-4559" # external services port range
   environment:
     # LocalStack configuration: https://docs.localstack.cloud/references/configuration/
#      - DEBUG=${DEBUG:-0}
     - DEBUG=1
#      - LOCALSTACK_AUTH_TOKEN="AUTH-TOKEN"
     - PERSISTENCE=1
     - RDS_CLUSTER_ENDPOINT_HOST_ONLY=1
   volumes:
     - "./localstack-volume:/var/lib/localstack"
     - "/var/run/docker.sock:/var/run/docker.sock"
```

Mise en place de l'environnement

Ensuite nous devons lancer le container docker :

```
$ docker compose up -d
```

Et créer les queue SQS awslocal :

```
$ awslocal sqs create-queue --queue-name test-queue
```

- SQS : Indique qu'on veut travailler avec le service Amazon SQS (Simple Queue Service).
- create-queue : Sous-commande qui permet de créer une nouvelle file.
- - -queue-name test-queue : Spécifie le nom de la file qui sera créée (ici : test-queue).

Pour vérifier que la queue a bien été créée :

```
$ awslocal sqs list-queues
```

Mise en place de l'environnement

Les commandes ci-dessous devront être exécutées par instance de terminal.
Nous devons ensuite setup les accès AWS vers notre container :

```
$ export AWS_ACCESS_KEY_ID="test"
$ export AWS_SECRET_ACCESS_KEY="test"
$ export AWS_DEFAULT_REGION="us-east-1"
```

Sur Windows :

```
$env:AWS_ACCESS_KEY_ID = "test"
$env:AWS_SECRET_ACCESS_KEY = "test"
$env:AWS_DEFAULT_REGION = "us-east-1"
```

Et faire pointer notre AWS CLI vers notre container :

```
$ aws --endpoint-url=http://localhost:4566 kinesis list-streams
```

- —endpoint-url=http://localhost:4566 : Indique à la CLI d'envoyer la requête non pas vers AWS, mais vers LocalStack, qui simule les services AWS en local.
- kinesis : Spécifie que l'on interagit avec le service Amazon Kinesis (service de streaming de données en temps réel).
- list-streams : Sous-commande qui liste tous les flux Kinesis existants.

Les Concepts de Base de Terraform

bstorm TRAINING

Introduction au modèle Terraform

Terraform est basé sur un modèle déclaratif qui permet de décrire l'infrastructure sous forme de code.

Points forts :

- Reproductible et versionnable
- Indépendant du cloud (multi-provider)
- Organisé en blocs logiques : Providers, Resources, Modules, Variables, Outputs

Illustration :
Terraform Configuration -> Terraform Plan -> Terraform Apply -> Infrastructure

Structure générale des fichiers Terraform

Un projet Terraform typique contient plusieurs fichiers .tf ou .tfvars :

```
my-terraform-project/
├── main.tf        # Configuration principale : ressources, modules
├── variables.tf   # Déclaration des variables
├── outputs.tf     # Déclaration des outputs
├── terraform.tfvars # Valeurs des variables
├── providers.tf   # Déclaration des providers (optionnel, peut être dans main.tf)
├── terraform.tf   # Spécification de la version de Terraform et des providers
└── modules/       # Dossiers de modules réutilisables
```

Types de fichiers et leur rôle : main.tf

Contient la configuration principale, c'est-à-dire :

- Les ressources (resource)
- Les modules (module)
- Les data sources (data)

```
resource "aws_instance" "app" {
 ami           = "ami-0c55b159cbfafe1f0"
 instance_type = "t2.micro"
 tags = { Name = "demo-app" }
}
```

Types de fichiers et leur rôle : variables.tf

Rôles :

- Déclare les variables utilisées dans le projet.
- Définition du type, valeur par défaut, description et éventuellement sensibilité.

```
variable "region" {
 description = "La région AWS"
 type        = string
 default     = "us-east-1"
}
```

Types de fichiers et leur rôle : terraform.tfvars

Ce fichier est utilisé pour fournir des valeurs concrètes aux variables.

```
region        = "eu-west-1"
instance_type = "t2.micro"
```

Types de fichiers et leur rôle : outputs.tf

Déclare les valeurs que Terraform doit exposer après avoir apply.

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

Types de fichiers et leur rôle : providers.tf

Déclare les providers utilisés et leur configuration (AWS, Azure, GCP...).

```
provider "aws" {
 region = var.region
}
```

Types de fichiers et leur rôle : terraform.tf

Spécifie la version minimale de Terraform et des providers pour assurer la compatibilité.

```
terraform {
 required_providers {
   aws = {
     source  = "hashicorp/aws"
     version = "~> 5.92"
   }
 }

 required_version = ">= 1.2"
}
```

Types de fichiers et leur rôle : modules

Dossier contenant des groupes de ressources réutilisables.

```
modules/
└── network/
    ├── main.tf
    ├── variables.tf
    └── outputs.tf
```

```
module "network" {
 source   = "./modules/network"
 vpc_cidr = "10.0.0.0/16"
}
```

Format des fichiers Terraform

Format : HCL (HashiCorp Configuration Language)

- Lisible par les humains, proche du JSON
- Utilisé pour tous les fichiers .tf

Exemple simple :

```
resource "aws_s3_bucket" "demo" {
 bucket = "terraform-demo-bucket"
}
```

Commandes utiles pour le formatage :

```
$ terraform fmt          # Formate tous les fichiers du projet
$ terraform validate     # Vérifie la validité de la configuration
```

Bonnes pratiques pour les fichiers Terraform

- Séparer les variables, outputs, providers et resources pour plus de clarté
- Versionner les fichiers .tf dans Git
- Éviter de stocker les secrets directement dans les fichiers .tf -> utiliser des variables sensibles ou un gestionnaire de secrets
- Organiser les modules pour la réutilisation et la lisibilité
- Documenter chaque fichier avec des commentaires

Gestion des États et des Backends

bstorm TRAINING

Introduction à l'état de Terraform

Concept :

- Terraform utilise un fichier d'état pour suivre l'infrastructure réelle et la comparer à la configuration déclarative.
- Ce fichier contient les informations sur toutes les ressources créées, modifiées ou supprimées.

Rôle clé :

- Permet à Terraform de savoir ce qui existe dans le cloud
- Facilite les plans d'exécution (terraform plan) précis
- Supporte les dépendances et références entre ressources

Où se trouve l'état

Par défaut : fichier terraform.tfstate dans le répertoire du projet

Peut être stocké à distance (Remote Backend) pour :

- Collaboration en équipe
- Historique des versions
- Sécurité et verrouillage

Exemples de backends :

- local -> fichier sur disque local
- s3 -> bucket S3 pour un projet AWS
- azurerm -> stockage dans un compte Azure
- consul -> backend HashiCorp Consul

Contenu d'un fichier d'état

- Liste de toutes les ressources gérées par Terraform
- Attributs réels de chaque ressource (ID, ARN, IP, tags...)
- Dépendances entre les ressources
- Métadonnées du module et de la configuration

