# Terraform

## Infrastructure As Code : support complet

Formation **Terraform** · orientée **expertise terrain** · cloud cible **Azure**

Édition **Retengr** · support **v1.1.1** · © Retengr 2026

# Au programme

1. **Introduction à l'IaC**
2. **Présentation de Terraform** & workflow
3. **Les providers**
4. **Syntaxe de base** (HCL)
5. **Variables & fonctions**
6. **State & backends**
7. **Les modules**
8. **Syntaxe avancée**
9. **Bonnes pratiques**
10. **Préparation certification 004** (annexe D + examen blanc)

💡 **Convention :** 💡 = retour terrain / expertise · 🎓 = point d'examen · ⚠️ = piège fréquent. **Annexes :** A — versions & cycle de vie · B — provider `azurerm` · C — state perdu & récupération · D — préparation certification 004.

🎓 **Certif 004 :** l'examen est évalué sur Terraform **1.12** et la terminologie **HCP Terraform** (ex-Terraform Cloud). Les encarts **Certif 004** signalent au fil des slides les points spécifiquement testés.

# Horaires & déroulé

## Journée type

- Matin : **9h00 – 12h00**
- Pause déjeuner : **12h00 – 14h00**
- Après-midi : **14h00 – 17h00**
- ☕ Pause café matin & après-midi

## Rythme

- **2 jours** · alternance théorie / **labs**
- **J1** — Fondamentaux : modules 01 → 06 · exercices 1, 2, 3
- **J2** — Structuration & pratiques : modules 07 → 09 + certif · exercices 4-5 (ouverture), 6-7, cas pratique

💡 **Remarque :** Un temps de théorie puis un **lab** pratique. Vos questions sont les bien venues à tout moment.

# Tour de table : faisons connaissance

## En deux minutes chacun

1. **Qui êtes-vous ?** Prénom, rôle, équipe
2. **D'où partez-vous ?** Terraform : jamais · un peu · en prod — et Azure ?
3. **Votre quotidien** : comment votre infra est-elle créée aujourd'hui ? (portail, scripts, tickets, autre outil IaC…)
4. **Vos attentes** : qu'est-ce qui fera que ces deux jours auront valu le coup ?

## Pourquoi on commence par là

- Ajuster le **rythme** et les exemples à vos niveaux
- Repérer les **cas concrets** que vous aimeriez automatiser — on s'en servira
- Savoir **à qui demander** : dans une salle, il y a toujours quelqu'un qui a déjà vécu votre problème

💡 **On note au tableau** les attentes de chacun : on y revient en fin de J2 pour vérifier qu'elles ont été couvertes.

# Le cloud de cette formation : Azure

Nos **labs** se déroulent sur **Microsoft Azure** (provider `azurerm`), depuis la **VM mise à disposition**. C'est un **choix pédagogique**, pas une contrainte de l'outil : le site des labs propose aussi chaque exercice en **AWS** et en **GCP**.

- Terraform est **agnostique** : le même workflow (`init → plan → apply`) pilote **Azure, AWS ou GCP**
- Nous **aurions pu prendre AWS** (`aws`) **ou GCP** (`google`) : seuls **les noms de ressources** changent, **pas la logique** Terraform — et l'exercice 5 vous fera utiliser **Azure et AWS dans le même projet** (la zone DNS est sur Route 53)
- Certains exemples de slides utilisent **AWS** (`aws_instance`, `aws_s3_bucket`…) car c'est le provider **historique de référence** dans la doc et la certification — ils restent **valables et transposables**

⚠️ **Le piège à éviter :** croire que Terraform rend l'infra **portable** entre clouds. Non : Terraform unifie le **workflow**, pas les ressources. Un `azurerm_linux_virtual_machine` n'a **pas** d'équivalent automatique en AWS — changer de cloud = **réécrire** le code, en gardant les mêmes réflexes.

💡 **Terrain :** ce qui se transfère d'un cloud à l'autre, c'est **votre méthode** — state, modules, précédence des variables, revue par `plan`. La syntaxe des ressources, elle, s'apprend par la doc du provider concerné.

MODULE 01

# Introduction à l'IaC

Concepts, finalité et bénéfices de l'Infrastructure as Code

# Qu'est-ce que l'Infrastructure as Code ?

L'**Infrastructure as Code (IaC)** : gérer l'infrastructure (réseaux, VMs, load balancers, DNS, IAM…) via un **modèle descriptif** versionné dans un système de gestion de sources.

- Partie intégrante de l'approche **DevOps**
- L'infrastructure est traitée comme **une application** : développée, revue, testée, versionnée (Git/GitLab)
- Le **code devient la documentation** de référence de l'infrastructure

🎓 **Certif 004 :** savoir **définir l'IaC** et citer ses bénéfices (cohérence, reproductibilité, versionnement) — domaine **« Understand IaC concepts »**.

💡 **Terrain :** la vraie valeur n'est pas « écrire du HCL » mais le **workflow de revue** : toute évolution d'infra passe par une Merge Request relue, exactement comme du code applicatif.

# Mutable vs Immutable

| | MUTABLE | IMMUTABLE |
|---|---|---|
| **Principe** | Modifiable en place — modèle classique | Non modifiable — recréation à chaque changement |
| **Contexte** | Legacy, infra existante | Cloud native |
| **Impact** | Modifications en place | L'architecture applicative doit le prévoir (stateless) |

# Drift & snowflake : les deux ennemis

- **Drift** : écart entre la configuration **déclarée** et l'état **réel** après une modification hors-outil. Concerne **directement** Terraform.
- **Snowflake** : serveurs mis à jour « au cas par cas », mal tracés. Concerne **indirectement** Terraform.

L'IaC **réduit** les deux et **diminue l'erreur humaine** : tout changement est poussé dans le VCS (qui, quand, quoi).

> 💡 **Terrain :** l'immutable réduit drastiquement le **snowflake**. Pattern courant : image dorée (Packer) + Terraform qui **remplace** les instances plutôt que de les patcher. Sur des ressources stateful (bases), on reste souvent mutable.

> ⚠️ **Piège :** modifier une ressource directement dans la console cloud crée du drift. Au prochain `plan`, Terraform voudra « corriger » — d'où l'importance d'interdire les changements manuels en prod.

# Le paysage des outils

| PROVISIONING D'INFRASTRUCTURE | PROVISIONING APPLICATIF (CONFIG) |
|---|---|
| Terraform / OpenTofu | Chef |
| CloudFormation (AWS) | Puppet |
| Blueprint (Google) | Ansible |
| Azure Resource Manager (ARM/Bicep) | |

🎓 **Certif 004 :** Terraform fait du **provisioning d'infrastructure** (créer les ressources), pas de la configuration applicative (installer/configurer des logiciels → Ansible). Distinction classiquement testée.

💡 **Terrain :** Terraform **+** Ansible se complètent : Terraform crée la VM et expose son IP en `output`, Ansible la configure. On évite les `provisioner` Terraform (fragiles, hors graphe d'état).

# Bénéfices & portabilité

- **Vitesse de livraison** : infrastructures déjà codées et réutilisables (modules)
- **Traçabilité** : chaque changement est un commit (qui, quand, quoi)
- **Partage** d'équipe facilité par Git
- **Portabilité** : code identique entre `dev`, `staging`, `prod` → réduit les écarts de configuration

> 💡 **Terrain :** « même code, valeurs différentes ». On ne duplique pas le code par environnement : on garde une base commune (modules) paramétrée par des `tfvars` distincts. Voir module 05.

## 🎓 Quiz : 5 questions

1. L'IaC stocke la configuration de l'infrastructure dans… ? **(VCS / RAM / wiki)**
2. Terraform fait du provisioning **d'infrastructure** ou **applicatif** ?
3. Qu'est-ce que le **drift** ?
4. **Vrai ou faux** : l'approche immutable consiste à patcher les serveurs en place.
5. Quel outil privilégier pour la **configuration applicative** ?

## ✅ Corrections

1. Un **système de gestion de versions (Git)** — l'infra est versionnée.
2. **Provisioning d'infrastructure** : créer/gérer les ressources.
3. L'**écart entre la config déclarée et l'état réel** de l'infrastructure.
4. **Faux** — l'immutable **recrée** la ressource ; c'est le **mutable** qui patche en place.
5. **Ansible** (provisioning applicatif, ≠ cœur de métier de Terraform).

MODULE 02

# Présentation de Terraform

Le binaire, le langage, les providers, le state — et le workflow

# Terraform en bref

- Créé par **HashiCorp en 2014**, écrit en **Go**
- Outil **open source** pour décrire et provisionner une infrastructure
- Langage **HCL** (HashiCorp Configuration Language)
- Approche **déclarative** : on décrit l'**état souhaité**, Terraform calcule le chemin pour y arriver

> 🎓 **Certif 004 :** quatre composants à connaître — un **binaire**, un **langage (HCL)**, des **providers**, un **fichier d'état** ( `terraform.tfstate` ).

# Déclaratif vs impératif

| | IMPÉRATIF | DÉCLARATIF |
|---|---|---|
| **Principe** | On décrit **comment** faire | On décrit **ce que** l'on veut |
| **Exemple** | « Crée une VM, ouvre le port 22… » | « Je veux une VM avec le port 22 ouvert » |
| **Outils** | Scripts shell, Ansible | Terraform, CloudFormation |
| **Idempotence** | À gérer soi-même | Garantie par l'outil |
| **Changements** | Réécrire le script | Modifier la description, Terraform calcule le **delta** |

💡 **Terrain :** l'idempotence « gratuite » est le vrai gain : relancer un `apply` sans changement de code ne fait **rien**. C'est ce qui rend le `plan` fiable comme outil de revue.

# Terraform & OpenTofu

- En **2023**, HashiCorp passe Terraform sous licence **BSL** (Business Source License)
- La communauté crée **OpenTofu** : fork open source, maintenu par la **Linux Foundation**
- Les deux sont **compatibles au niveau HCL**
- Le choix dépend du contexte : licence, support, écosystème, gouvernance

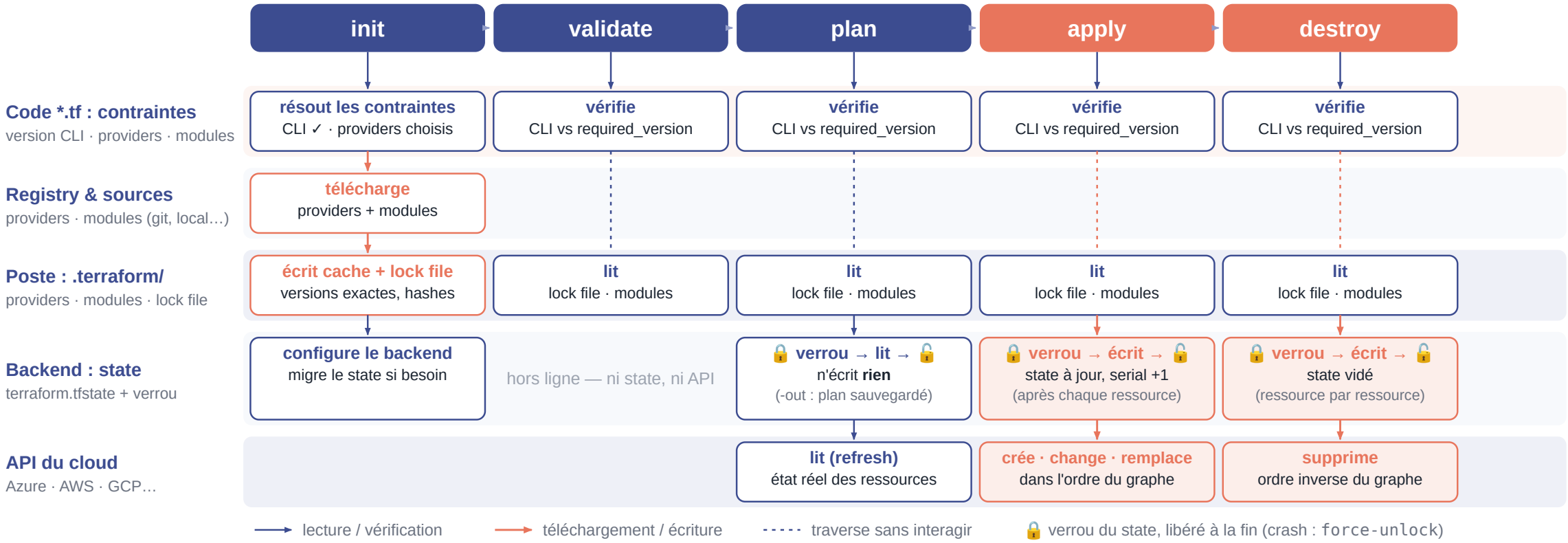
> 💡 **Terrain :** OpenTofu se substitue au binaire (`tofu` au lieu de `terraform`) sans réécriture. Quelques fonctions divergent (chiffrement de state natif côté OpenTofu). En entreprise, le choix est souvent dicté par la conformité licence BSL.

