OpenTofu
Le fork open source de Terraform

Atelier technique DevSecOps – alternance cours & TP

Thierry Gayet
thierry.gayet@gmail.com

Association LinuxMaine

25 mai 2026

Plan de la formation

Partie I — Concepts

- Dépôt Git & accès au Lab
- Introduction au Provisioning
- OpenTofu — présentation et rôle
- Comparatif Terraform vs OpenTofu
- Packer et alternatives open source

Partie II — Langage HCL

- HCL — le langage en profondeur
- Langage HCL — couverture complète

Partie III — Ateliers (TPs)

- Atelier — prise en main (TP01-02)
- Atelier — syntaxe & fonctionnalités (TP03-11)
- Exclusivités OpenTofu (TP12-13)
- Cloud Provisioning (TP27-34)
- CI/CD et tests (TP14-22)
- Packer / alternatives (TP23-26)

Partie IV — Cloud & Workflow

- Multi-cloud : exemples déploiement
- OpenTofu + Ansible : workflow combiné
- Qualité, sécurité et performance

Partie V — Annexes

- Lexique technique français
- Ouvrages recommandés FR & EN
- Refcards CI/CD (`.gitlab-ci.yml`, pre-commit)
- Liens officiels & chaines YouTube
- Refcard rapide OpenTofu

Prérequis

Linux, Git, notion de cloud. **Pas** de prérequis Terraform / Ansible.

Conventions de couleur

Bleu = cours Orange = TP / exercice Vert = astuce

Plan détaillé I

Dépôt Git & Accès au Lab

Cloner le dépôt

```
git clone https://framagit.org/linuxmaine/formations.git
cd formations/devsecops/opentofu/
```

Démarrer le lab

- `./opentofu-start.sh` — démarre la VM Debian 12
- `./opentofu-status.sh` — vérifie l'état
- `./opentofu-ssh.sh` — se connecte (no creds)
- `./opentofu-stop.sh [-destroy]`

Accès VM

- SSH : `ssh -i ~/.ssh/opentofu_ed25519 root@<ip>`
- Clé SSH générée automatiquement
- Image : Debian 12 bookworm (genericcloud)
- Provider IaC : libvirt (dmacvicar/libvirt)

Documentation locale du dépôt

Tout est dans le repo !

Le dépôt embarque sa propre documentation technique en français, consultable hors-ligne (HTML statique).

`labs/hcl/` — Doc HCL technique

- 11 pages HTML (index + 10 chapitres)
- Spec HCL native + JSON + types cty
- Expressions, templates, fonctions stdlib
- Bonnes pratiques
- Ouvrir : `xdg-open labs/hcl/index.html`

`ebooks/` — PDFs officiels

- `HCL-Language-Specification-Official.pdf` (38 p.)
- `OpenTofu Language Documentation.pdf`
- `OpenTofu Errors -Cheatsheet.pdf`

`labs/opentofu/` — Doc OpenTofu

- 20+ pages HTML (langage + CLI)
- Resources, providers, modules, state
- Exclusivités : encryption, early eval
- Toutes les commandes CLI
- Ouvrir : `xdg-open labs/opentofu/index.html`

À la racine du repo

- `Readme.opentofu.install.md` — 13 méthodes d'install
- `Readme.opentofu.refcard.md` — toutes les commandes
- `devsecops-opentofu.pdf` — ces slides

Qu'est-ce que le Provisioning ?

- Processus d'**installation et de configuration automatisée** d'un système
- Infrastructure as Code (IaC) : l'infrastructure décrite dans des fichiers versionnés
- **Idempotence** : exécuter N fois = même résultat qu'une seule fois
- Deux approches : **impérative** (comment faire) vs **déclarative** (quoi obtenir)
- Deux modèles : **Push** (le serveur envoie) vs **Pull** (le client récupère)

Objectif

Remplacer les opérations manuelles par du code reproductible, versionné et testable.

Pourquoi automatiser le provisioning ?