```
{
 "resources": [
   {
     "type": "aws_instance",
     "name": "app_server",
     "instances": [
       {
         "attributes": {
           "id": "i-0abcd1234efgh5678",
           "ami": "ami-0c55b159cbfafe1f0",
           "instance_type": "t2.micro"
         }
       }
     ]
   }
 ]
}
```

bstorm

Rôle de l’état dans la gestion des ressources

Création (apply)

- Terraform écrit les nouvelles ressources dans le fichier d’état après leur création.

Mise à jour (plan et apply)

- Terraform compare l’état actuel (terraform.tfstate) avec la configuration
- Détermine les changements nécessaires pour atteindre l’état désiré

Suppression (destroy)

- Terraform lit le fichier d’état pour savoir quelles ressources supprimer

Commandes utiles liées à l'état

Commande	Description
`terraform show`	Affiche l'état actuel (contenu de `tfstate`)
`terraform state list`	Liste toutes les ressources gérées
`terraform state show <resource>`	Détails d'une ressource spécifique
`terraform state rm <resource>`	Supprime une ressource du state (pas du cloud)
`terraform state mv <source> <destination>`	Renomme ou déplace une ressource dans le state

Bonnes pratiques pour l’état Terraform

- Ne jamais modifier le fichier .tfstate manuellement
- Utiliser un backend distant pour les équipes
- Versionner et sécuriser le state (IAM, chiffrement)
- Utiliser des workspaces pour gérer plusieurs environnements (dev, staging, prod)
- Sauvegarder régulièrement l’état

État local vs distant

État local :

- Simple à mettre en place
- Convient aux projets individuels
- Limité pour les équipes

État distant (Remote Backend) :

- Stockage centralisé (S3, Terraform Cloud, Azure Blob)
- Verrouillage automatique pour éviter les conflits
- Historique des modifications

Exemple backend S3

```
terraform {
 backend "s3" {
   bucket         = "my-terraform-state"
   key            = "prod/terraform.tfstate"
   region         = "us-east-1"
   dynamodb_table = "terraform-locks"  # pour le verrouillage
   encrypt        = true
 }
}
```

- bucket -> bucket S3 pour stocker l'état
- key -> chemin du fichier d'état dans le bucket
- dynamodb_table -> table DynamoDB pour verrouiller l'état pendant les opérations
- encrypt -> chiffrage côté serveur

Fonctionnement de la collaboration

- Stockage centralisé de l'état
- Verrouillage automatique pour éviter que plusieurs utilisateurs modifient le même état en même temps
- Chaque utilisateur exécute terraform plan et terraform apply -> état mis à jour dans le backend
- Historique et versionning -> possibilité de revenir à un état précédent si nécessaire

Commandes utiles avec backends distants

Commande	Description
`terraform init`	Initialise le backend distant et télécharge les providers
`terraform plan`	Compare la configuration avec l'état distant
`terraform apply`	Applique les changements et met à jour l'état distant
`terraform state pull`	Récupère l'état distant localement
`terraform state push`	Pousse un état local vers le backend distant

Bonnes pratiques

- Toujours utiliser un backend distant pour les équipes
- Activer le verrouillage de l'état pour éviter les conflits
- Protéger les backends distants avec des permissions et IAM
- Utiliser Terraform Cloud ou S3 + DynamoDB pour les projets multi-utilisateurs
- Séparer les environnements via des workspaces (dev, staging, prod)

Résumé

- Le state est le cœur de Terraform
- Permet de suivre l'infrastructure réelle et planifier des changements fiables
- Différence majeure : configuration déclarative vs état réel
- Stockage local ou distant, toujours sauvegardé et sécurisé

Création et Gestion d'Infrastructure

bstorm TRAINING

Provider

Les Providers permettent à Terraform de communiquer avec différents services cloud ou systèmes.

Exemples : AWS, Azure, GCP, Kubernetes, Docker, etc.

Points clés :

- Chaque provider a ses propres ressources et data sources
- Terraform télécharge les providers nécessaires lors de terraform init
- Permet de gérer des infrastructures multi-cloud

Provider

Plugin qui permet à Terraform de gérer un service (AWS, Azure, GCP, Kubernetes, Docker, etc.).

Pour le cloud AWS

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

Pour LocalStack

```
provider "aws" {
 region = "us-west-2"

 access key                  = "anaccesskey"
 secret key                  = "asecretkey"
 s3 use path style           = true
 skip credentials validation = true
 skip metadata api check     = true
 skip_requesting_account_id  = true

 endpoints {
   ec2 = "http://localhost:4566"
 }
}
```

Variables

Les variables permettent de paramétrer les configurations Terraform.
https://developer.hashicorp.com/terraform/language/block/variable

Avantages :

- Réutilisation du code
- Flexibilité et personnalisation
- Séparation entre configuration et valeurs

Types de variables :

- string -> texte
- number -> nombre
- bool -> vrai/faux
- list / map / object / tuple -> structures plus complexes

Variables

Définition d'une variable :

- description -> explique l'usage de la variable
- type -> type attendu
- default -> valeur par défaut si aucune valeur n'est fournie

```
variable "region" {
 description = "La région AWS où déployer les ressources"
 type        = string
 default     = "us-east-1"
}
```

Variables : Sensitive

```
variable db_password {
 description = "The password for the database"
 type        = string
 sensitive   = true
}
```

Les variables marquées comme “sensitive” ne seront pas affichées lors de l'exécution de la ligne de commande.

Variables

Définir les valeurs :

- Par terraform.tfvars ou dans un fichier :

```
region        = "eu-west-1"
instance_type = "t2.micro"
```

- Par la ligne de commande :

```
$ terraform apply -var="region=eu-west-1" -var="instance_type=t2.micro"
```

- Par les variables d'environment du terminal :

```
$ export TF_VAR_region="eu-west-1"
$ terraform apply
```

Variables Complexes