# Le workflow Terraform

```
terraform init      # Télécharge providers + modules, initialise le backend
terraform plan      # Simule les changements (dry run) — NE MODIFIE RIEN
terraform apply     # Applique le delta, met à jour le state
terraform destroy   # Détruit les ressources gérées
```

🎓 **Certif 004 — cœur de l'examen :** `init` est le **seul** à télécharger les providers ; `plan` est en **lecture seule** ; `apply` écrit le state ; le workflow principal est `write → plan → apply`.

# Le workflow : qui touche à quoi ?




💡 **Deux verrous à ne pas confondre :** `required_version` ne fige rien, il **refuse** un binaire hors contrainte ; `.terraform.lock.hcl` **fige** les versions (et hashes) des providers résolues par `init` → à **committer**. Les modules n'ont pas de lock file : épinglez `version` dans le bloc `module`. Module modifié ⇒ `init` (ou `get -update`).

# Démonstration : init & plan

```
$ terraform init
  Initializing provider plugins...
  - Installing hashicorp/aws v5.82.0...
  Terraform has been successfully initialized!

$ terraform plan
  # aws_instance.web will be created
  Plan: 1 to add, 0 to change, 0 to destroy.
```

💡 **Terrain :** en CI, on fige le plan : `terraform plan -out=tfplan` puis `terraform apply tfplan`. Le `apply` applique **exactement** ce qui a été relu — pas de recalcul entre la revue et l'exécution.

# Démonstration : apply & destroy

```
$ terraform apply
  aws_instance.web: Creating...
  aws_instance.web: Creation complete after 32s [id=i-0abc123]
  Apply complete! Resources: 1 added, 0 changed, 0 destroyed.

$ terraform destroy
  Destroy complete! Resources: 1 destroyed.
```

- Après `apply`, l'état est enregistré dans `terraform.tfstate`
- Chaque `plan` compare **code ↔ state ↔ réalité** → c'est le mécanisme de **détection du drift**

# Ce que Terraform gère / n'est pas

## Gère (des milliers de ressources)

- Réseau : VPC, subnets, SG, LB
- DNS : Route53, Cloud DNS, Cloudflare
- Bases : RDS, Cloud SQL
- Conteneurs : ECS, EKS, GKE, AKS
- IAM : users, rôles, policies

## N'est PAS

- Un outil pour installer des logiciels → **Ansible**
- Un langage universel multi-cloud : le HCL est **spécifique à chaque provider**
- Changer de cloud = réécrire le code

⚠️ **Idée reçue :** « Terraform rend mon infra portable entre clouds ». Faux : il unifie le **workflow**, pas les ressources. Un `aws_instance` n'a pas d'équivalent automatique en Azure.

# Installation & versioning

```
brew install terraform          # macOS
apt install terraform           # Debian/Ubuntu
choco install terraform         # Windows
terraform version               # Vérification
```

💡 **Terrain :** verrouillez la version de Terraform par projet avec **tfenv** / **asdf** et un bloc `required_version`. En équipe, des versions différentes du binaire peuvent produire des states incompatibles (le format de state n'est rétrocompatible que vers le haut).

```
terraform {
  required_version = ">= 1.5.0, < 2.0.0"
}
```

# Le cycle de vie des releases

## Versionnement (SemVer)

`MAJOR.MINOR.PATCH` — ex. `1.12.2`

- **PATCH** : corrections de bugs & sécurité
- **MINOR** : nouvelles fonctionnalités, **rétrocompatibles**
- **MAJOR** : changements de rupture (on est en **1.x** depuis 2021)
- **Cadence** : ~1 mineure / trimestre, patchs au fil de l'eau

## Compatibilité & support

- Depuis **1.0** (juin 2021) : on peut **monter** de n'importe quelle 1.x vers une 1.x plus récente
- **Code** (`.tf`) : un code écrit pour une ancienne 1.x tourne sur une plus récente ; l'inverse échoue seulement s'il utilise une **nouveauté** (ex. `import {}` → 1.5+)
- **State** : on peut monter, **pas redescendre** — un binaire récent peut écrire un state qu'une version plus ancienne ne sait plus relire
- HashiCorp maintient surtout la **dernière mineure**
- Licence **BSL** depuis **1.6** → fork **OpenTofu**

🎓 **Certif 004 :** évaluée sur Terraform **1.12**. Le bloc `required_version` contraint la version utilisable ; combiné à `required_providers`, il rend le projet reproductible.

# Jalons notables des versions

| VERSION | ANNÉE | APPORT CLÉ |
|---|---|---|
| **0.12** | 2019 | HCL2 — refonte du langage |
| **0.13** | 2020 | `for_each` / `count` sur les **modules** |
| **0.14** | 2020 | `.terraform.lock.hcl`, valeurs `sensitive` |
| **1.0** | 2021 | Stabilité & **promesse de compatibilité** |
| **1.5** | 2023 | Blocs `import` & `check` |
| **1.6** | 2023 | `terraform test` natif · licence **BSL** |
| **1.10** | 2024 | Ressources / valeurs `ephemeral` |
| **1.12** | 2025 | **Version testée par la certif 004** |
| **1.17** | 2026 | Branche courante (octobre 2026) — deux branches maintenues à la fois |

💡 **Terrain :** restez sur une mineure **récente mais éprouvée** (N-1). Montez régulièrement pour rester dans le périmètre supporté et bénéficier des correctifs de sécurité — jamais en plein cycle de livraison critique. (liste non exhaustive)

## 🎓 Quiz : 5 questions

1. Quelle commande **simule** les changements sans rien modifier ?
2. Terraform est-il **déclaratif** ou **impératif** ?
3. Comment s'appelle le **fork open source** de Terraform ?
4. Quel fichier représente l'**infrastructure déployée** ?
5. **Vrai ou faux** : Terraform rend votre code portable entre clouds.

## ✅ Corrections

1. `terraform plan` — en lecture seule (dry run).
2. **Déclaratif** — on décrit l'état souhaité, Terraform calcule le delta.
3. **OpenTofu** — maintenu par la Linux Foundation.
4. `terraform.tfstate` — le fichier d'état.
5. **Faux** — il unifie le **workflow**, pas les ressources ; le HCL est spécifique à chaque provider.

MODULE 03

# Les providers

Le pont entre Terraform et vos plateformes

# Qu'est-ce qu'un provider ?

Un **plugin** qui contient :

- un **annuaire** des `resources` et `data sources` utilisables ;
- la **couche technique (SDK)** qui parle à l'API de la plateforme.

Il gère le **cycle de vie CRUD** complet des ressources. Il est **versionné** et téléchargé au `terraform init`.

> Sans provider, Terraform ne sait **rien** faire.

> 🎓 **Certif 004 :** les providers sont installés par `init`, stockés dans `.terraform/`, et leurs versions exactes verrouillées dans `.terraform.lock.hcl`.

# L'écosystème & le Registry

| CATÉGORIE | EXEMPLES | MAINTENU PAR |
|---|---|---|
| Cloud public | AWS, Azure, GCP | HashiCorp / éditeur |
| Virtualisation | vSphere, Proxmox, Nutanix | Communauté / éditeur |
| Conteneurs | Kubernetes, Docker, Helm | HashiCorp |
| DNS / CDN | Cloudflare, Route53, OVH | Éditeur / communauté |
| SaaS | GitHub, GitLab, Datadog | Éditeur |

**Trois niveaux de confiance** sur `registry.terraform.io` : **Official** (HashiCorp) · **Partner** (éditeur vérifié) · **Community**.

> 🎓 **Certif 004 :** connaître les 3 tiers (Official / Partner / Community) et le rôle du Registry comme répertoire central providers **et** modules.

> https://registry.terraform.io/

# Déclarer un provider & contraintes de version

```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"   # namespace/nom
      version = "~> 5.0"          # contrainte
    }
  }
}
```

| SYNTAXE | SIGNIFICATION |
|---|---|
| `= 5.0.0` | Exactement cette version |
| `>= 5.0` | Cette version ou supérieure |
| `~> 5.0` | `>= 5.0.0` et `< 6.0.0` |
| `~> 5.0.1` | `>= 5.0.1` et `< 5.1.0` |

🎓 **Certif 004 :** l'opérateur pessimiste `~>` est **très testé**. `~> 5.0` autorise les mineures (5.x), `~> 5.0.1` seulement les patchs (5.0.x).

# Le lock file & init

```
$ terraform init
  - Finding hashicorp/aws versions matching "~> 5.0"...
  - Installing hashicorp/aws v5.82.0...
  - Installed hashicorp/aws v5.82.0 (signed by HashiCorp)
```

> 💡 **Terrain — committez `.terraform.lock.hcl` :** il garantit la même version de provider pour toute l'équipe et la CI.

# Anatomie de `.terraform.lock.hcl`

```
# This file is maintained automatically by "terraform init".
# Manual edits may be lost in future updates.

provider "registry.terraform.io/hashicorp/azurerm" {
  version     = "4.81.0"   # version EXACTE installée
  constraints = "~> 4.0"   # ce que le code demandait
  hashes = [
    "h1:3kQ…=",            # hash du zip, par plateforme
    "zh:0a4…",             # hashes du contenu
    "zh:1f9…",
  ]
}
```

## Ce qu'il fait

- Fige la **version exacte** de chaque provider résolue par `init` — la contrainte n'est plus réévaluée
- Vérifie les **hashes** : un provider altéré ou différent est **refusé** (protection **supply chain**)
- Ne couvre **que les providers** — ni la CLI, ni les modules

🎓 **Certif 004 :** créé par `init` (Terraform ≥ 0.14), verrouille les **providers** — pas les modules ; `init -upgrade` est la seule façon légitime de le faire évoluer.

# Le lock file : les 4 réflexes

1. **Committer** `.terraform.lock.hcl` — c'est l'inverse de `.terraform/` (ignoré) : l'équipe et la CI installent **exactement** les mêmes providers
2. **Ne jamais l'éditer à la main** : pour monter de version, on change la contrainte puis `terraform init -upgrade`
3. **Équipe multi-OS ou CI Linux** : les hashes `h1` sont **par plateforme** → sans ceux de la CI, son `init` échoue

```
terraform providers lock -platform=linux_amd64 -platform=darwin_arm64 -platform=windows_amd64
```

4. **Relire ses diffs** : un changement de ce fichier dans une PR **est** une montée de version de provider — on la traite comme du code (CHANGELOG, `plan` à blanc)

> 💡 **Terrain :** le symptôme classique du réflexe 3 — « ça marche sur mon Mac, la CI dit *provider hash mismatch* ». Ce n'est pas la CI qui est cassée : il manque le hash Linux dans le lock file.

# Configurer un provider : authentification

```
provider "aws" {                        # via variables d'env AWS_*
  region = "eu-west-3"
}
provider "google" {
  project     = "mon-projet-gcp"
  credentials = file("sa-key.json")    # service account
}
provider "azurerm" {
  features {}
  subscription_id = var.subscription_id
}
```

💡 **Notre cloud :** en labs nous utilisons **Azure** (bloc `azurerm` ci-dessus, `features {}` obligatoire). Les blocs `aws` et `google` montrent que **le même Terraform** pilote les trois : on aurait pu bâtir la formation sur AWS ou GCP à l'identique.

⚠️ **Sécurité :** jamais de credentials en dur dans le code. Privilégier variables d'environnement, profils, OIDC en CI, ou un coffre (Vault).

# Alias : plusieurs configs d'un même provider

```
provider "aws" { region = "eu-west-3" }          # par défaut

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

resource "aws_s3_bucket" "backup" {
  provider = aws.us                               # cible l'alias
  bucket   = "mon-backup-us"
}
```

💡 **Terrain :** indispensable en multi-régions / multi-comptes. Certaines ressources globales (ex. certificats ACM pour CloudFront) **exigent** `us-east-1` — l'alias est alors la seule solution propre.

# Resources vs Data sources

| | RESOURCE | DATA SOURCE |
|---|---|---|
| Rôle | Créer / modifier / supprimer | Interroger l'existant |
| Mode | Lecture / écriture | **Lecture seule** |
| Syntaxe | `resource "aws_instance" "web" {}` | `data "aws_ami" "ubuntu" {}` |
| Cycle de vie | Géré par `plan` / `apply` | Évalué à chaque refresh |

🎓 **Certif 004 :** une `data source` ne crée rien — elle **lit**. Sa documentation et la distinction avec `resource` sont au programme (domaine « configuration »).

# Déboguer un provider

| VARIABLE | CE QU'ELLE FAIT |
|---|---|
| `TF_LOG=DEBUG` | tout : le cœur de Terraform **et** les providers — très verbeux |
| `TF_LOG_PROVIDER=DEBUG` | **uniquement** le provider : ses requêtes à l'API du cloud et les réponses |
| `TF_LOG_PATH=./terraform.log` | écrit les logs dans un fichier au lieu du terminal |

```
TF_LOG_PROVIDER=DEBUG TF_LOG_PATH=./terraform.log terraform plan
grep -iE 'status|error' terraform.log      # le code HTTP et le message de l'API
```

💡 **Lire le résultat :** un **401** ou un **403** = problème d'**identité ou de droits**, pas de code Terraform. Côté Azure : `AuthorizationFailed` (rôle manquant, mauvais RG) ou secret refusé (faux, expiré). ⚠️ Les logs contiennent des valeurs sensibles : ne pas les committer, supprimer `terraform.log` après usage.

