# Terraform - Infrastructure-as-Code (IaC)

# Plan

- Introduction
- Concepts fondamentaux de Terraform
- Bonnes pratiques
- À vous de jouer !

# Plan

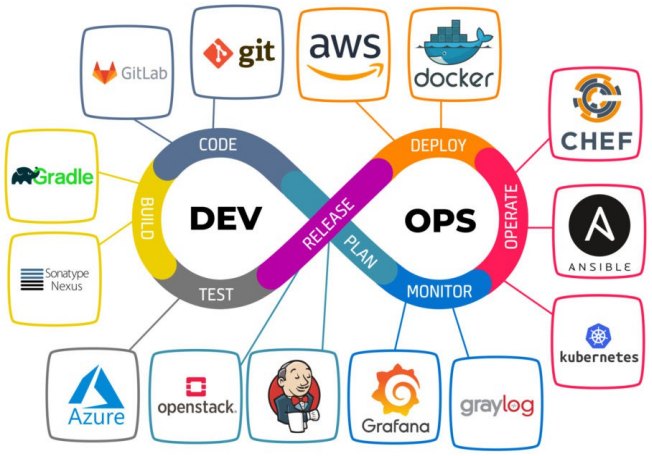
- Introduction
- Concepts fondamentaux de Terraform
- Bonnes pratiques
- À vous de jouer !

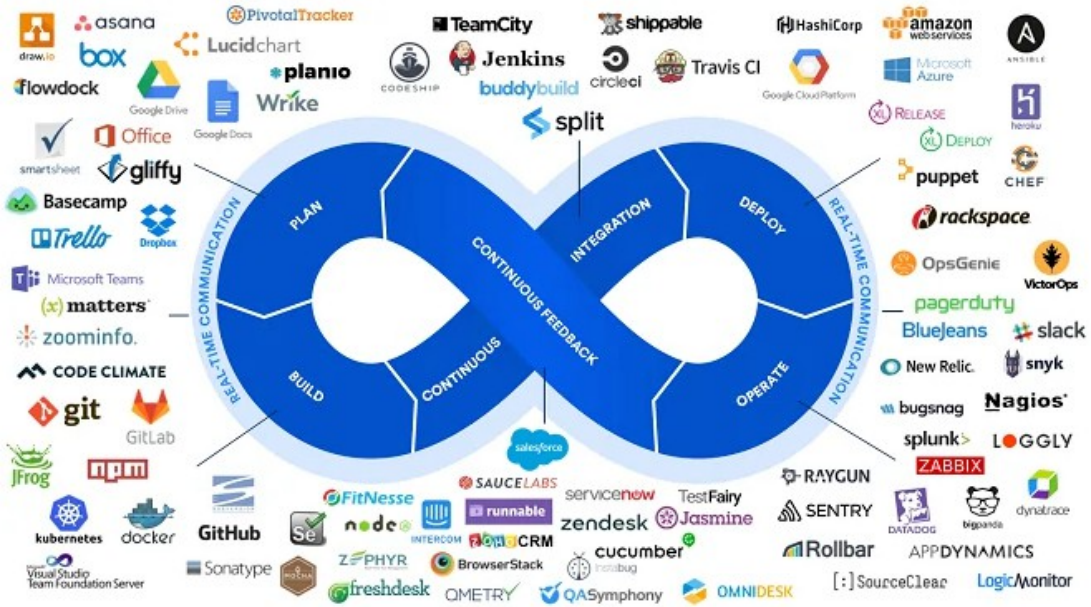
# Introduction

- Ce cours a pour objectif d'introduire le concept d'Infrastructure as Code (IaC) et de présenter Terraform comme un outil central de gestion de l'infrastructure moderne.
- Nous verrons comment automatiser la création, la modification et la suppression d'infrastructures,
  - aussi bien dans le cloud qu'en local, de manière fiable et reproductible.

# DevOps

- Les pratiques DevOps visent à réduire la durée des cycles de mise en production, à rendre le développement logiciel plus flexible (agile), et à combler le fossé entre les équipes de développement et d'exploitation.