```
variable "availability_zones" {
 type    = list(string)
 default = ["us-east-1a", "us-east-1b"]
}

variable "ami_map" {
 type = map(string)
 default = {
   "us-east-1" = "ami-0abcd1234"
   "eu-west-1" = "ami-0abcd5678"
 }
}

variable "server_config" {
 type = object({
   cpu    = number
   memory = number
 })
 default = {
   cpu    = 2
   memory = 4096
 }
}
```

Variables (Exemple d’utilisation)

```
variable "instance_type" {
 description = "Type de l’instance EC2"
 type        = string
 default     = "t2.micro"
}

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

 tags = {
   Name = "terraform-app"
 }
}
```

Variables : Bonnes pratiques

- Toujours documenter les variables (description)
- Utiliser des types pour éviter les erreurs
- Mettre des valeurs par défaut raisonnables
- Séparer les variables sensibles (sensitive = true) pour ne pas les afficher
- Versionner les fichiers .tfvars si nécessaire

Ressources

Les resources représentent les éléments réels à créer, modifier ou supprimer dans le cloud ou tout autre provider.

Chaque resource correspond à un service ou composant (VM, bucket, base de données, réseau...).

https://developer.hashicorp.com/terraform/language/block/resource

Ressources : Structure d'une ressource

La structure de base :

```
resource "<TYPE>" "<NAME>" {
 <ATTRIBUTS>
}
```

Ce qui donne :

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

 tags = {
   Name = "terraform-app"
 }
}
```

- TYPE : type de la ressource dans le provider (aws_instance, aws_s3_bucket, azurerm_resource_group, etc.)
- NAME : nom local pour référencer la ressource dans Terraform
- ATTRIBUTS : configuration spécifique à la ressource (AMI, type de machine, tags...)

Ressources : Exemple complet d'instance EC2

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

 tags = {
   Name = "terraform-app"
 }
}
```

- Crée une VM EC2 avec l'AMI spécifiée
- Type de machine : t2.micro
- Ajoute un tag pour identifier la ressource

Ressources : Dépendances

Les ressources peuvent dépendre d'autres ressources ; ici aws_subnet.subnet1 dépend de aws_vpc.main donc Terraform crée d'abord le VPC, puis le subnet.

Dépendance implicite :

```
resource "aws_vpc" "main" {
 cidr_block = "10.0.0.0/16"
}

resource "aws_subnet" "subnet1" {
 vpc_id     = aws_vpc.main.id
 cidr_block = "10.0.1.0/24"
}
```

Ressources : Dépendances Explicite

```
resource "aws_vpc" "main" {
 cidr_block = "10.0.0.0/16"
}

resource "aws_subnet" "subnet1" {
 # Valeur fixe ou variable au lieu d'une référence directe
 vpc_id     = "vpc-xxxxxxx"
 cidr_block = "10.0.1.0/24"

 depends_on = [
   aws_vpc.main
 ]
}
```

Ressources : Utilisation des variables

Les ressources peuvent être paramétrées via des variables ce qui permet de réutiliser le même code pour différents environnements.

```
variable "instance_type" {
 default = "t2.micro"
}

variable "ami" {
 default = "ami-0c55b159cbfafe1f0" # Amazon Linux 2 AMI
}

resource "aws_instance" "app" {
 ami           = var.ami
 instance_type = var.instance_type
}
```

Ressources : Ressources multiples avec count ou for_each

Count permet de créer 3 instances identiques

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

for_each permet de créer des instances pour chaque éléments dans la variable “bucket_names”

```
variable "bucket_names" {
 description = "List of S3 bucket names to create"
 type        = set(string)
 default     = ["my-unique-bucket-12345", "another-unique-bucket-67890"]
}

resource "aws_s3_bucket" "buckets" {
 for_each = var.bucket_names
 bucket   = each.value
}
```

Ressources : Utilisation du block Data

```
data "aws_ami" "ubuntu" {
 most_recent = true

 filter {
   name = "name"
   values = ["ubuntu/images/hvm-ssd-gp3/ubuntu-noble-24.04-amd64-server-*]"
 }
 owners = ["099720109477"] # Canonical
}

resource "aws_instance" "app_server" {
 ami           = data.aws_ami.ubuntu.id
 instance_type = "t2.micro"

 tags = {
   Name = "learn-terraform"
 }
}
```

Ressources : Utilisation du block Data (Exemple avec AWS)

On peut utiliser le block data pour récupérer des informations depuis le Cloud Provider où d'autres ressources.
Ici je fais une query vers AWS pour récupérer l'AMI (Amazon Machine Image) respectant le filtre. Ici je filtre sur le nom de l'image. Cela me permet de ne pas avoir l'AMI ID dans la configuration, ce qui permet de simplifier la compréhension de celle-ci, cela permet aussi d'éviter d'avoir des données Hardcodée et d'avoir une configuration dynamique.
https://developer.hashicorp.com/terraform/language/block/data