## 🎓 Quiz : 5 questions

1. Quelle commande **installe les providers** ?
2. La contrainte `~> 5.0` autorise-t-elle la version **6.0** ?
3. Citez les **3 niveaux de confiance** du Terraform Registry.
4. À quoi sert le fichier `.terraform.lock.hcl` ?
5. Comment gérer **deux régions** d'un même provider dans un projet ?

## ✅ Corrections

1. `terraform init` (seul à télécharger les providers).
2. **Non** — `~> 5.0` signifie `>= 5.0.0` et `< 6.0.0`.
3. **Official** / **Partner** / **Community**.
4. **Verrouiller les versions exactes** des providers — à **committer**.
5. Définir un `alias` de provider et cibler `provider = aws.us`.

🧪 ON PASSE À LA PRATIQUE

# Exercice 1 : Authentification

Votre VM, votre identité Azure

- **Objectif** : renseigner la page **Configuration** du site des labs (votre prénom, injecté dans tous les exercices), vérifier l'environnement Azure de votre VM (`env | grep ARM` : un **Service Principal**) et créer une ressource avec `az` — sans Terraform
- **Vous avez fini quand** : `az` répond avec votre Service Principal et votre ressource de test a été créée puis supprimée
- **Il vous faut** : la VM mise à disposition — rien à installer

📚 Site des labs Retengr → **Configuration**, puis **Authentification** › Azure

MODULE 04

# Syntaxe de base : HCL

Blocs, références, variables, outputs, interpolation

# Le langage HCL & ses 4 blocs

HCL organise la configuration en **blocs** composés de couples clé/valeur. Types : `string`, `number`, `bool`, `list`, `map`, `object`.

| BLOC | RÔLE |
|---|---|
| `resource` | Ressource à **créer** |
| `variable` | **Entrée** de la configuration |
| `output` | **Sortie** à afficher / exporter |
| `data` | Interroge l'existant (lecture seule) |

> 🎓 **Certif 004 :** savoir **lire un bloc** et identifier son type est central. Connaître aussi `terraform {}`, `provider {}`, `module {}`, `locals {}`.

# Déclarer & référencer une ressource

```
resource "aws_instance" "web" {        # <type> <nom local>
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t3.micro"
  tags = { Name = "mon-serveur-web" }
}

# Référence : <TYPE>.<NOM>.<ATTRIBUT>
aws_instance.web.public_ip
aws_instance.web.id
```

💡 **Terrain — graphe implicite :** en référençant `aws_instance.web.id` ailleurs, vous créez une **dépendance automatique**. Terraform ordonne la création/destruction tout seul. N'utilisez `depends_on` que pour les dépendances **invisibles** (IAM, ordre d'API).

# Variables : déclaration, usage, types

```
variable "region" {
  description = "Région AWS"
  type        = string
  default     = "eu-west-3"
}

provider "aws" { region = var.region }     # usage : var.<NOM>
```

| TYPE | EXEMPLE DE VALEUR | USAGE VAR.… |
|---|---|---|
| `string` / `number` / `bool` | `"eu-west-3"` / `3` / `true` | `var.region` · `var.nb_vm` · `var.debug` |
| `list(string)` | `zones = ["1","2","3"]` | `var.zones[0]` → `"1"` · `length(var.zones)` → 3 |
| `map(string)` | `tags = { env = "prod" }` | `var.tags["env"]` ou `var.tags.env` → `"prod"` |
| `object({ name=string, port=number })` | `srv = { name = "web", port = 80 }` | `var.srv.name` → `"web"` · `var.srv.port` → `80` |

🎓 **Certif 004 :** toujours typer les variables. Le `description` est une bonne pratique attendue.

# Outputs & interpolation

```
output "instance_ip" {
  description = "IP publique du serveur web"
  value       = aws_instance.web.public_ip
}

resource "aws_instance" "web" {
  tags = { Name = "server-${var.environment}" }   # interpolation ${...}
}
```

- Les `output` **affichent** après `apply` **et** exposent des valeurs **entre modules**
- Hors chaîne, la référence directe suffit (`var.region`) ; `${...}` sert à **insérer** dans une chaîne

> 💡 **Terrain :** marquez les outputs propageant un secret avec `sensitive = true` ; sinon ils s'affichent en clair dans la sortie `apply` et dans la CI.

# Structure d'un projet

```
mon-projet/
├── main.tf          # Ressources principales
├── variables.tf     # Déclaration des variables
├── outputs.tf       # Valeurs de sortie
├── terraform.tfvars # Valeurs des variables
└── providers.tf     # Providers + backend
```

🎓 **Certif 004 :** ces noms sont une **convention**, pas une obligation — Terraform charge **tous** les `*.tf` du répertoire, sans ordre garanti (le graphe de dépendances détermine l'ordre réel).

## 🎓 Quiz : 5 questions

1. Citez les **4 blocs fondamentaux** de HCL.
2. Quelle est la syntaxe pour **référencer un attribut** de ressource ?
3. Un bloc `data` peut-il créer une ressource ?
4. Comment **insérer une variable** dans une chaîne de caractères ?
5. **Vrai ou faux** : l'ordre des fichiers `.tf` détermine l'ordre d'exécution.

## ✅ Corrections

1. `resource`, `variable`, `output`, `data`.
2. `<TYPE>.<NOM>.<ATTRIBUT>` — ex. `aws_instance.web.id`.
3. **Non** — `data` interroge l'existant en **lecture seule**.
4. Avec l'**interpolation** `${...}`.
5. **Faux** — c'est le **graphe de dépendances** (via les références) qui ordonne.

MODULE 05

# Variables & Fonctions

Valorisation, précédence, locals, fonctions built-in

# Pourquoi (et jusqu'où) variabiliser

- Code **portable** (même code dev/staging/prod)
- **Séparer** configuration et code
- **Partager** sans exposer les secrets

> **Règle :** variabilisez ce qui **change** entre environnements, pas ce qui est constant.

> ⚠️ **Anti-pattern :** tout paramétrer rend le code illisible et le `plan` imprévisible. Une variable doit représenter une **vraie décision** d'environnement, pas un réflexe.

# Stratégies de valorisation : la précédence

Du plus **faible** au plus **fort** :

1. `default` dans la déclaration
2. Variable d'environnement `TF_VAR_key`
3. `terraform.tfvars` (auto-chargé)
4. `*.auto.tfvars` (auto-chargé, ordre alphabétique)
5. `-var-file="fichier.tfvars"` / `-var="key=value"` (ligne de commande — en cas de conflit, le **dernier écrit** l'emporte)

> 🎓 **Certif 004 — à connaître par cœur :** en cas de conflit, la valeur chargée en **dernier** l'emporte. Attention : `TF_VAR_*` est parmi les plus **faibles** (juste au-dessus de `default`), tandis que `-var` / `-var-file` sont les plus **forts** — c'est un piège classique de QCM.

# tfvars & surcharge

```
terraform apply -var-file="production.tfvars"   # par environnement
terraform apply -var="region=us-east-1"         # ponctuel
export TF_VAR_region="us-east-1"; terraform apply  # CI/CD
```

```
# terraform.tfvars
region        = "eu-west-3"
instance_type = "t3.micro"
environment   = "dev"
```

💡 **Terrain :** un fichier `tfvars` par environnement, **non secret** versionné, secrets injectés via `TF_VAR_*` en CI. Les `*.auto.tfvars` sont pratiques mais « magiques » : en équipe, préférez l'explicite `-var-file`.

# list, map & validation

```
variable "availability_zones" {
  type    = list(string)
  default = ["eu-west-3a", "eu-west-3b"]
}

variable "environment" {
  type = string
  validation {
    condition     = contains(["dev","staging","prod"], var.environment)
    error_message = "L'environnement doit être dev, staging ou prod."
  }
}
```

💡 **Terrain :** les blocs `validation` font échouer le `plan` **tôt**, avec un message clair, plutôt qu'une erreur d'API obscure 5 minutes plus tard. Avec `object({...})`, `optional(type, defaut)` rend des attributs facultatifs.

# Variables sensibles & scope

```
variable "db_password" {
  type      = string
  sensitive = true        # jamais affichée dans les logs
}
```

```
var.region                  # variable d'entrée
aws_instance.web.public_ip  # attribut de ressource
data.aws_ami.ubuntu.id      # attribut de data source
module.network.vpc_id       # output d'un module enfant
local.common_tags           # valeur locale
```

🎓 **Certif 004 :** `sensitive = true` masque la valeur en sortie — mais elle reste **en clair dans le state**. D'où le besoin d'un backend chiffré (module 06).

# Locals & fonctions built-in

```
locals {
  project_name = "demo-${var.environment}"
  common_tags  = {
    Project   = local.project_name
    ManagedBy = "terraform"
  }
}
resource "aws_instance" "web" { tags = local.common_tags }
```

```
$ terraform console
> upper("hello")              "HELLO"
> format("server-%02d", 3)    "server-03"
```

💡 **Terrain :** `locals` = valeurs **calculées internes** (DRY), `variable` = **entrées** de l'utilisateur. Ne confondez pas : un local ne se surcharge pas en ligne de commande.

# Catégories de fonctions & templatefile

| CATÉGORIE | EXEMPLES |
|---|---|
| String | `upper`, `format`, `split`, `join` |
| Numérique | `max`, `min`, `ceil`, `abs` |
| Collection | `length`, `merge`, `flatten`, `keys` |
| Réseau | `cidrsubnet`, `cidrnetmask` |
| Fichier | `file`, `templatefile`, `fileexists` |

```
user_data = templatefile("${path.module}/init.sh", {
  hostname    = "web-server"
  environment = var.environment
})
```

💡 **Terrain :** `terraform console` est votre bac à sable pour tester une expression avant de l'écrire. `cidrsubnet()` évite de calculer les masques à la main pour découper un VPC.

## 🎓 Quiz : 5 questions

1. En cas de conflit, qui l'emporte : `-var` ou `terraform.tfvars` ?
2. À quoi sert `sensitive = true` sur une variable ?
3. Une valeur sensible est-elle **chiffrée dans le state** ?
4. Quelle est la différence entre une `variable` et un `local` ?
5. Quelle commande permet de **tester une fonction** ?

## ✅ Corrections

1. `-var` — chargé en **dernier** (précédence la plus forte), l'emporte sur `terraform.tfvars`.
2. **Masquer la valeur** dans les logs et les sorties Terraform.
3. **Non** — elle reste **en clair** dans le state → backend chiffré obligatoire.
4. `variable` = entrée surchargeable ; `local` = valeur calculée interne, non surchargeable.
5. `terraform console`.

🧪 ON PASSE À LA PRATIQUE

# Exercice 2 : Premiers pas

Votre première VM, un seul `apply`

- **Objectif** : vérifier `terraform -version` (déjà installé sur la VM), créer le module root `~/www` (`main.tf` / `variables.tf` / `outputs.tf`), déclarer `azurerm`, puis déployer la VM en passant par des **variables**
- **Vous avez fini quand** : `terraform apply` est vert, `terraform state list` montre vos ressources et `terraform output` donne l'IP publique
- **Il vous faut** : l'exercice 1 réussi ; **la VM ne répond pas encore en SSH — c'est normal, c'est l'exercice 4**

📚 Site des labs Retengr → **Premiers pas** › Azure

MODULE 06

# State & Backends

Le cœur de Terraform — protégez-le

# Le fichier d'état

- Terraform stocke l'état de l'infra dans un fichier **JSON** (`terraform.tfstate`)
- C'est la **source de vérité** pour Terraform
- À chaque `plan`, il compare : **le code** (ce que vous voulez) ↔ **le state** (ce qu'il connaît) ↔ **la réalité** (ce qui existe)

> 🎓 **Certif 004 :** sans state, Terraform ne sait pas quelles ressources il gère. Le state stocke aussi les **attributs calculés** (IDs, IPs, ARN) et gère les dépendances. Domaine **« Implement and maintain state »**.