- Le provisioning manuel est **lent** et source d'**erreurs humaines**
- Garantir la **cohérence** entre les environnements (dev, staging, prod)
- **Reproductibilité** : recréer un environnement identique à tout moment
- **Scalabilité** : provisionner 1 ou 100 serveurs avec le même effort
- **Documentation as Code** : l'infra est décrite dans des fichiers lisibles
- **Piste d'audit** complète grâce à l'historique Git (qui, quoi, quand)

Infrastructure as Code — Les principes

- Décrire l'**état désiré** (desired state) plutôt que les étapes à exécuter
- **Versionner** le code d'infrastructure dans Git
- **Revue de code** via Pull Requests avant tout changement
- **Tester** l'infrastructure (lint, dry-run, tests d'intégration)
- **Idempotence** : appliquer N fois = même résultat qu'une seule fois
- Infrastructure **immutable** vs **mutable** : remplacer plutôt que modifier

IaC — Les bénéfices concrets

Sans IaC

- Connexions SSH manuelles serveur par serveur
- Documentation Word obsolète dès le lendemain
- Serveurs “snowflake” tous différents
- “Ca marche sur ma machine” en production
- Rollback = espérer et prier

Avec IaC

- Infrastructure reproductible à 100%
- Code versionné dans Git (historique complet)
- Testable : lint, dry-run, tests d’intégration
- Auditée : chaque changement est tracé
- Rollback = `git revert` + apply

Impératif vs Déclaratif

Impératif (comment faire)

- Scripts Shell / Bash
- Ansible ad-hoc (commandes ponctuelles)
- Chef recipes (Ruby procédural)
- On décrit les étapes séquentielles
- Ordre d'exécution important

Déclaratif (quoi obtenir)

- Terraform / OpenTofu (HCL)
- Puppet manifests (DSL déclaratif)
- Ansible playbooks (YAML déclaratif)
- Salt states (YAML déclaratif)
- Le moteur calcule les actions nécessaires

Push vs Pull

Push (le contrôleur pousse)

- Ansible : connexion SSH depuis le controller
- Terraform : appel API vers les providers
- Pas d'agent à installer sur les cibles
- Exécution à la demande (on-demand)
- Idéal pour les déploiements ponctuels

Pull (les noeuds tirent)

- Puppet agent : interroge le Puppet Server
- Chef client : interroge le Chef Server
- Salt minion : écoute le Salt Master
- Agent installé sur chaque noeud
- Exécution périodique (convergence)

Historique du Provisioning

- **2005** : Puppet — premier outil moderne de gestion de configuration
- **2009** : Chef — automatisation en Ruby DSL
- **2011** : SaltStack — orchestration haute performance (ZeroMQ)
- **2012** : Ansible — simplicité, sans agent, YAML
- **2014** : Terraform — IaC multi-cloud (HCL)
- **2019** : Pulumi — IaC en langages standards (Python, Go, TS)
- **2023** : OpenTofu — fork open source de Terraform (Linux Foundation)

Tendance : de l'impératif (scripts) vers le déclaratif (IaC), du pull vers le push

Panorama des outils

Avec agent (Pull)

- **Puppet** — DSL propre, entreprise, mature
- **Chef** — Ruby DSL, flexible, complexe
- **SaltStack** — YAML, rapide (ZeroMQ), push+pull
- **CFEngine** — C, très performant, bas niveau

Sans agent (Push)

- **Ansible** — YAML, SSH, simple, très populaire
- **Terraform** — HCL, infra cloud, multi-provider (BSL)
- **OpenTofu** — Fork open source de Terraform (MPL 2.0)
- **Pulumi** — Langages standards (Python, Go, TS)

DevSecOps — Rappel

- Extension de DevOps : **Security** intégrée à chaque étape
- **Shift-Left** : la sécurité commence dès le code, pas en fin de cycle
- Automatisation des contrôles de sécurité dans le pipeline CI/CD
- Outils clés : SAST, SCA, DAST, scanning d'images, SBOM
- Le provisioning sécurisé est un pilier du DevSecOps

Lien avec le provisioning

Chaque outil de provisioning doit être intégré dans une démarche DevSecOps : lint du code IaC, scanning de sécurité, audit des configurations.

OpenTofu : les “murs” de l’infrastructure

Analogie

OpenTofu construit **les murs** (le bâti, les fondations) : serveurs, réseaux, disques, IP, DNS, IAM, secrets stores...
Les outils de **configuration** (Ansible, Puppet, Chef...) aménagent **l’intérieur** (paquets, fichiers, services).

- **OpenTofu / Terraform** : provisioning *declaratif* de ressources **cloud / API**
- **Packer** : production d’**images** (AMI, qcow2, OCI) *baked-in*
- **Ansible / Puppet / Chef / Salt** : configuration **intra-VM** après boot
- **Kubernetes / Helm / ArgoCD** : applications **sur** l’infrastructure

En une phrase

OpenTofu **crée et raccorde les ressources**. Les autres outils **configurent leur contenu**.