# Quelle est la relation entre DevOps et le Cloud computing ?

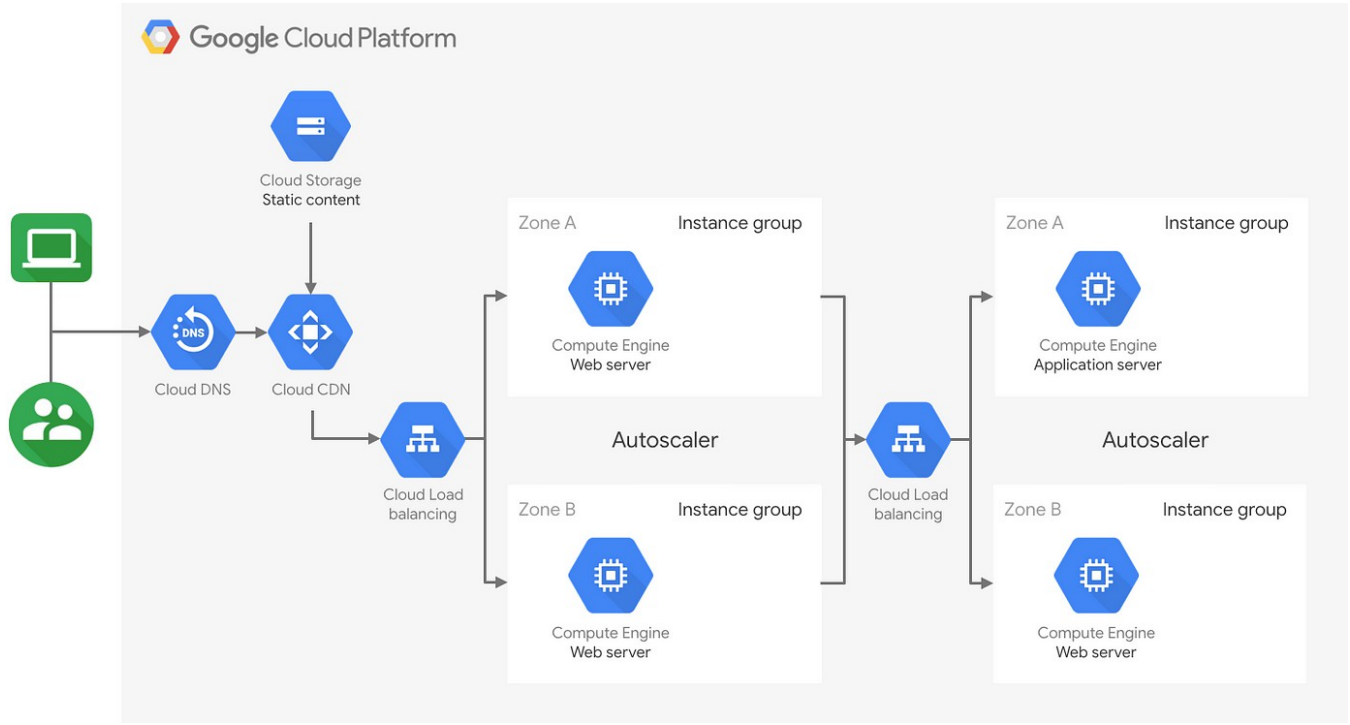
- De nombreuses applications sont aujourd'hui migrées vers des architectures micro-services et déployées dans le cloud, car :
  - les serveurs sont opérés par des fournisseurs tiers,
  - les entreprises ne paient que ce qu'elles consomment,
  - l'élasticité et la scalabilité sont faciles à mettre en œuvre,
  - l'infrastructure peut évoluer rapidement.
- Les cycles DevOps intègrent très souvent des opérations cloud.

# Infrastructures = APIs

- Le paradigme du cloud computing a transformé l'infrastructure (ainsi que les plateformes et les logiciels) en APIs externes accessibles par des requêtes.
- Aujourd'hui, les infrastructures se comportent comme des logiciels offrant des services.
- Par exemple, il est possible de :
  - faire une requête pour créer un bucket afin de stocker du contenu,
  - faire une requête pour provisionner un cluster GKE afin d'héberger des applications micro-services,
  - faire une requête pour créer des machines virtuelles destinées à héberger des bases de données ou des logiciels plus lourds,
  - etc.

# Exemple

- Architecture d'une application web scalable sur Google Cloud Platform, utilisant Cloud DNS, Cloud CDN, Cloud Load Balancing et des groupes d'instances Compute Engine répartis sur plusieurs zones, avec autoscaling pour assurer haute disponibilité et performance.



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

- L’Infrastructure as Code (IaC) vise à éviter les méthodes manuelles ou ponctuelles (création, mise à jour, suppression) pour gérer des infrastructures complexes.

- Elle consiste à considérer la gestion de l’infrastructure comme du code, pouvant être partagé, versionné, automatisé et maintenu de manière collaborative.

- Concepts associés :
  - approche impérative et approche déclarative
  - idempotence

# Différents types d'Infrastructure as Code

- Gestion de configuration
- Ces outils ont été conçus pour automatiser et rendre plus réutilisable et plus flexible la configuration des serveurs, des machines et des machines virtuelles.
- Exemples d'outils :
  - Puppet
  - Chef
  - Ansible

# Différents types d'Infrastructure as Code

- Outils de provisioning
- Ces outils ont été conçus pour automatiser et rendre plus réutilisable, plus flexible et plus sûr la gestion des infrastructures cloud.
- Exemples d'outils :
  - Heat (OpenStack)
  - CloudFormation (AWS)
  - **Terraform**
  - Pulumi

# Différents types d’Infrastructure as Code

- Outils d’orchestration
- Ces outils sont conçus pour orchestrer le cycle de vie d’un grand ensemble de conteneurs ainsi que leur déploiement sur des serveurs.
- Exemples d’outils :
  - Docker Swarm
  - Kubernetes
  - Nomad

# Différents types d’Infrastructure as Code

| Type d’outil IaC | Objectif principal | Rôle clé | Caractéristiques | Exemples d’outils |
|---|---|---|---|---|
| **Gestion de configuration** | Installer et maintenir la configuration des systèmes | Configurer des machines existantes | - Gestion des logiciels et services- Structure standardisée- Versionnable (Git)- Idempotent | Puppet<br>Chef<br>Ansible |
| **Outils de provisioning** | Créer et gérer l’infrastructure | Provisionner des ressources cloud | - Déclaratif- Gestion de l’état (state)- Multi-cloud (selon l’outil)- Idempotent | Heat (OpenStack)<br>CloudFormation (AWS)<br>**Terraform**<br>Pulumi |
| **Outils d’orchestration** | Gérer le cycle de vie des conteneurs | Orchestrer le déploiement applicatif | - Gestion de clusters- Scalabilité automatique- Haute disponibilité- Orchestration de conteneurs | Docker Swarm<br>Kubernetes<br>Nomad |

# Provisioning

- Dans ce module, nous nous concentrons sur un outil de provisioning déclaratif : Terraform.