# Anatomie de `terraform.tfstate`

```
{
  "version": 4,
  "terraform_version": "1.16.7",
  "serial": 17,
  "lineage": "9f1c2b3a-…-7e21",
  "resources": [
    {
      "mode": "managed",
      "type": "azurerm_linux_virtual_machine",
      "name": "vm",
      "provider": "provider[\"registry.terraform.io/hashicorp/azurerm\"]",
      "instances": [ {
        "schema_version": 0,
        "attributes": { "id": "/subscriptions/…/web1", "size": "Standard_B1s" },
        "dependencies": [ "azurerm_network_interface.main" ]
      } ]
    }
  ]
}
```

- **`version`** — format du state (4 depuis Terraform 0.12) ; ne bouge presque jamais
- **`terraform_version`** — le binaire qui l'a **écrit** ; un binaire plus ancien **refuse** de le relire (migration vers le haut seulement)
- **`serial`** — compteur **+1 à chaque écriture** : c'est le **verrou optimiste** du backend, qui refuse un state au serial plus petit que le sien
- **`lineage`** — identifiant de **naissance** (UUID), jamais modifié : deux states de lineages différents ne se mélangent pas (**lineage mismatch**)
- **`resources[]`** — `mode` (`managed` / `data`), adresse `type.name`, puis par instance les **attributs réels** renvoyés par l'API (**secrets en clair** compris) et ses `dependencies` (ordre de destruction)

⚠️ **On ne l'édite jamais à la main :** un `serial` ou un `lineage` faux, et le backend refuse le push (ou pire, l'accepte). Toutes les retouches passent par `terraform state …`, `moved {}`, `import` — qui gèrent `serial` pour vous.

# Pourquoi le state est critique

```
{
  "version": 4,
  "resources": [{
    "type": "aws_instance", "name": "web",
    "instances": [{ "attributes": {
      "id": "i-0abc123", "public_ip": "54.37.12.45"
    }}]
  }]
}
```

⚠️ **Sécurité :** le state contient des données **sensibles** en clair (mots de passe, clés, IPs). Ne le commitez **jamais** dans Git. Chiffrez-le côté backend.

# State local vs distant

| | LOCAL (DÉFAUT) | DISTANT (REMOTE BACKEND) |
|---|---|---|
| Stockage | `terraform.tfstate` sur disque | S3, GCS, Azure Blob, HCP Terraform |
| Usage | Développeur seul | **Indispensable en équipe** |
| Locking | Non | Oui (DynamoDB, GCS…) |
| Sécurité | Fichier en clair | Chiffrement côté serveur |
| Partage | Manuel | Automatique |

🎓 **Certif 004 :** savoir citer les types de backends distants et l'intérêt du **locking** (éviter deux `apply` concurrents qui corrompent le state).

# Configurer un backend S3 + locking

```
terraform {
  backend "s3" {
    bucket         = "mon-bucket-tfstate"
    key            = "projet/terraform.tfstate"
    region         = "eu-west-3"
    dynamodb_table = "terraform-locks"   # verrou
    encrypt        = true
  }
}
```

```
terraform init -migrate-state    # migrer un state local vers S3
```

💡 **Terrain :** le backend n'accepte **pas** de variables (évalué avant tout). Utilisez la **config partielle** : `terraform init -backend-config=prod.s3.tfbackend` pour cibler des buckets différents par environnement.

# Commandes de state (« state surgery »)

```
terraform state list                          # lister
terraform state show aws_instance.web         # détailler
terraform state rm aws_instance.web           # retirer du state (sans détruire)
terraform state mv aws_instance.web aws_instance.app  # renommer/déplacer
terraform state pull > backup.tfstate         # exporter
```

🎓 **Certif 004 :** bien distinguer `state rm` (oublie la ressource, qui survit dans le cloud) de `destroy` (supprime la ressource réelle). Sujet fréquent.

💡 **Terrain :** avant toute manip de state, `terraform state pull` pour une sauvegarde. En 1.7+, le bloc `moved {}` remplace souvent `state mv` (versionné, revu en MR).

# Drift : quand la réalité diverge

1. Quelqu'un modifie une ressource dans la console cloud.
2. Au prochain `terraform plan`, Terraform **détecte** l'écart (drift).
3. Deux options :
   - `terraform apply` pour **remettre l'infrastructure** dans l'état décrit par le code.
   - **Mettre à jour le code Terraform** si le changement manuel doit devenir la nouvelle référence.

💡 **Terrain :** Le code Terraform doit rester la **Single Source of Truth**. Une bonne pratique consiste à exécuter régulièrement un `terraform plan` (par exemple chaque nuit via la CI) afin de détecter rapidement les dérives (**drift**) et d'alerter l'équipe avant qu'elles ne s'accumulent.

# Drift : quand la réalité diverge

```
terraform apply -refresh-only
# Met à jour le state pour refléter l'état réel d'une ressource
# déjà présente dans le state, sans modifier l'infrastructure.
```

> ⚠️ **À ne pas confondre avec `terraform import` :**
>
> - `terraform import` (ou le bloc `import {}`) **ajoute une ressource existante au state** afin que Terraform commence à la gérer.
> - `terraform apply -refresh-only` **ne peut pas découvrir une nouvelle ressource**. Il synchronise uniquement le state des ressources **déjà gérées** par Terraform.

# plan et apply : qui compare quoi ? (sur un exemple)

| Code *.tf | | ce que je veux | ② | State | | ce que Terraform sait | ① | Azure (réalité) | | ce qui existe |
|---|---|---|---|---|---|---|---|---|---|---|
| web1 | D2 | | | web1 | D2 | | | web1 | D4 | modifiée à la main |
| web2 | D2 | | | web2 | D2 | | | web2 | D2 | |
| web3 | D2 | nouvelle dans le code | | web3 | – | inconnue | | web3 | – | n'existe pas |
| web4 | – | pas dans le code | | web4 | – | inconnue | | web4 | D2 | créée au portail |

### terraform plan

① **refresh** — demande à Azure des nouvelles de `web1` et `web2` (les seules du state)
→ `web1` est en **D4**. `web3`, `web4` ? pas dans le state → **pas demandées**

② **compare** au code : `web1` D2 ≠ D4 → **~ modifier** · `web3` absente → **+ créer**
`web4` ? Ni code, ni state → **pas une ligne dans le plan** : invisible.

### terraform apply

③ **exécute** le plan, rien d'autre : `web1` → D2, `web3` créée

④ **note** la réponse d'Azure dans le state : `web1` D2 · `web2` D2 · `web3` D2
`web4` : **intouchée** — ni détruite, ni adoptée. Tourne, facturée, sans pilote.
apply ne compare rien : il agit puis écrit. Adopter web4 → `import` (annexe C)

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

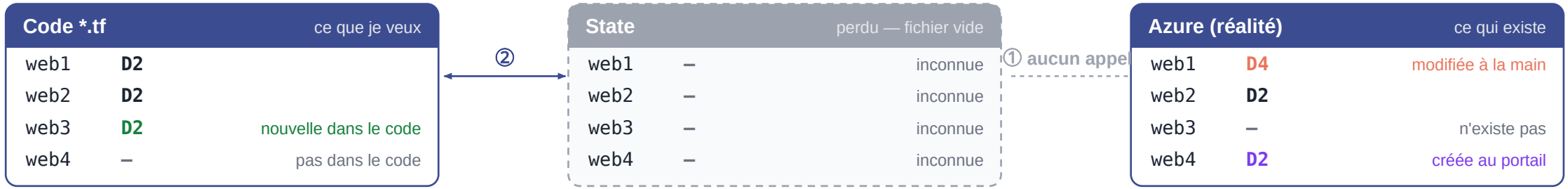
→

```
Apply complete! 1 added, 1 changed, 0 destroyed.
```

**À retenir :** **plan** = code **contre** state, après avoir mis le state à jour depuis le cloud — **uniquement pour les ressources qu'il liste** · **apply** = agir, **puis** écrire.

💡 **Image mentale :** le state est un **carnet d'inventaire**. *refresh* = recompter **les rayons listés dans le carnet** ; *plan* = comparer la commande (code) au carnet recompté ; *apply* = acheter / jeter, puis mettre le carnet à jour. Un rayon absent du carnet n'existe pas.

# … et le même film sans state




### terraform plan

① **refresh** — le state est vide : **aucune question** posée à Azure

② **compare** au code : `web1`, `web2`, `web3` absentes du state → **+ créer** ×3

Terraform **ignore** que `web1` et `web2` existent déjà : il ne les a pas regardées

`web4` : toujours invisible. Le plan paraît **normal** — c'est ça le piège.

### terraform apply

③ **exécute** : crée `web3` ✓ · crée `web1`, `web2` → **409 AlreadyExists**

(nom déjà pris dans le RG — azurerm refuse et demande un `import`)

④ **note** dans le state ce qui a réussi : `web3` D2 — **seulement**

`web1` (D4), `web2`, `web4` : hors Terraform. Noms générés → **doublons**.

```
Plan: 3 to add, 0 to change, 0 to destroy.
```

```
Error: A resource with the ID ".../web1" already exists
```

**À retenir :** sans state, le plan est **faux mais plausible** ; l'apply casse (409) ou duplique. On **reconstruit le state** (`import`) **avant** — annexe C.

⚠️ **Le vrai danger n'est pas l'erreur, c'est l'absence d'erreur :** avec des noms uniques ou générés (`random_id`, suffixes, AWS sans nom), l'`apply` **réussit** — et vous avez deux fois l'infra, l'ancienne orpheline et facturée. Réflexe avant tout `apply` suspect : `terraform state list` doit lister ce que vous savez exister.

## 🎓 Quiz : 5 questions

1. Le **state** est-il la source de vérité pour Terraform ?
2. `terraform state rm` détruit-il la ressource réelle ?
3. À quoi sert le **locking** ?
4. Quelle commande **synchronise le state** sans modifier les ressources ?
5. Citez **deux backends distants**.

## ✅ Corrections

1. **Oui** — il représente ce que Terraform connaît de l'infra.
2. **Non** — il la **retire du state** ; la ressource survit dans le cloud.
3. **Empêcher deux `apply` concurrents** de corrompre le state.
4. `terraform apply -refresh-only`.
5. **S3**, **GCS**, **Azure Blob**, **HCP Terraform** (deux suffisent).

🧪 ON PASSE À LA PRATIQUE

# Exercice 3 : State distant

Mettre l'état à l'abri avant de rentrer

- **Objectif** : migrer le state local vers le **backend Azure Storage** partagé (compte de stockage et container `tfstate` indiqués dans l'énoncé) et observer le verrou
- **Vous avez fini quand** : votre state est visible dans le container (`az storage blob list`) et un `plan` lancé pendant un autre `plan` est refusé (verrou)
- **Il vous faut** : votre projet de l'exercice 2 intact

📚 Site des labs Retengr → **Gestion de l'état** › Azure

🧪 ON PASSE À LA PRATIQUE

# Exercices 4 & 5 : Accéder à la VM, puis lui donner un nom

Réveil J2 : la VM d'hier répond enfin, et elle a un nom

- **Objectif** : **4** : générer une clé SSH, l'associer à la VM (utilisateur `ubuntu`), créer le NSG · **5** : un enregistrement DNS `<prénom>.labs.retengr.com` — la zone est chez **AWS** (Route 53) : vous déclarez un **second provider**, `aws`, dans le même projet
- **Vous avez fini quand** : `ssh -i thekey ubuntu@<prénom>.labs.retengr.com` répond
- **Il vous faut** : rien de nouveau côté théorie — un provider de plus, c'est tout

📚 Site des labs Retengr → **Accéder à votre VM** › Azure, puis **Gestion du DNS** › Azure

MODULE 07

# Les modules

Organiser, réutiliser, partager

# Module : définition & intérêt

Un **module** est un conteneur d'un ensemble de ressources. **Tout projet Terraform est un module** (le **Root module**).

- **Organisation** : découper un projet complexe
- **Réutilisation** : un module pour dev/staging/prod
- **Encapsulation** : cacher la complexité derrière des variables simples
- **Partage** : Registry ou dépôt Git

> 🎓 **Certif 004 :** distinguer **root module** (où l'on lance `apply`, un seul par projet) et **child module** (appelé, réutilisable, instanciable plusieurs fois).

# Structure & appel d'un module

```
modules/webserver/
├── main.tf       # Ressources
├── variables.tf  # Entrées
├── outputs.tf    # Sorties
└── README.md     # Documentation
```

```
module "webserver" {
  source        = "./modules/webserver"
  instance_type = "t3.micro"
  environment   = "production"
  server_count  = 3
}
```

> 💡 **Terrain :** un module ≠ une ressource. Créez un module quand un **ensemble cohérent** se répète (un « webserver » = instance + SG + EIP + DNS). Sur-modulariser nuit autant que ne pas modulariser.

# Module du Registry & versioning

```
module "vpc" {
  source  = "terraform-aws-modules/vpc/aws"
  version = "~> 5.0"
  name            = "mon-vpc"
  cidr            = "10.0.0.0/16"
  azs             = ["eu-west-3a", "eu-west-3b"]
  private_subnets = ["10.0.1.0/24", "10.0.2.0/24"]
}
```

🎓 **Certif 004 :** l'argument `version` n'est valable que pour les sources **Registry** (pas pour les sources locales/Git, où l'on épingle via la ref Git).

💡 **Terrain :** réutilisez les modules communautaires éprouvés ( `terraform-aws-modules` ) plutôt que réinventer un VPC. Mais **épinglez la version** : une montée majeure peut recréer des ressources.