Présentation d'OpenTofu

- **Fork communautaire** de Terraform, lancé en **septembre 2023**
- Hébergé par la **Linux Foundation** : gouvernance ouverte
- Licence **MPL 2.0** (vs BSL 1.1 de Terraform)
- **150+** signataires du manifeste OpenTofu
- CLI `tofu` : *drop-in* de `terraform` pour les versions ≤ 1.5
- Compatibilité des fichiers `.tf` et des providers

Pérennité

OpenTofu garantit la disponibilité d'un IaC multi-cloud sous licence libre, audité par la communauté Linux Foundation.

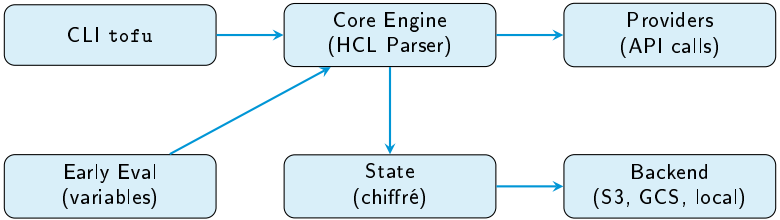
Documentation locale

Doc technique française dans `labs/opentofu/` (20+ pages HTML) — consultable hors-ligne.

Historique

- **Août 2023** : HashiCorp passe Terraform de MPL 2.0 à **BSL 1.1**
- **Septembre 2023** : Manifeste OpenTofu, 150+ signatures
- **Octobre 2023** : Le projet rejoint la **Linux Foundation**
- **Janvier 2024** : **v1.6.0** — première version stable
- **Mai 2024** : **v1.7.0** — *State Encryption*, *Early Variable Evaluation*
- **Sept. 2024** : **v1.8.0** — enrichissement des providers
- **Déc. 2024** : **v1.9.0** — améliorations performance
- **2025** : **v1.10 à v1.12** — divergence progressive, *tofu test* natif

Architecture



- **State Encryption** : chiffrement natif du fichier d'état (exclusif OpenTofu)
- **Early Variable Evaluation** : variables dans `terraform{}` (exclusif)
- **Registry** : `registry.opentofu.org` (mirror des providers Terraform + propres)

Terraform vs OpenTofu — Comparatif détaillé

Critère	OpenTofu	Terraform (HashiCorp)
Licence	MPL 2.0 (libre)	BSL 1.1 (source-available)
Gouvernance	Linux Foundation	HashiCorp / IBM
Première release	Janvier 2024	Juillet 2014
Compatibilité HCL	TF 1.5.x à 100%	—
Registry	registry.opentofu.org	registry.terraform.io
State encryption	Natif (depuis 1.7)	Non
Early var. eval	Oui (1.7)	Non
`tofu test` / `terraform test`	Natif	Natif (depuis 1.6)
Bloc `import`	Oui (génération .tf)	Oui
Bloc `removed`	Oui (1.7)	Oui (1.7)
Provider protocol	v5+v6	v5+v6
Workspaces	Identique	Identique
Cloud (SaaS)	Spacelift, env0...	Terraform Cloud / Enterprise
Communauté	Linux Foundation	HashiCorp + écosystème

En pratique

Pour un projet **nouveau** : OpenTofu (libre, fonctionnalités exclusives).
Pour un projet **existant** en Terraform $\leq$ 1.5 : migration triviale.

Provisioning IaC — comparatif des outils

Outil	Langage	Approche	Multi-cloud	Licence	Note
OpenTofu	HCL	Déclarative	Oui (providers)	MPL 2.0	Fork libre TF
Terraform	HCL	Déclarative	Oui	BSL 1.1	Référence
Pulumi	Python/Go/TS	Déclarative	Oui	Apache 2.0	Code général
AWS CloudFormation	YAML/JSON	Déclarative	AWS only	Propriétaire	Natif AWS
AWS CDK	TS/Python...	Impérative	AWS only	Apache 2.0	Synth en CFN
GCP Deployment Mgr	YAML/Jinja	Déclarative	GCP only	Propriétaire	Déprécié
Azure ARM/Bicep	Bicep/JSON	Déclarative	Azure only	MIT	Natif Azure
Crossplane	YAML K8s	Déclarative	Oui	Apache 2.0	K8s-native
Ansible	YAML	Impérative	Oui (modules)	GPLv3	Push, sans agent

Choix

HCL (TF/OpenTofu) reste le standard de facto, large écosystème de providers. **Pulumi** convient si l'équipe veut du Python/Go pur. **Crossplane**, pour une approche 100% Kubernetes.