- *state* : représente à la fois l'état courant de l'infrastructure et l'état désiré défini par l'utilisateur. Ces deux états peuvent évoluer au cours du temps.
- *reconciliation* : Les outils IaC déclaratifs cherchent à réconcilier l'état courant avec l'état désiré.
- *plan* : Pour effectuer cette réconciliation, les outils IaC déclaratifs génèrent automatiquement un plan décrivant les actions à exécuter ou à appliquer.

# Pourquoi utiliser un outil de provisioning ?

- « Je peux faire cela via les interfaces graphiques des fournisseurs cloud ! »
- Oui, mais…
  - Les procédures manuelles sont longues et sources d'erreurs.
  - La collaboration entre plusieurs personnes est difficile et risquée.
  - Il n'existe pas de vision claire et centralisée de l'état de l'infrastructure.
  - Cette approche ne passe pas à l'échelle.

- « Je peux le faire avec les CLI et des scripts des fournisseurs cloud ! »
- Oui, mais…
  - Il faut maîtriser autant de CLI que de fournisseurs cloud utilisés.
  - Un script est moins spécialisé et moins structuré que l'IaC, ce qui le rend plus difficile à écrire, lire et maintenir.
  - La gestion de l'état de l'infrastructure doit être faite manuellement, ce qui est complexe et source d'erreurs.

# Autres outils IaC de provisioning spécifiques aux fournisseurs cloud

- CloudFormation : utilise des classes TypeScript pour AWS
- Azure Resource Manager (ARM) : utilise un DSL spécifique à Azure
- Heat : utilise des templates YAML pour OpenStack

- Comparativement, Terraform permet de gérer l'ensemble des fournisseurs cloud à l'aide d'un seul fichier d'état.

- Pulumi
  - Concurrent direct de Terraform, avec des objectifs similaires
  - Réutilise les providers Terraform existants
  - Indépendant du langage : les équipes DevOps peuvent utiliser Python, Node.js, .NET, Go ou YAML

# Plan

- Introduction
- Concepts fondamentaux de Terraform
- Bonnes pratiques
- À vous de jouer !

# Terraform – Concepts clés (vue d'ensemble)

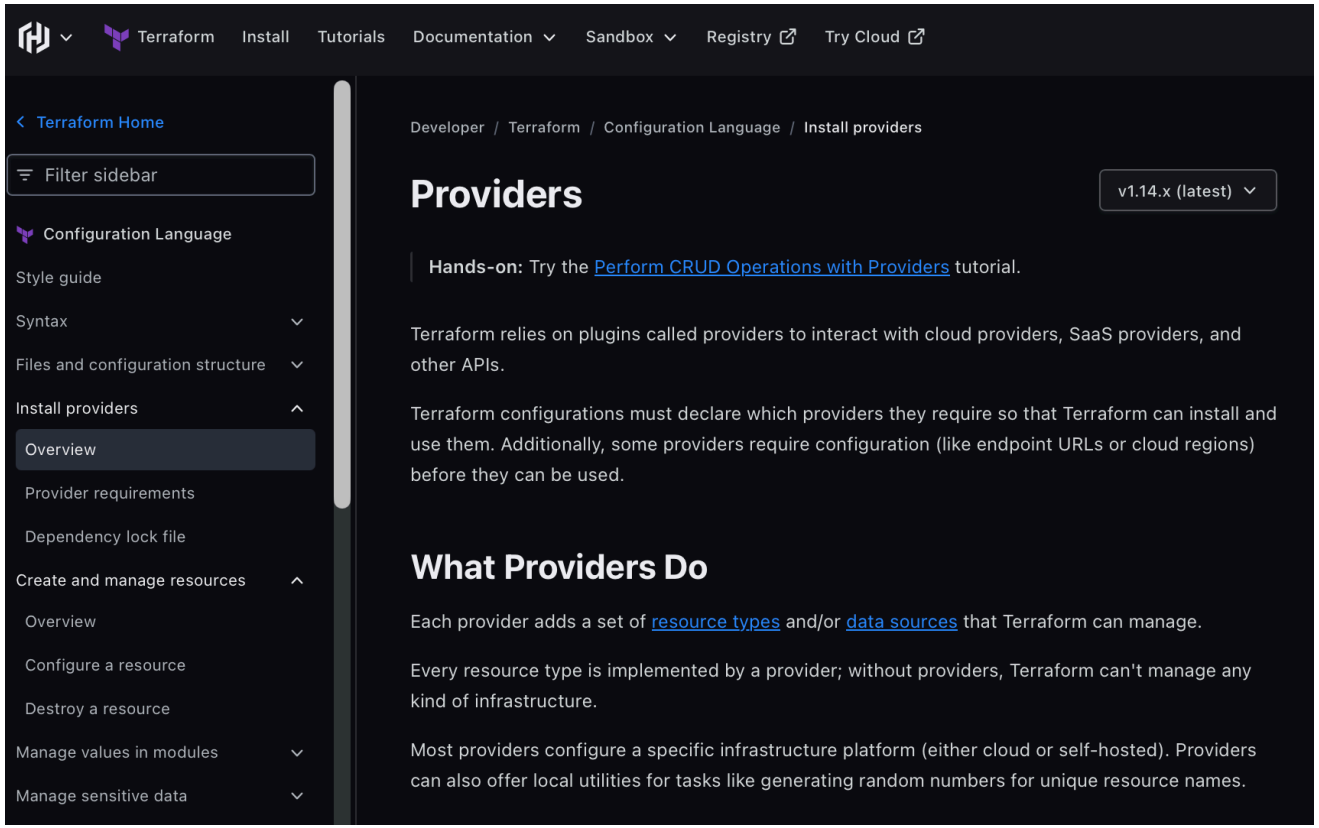
- Terraform repose sur une séparation claire des responsabilités
- L'utilisateur décrit l'intention
- Terraform Core calcule comment y parvenir
- Les providers exécutent les actions via des APIs
- Le state maintient la cohérence entre code et réalité
- Concepts clés :
  - Providers
  - Resources
  - State
  - Plan & Apply