```
data "aws_ami" "ubuntu" {
 most_recent = true

 filter {
   name = "name"
   values = ["ubuntu/images/hvm-ssd-gp3/ubuntu-noble-24.04-amd64-server-*]"
 }
 owners = ["099720109477"] # Canonical
}
```

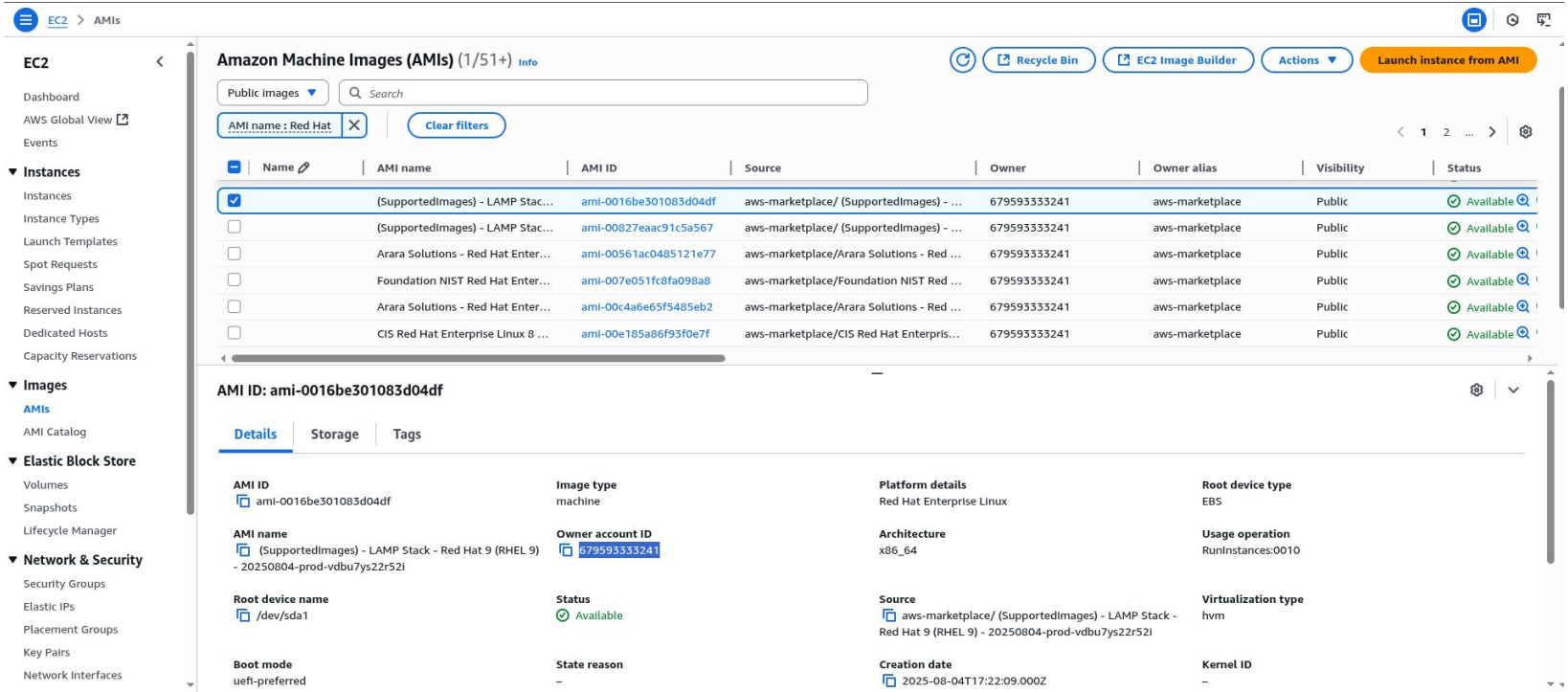
Le paramètre "*owners*" permet de restreindre la recherche aux images publiées par un propriétaire donné. Ici : 099720109477 = l'ID officiel du compte Canonical (éditeur d'Ubuntu). Ça garantit qu'on utilise une AMI officielle et fiable, et pas une image publiée par un inconnu.

Ressources : Utilisation du block Data

Pour avoir la liste des AMI :
https://eu-north-1.console.aws.amazon.com/ec2/home?region=eu-north-1#AMICatalog

Exemple de recherche :
https://eu-north-1.console.aws.amazon.com/ec2/home?region=eu-north-1#Images:visibility=public-images;imageName=:Red%20Hat;v=3;$case=tags:false%5C,client:false;$regex=tags:false%5C,client:false

Ressources : Utilisation du block Data



bstorm

Ressources : Utilisation du block Data

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

 tags = {
   Name = "learn-terraform"
 }
}
```

- ami : l'AMI à utiliser (récupérée depuis le block data)
- instance_type : Définit le type de machine EC2 : CPU, RAM, capacité réseau. (t2.micro => serveur gratuit AWS)
- tags : Permet de taguer la ressource pour l'identifier dans AWS.

Ressources : Bonnes pratiques

- Nommer les resources clairement et de manière cohérente
- Utiliser variables et outputs pour paramétrer et réutiliser les valeurs
- Documenter les tags et noms pour la gestion et la facturation
- Utiliser count / for_each pour gérer des ressources multiples de façon dynamique
- Séparer les resources critiques dans des fichiers dédiés pour plus de lisibilit

Outputs

Les outputs permettent de récupérer des valeurs d'une configuration Terraform après le déploiement.
https://developer.hashicorp.com/terraform/language/block/output

Avantages :

- Facilite la réutilisation des valeurs dans d'autres modules ou projets
- Permet d'afficher les informations importantes après terraform apply
- Intégration facile avec CI/CD ou scripts externes

Outputs

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

- description -> explique l'usage de l'output
- value -> la valeur à retourner
- Affiché après un terraform apply

Outputs : Afficher les outputs

- Après avoir “apply” :

```
$ terraform apply
```

Sortie :

Apply complete! Resources: 1 added, 0 changed, 0 destroyed.

Outputs:

instance_id = "i-0abcd1234efgh5678"

- Avec la commande dédiée :

Pour voir toutes les outputs :

```
$ terraform output
```

Pour voir une output d’une instance spécifique :

```
$ terraform output instance_id
```

Outputs : Exemple

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

 tags = {
   Name = "terraform-app"
 }
}

output "app_instance_id" {
 description = "ID de l'instance EC2"
 value       = aws_instance.app.id
}

output "app_public_ip" {
 description = "Adresse IP publique"
 value       = aws_instance.app.public_ip
}
```

Outputs Complexes

```
variable db_password {
 description = "The password for the database"
 type        = string
 sensitive   = true
}

output "instance_info" {
 value = {
   id   = aws_instance.app.id
   ip   = aws_instance.app.public_ip
   type = aws_instance.app.instance_type
 }
}

output "db_password" {
 value     = var.db_password
 sensitive = true
}
```

Les outputs marquée “sensitive” ne seront pas affichés dans la sortie standard pour protéger les informations sensibles.

Outputs : Bonnes Pratiques

- Documenter les outputs (description)
- Nommer les outputs de manière claire et descriptive
- Utiliser sensitive = true pour les valeurs sensibles
- Limiter les outputs aux valeurs réellement utiles pour éviter le bruit
- Utiliser les outputs pour réutiliser des valeurs entre modules

Gestion des Providers et des Modules

bstorm TRAINING

Introduction aux Providers Terraform

Concept :

- Un provider est un plugin qui permet à Terraform de communiquer avec un service externe (cloud, SaaS, Kubernetes...).
- Terraform utilise les providers pour créer, lire, mettre à jour et supprimer des ressources.

Exemples de providers :

- Cloud : aws, azure, google
- Orchestration : kubernetes, docker
- Services : vault, github, datadog

Rôle du provider

Fonctions principales :

- Authentification auprès du service (API, credentials)
- Fournir les types de ressources spécifiques au service
- Gérer l'état des ressources créées
- Permettre l'interopérabilité multi-cloud avec d'autres providers

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

Ce que fait ce code :

- Connecte Terraform à AWS
- Permet d'utiliser des ressources comme aws_instance, aws_s3_bucket, etc.

Déclaration de plusieurs providers

Terraform permet d'utiliser plusieurs providers dans le même projet
Exemple : gérer AWS et Azure dans le même plan

```
provider "aws" {
 alias  = "us" # alias -> permet de différencier plusieurs instances d'un même provider
 region = "us-east-1"
}