# Entrées, sorties & sources

```
# Dans le module : outputs.tf
output "public_ips" { value = aws_instance.web[*].public_ip }

# Dans le root : consommer l'output
output "server_ips" { value = module.webserver.public_ips }
```

| SOURCE | EXEMPLE |
|---|---|
| Locale | `./modules/webserver` |
| Registry | `terraform-aws-modules/vpc/aws` |
| GitHub | `github.com/org/repo//modules/vpc` |
| Git | `git::https://example.com/module.git?ref=v1.2.0` |

💡 **Terrain :** le `//` cible un sous-dossier, le `?ref=` épingle un tag/commit Git. Toujours épingler une **ref immuable** (tag), jamais une branche mouvante comme `main`.

# Passer des providers & itérer sur un module

```
module "bucket_us" {
  source    = "./modules/bucket"
  providers = { aws = aws.us }      # injecter un alias
}

module "team" {
  source   = "./modules/user"
  for_each = toset(["alice", "bob"])
  username = each.key
}
```

💡 **Terrain :** `count` et `for_each` s'appliquent aussi aux **modules**, pas seulement aux ressources — puissant pour instancier un même périmètre par région/compte. Passez les providers explicitement quand le module crée des ressources multi-régions.

## 🎓 Quiz : 5 questions

1. Tout projet Terraform est-il un **module** ?
2. Comment **référencer un output** d'un module enfant ?
3. L'argument `version` fonctionne pour quelle **source** de module ?
4. Combien de **root modules** par projet ?
5. Comment **épingler** un module servi via Git ?

## ✅ Corrections

1. **Oui** — c'est le **root module**.
2. `module.<NOM>.<OUTPUT>` — ex. `module.webserver.public_ips`.
3. Les sources **Registry** uniquement (pas locale ni Git).
4. **Un seul**.
5. Avec `?ref=<tag ou commit>` — jamais une branche mouvante.

🧪 ON PASSE À LA PRATIQUE

# Exercices 6 & 7 : Datasources, puis votre module

Lire l'existant, puis factoriser

- **Objectif** : **6** : remplacer le `zone_id` écrit en dur par une **data source** Route 53, et créer une VM sur l'**image personnalisée** du lab (`data "azurerm_image"`) · **7** : sortir la VM dans un module local `./modules/vm`, l'appeler (`module "mon_serveur"`), puis découvrir les modules du Registry
- **Vous avez fini quand** : `terraform state list` montre `module.mon_serveur.…` et votre nom DNS répond toujours
- **Il vous faut** : les exercices 4 et 5 terminés

📚 Site des labs Retengr → **Les datasources** › Azure, puis **Les Modules** › Azure

MODULE 08

# Syntaxe avancée

count, for_each, expressions, lifecycle, dépendances

# count : N ressources identiques

```
resource "aws_instance" "app" {
  count         = 3
  ami           = var.ami_id
  instance_type = "t3.micro"
  tags = { Name = "app-${count.index}" }   # app-0, app-1, app-2
}
# Référence : aws_instance.app[0]
```

⚠️ **Limite de `count` :** les ressources sont indexées (0,1,2…). Supprimer l'élément du **milieu** décale tous les suivants → Terraform **détruit et recrée** les ressources décalées. Dangereux sur du stateful.

# for_each : collection à clés stables

```
variable "users" {
  type    = map(string)
  default = { alice = "admin", bob = "developer", carol = "viewer" }
}
resource "aws_iam_user" "team" {
  for_each = var.users
  name     = each.key
  tags     = { Role = each.value }
}
# Référence : aws_iam_user.team["alice"]
```

| | COUNT | FOR_EACH |
|---|---|---|
| Identifiant | Index numérique | Clé de la map/set |
| Suppression | **Décale** les index | **Stable** |
| Usage | Homogène/jetable | Paramétré |

🎓 **Certif 004 :** savoir choisir. Règle : **`for_each` par défaut**, `count` seulement pour un simple compteur ou un on/off conditionnel. Distinguer `count.index` de `each.key` / `each.value` .

# Expressions for & splat

```
[for s in var.names : upper(s)]               # ["ALICE","BOB"]
[for s in var.names : upper(s) if s != ""]    # filtrer
{for s in var.names : s => upper(s)}          # produire une map

# splat : équivalents
[for i in aws_instance.app : i.public_ip]
aws_instance.app[*].public_ip
```

💡 **Terrain :** les expressions `for` transforment des collections sans boucle externe. Le splat `[*]` est le raccourci idiomatique pour récupérer un attribut sur toutes les instances d'un `count` / `for_each`.

# Datasources : interroger l'existant

```
data "aws_ami" "ubuntu" {
  most_recent = true
  owners      = ["099720109477"]   # Canonical
  filter {
    name   = "name"
    values = ["ubuntu/images/hvm-ssd/ubuntu-jammy-22.04-amd64-*"]
  }
}
resource "aws_instance" "web" { ami = data.aws_ami.ubuntu.id }
```

⚠️ **Piège :** `most_recent = true` + filtre large = l'AMI peut **changer** entre deux `plan`, provoquant un remplacement d'instance surprise. Épinglez l'AMI ou filtrez précisément en prod.

# Conditionnelles & lifecycle

```
# 1. Expression conditionnelle : si var.environment = prod → t3.large, sinon → t3.micro
instance_type = var.environment == "prod" ? "t3.large" : "t3.micro"

# 2. Ressource optionnelle : créée uniquement si create_eip est vrai
resource "aws_eip" "web" {
  count = var.create_eip ? 1 : 0      # 1 = créée, 0 = aucune
}

# 3. lifecycle : piloter le cycle de vie de la ressource
resource "aws_instance" "web" {
  lifecycle {
    prevent_destroy       = true       # interdit la destruction
    ignore_changes        = [tags]     # ignore les changements de tags
    create_before_destroy = true       # crée le nouveau avant de détruire l'ancien
  }
}
```

🎓 **Certif 004 :** connaître les méta-arguments `lifecycle` (`create_before_destroy`, `prevent_destroy`, `ignore_changes`) et le pattern `count = condition ? 1 : 0` pour rendre une ressource optionnelle.

# Valider à trois niveaux

| MÉCANISME | PORTE SUR | ÉVALUÉ | SI ÇA ÉCHOUE |
|---|---|---|---|
| `validation` (bloc `variable`) | une **entrée** | au `plan`, avant tout | **erreur** : rien ne bouge |
| `precondition` (dans `lifecycle`) | une ressource / data source, **avant** | au `plan` (ou à l'`apply` si valeur inconnue) | **erreur** : la ressource n'est pas touchée |
| `postcondition` (dans `lifecycle`) | le **résultat** de la ressource (`self`) | après création / lecture | **erreur** : l'`apply` s'arrête, les dépendants ne sont pas créés |
| `check` (1.5+, bloc autonome) | l'**infra déployée**, dans son ensemble | à **chaque** `plan` et `apply` | **avertissement** seulement — n'empêche rien |

💡 **La règle :** `validation`, `precondition`, `postcondition` **bloquent** — ce sont des garde-fous. `check` **observe** — c'est un test de santé qui tourne à chaque exécution.

🎓 **Certif 004 :** la question type — « lequel n'empêche pas l'apply ? » → `check`. Et `self` n'est utilisable que dans une `postcondition`.

# Exemple : garde-fou et test de santé

```
resource "azurerm_linux_virtual_machine" "vm" {
  # ...
  lifecycle {
    precondition {
      condition     = contains(["Standard_B1s", "Standard_B2s"], var.vm_size)
      error_message = "Taille hors du catalogue autorisé."
    }
  }
}

resource "azurerm_public_ip" "main" {
  # ...
  lifecycle {
    postcondition {
      condition     = self.ip_address != ""
      error_message = "Pas d'IP attribuée : sku/allocation ?"
    }
  }
}
```

```
# check : la page web répond-elle ?
check "site_web" {
  data "http" "page" {          # data source « scopée »
    url = "http://${var.prenom}.labs.retengr.com"
  }

  assert {
    condition     = data.http.page.status_code == 200
    error_message = "La page ne répond pas 200."
  }
}
```

```
Warning: Check block assertion failed
  on checks.tf line 7, in check "site_web":
  La page ne répond pas 200.
```

💡 **Terrain :** le `check` automatise le test « mon service répond-il ? » — il tourne à chaque `plan`, donc il signale une panne **sans qu'on le lui demande**. Il exige le provider `hashicorp/http`.

# depends_on : dépendances explicites

```
resource "aws_instance" "web" {
  ami           = data.aws_ami.ubuntu.id
  instance_type = "t3.micro"
  depends_on    = [aws_iam_role_policy.web_policy]
}
```

💡 **Terrain :** n'utilisez `depends_on` que lorsque la dépendance **n'est pas** détectable via une référence (ex. une policy IAM qui doit exister avant que l'instance puisse l'utiliser, sans attribut partagé). En abuser sérialise inutilement le graphe et ralentit l'`apply`.

🎓 **Certif 004 :** Terraform construit les dépendances **implicitement** via les références ; `depends_on` est l'exception explicite.

# Renommer sans détruire : moved

Renommer une ressource ou la déplacer dans un module **change son adresse** : sans précaution, Terraform la **détruit puis la recrée**. Le bloc `moved` (Terraform 1.1+) déclare que c'est **la même ressource, à une autre adresse**.

```
# moved.tf — un bloc par ressource déplacée
moved {
  from = azurerm_linux_virtual_machine.vm
  to   = module.mon_serveur.azurerm_linux_virtual_machine.vm
}
```

```
# azurerm_linux_virtual_machine.vm has moved to module.mon_serveur.azurerm_linux_virtual_machine.vm
Plan: 0 to add, 0 to change, 0 to destroy.
```

# moved : les règles d'usage

- Les adresses sont **relatives au module qui contient le bloc** : `moved.tf` va dans le module **qui appelle** (ici le root), jamais dans le module déplacé
- L'`apply` reste nécessaire, même à « 0 to add » : c'est lui qui **écrit les nouvelles adresses** dans le state
- À garder tant que **tous** les environnements n'ont pas appliqué — puis à retirer, ou à conserver dans un module publié pour ses utilisateurs
- L'équivalent impératif, `terraform state mv`, fait la même chose **sans laisser de trace** dans le code : à réserver au dépannage

💡 **Terrain :** à faire pendant l'exercice 7, avant d'appliquer — regardez le `plan` **sans** `moved` (5 à détruire), puis **avec** (0 à détruire).

## 🎓 Quiz : 5 questions

1. Faut-il utiliser `count` ou `for_each` pour des clés stables ?
2. Comment **référencer** un élément créé par `for_each` ?
3. Que fait le méta-argument `prevent_destroy` ?
4. À quoi sert `count = var.x ? 1 : 0` ?
5. Quand faut-il utiliser `depends_on` ?

## ✅ Corrections

1. `for_each` — clé stable, pas de décalage à la suppression.
2. Par sa **clé** — ex. `aws_iam_user.team["alice"]`.
3. Il **empêche la destruction** de la ressource (garde-fou).
4. Créer une ressource **conditionnellement** (1 = créée, 0 = absente).
5. Seulement quand la dépendance **n'est pas détectable** via une référence.

🧪 ON PASSE À LA PRATIQUE

# Cas pratique : Une flotte avec `for_each`

Le bouquet final (optionnel)

- **Objectif** : déployer une infrastructure complète pour **une liste de participants** (`for_each` sur une `map`), avec le provider **AWS ou GCP** — même méthode, autre cloud
- **Vous avez fini quand** : une ressource par participant, puis `terraform destroy` laisse un state vide
- **Il vous faut** : l'exercice 7 terminé

📚 Site des labs Retengr → **Cas pratique**

MODULE 09

# Bonnes pratiques

Qualité, sécurité, migration, environnements, CI/CD

# Formater & valider

```
terraform fmt -recursive      # formate tout le code HCL
terraform validate            # valide syntaxe & cohérence (HORS cloud)
# → Success! The configuration is valid.
```

🎓 **Certif 004 :** `fmt` reformate, `validate` vérifie **sans contacter** le cloud (donc utilisable hors authentification, en CI early-stage). Piège classique de QCM.

💡 **Terrain :** automatisez via **pre-commit** (`fmt -check`, `validate`, `tflint`) pour bloquer le code mal formé avant le commit, pas en revue.

# `terraform test` : tester son code (1.6+)

Des fichiers `*.tftest.hcl` (dans `tests/`), lancés par `terraform test` : chaque bloc `run` exécute la configuration puis vérifie des `assert`.

| `run` EN… | CE QUI SE PASSE | COÛT / DURÉE | SERT À |
|---|---|---|---|
| `command = plan` | plan seulement, rien n'est créé | gratuit, secondes | test **unitaire** : noms, tailles, logique, conditions |
| `command = apply` (défaut) | crée **pour de vrai**, puis **détruit** à la fin | facturé, minutes | test **d'intégration** : l'infra fonctionne |
| + `mock_provider` (1.7+) | le provider est **simulé** : aucun appel API | gratuit, sans credentials | tester un module **hors cloud**, en CI |

- `expect_failures = [var.x]` : le test **réussit si** la `validation` (ou la condition) **échoue** — on teste les garde-fous
- Les tests ciblent le **module du dossier courant** : idéal pour une brique réutilisable comme `modules/vm`

💡 **Terrain :** `validate` dit « le code est bien écrit », `plan` dit « voici ce qui changera », `test` dit « **le code fait ce que j'attends** ». En CI : `fmt -check` → `validate` → `test` (mocks) → `plan`.