Packer — Construire des images, pas des VMs

- **HashiCorp Packer** : outil de *baking* d'images machines
- Produit des artefacts : AMI (AWS), images GCP/Azure/OVH, qcow2 libvirt, OCI Docker...
- Workflow : `packer init` → `packer build template.pkr.hcl`
- Licence **BSL 1.1** (depuis aout 2023), même histoire que Terraform
- Complémentaire d'OpenTofu : **Packer cuit l'image** ⇒ **OpenTofu la déploie**

Workflow type

`Packer build` (Debian + nginx + app) → image AMI → `OpenTofu apply` (VPC + ASG + LB + R53) → `Ansible playbook` (config dynamique)

Packer — Alternatives open source

- **Pas (encore) de fork open source officiel de Packer**. La communauté OpenTofu/Linux Foundation **n'a pas** forké Packer (pour l'instant).
- Alternatives :
  - **diskimage-builder** (OpenStack, Apache 2.0) — production d'images cloud Linux
  - **virt-builder** / **virt-customize** (libguestfs, LGPL) — images qcow2
  - **Kiwi** (openSUSE, GPL) — images Linux multi-format
  - **Earthly** (MPL 2.0) — build d'images container/VM via DSL
  - **Buildah** / **Podman** (Apache 2.0) — pour les images OCI/Docker
  - **nixpkgs** (NixOS) — images déclaratives reproductibles
- **Packer reste open source** dans la pratique (binaire téléchargeable, code lisible), mais sous BSL : usage interne autorisé, redistribution restreinte.

Stratégie

Pour un usage **production interne** : Packer fonctionne. Pour une **redistribution** ou un **produit SaaS**, préférer diskimage-builder / virt-builder.

Packer — Installation (toutes plateformes)

Debian / Ubuntu (APT) :

```
wget -O- https://apt.releases.hashicorp.com/gpg | sudo gpg --dearmor -o /etc/apt/keyrings/hashicorp.gpg
echo "deb [signed-by=/etc/apt/keyrings/hashicorp.gpg] https://apt.releases.hashicorp.com $(lsb_release -cs) main" \
  | sudo tee /etc/apt/sources.list.d/hashicorp.list
sudo apt-get update && sudo apt-get install -y packer
```

Fedora / RHEL (DNF) :

```
sudo dnf install -y dnf-plugins-core
sudo dnf config-manager --add-repo https://rpm.releases.hashicorp.com/fedora/hashicorp.repo
sudo dnf install -y packer
```

Homebrew / macOS / Linux :

```
brew tap hashicorp/tap
brew install hashicorp/tap/packer
```

Standalone (toutes plateformes) :

```
VER=1.11.2; curl -fsSL -o packer.zip \
  "https://releases.hashicorp.com/packer/${VER}/packer_${VER}_linux_amd64.zip"
sudo unzip packer.zip -d /usr/local/bin && packer version
```

Packer — CLI principale

Commandes courantes

- `packer init template.pkr.hcl` — télécharge les plugins
- `packer fmt template.pkr.hcl` — formate
- `packer validate template.pkr.hcl` — valide
- `packer inspect template.pkr.hcl` — inspecte builds/vars
- `packer build template.pkr.hcl` — exécute le build
- `packer version` — version installee

Options utiles

- `-only=src1.src2` — builders cibles
- `-except=src1` — exclusions
- `-var "name=val"` — variable inline
- `-var-file=values.pkrvars.hcl`
- `-debug` — pause après chaque step
- `-on-error=ask|abort|cleanup|run-cleanu`
- `-parallel-builds=N`
- `-force` — ecrase artifact existant

Packer — Structure d'un template HCL

```
# packer { } : declaration des plugins requis
packer {
  required_plugins {
    qemu = {
      source  = "github.com/hashicorp/qemu"
      version = "~> 1.0"
    }
  }
}

# variable : parametres d'entree
variable "iso_url" { type = string }

# source : un BUILDER (cible : qemu, amazon-ebs, googlecompute...)
source "qemu" "debian" {
  iso_url      = var.iso_url
  iso_checksum = "file:./debian-checksums.txt"
  disk_size    = "10G"
  format       = "qcow2"
  accelerator  = "kvm"
  output_directory = "output"
}

# build : enchaine SOURCES + PROVISIONERS + POST-PROCESSORS
build {
  sources = ["source.qemu.debian"]
  provisioner "shell" {
    inline = ["apt-get update", "apt-get install -y nginx"]
  }
}
```