# État déclaratif : on décrit ce que l'on veut, pas comment l'obtenir

- Terraform adopte une approche déclarative : on décrit ce que l'on veut, pas comment le faire.
- L'état désiré de l'infrastructure est défini par l'équipe DevOps *(ou l'administrateur,…)* dans des fichiers .tf.
  - L'ordre de provisionnement est déterminé automatiquement par Terraform.
  - Terraform crée l'infrastructure dans le bon ordre.
  - Cet ordre est déduit lorsque les ressources font référence les unes aux autres.
  - Les changements dans l'état déclaré sont comparés au fichier de state.
- Il est ainsi possible de créer plusieurs versions d'une même infrastructure répliquée (par exemple : développement, production).

# Architecture de base - Providers

- Un provider est un plugin Terraform
- Il permet à Terraform de communiquer avec une plateforme externe
- Il définit :
  - les types de ressources disponibles
  - les data sources
  - le format des paramètres
  - les appels API (Create, Read, Update, Delete)
- Exemples de providers :
  - local
  - azurerm
  - aws
  - google
  - kubernetes
  - docker




https://developer.hashicorp.com/terraform/language/providers

PROVIDER
composant (plug-in)
permettant de se connecter
à différents services

Official 

Distribués et maintenus
par Hashicorp

Partner 


Distribués et maintenus
par des tiers

Community

Distribués et maintenus par des
contributeurs individuels

# Architecture de base – Providers (local)

```
terraform {
  required_providers {
    local = {
      source  = "hashicorp/local"
      version = "~> 2.4"
    }
  }
}

provider "local" {}
```

- Le provider local permet de gérer des fichiers locaux
- Très utile pour apprendre Terraform sans cloud

# Architecture de base – Providers (local)

```
resource "local_file" "example" {
  filename = "hello.txt"
  content  = "Hello Terraform"
}
```

- Terraform gère ici une ressource locale
- Le fichier est créé, modifié ou supprimé via Terraform
- Le state garde la trace du fichier

# Architecture de base – Providers (cloud)

- Le provider ***azurerm*** permet de gérer Azure
- Il nécessite :
  - un abonnement Azure
  - une authentification (CLI, service principal, etc.)
- Terraform ne **remplace pas Azure** : il pilote les APIs Azure