# Exemple : tester le module `vm` sans Azure

```
# modules/vm/tests/vm.tftest.hcl
mock_provider "azurerm" {}        # aucun appel à Azure

variables {
  nom                 = "test"
  resource_group_name = "rg-test"
  location            = "francecentral"
  subnet_id           = "/subscriptions/0000/…/subnets/s"
  ssh_public_key      = file("../../thekey.pub")
  image_id            = "/subscriptions/0000/…/images/image4lab"
}

run "nommage_et_taille" {
  command = plan
  assert {
    condition     = azurerm_linux_virtual_machine.vm.name == "vm-test"
    error_message = "La VM doit s'appeler vm-<nom>."
  }
}

run "taille_hors_catalogue_refusee" {
  command   = plan
  variables { vm_size = "Standard_D4s_v5" }
  expect_failures = [var.vm_size]
}
```

```
# modules/vm/variables.tf : le garde-fou testé
variable "vm_size" {
  type    = string
  default = "Standard_B1s"
  validation {
    condition     = contains(["Standard_B1s", "Standard_B2s"], var.vm_size)
    error_message = "Taille hors du catalogue autorisé."
  }
}
```

```
$ cd modules/vm && terraform init && terraform test
tests/vm.tftest.hcl... in progress
  run "nommage_et_taille"... pass
  run "taille_hors_catalogue_refusee"... pass
tests/vm.tftest.hcl... pass

Success! 2 passed, 0 failed.
```

🎓 **Certif 004 :** `.tftest.hcl` · `run` / `assert` · `plan` vs `apply` (défaut, détruit à la fin) · `expect_failures`

# Organisation & .gitignore

```
# .gitignore Terraform
.terraform/             # plugins téléchargés
*.tfstate               # state (données sensibles)
*.tfstate.backup
*.tfvars                # variables sensibles
!terraform.tfvars       # …sauf celui-ci si non secret
crash.log
```

⚠️ **À ne PAS ignorer :** committez **toujours** `.terraform.lock.hcl` (reproductibilité des versions de providers). C'est l'inverse du `.terraform/`.

## Sécurité

- **Jamais** de credentials en dur → variables d'env, OIDC, gestionnaire de secrets
- Variables sensibles : `sensitive = true`
- Restreindre le réseau : **pas de** `0.0.0.0/0` en production
- **Chiffrer** le state (backend S3 avec `encrypt = true`)

> 💡 **Terrain — outils :** `tfsec` / `checkov` / `trivy` scannent les configs à risque (bucket public, SG ouvert) ; `infracost` estime le coût d'un `plan` ; `terraform-docs` génère la doc des modules. À brancher en CI.

# Gérer les secrets : principes

## D'où viennent les secrets ?

- Variables d'environnement `TF_VAR_*` (injectées en CI)
- Coffre-fort : **HashiCorp Vault**, AWS Secrets Manager, GCP Secret Manager
- Fichiers chiffrés : **SOPS** + KMS/age
- Store de secrets de la CI (GitLab/GitHub)

## Règles d'or

- **Jamais** de secret en clair dans le code ni Git
- `sensitive = true` masque les **logs**, pas le **state**
- Pas de secret dans un `output` non `sensitive`
- Restreindre **qui peut lire le state**

⚠️ **Le maillon faible, c'est le state :** tout secret lu ou créé par Terraform finit **en clair dans le state**. La protection passe d'abord par un **backend chiffré à accès restreint** (S3 + KMS, RBAC).

# HashiCorp Vault avec Terraform

Le **provider** `vault` lit des secrets via des **data sources** :

```
provider "vault" {
  address = "https://vault.example.com:8200"   # token via VAULT_TOKEN / AppRole / OIDC
}

data "vault_kv_secret_v2" "db" {
  mount = "secret"
  name  = "prod/database"
}

resource "aws_db_instance" "main" {
  username = "admin"
  password = data.vault_kv_secret_v2.db.data["password"]   # ⚠ transite par le state
}
```

🎓 **À retenir :** un secret lu via une **data source** est **écrit dans le state**. Lire depuis Vault ne dispense **pas** de chiffrer le state.

# Vault : secrets dynamiques & ephemeral

## Secrets statiques (KV)

- Mot de passe stocké, lu tel quel
- Simple, mais **persisté dans le state**
- Rotation manuelle

## Secrets dynamiques

- Vault **génère à la volée** des credentials cloud/DB temporaires (TTL court)
- Révoqués automatiquement
- Terraform s'authentifie avec des creds **éphémères**

💡 **Terrain :** en CI, authentifiez Vault via **AppRole / OIDC** (pas de token statique). Depuis Terraform **1.10+**, les ressources/valeurs `ephemeral` et les arguments **write-only** permettent de manipuler un secret **sans l'écrire dans le state** — la direction recommandée pour les nouveaux projets.

# Importer l'existant

```
terraform import aws_instance.web i-0abc123def456   # CLI (puis écrire le HCL)
```

```
import {                                # bloc import — Terraform 1.5+
  to = aws_instance.web
  id = "i-0abc123def456"
}
# Générer le code :
# terraform plan -generate-config-out=generated.tf
```

# Importer l'existant

```
resource "azurerm_resource_group" "rg" {
  name     = "rg-prod"
  location = "France Central"
}

import {
  to = azurerm_resource_group.rg
  id = "/subscriptions/xxxxxxxx/resourceGroups/rg-prod"
}
```

🎓 **Certif 004 :** l' `import` rattache une ressource existante au state. Le **bloc `import`** (1.5+) est déclaratif et versionnable, et peut générer le HCL — approche recommandée vs la commande impérative.

# Workspaces & environnements

```
terraform workspace new staging
terraform workspace select production
```

```
instance_type = terraform.workspace == "production" ? "t3.large" : "t3.micro"
```

⚠️ **Débat terrain :** les **workspaces** partagent le même backend/key et conviennent à des variantes **légères**. Pour des environnements réellement isolés (prod/dev), beaucoup préfèrent des **répertoires/backends séparés** — isolation du state et du blast radius.

# Arborescence multi-environnements

```
infra/
├── modules/                # code partagé
│   ├── network/
│   ├── compute/
│   └── database/
└── environments/
    ├── dev/
    │   ├── main.tf         # appelle les modules
    │   ├── backend.tf      # state dev (clé dédiée)
    │   └── terraform.tfvars
    ├── staging/
    │   ├── main.tf
    │   ├── backend.tf      # state staging
    │   └── terraform.tfvars
    └── prod/
        ├── main.tf
        ├── backend.tf      # state prod
        └── terraform.tfvars
```

## Principe

- Un **répertoire par environnement** = un **state isolé**
- Le **code commun** vit dans `modules/`, appelé partout
- Seuls les `tfvars` et le `backend` diffèrent

```
# environments/prod/main.tf
module "network" {
  source = "../../modules/network"
  cidr   = var.cidr
}
```

💡 **Terrain :** on lance `terraform apply` **depuis le dossier de l'environnement**. Avantages vs workspaces : backends/clés distincts, droits IAM séparés par env, et un `destroy` sur `dev` ne peut **pas** toucher `prod`.

# CI/CD & pièges courants

```
stages: [validate, plan, apply]
validate: { script: [terraform fmt -check, terraform init, terraform validate] }
plan:     { script: [terraform plan -out=tfplan] }
apply:    { script: [terraform apply tfplan], when: manual }   # approbation humaine
```

**Pièges à éviter :** committer le state · modifier l'infra à la main · ignorer le `plan` · hardcoder les valeurs · négliger le nommage.

> 💡 **Terrain :** `apply` applique le **plan figé** (`tfplan`), jamais un recalcul. Gate manuel sur prod, `plan` automatique en MR pour la revue.

## 🎓 Quiz : 5 questions

1. `terraform validate` contacte-t-il le cloud ?
2. Faut-il **committer** `.terraform.lock.hcl` ?
3. Faut-il **committer** les fichiers `*.tfstate` ?
4. Quel **bloc** importe une ressource existante (Terraform 1.5+) ?
5. À quoi sert un **workspace** ?

## ✅ Corrections

1. **Non** — il valide la syntaxe et la cohérence **hors-ligne**.
2. **Oui** — pour garantir des versions de providers reproductibles.
3. **Non** — données sensibles ; utilisez un **backend distant**.
4. Le bloc `import {}` (générable en HCL via `-generate-config-out`).
5. **Gérer plusieurs déploiements/états** avec le même code.

ANNEXE A

# Versions & cycle de vie de Terraform

Rester à jour sans casser la prod — pinner, verrouiller, monter par petits sauts

# Terraform ≠ un seul produit

Avant de parler « version », il faut distinguer les produits, qui n'ont **pas le même cycle** :

| PRODUIT | DESCRIPTION | VERSIONNAGE |
|---|---|---|
| **Terraform CLI (core)** | L'outil en ligne de commande, open source* | SemVer `X.Y.Z` |
| **Terraform Enterprise** | Plateforme self-hosted | Trimestriel `x.y.z` (depuis août 2025) |
| **HCP Terraform** | SaaS managé (ex-Terraform Cloud) | Continu (rolling) |
| **OpenTofu** | Fork open source communautaire (Linux Foundation) | SemVer indépendant |

* Licence BUSL-1.1 depuis août 2023 — d'où l'existence du fork OpenTofu.

# Le versionnage de la CLI : SemVer

Format **`MAJOR.MINOR.PATCH`** (ex. `1.16.7`) :

- **PATCH** (`.Z`) → corrections de bugs uniquement, **iso-fonctionnel**
- **MINOR** (`.Y`) → nouvelles fonctionnalités, **compatibilité préservée**
- **MAJOR** (`.X`) → changements potentiellement **cassants**

> 💡 **Terrain :** un passage `1.16.3 → 1.16.7` est quasi sans risque. Un passage `1.16 → 1.17` mérite un coup d'œil au **CHANGELOG**.

# Le cycle de vie (EoL Policy HashiCorp)

- Une release **GA** (General Availability) est supportée jusqu'à **2 ans**
  - alpha → beta → release candidate (RC) → GA
- Les correctifs (sécurité, bugs) arrivent via une **nouvelle release mineure**
- Support limité aux **2 dernières branches** (X.Y)
- Terraform Enterprise : politique **« Advisory » de 2 ans** (conseils & bonnes pratiques)

⚠️ **À retenir :** une fois la branche en **End of Life**, plus aucun patch de sécurité. Rester sur une version EoL = dette technique + risque **CVE** non corrigé.

# Photo des versions (novembre 2026)

| BRANCHE | SORTIE | SUPPORT SÉCURITÉ | DERNIER PATCH |
|---|---|---|---|
| **1.17** | Oct 2026 | ✅ Actif | 1.17.2 (nov. 2026) |
| **1.16** | Aoû 2026 | ✅ Actif | 1.16.7 |
| 1.15 | Avr 2026 | ❌ Terminé (oct 2026) | 1.15.9 |
| 1.14 | Nov 2025 | ❌ Terminé (aoû 2026) | 1.14.9 |
| ≤ 1.13 | — | ❌ Terminé | — |

💡 **Cible du jour : 1.16.x** (stable, éprouvée) ou **1.17.x** (dernière). Règle observée : **deux branches** maintenues à la fois, chaque sortie de mineure **retire** la plus ancienne. Vérifier en continu sur `endoflife.date/terraform`. (instantané au 18/11/2026 — à réactualiser)

# Guideline : Épingler la version

La règle d'or : **pinner, mais pas trop**.

```
terraform {
  # Module racine : on borne pour éviter les mauvaises surprises
  required_version = "~> 1.16.0"   # accepte 1.16.x, refuse 1.17

  required_providers {
    azurerm = {
      source  = "hashicorp/azurerm"
      version = "~> 4.0"           # accepte 4.x, refuse 5.x (majeure de juil. 2026)
    }
  }
}
```

- **Module racine** → contraintes **strictes** (`~> 1.16.0`)
- **Modules réutilisables** → contraintes **minimales** (`>= 1.15.0`) pour laisser de la flexibilité aux consommateurs

# Guideline : Verrouiller l'environnement

Reproductibilité = même version partout (dev, CI, prod) :

- `.terraform.lock.hcl` → versionner ce fichier dans Git (verrouille les **providers**)
- `tfenv` / `tenv` + un fichier `.terraform-version` → verrouille la **CLI** par projet
- **Image CI figée** → ne jamais utiliser `latest` dans le pipeline

```
# .terraform-version
1.15.9
```

> 💡 **Objectif :** « ça marche sur ma machine » ≡ « ça marche en CI ».