Packer — Builders, Provisioners, Post-processors

Builders

- qemu (qcow2)
- virtualbox-iso
- vmware-iso
- amazon-ebs (AMI)
- googlecompute
- azure-arm
- openstack
- docker
- lxd, proxmox

Provisioners

- shell (script inline / fichier)
- shell-local
- file (upload)
- ansible
- ansible-local
- chef-solo
- puppet-masterless
- salt-masterless
- powershell

Post-processors

- compress
- shell-local
- manifest
- checksum
- vagrant (.box)
- docker-push
- amazon-import
- googlecompute-export

Packer — Script de build complet (qcow2 Debian 12)

```
packer {
  required_plugins {
    qemu = { source = "github.com/hashicorp/qemu", version = "~> 1.0" }
  }
}
variable "iso_url"      { default = "https://.../debian-12-genericcloud-amd64.qcow2" }
variable "iso_checksum" { default = "sha256:..." }
variable "image_name"   { default = "debian12-baked" }
source "qemu" "debian" {
  iso_url          = var.iso_url
  iso_checksum     = var.iso_checksum
  disk_image       = true
  format           = "qcow2"
  accelerator      = "kvm"
  memory           = 2048
  cpus             = 2
  disk_size        = "10G"
  headless         = true
  net_device       = "virtio-net"
  disk_interface   = "virtio"
  ssh_username     = "root"
  ssh_private_key_file = "~/.ssh/packer_ed25519"
  ssh_timeout      = "10m"
  output_directory = "output-${var.image_name}"
  shutdown_command = "shutdown -h now"
}
build {
  name    = "debian-baked"
  sources = ["source.qemu.debian"]
  provisioner "shell" {
    inline = [
      "apt-get update -y",
      "apt-get install -y nginx qemu-guest-agent",
      "systemctl enable nginx qemu-guest-agent",
      "apt-get clean && rm -rf /var/lib/apt/lists/*"
    ]
  }
  post-processor "manifest" { output = "manifest.json" }
}
```

Packer — Build d'une image Docker (OCI)

```
source "docker" "nginx-debian" {
  image       = "debian:12-slim"
  commit      = true
  changes     = [
    "EXPOSE 80",
    "CMD [\"nginx\", \"-g\", \"daemon off;\"]"
  ]
}

build {
  sources = ["source.docker.nginx-debian"]

  provisioner "shell" {
    inline = [
      "apt-get update && apt-get install -y nginx",
      "echo 'Bake by Packer' > /var/www/html/index.html",
      "apt-get clean && rm -rf /var/lib/apt/lists/*"
    ]
  }

  post-processor "docker-tag" {
    repository = "registry.linuxmaine.org/nginx"
    tags       = ["v1", "latest"]
  }
}
```

Workflow CI

`packer init` → `packer validate` → `packer build` → `docker push` → déploiement Kubernetes / Nomad.

Packer + OpenTofu — chaine complète

Pipeline d'un déploiement immuable

1. **packer build** : produit une image (AMI, qcow2, OCI)
2. Le `manifest.json` expose l'ID de l'artefact (`ami-xxxx`)
3. **OpenTofu** lit le manifest via `file()/jsondecode()` et le passe à `aws_instance.ami`
4. **tofu apply** : déploie l'infra avec l'image *bakeée*
5. Pas de provisioner runtime : tout est dans l'image

- **Bénéfice** : boot rapide (image prête), reproductible, scannable
- **Sécurité** : image scannée par `trivy` / `grype` avant publication, signature avec `cosign`
- **Coupling** : Packer connait son artefact, OpenTofu le consomme via tag/manifest, jamais d'effet de bord

TP23 `tp23-packer-qcow2/` — Build d'une image qcow2

Objectif pédagogique

Construire une image Debian 12 *baked-in* avec nginx préinstallé via Packer + builder `qemu`.

- Template `debian.pkr.hcl` : variables, builder qemu, provisioner shell
- `packer init` télécharge le plugin `qemu`
- `packer build` produit `output/packer-debian-baked` (qcow2)
- L'image peut ensuite être utilisée par OpenTofu (TP02 modifié)

Scripts du TP : `tp23-do-run.sh` · `tp23-do-reset.sh` · `tp23-do-check.sh`

TP24 `tp24-packer-docker/` — Build d'une image OCI

Objectif pédagogique

Construire une image Docker nginx *bakeée* via Packer + builder `docker`.

- Builder `docker`, provisioner `shell`, post-processor `docker-tag`
- Idempotence : le `COMMIT` gère le checksum de l'image résultante
- Scan sécurité en post-processor (`trivy`)
- Comparaison avec un `Dockerfile` équivalent (âge d'or de Packer pour OCI ?)