# Architecture de base – Providers (cloud)

```
terraform {
  required_providers {
    azurerm = {
      source  = "hashicorp/azurerm"
      version = "~> 3.0"
    }
  }
}

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

- features {} est obligatoire
- L'authentification peut se faire via :
  - az login
  - variables d'environnement

# Architecture de base – Providers (cloud)

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

- Terraform crée un Resource Group Azure
- La ressource est décrite de manière déclarative
- Terraform sait la recréer ou la supprimer

# Architecture de base – Ressources

- Une ressource peut représenter n'importe quel élément de l'infrastructure,
  - par exemple : une machine virtuelle, une image Docker, un réseau virtuel, une adresse IP, un utilisateur, un compte, un rôle, etc.

- Les providers fournissent une API qui décrit :
  - les types de ressources disponibles,
  - les paramètres associés à chaque type de ressource.

- Exemple resource possède :
  - un type (azurerm_resource_group)
  - un nom logique (rg)
  - une configuration

# Notions fondamentales | resource

RESOURCE
objet pouvant être géré par Terraform

Fichiers

Machines virtuelles

Bases de données

Services cloud

network, rules, policies etc…

# Architecture de base – Terraform core

- Tous les fichiers **.tf** contiennent les déclarations des ressources.
- Le répertoire courant constitue le module racine (**root module**).
- Le fichier de **state** contient l'état actuel des ressources gérées par Terraform.
- À chaque appel de la CLI, le state est **rafraîchi** à partir de l'état réel des ressources.
- Terraform détecte les changements dans la configuration et planifie les appels API nécessaires en conséquence.

# Terraform State : pourquoi est-il crucial ?

- Le state garde la trace des ressources créées et permet à Terraform de savoir ce qui existe réellement.

- Le state est un fichier qui :
  - mappe le code aux ressources réelles
  - stocke les IDs des ressources

- Il permet à Terraform de :
  - savoir ce qui existe déjà
  - calculer les différences
  - éviter les doublons

- Sans state fiable, Terraform est dangereux.

STATE
représentation de l'infrastructure actuelle telle qu'elle existe dans le cloud ou dans l'environnement géré par Terraform

Fichier d'état

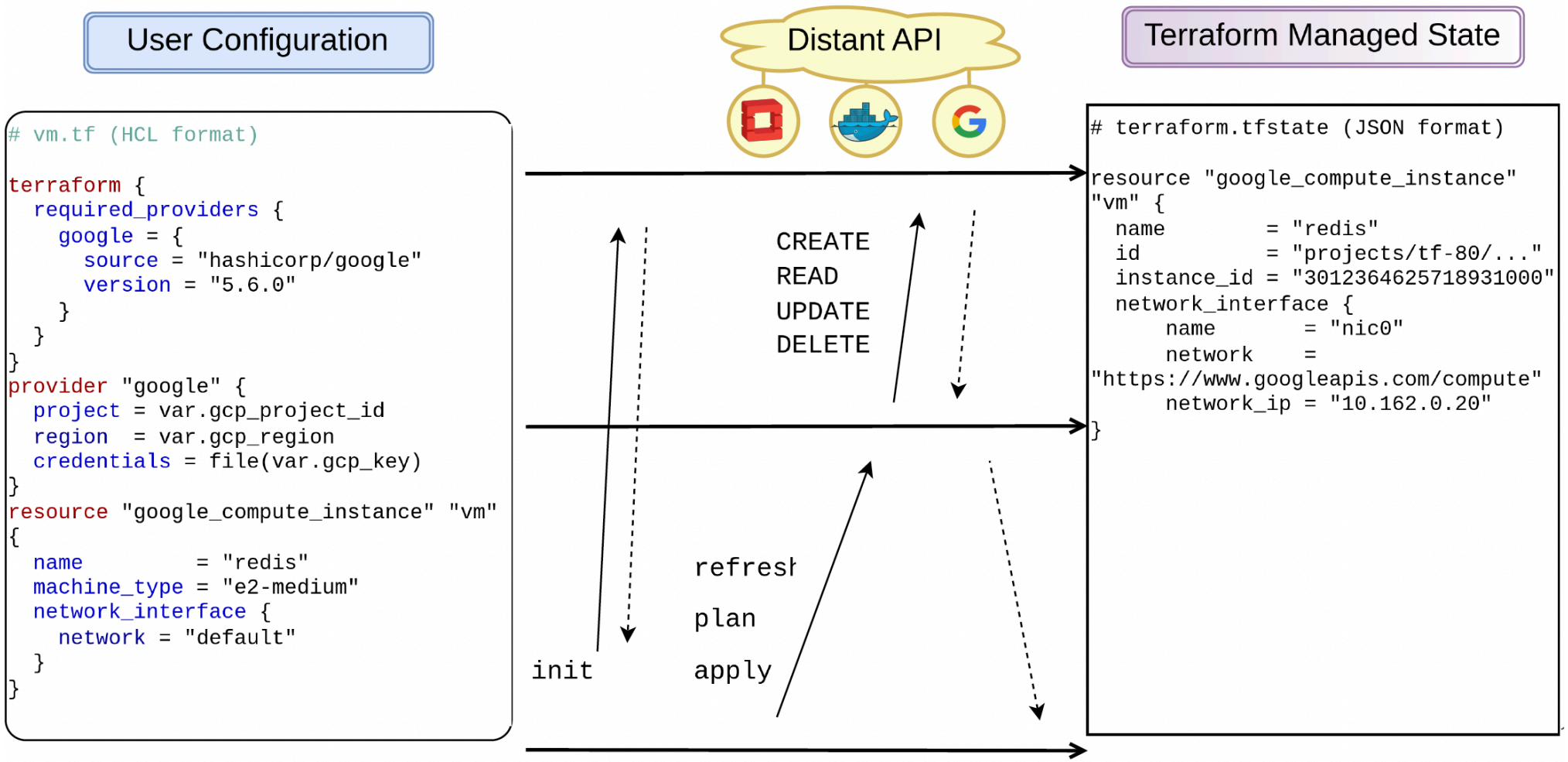
Machines virtuelles

Bases de données

# Local state vs Remote state

- Local state
  - Stocké dans terraform.tfstate
  - Simple, mais peu adapté au travail en équipe
- Remote state
  - Stocké dans un backend distant (Azure Storage, S3, etc.)
  - Partagé, sécurisé, verrouillable
- Indispensable en entreprise

# Schéma du flux de travail Terraform



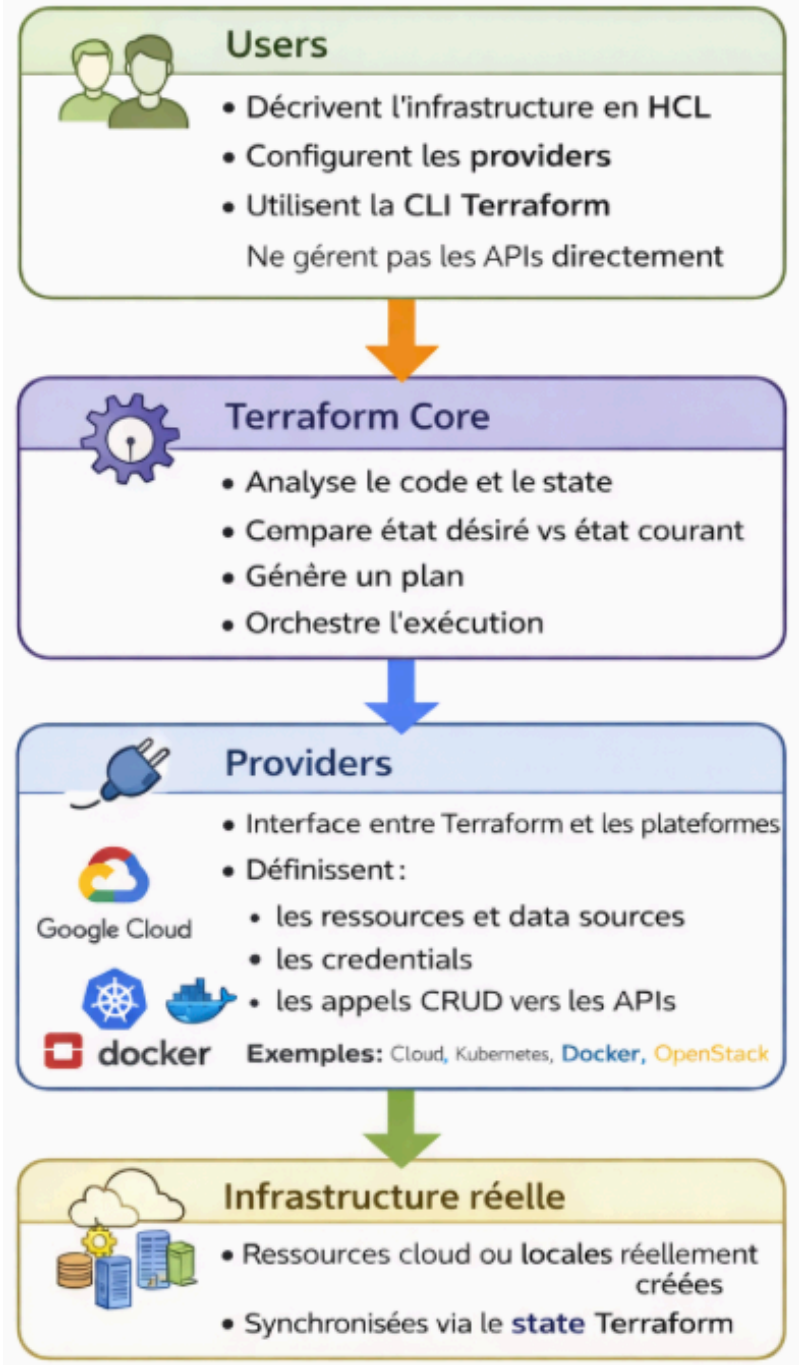
# Terraform – Concepts clés (vue d’ensemble)

## Providers

- Les providers sont des plugins Terraform permettant de communiquer avec des plateformes comme Azure, AWS ou le système local.
- https://developer.hashicorp.com/terraform/language/providers
- Types de Provider
  - Provider local : provider "local" {}
    - Permet de gérer des fichiers et ressources locales.
  - Provider cloud
    - provider "azurerm" {
      features {}
      }

# Terraform – Concepts clés (vue d'ensemble)

- Terraform sépare clairement l'intention (code), l'exécution (providers) et la réalité (state).



# Commandes pour différentes étapes

- *terraform init*
  - Initialise le répertoire de travail et télécharge les providers nécessaires.
- *terraform plan*
  - Génère un plan d'exécution qui détaille ce qui sera ajouté/supprimé/modifié en comparant les fichiers .tf avec le fichier de state.
  - Les plans peuvent être sauvegardés pour être appliqués ultérieurement.
- *terraform apply*
  - Génère un plan puis l'exécute pour appliquer les changements.
  - L'exécution peut échouer si le provider ne peut pas effectuer les appels API demandés par Terraform.
- *terraform destroy*
  - Appelle le provider pour supprimer toutes les ressources gérées par Terraform.

# Syntaxe du langage HCL (1) – Attributs

- Les attributs, aussi appelés arguments ou champs, sont définis à l'aide du signe = qui représente une affectation.
- La valeur d'un attribut peut être n'importe quelle expression : appels de fonctions, listes, objets, références à d'autres ressources, etc.
- Exemples :