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

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

Utilisation d’un provider spécifique

On peut lier une ressource à un provider particulier avec provider = <provider>.

```
resource "aws_instance" "us_server" {
 provider      = aws.us
 ami           = "ami-0c55b159cbfafe1f0"
 instance_type = "t2.micro"
}

resource "aws_instance" "eu_server" {
 provider      = aws.eu
 ami           = "ami-0c55b159cbfafe1f0"
 instance_type = "t2.micro"
}
```

Exemple multi-cloud (AWS + Azure)

```
provider "aws" {
 region = "us-east-1"
}

provider "azurerm" {
 features {}
}

resource "aws_s3_bucket" "my_bucket" {
 bucket = "my-terraform-bucket"
}

resource "azurerm_resource_group" "rg" {
 name     = "tf-demo-rg"
 location = "West Europe"
}
```

Bonnes pratiques multi-provider

- Toujours nommer les alias pour éviter les confusions
- Séparer les ressources par provider dans le code pour plus de lisibilité
- Utiliser workspaces ou modules pour gérer différents clouds
- Valider chaque provider indépendamment avant un terraform apply global

Modules

Les modules permettent de regrouper et réutiliser des ressources Terraform.
https://developer.hashicorp.com/terraform/language/block/module

Un module peut être :

- Local : dans le même projet
  - https://developer.hashicorp.com/terraform/language/modules/develop
- Remote : depuis le registre Terraform (registry.terraform.io) ou Git
  - https://developer.hashicorp.com/terraform/language/modules/configuration

Avantages :

- Réutilisabilité du code
- Meilleure organisation
- Facilite la maintenance et les bonnes pratiques

Modules (Exemple d'utilisation de module distant)

Comment trouver des modules distants : https://registry.terraform.io/browse/modules

Avantages des modules distant :

- Ne pas réinventer la roue
- Modules validés par la communauté
- Maintenance simplifiée

Modules (Exemple d'utilisation de module distant)

https://registry.terraform.io/modules/terraform-aws-modules/vpc/aws/latest

```
module "vpc" {
 source  = "terraform-aws-modules/vpc/aws"
 version = "4.0.0"

 name = "my-vpc"
 cidr = "10.2.0.0/16"

 azs             = ["us-east-1a", "us-east-1b"]
 public_subnets  = ["10.2.1.0/24", "10.2.2.0/24"]
 private_subnets = ["10.2.101.0/24", "10.2.102.0/24"]
}
```

Modules (Exemple de création de module local)

```
modules/
└── network/
    ├── main.tf      # Définition des resources (VPC, subnets, etc.)
    ├── variables.tf  # Variables d'entrée du module
    └── outputs.tf    # Variables de sortie du module
```

Modules (Exemple de création de module local)

Fichier modules/network/variables.tf

```
variable "vpc_cidr" {
 description = "CIDR block for the VPC"
 type        = string
 default     = "10.0.0.0/16"
}
```

Modules (Exemple de création de module local)

Fichier modules/network/outputs.tf :

```
output "vpc_id" {
 value = aws_vpc.main.id
}
```

Modules (Exemple de création de module local)

Fichier modules/network/main.tf :

```
resource "aws_vpc" "main" {
 cidr_block = var.vpc_cidr
 tags = {
   Name = "terraform-demo-vpc"
 }
}

resource "aws_subnet" "subnet1" {
 vpc_id            = aws_vpc.main.id
 cidr_block        = "10.0.1.0/24"
 availability_zone = "us-east-1a"
}
```

Modules (Exemple de création de module local)

Fichier main.tf :

```
module "network" {
 source   = "./modules/network"
 vpc_cidr = "10.1.0.0/16"
}

output "my_vpc_id" {
 value = module.network.vpc_id
}
```

Modules (Exemple de création de module local)

Bonnes pratiques :

- Nommer les modules de manière claire et descriptive
- Toujours versionner les modules distants
- Utiliser des variables et outputs pour l'interface du module
- Documenter les paramètres requis et optionnels
- Tester le module en local avant intégration

Planification, Application et Gestion des Modifications

bstorm TRAINING

Introduction aux commandes Terraform

Terraform suit un modèle déclaratif : vous décrivez l'infrastructure souhaitée et Terraform s'occupe de la créer ou de la modifier.

Les trois commandes clés pour gérer l'infrastructure :

- terraform plan -> prévisualisation des changements
- terraform apply -> application des changements
- terraform destroy -> suppression des ressources

terraform plan

Objectif :

- Générer un plan d'exécution qui montre les actions que Terraform effectuera.
- Permet de vérifier les changements avant de les appliquer.

Exemple :

```
$ terraform plan
```

Sortie :

Plan: 2 to add, 1 to change, 0 to destroy.

Avantages :

- Éviter les erreurs
- Vérifier que les modifications correspondent aux attentes

terraform apply

Objectif :

- Appliquer réellement les changements dans le cloud ou l'infrastructure.
- Terraform se base sur le plan généré (ou crée un plan automatiquement).

Exemple :

```
$ terraform apply
```

Appliquer une planification :

```
$ terraform plan -out=tfplan
$ terraform apply tfplan
```

Permet de séparer planification et application, utile en CI/CD

Gestion des mises à jour

Terraform compare l'état actuel (state) avec la configuration et applique :

- Ajout de nouvelles ressources
- Modification des ressources existantes
- Suppression des ressources supprimées du code

```
resource "aws_instance" "app" {
 ami           = "ami-0c55b159cbfafe1f0"
 instance_type = "t2.small"  # changement depuis t2.micro
}
```

- terraform plan -> montre modification du type d'instance
- terraform apply -> met à jour l'instance

terraform destroy

Objectif :

- Supprimer toutes les ressources gérées par Terraform ou celles ciblées.

```
$ terraform destroy
```

- Affiche un plan de suppression avant exécution
- Demande confirmation (yes)
- Utile pour nettoyer un environnement de test ou supprimer un projet complet

Supprime uniquement la ressource spécifiée :

```
$ terraform destroy -target=aws_instance.app
```

Bonnes pratiques pour plan, apply, destroy

- Toujours exécuter terraform plan avant apply pour éviter les surprises
- Versionner la configuration pour pouvoir revenir en arrière si nécessaire
- Séparer les environnements (dev, staging, prod) avec des workspaces
- Pour destroy, être prudent sur les environnements de production
- Utiliser plans sauvegardés (-out=tfplan) pour CI/CD fiable