Scripts du TP : `tp24-do-run.sh` · `tp24-do-reset.sh` · `tp24-do-check.sh`

Alternatives à Packer — pourquoi ?

- **Packer est sous BSL 1.1** (même licence que Terraform pré-fork) : restrictions sur la redistribution commerciale
- Pas de fork “OpenPacker” officiel par la Linux Foundation (à ce jour)
- Selon le besoin, des outils spécialisés sont **plus simples**, **plus rapides** ou **vraiment libres** (Apache 2.0, GPL...)
- Deux familles principales : **images VM** (qcow2, raw, AMI) et **images OCI** (Docker)

Comment choisir

Si on a *déja* besoin d'Ansible/shell : préférer Packer.
Si on veut le *strict minimum* : virt-builder ou Buildah.
Si on vise la *reproductibilité bit-à-bit* : Nix.
Si on est OpenStack-centric : diskimage-builder.

Alternatives Packer — Images VM (qcow2, raw, AMI)

Outil	Licence	Approche	Forces	Limites
virt-builder (libguestfs)	LGPL 2.1+	CLI impératif	Très rapide, pas de VM jetable	Pas de DSL ; recette unique
diskimage-builder (DIB)	Apache 2.0	Elements (chroot)	Standard OpenStack ; très modulaire	Courbe d'apprentissage
mkosi (systemd)	LGPL 2.1+	Config TOML	Moderne, partition GPT, support TPM/UKI	Linux-only (systemd)
Kiwi (openSUSE)	GPL 2.0	XML déclaratif	Multi-format (ISO/qcow2/OCI/PXE)	Verbeux
Nix / nixpkgs	MIT	Fonctionnelle déclarative	Reproductibilité parfaite, cache binaire	Apprentissage Nix-lang
image-builder (Fedora)	Apache 2.0	Blueprints TOML	Backend osbuild	Fedora/RHEL-centrique
vmdb2 (Debian)	GPL 3	YAML déclaratif	Pur Debian, scriptable	Surtout ARM/embedded

Recommandation rapide

virt-builder pour un qcow2 rapide. **mkosi** pour un setup moderne systemd. **DIB** en environnement OpenStack. **Nix** si on veut la reproductibilité bit-à-bit.

Alternatives Packer — Images OCI (Docker)

Outil	Licence	Approche	Forces	Limites
Buildah (Red Hat)	Apache 2.0	Daemonless, CLI ou shell	Rootless, scriptable, OCI standard	Linux-only
kaniko (Google)	Apache 2.0	Dans un container	Build sans Docker daemon (K8s/CI)	Cache limité
img	MIT	Daemonless, rootless	Léger, basé sur BuildKit	Déprécié en faveur de buildctl
BuildKit (Docker)	Apache 2.0	API backend	Cache avancé, multi-arch	Daemon ou frontend (buildctl)
Earthly	MPL 2.0 (libre)	DSL déclaratif	Combine Make+Docker+CI, repeatable	Outil tiers
ko (Google)	Apache 2.0	Spécifique Go	Builds Go ultra-rapides, distroless	Go uniquement
Jib (Google)	Apache 2.0	Spécifique JVM	Builds Java reproductibles	JVM uniquement
Nix + dockerTools	MIT	Fonctionnelle	Image minimale, reproductible	Apprentissage

Quand quoi

Buildah/Podman en mainstream Linux. **kaniko** en CI Kubernetes (pas de Docker dans le pod). **ko/Jib** pour des binaires unilangues. **Earthly** si l'on veut un DSL.

virt-builder — Exemple (qcow2 Debian)

```
sudo apt install -y libguestfs-tools  # Debian/Ubuntu

# Lister les templates disponibles
virt-builder --list | grep debian

# Construire une image Debian 12 avec nginx, en 1 commande :
virt-builder debian-12 \
  --size 10G \
  --format qcow2 \
  --hostname web01 \
  --root-password password:linuxmaine \
  --ssh-inject root:file:~/.ssh/id_rsa.pub \
  --install nginx,qemu-guest-agent \
  --run-command 'systemctl enable nginx qemu-guest-agent' \
  --run-command 'echo "<h1>virt-builder</h1>" > /var/www/html/index.html' \
  --output debian12-baked.qcow2
```

- **Pas de VM jetable** : monte l'image en local et la modifie via libguestfs
- **Très rapide** : 1 min vs 15 min pour Packer
- Templates upstream signés (GPG) et vérifiés automatiquement