```
name = "redis server"                    # Nom du serveur
credentials = file("./creds.json") # Lit le fichier creds.json et utilise son contenu comme informations d'authentification
labels = {                                # Définition des labels (métadonnées)
        app = "redis"                     # Indique que la ressource appartient à l'application Redis
}
image = docker_image.redis.name  # Référence le nom de l'image Docker appelée "redis"
```

- Chaque attribut décrit une propriété de la ressource ou du provider.

# Syntaxe du langage HCL (1) – Attributs

- Unicité des attributs et méta-attributs
- Un attribut ne peut être défini qu'une seule fois : les affectations multiples sont interdites.
- Chaque attribut est à affectation unique (single assignment).
- En plus des arguments classiques dans un bloc, il existe des méta-attributs ayant une signification particulière, comme :
  - count
  - for_each
  - depends_on

| Méta-attribut | Sert à |
|---|---|
| `count` | Créer N fois la même ressource |
| `for_each` | Créer une ressource par élément (liste/map) |
| `depends_on` | Forcer l'ordre de création |

  - Ces méta-attributs permettent de contrôler le nombre de ressources, les boucles et les dépendances entre ressources.

# Syntaxe du langage HCL (1) – Bocks

- Exemple :
  - resource "docker_image" "redis" { ... }

- Un bloc possède un mot-clé obligatoire,
  - ici resource, qui définit sa signification dans le contexte Terraform.
- Des chaînes de caractères peuvent être associées au bloc,
  - ici docker_image et redis.
- Ces identifiants permettent de référencer ce bloc ailleurs dans la configuration .tf.

# Syntaxe du langage HCL – Bocks

Qu’est-ce qu’un block ?

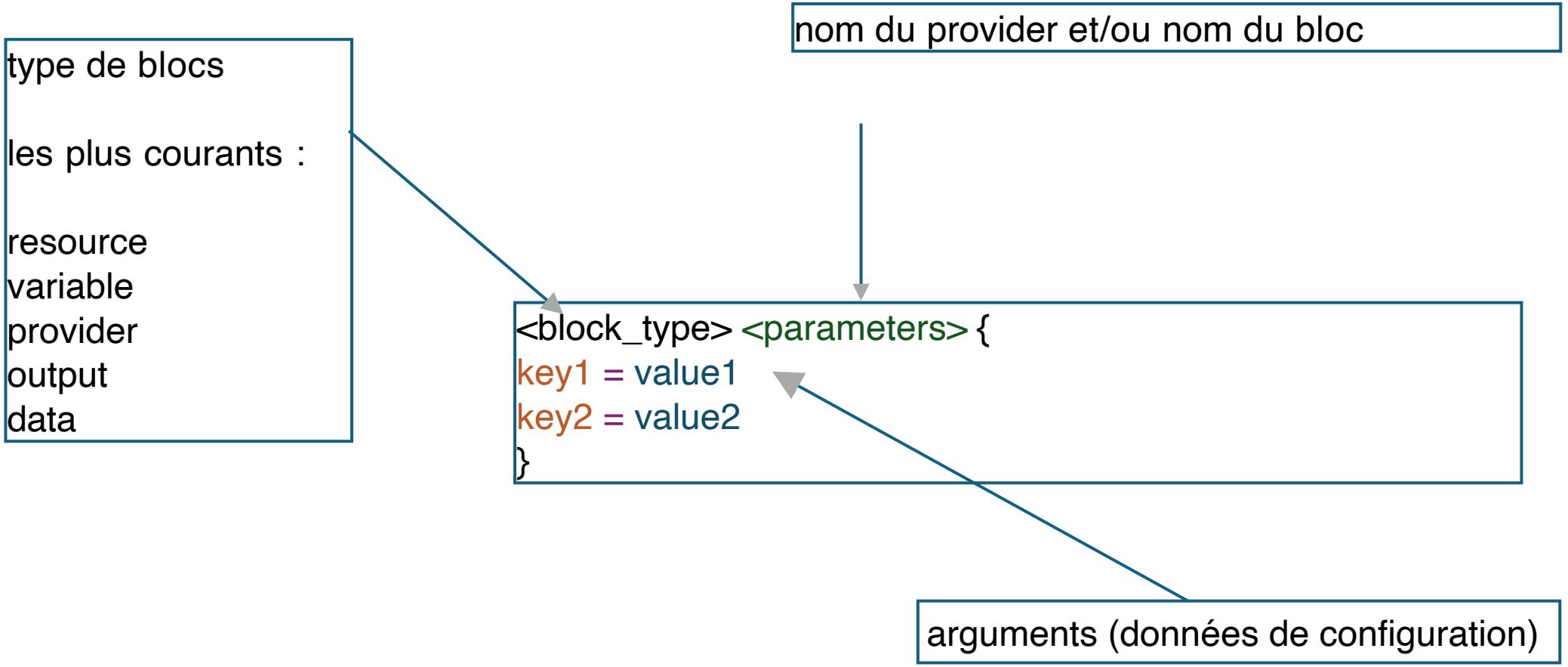
- Un block est la brique de base de Terraform.
- C’est la structure de base du langage HCL
- Tout dans Terraform est défini dans des blocks
- Un block décrit un objet Terraform
- Exemples de blocks :
  - resource
  - provider
  - variable
  - output
  - module

# Forme générale d’un block