Résumé

Commande	Objectif
`terraform plan`	Prévisualiser les changements
`terraform apply`	Appliquer les changements dans l'infrastructure
`terraform destroy`	Supprimer les ressources gérées par Terraform

- Processus recommandé : plan -> apply -> destroy (si nécessaire)
- Ces commandes permettent de gérer l'infrastructure en toute sécurité et de façon prévisible

Orchestration et Collaboration avec Terraform

bstorm TRAINING

Gestion des environnements avec Terraform

Problématique :

- Une même infrastructure doit exister en plusieurs environnements :
  - Dev → tests rapides, faible coût
  - Test/Staging → proche de la prod, validations
  - Prod → hautement disponible et robuste

Défis :

- Garder une configuration cohérente
- Pouvoir paramétrer chaque environnement (ex: tailles d'instances, régions)
- Éviter les erreurs (ex : déploiement accidentel en prod)

Approches pour gérer les environnements

```
environments/
├── dev/
│   ├── main.tf
│   └── terraform.tfvars
├── staging/
└── prod/
```

Séparation par workspaces

- Un seul code, plusieurs contextes
- Terraform choisit le bon workspace pour isoler les ressources

Séparation par projets/repos

- Utilisé dans les grandes organisations
- Chaque environnement = code et état distinct

Workspaces Terraform

Concept :

- Les workspaces permettent d'avoir plusieurs instances d'un même code Terraform, avec des états différents.
- Pratique pour gérer dev, test, prod dans une seule configuration.

Commandes principales

```
$ terraform workspace list        # Lister les workspaces
$ terraform workspace new dev     # Créer un workspace dev
$ terraform workspace select prod # Basculer sur prod
```

Exemple d’utilisation des workspaces

```
variable "instance_type" {
 default = "t2.micro"
}

resource "aws_instance" "app" {
 ami           = "ami-0c55b159cbfafe1f0"
 instance_type = terraform.workspace == "prod" ? "t3.large" : "t2.micro"
}
```

- En dev/test -> petites machines peu coûteuses
- En prod -> machines plus puissantes

Collaboration avec Terraform Cloud

Terraform Cloud offre :

- Stockage centralisé de l'état
- Verrouillage automatique pour éviter les conflits
- Workspaces gérés directement depuis l'interface
- Exécution distante (plus besoin de lancer Terraform en local)
- Gestion des variables et secrets

Exemple configuration :

```
terraform {
 backend "remote" {
   organization = "my-org"
   workspaces {
     name = "prod"
   }
 }
}
```

Backends distants partagés

Objectif :

- Stocker l'état Terraform de manière sécurisée et collaborative.

Exemples :

- AWS S3 + DynamoDB → stockage + verrouillage
- Azure Storage
- Google Cloud Storage (GCS)
- Terraform Cloud

```
terraform {
 backend "s3" {
   bucket         = "my-tf-state"
   key            = "prod/terraform.tfstate"
   region         = "us-east-1"
   dynamodb_table = "tf-locks"
 }
}
```

Bonnes pratiques multi-environnements

- Toujours isoler les environnements (dossiers, workspaces, backends)
- Nommer clairement les workspaces (dev, staging, prod)
- Restreindre les droits IAM → éviter qu'un dev déploie en prod par erreur
- Utiliser un backend distant pour la collaboration
- Automatiser avec CI/CD pour uniformiser les déploiements

Provisioner

bstorm TRAINING

Introduction

Les provisioners dans Terraform sont des mécanismes permettant d'exécuter des actions additionnelles sur des ressources après leur création. Ils servent principalement à installer, configurer ou initialiser des machines virtuelles une fois qu'elles sont disponibles.

Utilité principale

- Gestion et configuration post-déploiement
- Injection de fichiers ou scripts
- Intégration complémentaire avec des systèmes externes

Contexte d'utilisation

Terraform adopte un modèle déclaratif, tandis que les provisioners représentent un comportement impératif. Leur usage doit rester exceptionnel, car ils introduisent des risques de non-déterminisme.

Terraform recommande de privilégier des outils spécialisés tels qu'Ansible, Packer, cloud-init ou Systems Manager, mais maintient les provisioners pour les cas non couverts par le modèle déclaratif.

Fonctionnement dans le cycle Terraform

Les provisioners s'exécutent exclusivement après que la ressource concernée est créée et disponible. Ils ne modifient pas l'état Terraform directement.

Ils peuvent être configurés avec deux comportements :

- on_failure = continue (défaut) permet de continuer même si la commande échoue
- on_failure = fail provoque un arrêt et un statut d'erreur

Limitation critique :

En cas d'échec, Terraform peut perdre la synchronisation sur l'état de la ressource. Il peut alors tenter de détruire et recréer la ressource, augmentant les risques de déploiement instable.

Liste des provisioners natifs Terraform

Terraform propose trois provisioners intégrés :

- file
  Copie de fichiers entre la machine exécutant Terraform et une instance distante

- remote-exec
  Exécution de commandes ou scripts sur une machine distante

- local-exec
  Exécution de commandes localement sur l'hôte Terraform

Les autres provisioners (Chef, Puppet, Salt, Habitat) sont désormais dépréciés ou fortement déconseillés.

Provisioner : File

Le provisioner file permet de transférer des fichiers ou répertoires locaux vers une machine cible distante.

Cas d'usage typiques

- Déploiement de fichiers de configuration
- Téléchargement d'un binaire avant installation
- Préparation avant exécution d'un script via remote-exec

Paramètres disponibles :

Paramètre	Description
`source`	Fichier ou répertoire local à copier
`content`	Contenu à écrire directement dans un fichier distant
`destination`	Emplacement remote (fichier ou dossier)
`connection {}`	Bloc de configuration SSH ou WinRM

Bloc de connexion (connection)

Il définit la méthode d'accès à la machine distante.