Buildah — Exemple (image OCI nginx)

```
sudo apt install -y buildah     # Debian/Ubuntu

# Workflow en shell pur (pas de Dockerfile)
ctr=$(buildah from debian:12-slim)
buildah run    $ctr -- apt-get update
buildah run    $ctr -- apt-get install -y nginx
buildah run    $ctr -- bash -c 'echo "<h1>Buildah</h1>" > /var/www/html/index.html'
buildah config --port 80 \
               --workingdir /var/www/html \
               --entrypoint '["/usr/sbin/nginx"]' \
               --cmd '["-g", "daemon off;"]' \
               $ctr
buildah commit $ctr linuxmaine/nginx:buildah
buildah images | grep buildah

# Compatible Dockerfile aussi :
buildah bud -t linuxmaine/nginx:bud -f Dockerfile .
```

- **Daemonless, rootless** : pas besoin de socket Docker, pas d'élévation
- **Compatible OCI** : push sur n'importe quel registry Docker/OCI
- **Bien adapté en CI** (`kaniko`-like sans être dans K8s)

TP25 `tp25-virt-builder/` — qcow2 sans VM jetable

Objectif pédagogique

Construire une image Debian 12 + nginx **sans VM de build** avec virt-builder, et la comparer au résultat du TP23 (Packer + builder qemu).

- Outil : `virt-builder` (libguestfs-tools)
- Temps de build : $\leq$ **2 min** (contre 15-20 min pour Packer)
- Verif post-build : `virt-cat`, `virt-ls` (inspection sans boot)
- Sortie : `output/debian12-vbuild.qcow2` + manifest JSON

Scripts du TP : `tp25-do-run.sh` · `tp25-do-reset.sh` · `tp25-do-check.sh`

TP26 `tp26-buildah/` — Image OCI daemonless

Objectif pédagogique

Construire une image OCI nginx **sans Docker daemon** via Buildah, en shell pur, et la comparer au résultat du TP24 (Packer + builder docker).

- Outil : `buildah` (Red Hat, Apache 2.0)
- Approche : commandes shell + `buildah commit`
- Pas de privilège root (podman-compatible)
- Sortie : `linuxmaine/nginx:buildah` importable par Docker ou Podman

Scripts du TP : `tp26-do-run.sh` · `tp26-do-reset.sh` · `tp26-do-check.sh`

Prérequis de la formation

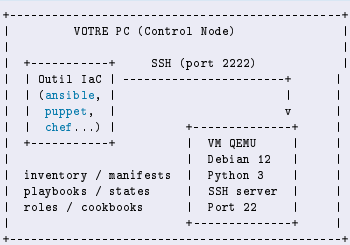
Matériel & Système

- PC avec **Linux** (Debian/Ubuntu ou Fedora)
- Minimum **4 Go de RAM** (2 Go pour la VM)
- **10 Go** d'espace disque libre
- Processeur supportant la virtualisation (**VT-x** / AMD-V)
- Accès Internet pour télécharger les images

Logiciels requis

- **QEMU/KVM** — émulateur et hyperviseur
- **cloud-image-utils** — générer le seed cloud-init
- **OpenSSH client** — connexion SSH
- **Python 3.10+** — requis par les outils modernes
- **Éditeur** : VS Code (extensions IaC) ou vim/nano
- **Git** — versionner le code d'infrastructure

Architecture du Lab



Principe

Un seul PC suffit : tout tourne en local. La VM simule un serveur distant.

QEMU — C'est quoi ?

Présentation

- **Quick EMUlator** : émulateur et virtualiseur open source
- Émule une machine complète (CPU, RAM, disque, réseau)
- Avec **KVM** : performances quasi-natives (accélération matérielle)
- Utilisé par : **libvirt**, Proxmox, OpenStack, Android Studio
- Léger : pas besoin de VirtualBox ou VMware

Concepts clés

- **qcow2** : format natif QEMU (snapshots, compression)
- **raw** : format brut (rapide, mais gros fichier)
- **qemu-img** : créer/convertir/redimensionner des images
- **Mode user** : NAT simple, pas de config réseau
- **hostfwd** : `tcp::2222-:22` redirige le port local

Pourquoi QEMU ? Léger, en ligne de commande, scriptable, idéal pour les labs automatisés.