```
mot_cle "label1" "label2" {
  attribut = valeur
}
```

- mot_cle → type du block (resource, provider, etc.)
- label1, label2 → identifiants
- { } → contenu du block
- À l’intérieur, on met des attributs et parfois d’autres blocks

# HCL - HashiCorp Configuration Language | basics



# Exemple simple

- Un block resource décrit un objet réel.

```
resource "local_file" "hello" {
  filename = "hello.txt"
  content  = "Bonjour"
}
```

Terraform voit ceci comme :
“Je veux un fichier local appelé *hello* de type *local_file*”

- resource → type de block
- local_file → type de ressource
- hello → nom logique
- Le contenu définit comment créer le fichier

# Référencer un block

- On peut réutiliser un block ailleurs.

```
local_file.hello.filename
```

Donne-moi l’attribut filename de la ressource hello de type local_file

# Blocks imbriqués

- Les blocks peuvent contenir d'autres blocks

```
resource "docker_container" "redis" {
  image = "redis"
  mounts {
    target = "/data"
    volume_options { no_copy = true }
  }
}
```

**mounts** est un block dans la resource

**volume**_options est un block dans mounts

- Les blocks imbriqués représentent une structure hiérarchique, comme en JSON
  - Terraform utilise des blocks pour représenter des **objets complexes**.

# Plusieurs blocks du même type

- Plusieurs blocks identiques forment une liste.

```
mounts { target = "/data" }
mounts { target = "/backup" }
```

- Terraform les transforme en liste :

```
mounts = [ {...}, {...} ]
```

# Accéder à un block imbriqué

- On accède via des index.

```
mounts[0].target
```

- Si un block contient un autre block :
  - Par exemple, pour accéder à une option de montage spécifique :

```
mounts[0].volume_options[0].no_copy
```

# Types de blocs de premier niveau (top-level) dans Terraform

- Terraform définit plusieurs types de blocs pouvant être déclarés au niveau racine.
  Les principaux sont : resource, data, provider et variable.

- La liste et la description de ces blocs sont disponibles dans la documentation des providers : https://registry.terraform.io/providers/
  - Le bloc **resource** décrit les ressources gérées par Terraform.
  - Le bloc **data** permet de lire des ressources existantes en *lecture seule*.
  - Le bloc **provider** configure les paramètres d'un provider.
  - Les blocs **variable** définissent les variables des modules.
- Il existe également d'autres blocs de premier niveau, comme : **module**, **check** et **import**.

- Terraform relie tout avec des références.
- Les variables, les ressources et les data sources forment un graphe de dépendances.

# Variables et références dans Terraform

- Terraform permet de stocker des valeurs dans trois endroits :

| Type | À quoi ça sert |
| --- | --- |
| **Input (`variable`)** | Valeurs fournies par l'utilisateur |
| **Local (`locals`)** | Valeurs calculées dans le code |
| **Output (`output`)** | Valeurs renvoyées après l'exécution |

# Variables et références dans Terraform - Définir des variables