Possibilités :

- type = "ssh" (Linux, Unix)
- type = "winrm" (Windows Server)

Paramètres spécifiques SSH :

- user
- private_key ou password
- host ou self.public_ip
- port optionnel

Paramètres spécifiques WinRM :

- user
- password
- https vrai ou faux
- timeout configurable

Provisioner : File

```
resource "aws_instance" "app" {
 ami                    = var.ami
 instance_type          = var.instance_type
 subnet_id              = var.subnet_id
 vpc_security_group_ids = var.security_group_ids

 tags = {
   Name = "terraform-demo"
 }

 provisioner "file" {
   source      = "conf/myapp.conf"
   destination = "/tmp/myapp.conf"

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

bstorm

Provisioner : Remote-Exec

Ce provisioner exécute des commandes sur la machine distante pour effectuer :

- installation de packages
- lancement ou configuration de services
- initialisation d'applications ou conteneurs

Paramètre	Description
`inline`	Liste de commandes exécutées directement
`script`	Script local envoyé et exécuté sur la machine
`scripts`	Liste de scripts locaux à exécuter
`connection {}`	Bloc de configuration SSH ou WinRM

Provisioner : Remote-Exec

```
resource "aws_instance" "app" {
 ami                    = var.ami
 instance_type          = var.instance_type
 subnet_id              = var.subnet_id
 vpc_security_group_ids = var.security_group_ids
 tags = {
   Name = "terraform-demo"
 }
 provisioner "remote-exec" {
 inline = [
   "sudo mv /tmp/myapp.conf /etc/myapp.conf",
   "sudo chmod 644 /etc/myapp.conf",
   "echo 'Fichier copié avec remote-exec'"
 ]
  connection {
    type        = "ssh"
    user        = "ubuntu"
    private_key = file("~/.ssh/id_rsa")
    host        = self.public_ip
  }
 }
}
```

bstorm

Provisioner : Local-Exec

Le provisioner local-exec exécute une commande localement sur la machine qui exécute Terraform. Il ne dépend d'aucune connexion distante.

Cas d'usage recommandés

- Génération de fichiers locaux (journalisation, inventaires)
- Interactions avec API externes
- Appels à d'autres orchestrateurs (Ansible, scripts CI/CD)

Paramètre	Description
`command`	Ligne de commande à exécuter localement
`working_dir`	Répertoire de travail
`environment`	Variables d'environnement injectées

Provisioner : Local-Exec

```
resource "aws_instance" "app" {
 ami                    = var.ami
 instance_type          = var.instance_type
 subnet_id              = var.subnet_id
 vpc_security_group_ids = var.security_group_ids

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

Gestion des erreurs provisioners

Option	Effet
`on_failure = "continue"`	Continue même si la commande échoue
`on_failure = "fail"`	Arrête le plan avec erreur

Sécurité et bonnes pratiques

Mises en garde officielles HashiCorp :

- Ne pas injecter de secrets en clair dans les scripts ou fichiers
- Risques élevés d'incohérence d'état Terraform
- Préférer les outils déclaratifs d'initialisation (User Data, cloud-init)
- Utiliser depends_on si un ordre strict est requis

Dans les environnements à grande échelle, les provisioners augmentent le risque de drift et ralentissent fortement les déploiements.

User Data

bstorm TRAINING

User Data

Le User Data correspond à un mécanisme d'initialisation automatisée d'une machine virtuelle lors de son premier démarrage. Il permet d'exécuter un script d'initialisation ou de configuration immédiatement après la création de l'instance, sans interaction humaine.

Objectifs principaux

- Configurer automatiquement le système à la première exécution
- Installer des packages et services
- Déployer des fichiers applicatifs
- Lancer des agents de supervision ou sécurité
- Initialiser des conteneurs ou orchestrateurs (Docker, Kubernetes, etc.)

Le user_data constitue un mécanisme déclaratif en amont du démarrage, contrairement aux provisioners Terraform exécutés après la création.

Fonctionnement technique général

Au démarrage d'une VM, l'hyperviseur ou la plateforme Cloud :

- Injecte le contenu du user_data dans l'environnement de l'OS invité
- Spécifie un interpréteur selon le type de script fourni
- Exécute les commandes lors du premier boot uniquement, sauf directive contraire

Support natif par cloud-init
Sur la majorité des distributions Linux cloud, l'exécution du user_data s'appuie sur cloud-init.

cloud-init assure

- Création d'utilisateurs
- Installation de paquets
- Configuration réseau
- Injection de clés SSH
- Gestion des fichiers, services, conteneurs

La présence d'un header permet au système de déterminer le type de contenu. Exemple :
#!/bin/bash
#cloud-config

Cas d’usage typiques

- Déployer une pile applicative sans intervention manuelle
- Déployer des secrets via paramétrage sécurisé (SSM, Vault, IMDS)
- Enregistrer l’instance auprès d’un cluster ou load balancer
- Adapter l’image générique à des besoins métiers spécifiques
- Initialiser Docker, un runtime ou configuration Kubernetes

Le user_data constitue une alternative solide aux provisioners remote-exec.

Déclaration du user_data avec AWS (Shellscript)

```
resource "aws_instance" "vm" {
 ami           = "ami-0abcd1234abcd5678"
 instance_type = "t3.micro"

 user_data = <<-EOF
   #!/bin/bash
   apt update -y
   apt install nginx -y
   systemctl enable nginx
 EOF
}
```

Déclaration du user_data avec AWS (cloud-init YAML)

```
resource "aws_instance" "vm" {
 ami           = "ami-0abcd1234abcd5678"
 instance_type = "t3.micro"

 user_data = <<-EOF
 #cloud-config
 package_update: true
 packages:
   - docker.io
 runcmd:
   - systemctl enable docker
   - systemctl start docker
 EOF
}
```

Paramètre	Rôle	Notes
`user_data`	Contenu inline non encodé	Lecteur direct
`user_data_base64`	Version encodée en base64	Recommandée pour éviter les erreurs d’encodage
`file()`	Lexer un fichier local	Sans encodage
`templatefile()`	Fichier avec variables	Idéal pour environnements dynamiques

```
resource "aws_instance" "vm" {
 ami           = "ami-0abcd1234abcd5678"
 instance_type = "t3.micro"

 user_data = file("init/script.sh")
}
```

```
user_data = templatefile("init/cloud-init.yaml", {
 environment = var.environment
})
```

Règles de sécurité fondamentales

Danger	Risque	Mitigation
Secrets en clair dans le user_data	Exposition via console Cloud ou journaling	Récupérer via IMDS, SSM Parameter Store, Vault
Exécution infinie ou en erreur	Machine inutilisable	Utiliser cloud-init modules et logs `/var/log/cloud-init.log`
Absence d'idempotence	Drift configurationnel	Modules `cloud-config` déclaratifs

Bonnes pratiques recommandées

- Préférer cloud-init au bash brut
- Encapsuler avec templatefile() pour supporter différents environnements
- Tester les scripts en local avec cloud-localds ou cloud-init analyze
- Stocker les scripts dans Git et non inline dans le code TF
- Structurer la configuration pour maintenir l'environnement stateless

Axe	User Data	Provisioners
Moment d'exécution	Au premier boot	Après création de ressource
Couplage	Direct au Cloud Provider	Couplé à Terraform
Robustesse	Stable, natif	Risque de désynchronisation
Idempotence	Vérifiée par cloud-init	Non garantie
Sécurité	Meilleure gestion des secrets	Souvent risqué