# En résumé : les trois verrous de version

| QUOI | CONTRAINTE (DANS LE CODE) | VERROU (VERSION EXACTE) | QUI L'ÉCRIT | DANS GIT ? | POUR MONTER |
|---|---|---|---|---|---|
| **CLI Terraform** | `required_version = "~> 1.16.0"` — **refuse** un binaire hors plage | `.terraform-version` → `tenv` / `tfenv` (outil externe, pas Terraform) | vous | ✅ | éditer le fichier, `tenv install` |
| **Providers** | `required_providers { version = "~> 4.0" }` | `.terraform.lock.hcl` — version **+ hashes** | `terraform init` | ✅ **toujours** | changer la contrainte, `init -upgrade` |
| **Modules** | `version = "~> 2.0"` (Registry) ou tag/ref git dans `source` | **aucun** — la contrainte **est** le verrou | vous | ✅ (le code) | changer la contrainte, `init -upgrade` |

## Ce qui n'est jamais dans Git

- `.terraform/` (providers et modules téléchargés)
- `*.tfstate` (secrets, cf. module 06)

## La règle d'or

Une contrainte **dit ce qu'on accepte** ; un verrou **dit ce qu'on a**. Sans verrou, deux `init` à un mois d'écart n'installent pas la même chose — et personne ne le voit.

⚠️ **Les deux oublis qui coûtent cher :** un lock file non committé (chacun sa version de provider, *drift d'outillage*) · un module sans `version` ni tag (le code change sous vos pieds au prochain `init -upgrade`).

# Guideline : Stratégie de montée de version

| TYPE DE SAUT | RISQUE | PROCESS |
|---|---|---|
| **Patch** (`1.16.3→1.16.7`) | Faible | Auto via Renovate/Dependabot, merge après CI verte |
| **Mineure** (`1.16→1.17`) | Moyen | Lire le CHANGELOG, tester en sandbox, `terraform plan` à blanc |
| **Majeure** (`1.x→2.x`) | Élevé | Lire le **guide de migration**, environnement non-critique d'abord, rollback planifié |

⚠️ **Jamais** de `terraform apply` direct en prod après un saut majeur.

# Guideline : Cadence réaliste

Adapter le rythme à **votre contexte**, pas à l'idéal théorique :

- **Patchs de sécurité** → dès qu'ils sortent (semaine)
- **Mineures** → tous les 1-3 mois, dans une fenêtre planifiée
- **Ne jamais dépasser EoL** → garder ≥ 1 branche d'avance sur la fin de support
- **Automatiser la veille** → Renovate scanne et ouvre les PR ; vous gardez la main sur le merge

> 💡 **Le piège classique :** laisser dériver 3-4 versions, puis devoir tout migrer d'un coup. Mieux vaut de **petits sauts fréquents** qu'un grand saut douloureux.

# Guideline : Checklist avant upgrade

1. ☐ Lire le **CHANGELOG** de la version cible
2. ☐ Vérifier la compatibilité des **providers** (azurerm, etc.)
3. ☐ Mettre à jour `required_version` + `.terraform-version`
4. ☐ `terraform init -upgrade` puis **`terraform plan`** (diff à blanc)
5. ☐ Tester sur un **workspace non-prod** d'abord
6. ☐ Valider en **CI** (lint `tflint`, scan `tfsec` / `trivy`)
7. ☐ Appliquer en prod dans une **fenêtre** avec plan de **rollback**

# Où trouver l'info dans la doc

| BESOIN | OÙ |
|---|---|
| Releases + CHANGELOG | `github.com/hashicorp/terraform/releases` |
| EoL & versions supportées | `endoflife.date/terraform` |
| Contraintes de version | `developer.hashicorp.com/terraform/language/expressions/version-constraints` |
| Guides de migration (upgrade) | `developer.hashicorp.com/terraform/language/upgrade-guides` |
| Cycle Terraform Enterprise | `developer.hashicorp.com/terraform/enterprise/releases` |
| Vérifier sa version locale | `terraform version` |

## En résumé : Versions & cycle de vie

- **Distinguer** les produits (CLI, Enterprise, HCP, OpenTofu)
- **Pinner** intelligemment (`required_version`, `required_providers`)
- **Verrouiller** l'environnement (lock file, `tfenv`, CI figée)
- **Automatiser** la veille · **petits sauts** fréquents · **jamais** d'EoL en prod

ANNEXE B

# Cycle de vie du provider `azurerm`

Le plugin Terraform qui pilote Azure Resource Manager — versionné à part

## Qu'est-ce que le provider `azurerm` ?

- Le **plugin** qui permet à Terraform de parler aux **API Azure Resource Manager**
- Source : `hashicorp/azurerm` sur le Terraform Registry
- **Co-maintenu** par l'équipe Azure de Microsoft **et** l'équipe Terraform de HashiCorp
- Gère le cycle **create / update / destroy** de chaque ressource Azure

> ⚠️ **À ne pas confondre** avec le cycle de **Terraform Core** (la CLI) : le provider a son **propre versionnage**, totalement découplé.

# Deux cycles distincts à ne pas mélanger

| | TERRAFORM CORE (CLI) | PROVIDER AZURERM |
|---|---|---|
| **Rôle** | Moteur d'exécution | Connecteur Azure |
| **Versionnage** | SemVer `1.X.Z` | SemVer `4.X.Z` |
| **Cadence** | Mineures ~tous les 3-6 mois | Mineures **quasi hebdomadaires** |
| **Déclaré dans** | `required_version` | `required_providers` |

💡 **Terrain :** on épingle **les deux séparément**, et une version majeure du provider impose souvent une **version minimale de Core**.

# Le versionnage du provider (SemVer)

Format `MAJOR.MINOR.PATCH`, mais avec une dynamique propre :

- **PATCH / MINOR** → nouvelles ressources, support de nouvelles API Azure, fixes
  → publiées **très fréquemment** (Azure évolue en permanence)
- **MAJOR** → changements **cassants** : suppression de ressources dépréciées, changement de schéma, defaults modifiés

> 💡 **Reco HashiCorp :** pas plus d'**une majeure par an** pour laisser le temps de migrer.

# Historique des versions majeures

| VERSION | SORTIE | MARQUEUR |
|---|---|---|
| **1.0** | 2018 | Première version stable |
| **2.0** | Févr. 2020 | Refonte, séparation des ressources |
| **3.0** | Mars 2022 | Auth MSAL, ressources figées |
| **4.0** | Août 2024 | Exige **Terraform ≥ 1.0**, nettoyage majeur |
| 4.81 | 14 juil. 2026 | Dernière mineure de la branche 4.x |
| **5.0** | **28 juil. 2026** | Nouvelle majeure (voir slide suivante) |
| 5.14 | 13 nov. 2026 | Dernière mineure en date |

La v4 a accumulé 81 mineures en moins de 2 ans, et la v5 en sort **une par semaine** : c'est le rythme à absorber. (instantané au 18/11/2026)

# azurerm 5.0 : une majeure « en direct »

Sortie le **28 juillet 2026**. Ce qu'elle change, et pourquoi les labs restent en 4.x (épinglés en `4.14.0`) :

| CHANGEMENT 5.0 | IMPACT CONCRET |
|---|---|
| `resource_provider_registrations` vaut **`none`** **par défaut** ; `skip_provider_registration` **supprimé** | Le provider n'enregistre plus rien sur la souscription — exactement ce que nous forçons déjà en 4.x |
| `enhanced_validation` déplacé **dans** **`features {}`**, `preflight_enabled` (API Preflight au `plan`) | Nouvelles options, pas de rupture si on ne les utilisait pas |
| **Ressources supprimées** : `azurerm_app_service`, `azurerm_function_app`, la plupart des `azurerm_postgresql_*`, `azurerm_hpc_cache_*` | Un projet qui les utilise encore **ne passe plus** **`init`** → c'est le « saut de majeure se prépare » |
| `azurerm_subnet` : `service_endpoints` → bloc `service_endpoint` ; `azurerm_key_vault` : `rbac_authorization_enabled` **obligatoire** | Diffs de code à faire, visibles au `plan` |

💡 **Terrain :** pour les labs (VM, VNet, NSG) la 5.0 ne change rien — mais on reste en 4.x jusqu'à avoir **lu le guide de migration** et rejoué le `plan`. C'est la règle, pas l'exception. 📚 Source : *azurerm 5.0 Upgrade Guide*, HashiCorp, juillet 2026.

# Le cycle de vie d'une ressource

Une ressource ne disparaît jamais brutalement — elle suit un parcours :

**Active → Dépréciée → Feature-frozen → Supprimée (majeure suivante)**

- **Dépréciée** : un warning apparaît au `plan`, une remplaçante existe
- **Feature-frozen** : la ressource reste utilisable mais ne reçoit **plus aucune évolution**
- **Supprimée** : uniquement lors d'un **saut de version majeure**

> 💡 **Terrain :** c'est pour ça qu'un saut de majeure **se prépare** — c'est là que les ressources dépréciées sautent.

# Couplage avec Terraform Core

Le provider impose une **version minimale de Core** :

```
terraform {
  required_version = ">= 1.0"          # exigé par azurerm v4

  required_providers {
    azurerm = {
      source  = "hashicorp/azurerm"
      version = "~> 4.0"               # accepte 4.x, refuse 5.x
    }
  }
}
```

⚠️ **Effet domino :** un Core en fin de vie finit par ne plus être testé par le provider → `terraform init` ou `plan` peut casser même si le binaire « tourne » encore.

# Guideline : Épingler intelligemment

| CONTEXTE | CONTRAINTE | EFFET |
|---|---|---|
| **Prod critique** | `= 4.81.0` | Version exacte, zéro surprise |
| **Standard** | `~> 4.0` | Toutes les 4.x, bloque la 5.x |
| **Module réutilisable** | `>= 4.0` | Souple pour les consommateurs |

💡 **Toujours** versionner le `.terraform.lock.hcl` dans Git : il fige la version exacte **et** les checksums (intégrité supply-chain).

# Guideline : Cadence de mise à jour

Le rythme du provider est rapide : adopter une stratégie **incrémentale**.

- **Mineures/patchs** → tous les 1-3 mois, dans une fenêtre planifiée
  → automatisable via **Renovate / Dependabot** (PR auto + CI verte)
- **Majeure** (`4.x → 5.x`) → projet à part entière :
  i. Lire le **Upgrade Guide** officiel
  ii. Tester en **sandbox** + `terraform plan` à blanc
  iii. Traiter les ressources dépréciées **avant** le saut
- **Ne jamais** sauter plusieurs majeures d'un coup

## Guideline : Checklist avant montée de version

1. ☐ Lire le **CHANGELOG** (et le Upgrade Guide si majeure)
2. ☐ Vérifier la version **minimale de Core** exigée
3. ☐ Identifier les ressources **dépréciées** utilisées
4. ☐ Mettre à jour la contrainte + `terraform init -upgrade`
5. ☐ **`terraform plan`** : analyser le diff (drift, recréations)
6. ☐ Tester sur **non-prod**, valider CI (`tflint`, `tfsec`)
7. ☐ Mettre à jour et **committer** le lock file

# Où trouver l'info dans la doc

| BESOIN | OÙ |
|---|---|
| Doc + version courante | `registry.terraform.io/providers/hashicorp/azurerm/latest` |
| Upgrade guides (3.0, 4.0…) | Section **Guides** du Registry |
| Toutes les releases / binaires | `releases.hashicorp.com/terraform-provider-azurerm` |
| CHANGELOG détaillé | `github.com/hashicorp/terraform-provider-azurerm/releases` |
| Historique versions (Microsoft) | `learn.microsoft.com/azure/developer/terraform` |

## En résumé : Provider `azurerm`

- Provider **découplé** de Core, avec son **propre** SemVer
- **Mineures fréquentes** (Azure bouge en permanence)
- Ressources **dépréciées** → **figées** → **supprimées** à la majeure
- **Pinner** + lock file · **petites montées** régulières · **préparer** les sauts majeurs

ANNEXE C

# State perdu : récupération

Ne pas paniquer : votre infra est toujours là — c'est la mémoire de Terraform qu'il faut reconstruire

## D'abord : comprendre ce qui s'est passé

Le state perdu **≠ infrastructure détruite**.

- Le state est juste le **fichier de correspondance** entre le code HCL et les ressources réelles
- Les ressources Azure/AWS/etc. **continuent de tourner** normalement
- Terraform a simplement « **oublié** » qu'il les gérait

> 💡 **Le problème** n'est pas la prod, c'est la **mémoire** de Terraform. Objectif : reconstruire cette mémoire, **sans toucher** aux ressources.

# Triage : évaluer la situation

Avant d'agir, répondre à 3 questions :

1. **Backend local ou distant ?** (`terraform.tfstate` sur disque vs Azure Blob / S3 / HCP)
2. **Le versionnage est-il activé** sur le backend distant ?
3. **Reste-t-il une copie quelque part ?** (backup local, snapshot, autre poste, CI)

> 💡 **Stratégie :** elle dépend entièrement de ces réponses. On va du plus simple (**restaurer**) au plus coûteux (**reconstruire**).