```
variable "env" {
  default = "dev"
}

locals {
  app_name = "redis"
}

output "full_name" {
  value = "${local.app_name}-${var.env}"
}
```

# Variables et références dans Terraform
# Comment référencer quelque chose

```
resource "docker_image" "redis" {
  name = "redis:latest"
}

output "image_id" {
  value = docker_image.redis.image_id
}
```

- Terraform utilise des mots-clés pour savoir ce que tu veux lire :

| **Ce que tu veux** | **Comment l'écrire** |
|---|---|
| Une variable d'entrée | `var.env` |
| Une variable locale | `local.app_name` |
| Une ressource | `docker_image.redis.image_id` |
| Une source de données | `data.docker_image.redis.image_id` |

# Plan

- Introduction
- Concepts fondamentaux de Terraform
- Bonnes pratiques
- À vous de jouer !

# Bonnes pratiques

- Objectif : éviter le dépannage et les incidents
- Lire et comprendre attentivement chaque déclaration et chaque plan Terraform avant de l'appliquer.
- Versionner le code Terraform dans un système de contrôle de version (Git), en veillant à ne jamais y inclure de secrets.
- Intégrer Terraform dans des pipelines CI/CD pour automatiser les déploiements d'infrastructure.
- Stocker les fichiers de state Terraform dans des stockages distants avec des mécanismes de verrouillage afin d'éviter les conflits.

# Plan

- Introduction
- Concepts fondamentaux de Terraform
- Bonnes pratiques
- À vous de jouer !

# Feuille de route

1.- Installer son environnement de travail
- IDE + Terraform

2.- Découvrir et essayer en local

3.- Travailler sur Azure
- Installer Azure CLI
- Se connecter à Azure avec la console

Notions à explorer

*resource*
*variable, local*
*state  provider*
*data*
*init / plan / apply / destroy*

# Exercice 1 | fichier local

**Objectif**

Prendre en main Terraform en créant un fichier local sur votre machine avec du contenu spécifique et en définissant les droits sur ce fichier.

**Consignes**

- Depuis votre dossier de projet Terraform, créez un sous-dossier d'exercice appelé exercice_1.
- À l'intérieur de ce dossier, créez un fichier main.tf pour définir votre première ressource. Créez une ressource local_file qui génère un fichier texte appelé hello_world.txt avec le contenu : "Bienvenue dans Terraform !".
- Ajoutez également des arguments pour définir les permissions de lecture et écriture sur le fichier (ex: 0755).
- Exécutez les commandes nécessaires pour obtenir la création du fichier local.

**Ressources**

https://registry.terraform.io/providers/hashicorp/local/latest/docs/resources/file

# Exercice 2 | variables

**Objectif**

Découvrir l'utilisation des variables dans Terraform pour rendre la configuration flexible.
Vous allez créer un fichier dont le nom et le contenu seront définis par des variables.

**Consignes**

- Depuis votre dossier de projet Terraform, créez un sous-dossier d'exercice appelé exercice_2.
- À l'intérieur de ce dossier, créez un fichier main.tf et un fichier variables.tf.
- Dans le fichier variables.tf, définissez deux variables :
  - file_name : une chaîne de caractères qui représente le nom du fichier à créer.
  - file_content : une chaîne de caractères qui sera utilisée pour remplir le fichier.
- Dans le fichier main.tf utilisez ces variables pour créer un fichier avec Terraform.
- Exécutez les commandes nécessaires pour obtenir la création du fichier local.

**Ressources**

https://registry.terraform.io/providers/hashicorp/local/latest/docs/resources/file
https://developer.hashicorp.com/terraform/language/values/variables

# Exercice 3 | data source + http

**Objectif**

Utiliser le provider HTTP de Terraform pour télécharger un fichier à partir d'une URL et le stocker localement.

Le fichier se situe à l'adresse suivante : https://cdn.wsform.com/wp-content/uploads/2018/09/country_full.csv

**Consignes**

- Depuis votre dossier de projet Terraform, créez un sous-dossier d'exercice appelé exercice_3.
- À l'intérieur de ce dossier, créez un fichier main.tf pour définir le téléchargement et la sauvegarde du fichier.
- Créez une source de données data basée sur le fichier à télécharger.
- Créez une ressource local_file pour sauvegarder ce fichier localement sous le nom downloaded_file.txt.
- Exécutez les commandes nécessaires pour procéder au téléchargement du fichier.

**Ressources**

https://registry.terraform.io/providers/hashicorp/http/latest/docs

# Exercice 4 | multi providers

**Objectif**

Utiliser Terraform pour générer un ensemble de 10 mots de passe aléatoires et les stocker dans un fichier local. Cet exercice vous permet de travailler avec deux ressources Terraform où l'une dépend de l'autre : une ressource random pour générer des mots de passe et une ressource local_file pour les sauvegarder.

**Consignes**

- Depuis votre dossier de projet Terraform, créez un sous-dossier d'exercice appelé exercice_4.
- À l'intérieur de ce dossier, créez un fichier main.tf pour définir les ressources.
- Utilisez la ressource adéquate du provider random pour générer 10 mots de passe aléatoires.
- Votre code doit contenir 2 ressources.
- Exécutez les commandes nécessaires pour procéder au téléchargement du fichier.

**Ressources**

https://registry.terraform.io/providers/hashicorp/random/latest/docss
https://developer.hashicorp.com/terraform/language/meta-arguments/for_each