Déploiement d'un container docker en bash

```
resource "aws_instance" "docker_vm_bash" {
 ami           = "ami-0abcd1234abcd5678" # AMI Ubuntu ou Debian
 instance_type = "t3.micro"
 user_data = <<-EOF
   #!/bin/bash
   apt-get update -y
   apt-get install -y \
     apt-transport-https \
     ca-certificates \
     curl \
     gnupg-agent \
     software-properties-common
   curl -fsSL https://download.docker.com/linux/ubuntu/gpg | apt-key add -
   add-apt-repository \
     "deb [arch=amd64] https://download.docker.com/linux/ubuntu \
     $(lsb_release -cs) \
     stable"
   apt-get update -y
   apt-get install -y docker-ce docker-ce-cli containerd.io
   systemctl enable docker
   systemctl start docker
   docker run -d -p 80:80 --name web nginx
 EOF
```

Déploiement d'un container docker en cloud-init

```
resource "aws_instance" "docker_vm_cloudinit"{
 ami           = "ami-0abcd1234abcd5678" # AMI Ubuntu compatible cloud-init
 instance_type = "t2.micro"

 user_data = <<-EOF
 #cloud-config
 package_update: true
 packages:
   - docker.io
 runcmd:
   - systemctl enable docker
   - systemctl start docker
   - docker run -d -p 80:80 --name web nginx
 EOF
}
```

Différences et recommandations

Critère	Bash inline	cloud-init YAML
Robustesse	Moyenne	Excellente
Maintenabilité	Faible	Forte
Idempotence	Non garantie	Gérée
Logique de haut niveau	Limitée	Modules cloud-init disponibles
Bonnes pratiques DevOps	Déconseillé	Recommandé

Déploiement d'un docker-compose

```
resource "aws_instance" "vm_docker_compose"{
 ami           = "ami-0abcd1234abcd5678" # AMI Ubuntu compatible cloud-init
 instance_type = "t3.micro"

 # Convertir automatiquement YAML en base64 pour éviter les erreurs d'encodage
 user_data = base64encode(templatefile("deploy/cloud-init-docker-compose.yaml", {
   compose_file = file("deploy/docker-compose.yml")
 }))
}
```

cloud-init-docker-compose.yml

```
#cloud-config
package_update: true
packages:
 - docker.io
 - docker-compose

write_files:
 - path: /opt/docker/docker-compose.yml
   permissions: "0644"
   content: |
     ${compose_file}

runcmd:
 - systemctl enable docker
 - systemctl start docker
 - cd /opt/docker
 - docker-compose up -d
```

docker-compose.yml

```yaml
version: "3.9"

services:
 web:
   image: nginx
   ports:
     - "80:80"
```

Bonnes Pratiques et Sécurité avec Terraform

bstorm TRAINING

Gestion de la sécurité et des accès avec Terraform

Principes clés :

- Protéger les credentials et secrets
- Limiter les droits via le principe du moindre privilège (PoLP)
- Sécuriser l'accès aux fichiers d'état (terraform.tfstate)

Bonnes pratiques :

- Ne jamais mettre les clés dans le code
- Utiliser des variables d'environnement ou un gestionnaire de secrets (Vault, AWS Secrets Manager, Azure Key Vault)
- Stocker l'état dans un backend sécurisé (S3 + DynamoDB, Terraform Cloud, etc.)
- Mettre en place des rôles IAM restreints pour Terraform

```
provider "aws" {
 region = "us-east-1"
 # Mauvaise pratique : hardcoding !
 access_key = "XXXX"
 secret_key = "YYYY"
}
```

Contrôle d’accès et sécurité avancée

IAM (Identity and Access Management) / RBAC (Role-Based Access Control) : donner uniquement les permissions nécessaires à Terraform
Verrouillage de l’état : empêche les exécutions concurrentes destructives
Terraform Cloud / Enterprise :

- Intégration SSO
- Contrôle d’accès basé sur les rôles (RBAC)
- Politique Sentinel pour imposer des règles (ex : taille max d’instance)

Versionner les configurations Terraform

Pourquoi versionner ?

- Historique complet des changements
- Collaboration entre équipes
- Possibilité de rollback en cas de problème

Outils :

- Git (GitHub, GitLab, Bitbucket)
- GitOps → automatiser les déploiements à partir de Git

Bonnes pratiques Git :

- Un dépôt par projet Terraform
- Branches par environnement (dev, staging, prod)
- Revue de code (Pull Request/Merge Request) obligatoire avant apply

Audit et traçabilité

Git log -> qui a changé quoi et quand
Terraform plan dans les pipelines CI/CD -> garder un journal des changements planifiés
Terraform Cloud / Enterprise : audit log des exécutions
Tagging obligatoire des ressources pour traçabilité

```
resource "aws_instance" "app" {
 ami           = "ami-12345"
 instance_type = "t2.micro"
 tags = {
   Environment = terraform.workspace
   Owner       = "Team-Infra"
 }
}
```

Optimisation des configurations

Objectifs :

- Améliorer la performance des déploiements
- Garantir la lisibilité et la maintenabilité du code

Bonnes pratiques :

- Utiliser des modules -> factoriser le code réutilisable
- Variables et outputs clairs -> éviter le hardcoding
- Découpage par fichiers (main.tf, variables.tf, outputs.tf)
- Limiter les dépendances implicites
- Éviter les ressources inutiles (minimiser l'empreinte infra)

Exemple d’optimisation avec modules

Sans module (duplication) :

```
resource "aws_instance" "web1" { ... }
resource "aws_instance" "web2" { ... }
```

Avec module (réutilisable) :

```
module "web_servers" {
 source = "./modules/ec2"
 count  = 2
 instance_type = "t3.micro"
}
```

Avantages :

- Moins de duplication
- Plus facile à maintenir

Optimisation des performances Terraform

- terraform plan -parallelism=10 -> accélère les exécutions
- State backend distant optimisé (S3/DynamoDB, Terraform Cloud)
- Data sources -> éviter les valeurs codées en dur mais attention à la surcharge d'API
- Modules officiels validés (Terraform Registry) plutôt que réinventer la roue

Résumé

- Sécurité : pas de secrets en clair, IAM minimal, état protégé
- Versionnement/Audit : Git obligatoire, historique des changements, logs
- Optimisation : modules, variables, découpage clair, performance via parallélisme

Merci pour votre attention.

bstorm
TRAINING