# Piste 1 : Le backup local automatique

Terraform garde **automatiquement** une copie de l'avant-dernier state :

```
ls -la
# terraform.tfstate          <- perdu / corrompu
# terraform.tfstate.backup   <- la copie de secours !

cp terraform.tfstate.backup terraform.tfstate
terraform plan   # vérifier que tout colle
```

💡 **Ce fichier** `.backup` est écrit à chaque `apply`. C'est la récupération la plus rapide — à tenter **en premier** en local.

# Piste 2 : L'historique du backend distant

Si le state est sur un backend distant **avec versionnage**, tout est récupérable.

| BACKEND | MÉCANISME |
|---|---|
| **Azure Blob Storage** | **Blob versioning** / **soft delete** → restaurer la version |
| **AWS S3** | **Bucket versioning** → restaurer une version antérieure |
| **HCP Terraform** | Onglet **States** → historique, **rollback** en 1 clic |

💡 **Leçon :** un backend distant **versionné** transforme une catastrophe en non-événement. C'est la raison n°1 d'utiliser un remote backend.

# Piste 3 : Reconstruire par import

Aucun backup ? Il faut **reconstruire le state depuis l'infra existante**.

```
# Commande terraform import (historique, une ressource à la fois)
terraform import azurerm_resource_group.rg \
  /subscriptions/xxxx/resourceGroups/mon-rg
```

```
# Blocs import {} (Terraform 1.5+, déclaratif, idéal en masse)
import {
  to = azurerm_resource_group.rg
  id = "/subscriptions/xxxx/resourceGroups/mon-rg"
}
```

⚠️ **L'import** remplit **le state uniquement** — le HCL doit déjà exister (ou être généré via `-generate-config-out`).

# Idée reçue : « un `plan` va reconstruire le state »

**Non.** `terraform plan` est en **lecture seule** : il ne modifie jamais le state.

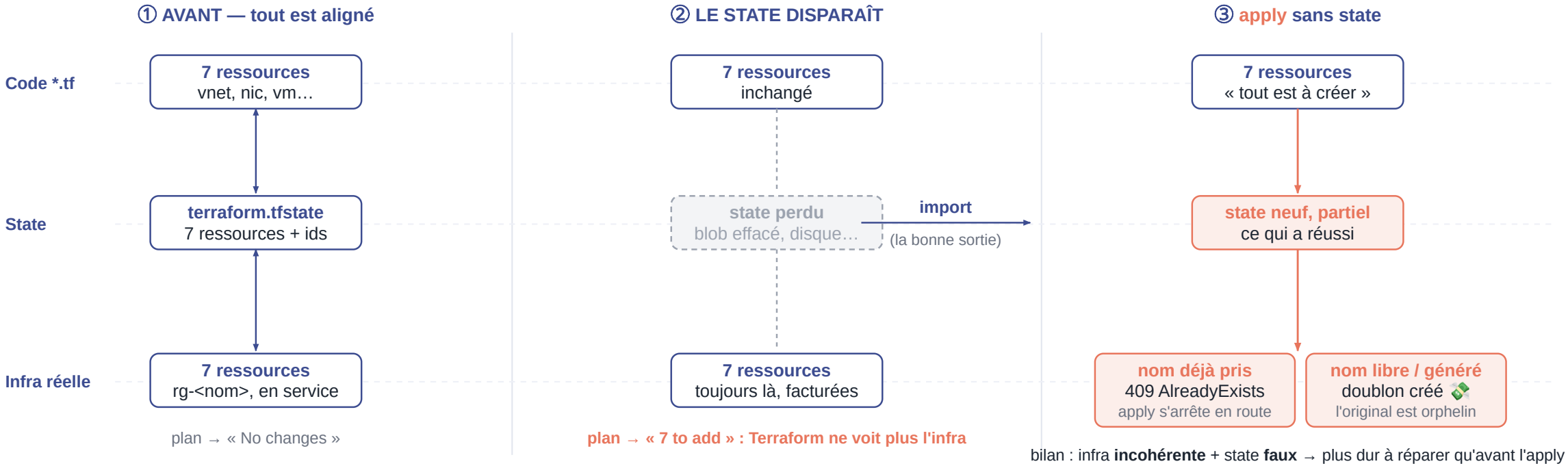
- Il **lit** le state, **refresh** via l'API, et **compare** au HCL → produit un **rapport**
- Il ne regarde que les ressources **déjà** dans le state — il ne « découvre » rien

Avec un state vide, le plan affiche donc :

```
Plan: 12 to add, 0 to change, 0 to destroy.
```

⚠️ **Piège :** Terraform croit que tout est à **créer** → un `apply` ferait des **doublons**. C'est l'**import** qui reconstruit le state, jamais le plan.

# Apply sans state : le film en trois temps




**Règle :** sans state, Terraform ne « découvre » jamais l'existant — `plan` ne compare le code qu'au state, pas au cloud. Reconstruire le state (`import`, blocs `import {}`, `aztfexport`) **avant** tout `apply`.

# La séquence correcte

```
import        # crée le lien state ↔ ressource réelle
   ↓
plan          # vérifie l'écart code ↔ réalité (lecture seule)
   ↓
ajuster HCL   # corriger valeurs / defaults / attributs
   ↓
plan vide     # "No changes" = state reconstruit et cohérent ✅
```

💡 **Nuance :** avec les blocs `import {}` (1.5+), le `plan` **prévisualise** l'import, mais c'est l'`apply` qui l'exécute et écrit le state.

# Piste 3 bis : Import en masse (Azure)

Réimporter à la main des dizaines de ressources est pénible. Pour **Azure** :

`aztfexport` (Azure Export for Terraform, maintenu par Microsoft)

```
# Réimporte un resource group entier : génère HCL + state
aztfexport resourcegroup mon-rg

# Mode --append pour s'intégrer à un backend distant existant
aztfexport resource --append <resource_id>
```

> 💡 **Multi-cloud : Terraformer** ; ou les blocs `import` avec `for_each` (TF **1.7+**) pour l'import scripté.

# Après l'import : valider jusqu'au plan propre

L'import ne suffit pas — il faut **converger** code ↔ réalité.

1. `terraform plan`
2. Lire le diff : tout écart = mismatch entre le HCL et la ressource réelle
3. Ajuster le HCL (valeurs, defaults, attributs computed)
4. Répéter **jusqu'à un plan vide** (`No changes`)

> ⚠️ **Un `apply`** sur un plan non vide après import peut **recréer ou détruire** des ressources. Le plan propre est la condition de sécurité absolue.

# Ce qu'il NE faut PAS faire

- ❌ **Éditer le state à la main** (JSON) → corruption de **lineage**, state « forké »
- ❌ Lancer un `apply` **à l'aveugle** → risque de recréation/destruction
- ❌ Refaire un `apply` **complet** en pensant « recréer » → doublons et conflits
- ❌ Travailler **sans lock** pendant la récup → deux personnes, deux états divergents

> 💡 **Toujours** préférer `terraform state mv` / `terraform state rm` aux éditions manuelles.

# Prévention : ne plus jamais revivre ça

| MESURE | BÉNÉFICE |
|---|---|
| **Backend distant** (Azure Blob, S3, HCP) | Plus de fichier sur un poste |
| **Versionnage activé** sur le backend | Rollback en 1 clic |
| **State locking** | Pas de corruption concurrente |
| **Backups / snapshots** réguliers | Filet de sécurité supplémentaire |
| **State jamais dans Git** | Pas de fuite de secrets, pas de conflit |

💡 **5 minutes** de configuration backend = des heures de réimport évitées.

# Arbre de décision

```
State perdu ?
 ├─ Backup local (.tfstate.backup) ?  → restaurer ✅
 ├─ Backend distant versionné ?       → rollback version ✅
 ├─ Snapshot / autre copie ?          → restaurer ✅
 └─ Rien du tout ?
      └─ Reconstruire par import
           ├─ Azure  → aztfexport
           ├─ Multi  → Terraformer / blocs import
           └─ puis : plan → ajuster → plan vide ✅
```

# En résumé : State perdu

- L'infra **survit** à la perte du state
- Toujours **chercher un backup d'abord** (local, puis backend versionné)
- **Reconstruire par import** en dernier recours · valider jusqu'au **plan vide**
- **Backend distant + versionnage** = la meilleure assurance

ANNEXE D

# Préparation certification 004

HashiCorp Terraform Associate (004)

# Les certifications Terraform disponibles

| | ASSOCIATE (004) | AUTHORING & OPERATIONS PRO |
|---|---|---|
| Niveau | Fondamentaux | Professionnel |
| Format | QCM | **Labs pratiques** + QCM |
| Durée | **1 h** | **4 h** (pause 15 min incluse) |
| Passage | En ligne surveillé | En ligne surveillé |
| Version testée | Terraform **1.12** | Terraform **1.6** |
| Prix | **70,50 $** US | **295 $** US (retake inclus) |
| Validité | 2 ans | 2 ans |
| Prérequis | Bases terminal & cloud | **Associate** recommandé · HCL/CLI avancé |

## Associate — objectifs

IaC, fondamentaux, workflow `init/plan/apply`, config (resources/data), modules, state & backends, HCP Terraform.

## Professional — objectifs

Cycle de vie des ressources, config dynamique, workflows collaboratifs, **création/maintenance de modules**, providers, workspaces HCP Terraform.

Source : developer.hashicorp.com/certifications/infrastructure-automation — tarifs indicatifs hors taxes.

# L'examen 004 en bref

## Format

- ≈ **57 questions** · **1 heure**
- En ligne, **surveillé** (webcam)
- QCM, réponses multiples, vrai/faux
- Validité **2 ans**

## Spécificités 004

- Évalué sur **Terraform 1.12**
- Terminologie **HCP Terraform**
- Nouveaux blocs : `import`, `moved`, `removed`, `check`
- `terraform test`, ressources **ephemeral**

🎓 **Réviser :** `developer.hashicorp.com/terraform/tutorials/certification-004` — Exam Content List + Study Guide + sample questions.

# Votre examen blanc : 50 questions, 1 heure

## Fourni par le formateur, en PDF

- 50 QCM alignés sur les domaines officiels, **une question par page**
- Corrigé **détaillé** dans un PDF séparé, une justification par question
- À passer après la formation, au calme

## Mode d'emploi — comme le jour J

1. **1 heure chrono**, seul, sans doc
2. Réponses sur papier : n° → lettre
3. Corrigé **ensuite** seulement
4. Cible : **≥ 70 %** — et chaque erreur comprise

💡 **Terrain :** le vrai examen pardonne peu les questions « quelle commande / quel bloc » : si vous hésitez entre `taint` et `-replace`, entre `moved` et `state mv`, c'est là que se joue la marge. Refaites les questions ratées une semaine plus tard.

# Les 9 domaines d'objectifs

| # | DOMAINE | MODULES |
|---|---|---|
| 1 | Concepts de l'IaC | 01 |
| 2 | Finalité de Terraform | 01 · 02 |
| 3 | Fondamentaux de Terraform | 02 · 03 · 04 |
| 4 | CLI hors workflow principal | 06 · 08 · 09 |
| 5 | Interagir avec les modules | 07 |
| 6 | Workflow principal | 02 · 04 |
| 7 | Implémenter & maintenir le state | 06 |
| 8 | Lire, générer, modifier la config | 04 · 05 · 08 |
| 9 | Capacités de HCP Terraform | Annexe D |

# Commandes CLI à maîtriser

| COMMANDE | RÔLE |
|---|---|
| `init` | Providers, backend, modules |
| `plan` / `apply` / `destroy` | Workflow principal |
| `fmt` / `validate` | Qualité (validate **hors-ligne**) |
| `state list/show/mv/rm` | Manipulation du state |
| `import` | Rattacher l'existant |
| `output` / `console` | Sorties / évaluer une expression |
| `taint` / `-replace` | Forcer une recréation |
| `workspace` | États multiples |

# HCP Terraform (domaine 9)

- **Workspaces distants** : state géré et chiffré côté serveur
- **Remote runs** : `plan` / `apply` exécutés dans le cloud
- Intégration **VCS** (déclenchement sur push)
- **Private Registry** de modules/providers
- **Sentinel / OPA** : **policy as code** (garde-fous avant `apply`)

> 💡 **Terrain :** HCP Terraform évite de maintenir un backend « maison ». Alternative fréquente : GitHub/GitLab Actions + backend S3/GCS + OIDC, avec policies (Conftest/OPA) en pipeline.

# Pièges fréquents à l'examen

- Croire que `plan` modifie l'infra → il est **en lecture seule**
- Croire que `validate` contacte le cloud → **hors-ligne**
- Confondre `state rm` (oublie) et `destroy` (supprime)
- Oublier l'**ordre de précédence** des variables
- Mélanger `count.index` et `each.key`
- Oublier que le **state** peut exposer des secrets
- Confondre **provisioning** (Terraform) et **configuration** (Ansible)

> 🎓 **Réflexe :** en cas de doute, demandez-vous « qu'écrirait Terraform dans le **plan** ? ».

# Merci !

Des questions ?

**Formation Terraform** — Retengr · version 1.1.1