Cloud-init — Configuration initiale des VMs

Présentation

- Outil **standard** de configuration initiale des VMs cloud
- Utilisé par AWS, GCP, Azure, OpenStack, Proxmox, QEMU
- Configure au **premier boot** : hostname, users, SSH keys, paquets
- Fichier `user-data` en YAML
- Seed ISO : `cloud-localds seed.img cloud-init.yml`

Exemple cloud-init.yml

```
#cloud-config
hostname: lab-vm
users:
  - name: deploy
    sudo: ALL=(ALL) NOPASSWD:ALL
    ssh_authorized_keys:
      - ssh-ed25519 AAAA... user@host
packages:
  - python3
  - curl
  - vim
```

À retenir — Avant les TP 0 et 1

Chemin d'installation

1. Installer QEMU + cloud-localds
2. Générer une clé SSH ed25519
3. Démarrer la VM avec `tp00-setup.sh`
4. Attendre que cloud-init termine (60s)
5. Tester en SSH puis installer Ansible

Checks après TP 0 et 1

- `ssh -p 2222 root@127.0.0.1 hostname` → `debian-lab`
- `ansible -version` → core 2.x
- `ansible lab -m ping` → `pong`

Démarrage du Lab — TP00 labs/tp00/

Objectif

Préparer une VM Debian 12 avec libvirt/QEMU et cloud-init. Atelier **autonome** : aucune dépendance avec l'atelier Ansible.

Script tout-en-un : labs/tp00/tp00-setup.sh

```
$ cd labs/tp00/
$ ./tp00-setup.sh --bg        # demarre la VM (libvirt + cloud-init)
$ ./tp00-setup.sh --status    # etat (IP, SSH)
$ ./tp00-setup.sh --ssh       # shell SSH sans creds
$ ./tp00-setup.sh --stop      # arret propre (images preservees)
$ ./tp00-setup.sh --reset     # arret + suppression disque/seed
$ ./tp00-setup.sh --purge     # tout supprimer (y compris image base)
$ ./tp00-setup.sh --help      # aide

# Equivalent direct depuis la racine du repo :
$ ./opentofu-start.sh ; ./opentofu-status.sh ; ./opentofu-ssh.sh
```

Ce qui est automatisé

1. Vérif prérequis (virsh, qemu-img, genisoimage)
2. Téléchargement image Debian 12 cloud
3. Génération clé SSH ed25519 (si absente)
4. Création disque overlay qcow2 (15 Go)
5. Seed ISO cloud-init (user-data + network)
6. Lancement VM + attente DHCP + SSH OK

Images stockées dans labs/tp00/ (gitignorées).

Installation d'OpenTofu (toutes plateformes)

Debian / Ubuntu :

```
curl -fsSL https://get.opentofu.org/install-opentofu.sh | sudo bash -s -- --install-method deb
tofu --version
```

Fedora / RHEL :

```
curl -fsSL https://get.opentofu.org/install-opentofu.sh | sudo bash -s -- --install-method rpm
```

macOS / Linux (Homebrew) :

```
brew install opentofu
```

Multi-version (asdf / tenv) :

```
asdf plugin add opentofu && asdf install opentofu 1.10.6 && asdf global opentofu 1.10.6
# ou : tenv tofu install latest
```

Vérification

`tofu -version` → OpenTofu v1.10.x ou supérieur. *Toutes les méthodes sont détaillées dans `Readme.opentofu.install.md`.*

TP01 tp01-installation/ — Installation et vérification

Objectif pédagogique

Installer OpenTofu et valider le binaire ; préparer libvirt pour la suite des TP.

- Script `tp01-installation.sh` : détecte l'OS, installe le paquet, configure libvirt, exécute un mini-projet HCL pour valider
- Sortie : `tofu -version` affiche v1.10+ ; libvirt accessible

Cible après ce TP

Pouvoir lancer `tofu init / plan / apply` sur un projet **local** (provider `local`, ressources `local_file`).

Scripts du TP : `tp01-do-run.sh` · `tp01-do-reset.sh` · `tp01-do-check.sh`

Premier projet — structure des fichiers

Un projet OpenTofu, c'est un **répertoire** avec un ou plusieurs fichiers `.tf` :

```
projet/
|-- main.tf            # ressources et providers
|-- variables.tf       # declarations de variables
|-- outputs.tf         # sorties
|-- terraform.tfvars   # valeurs de variables (auto-loaded)
|-- versions.tf        # required_version, required_providers
`-- .terraform.lock.hcl  # lockfile providers (commit !)
```

- Tous les `.tf` du répertoire courant sont **fusionnés**
- Les fichiers `*.auto.tfvars` sont chargés automatiquement
- Les sous-répertoires ne sont pas inclus, sauf via `module {}`
- Le bloc `terraform{}` configure la version requise et le backend

Syntaxe HCL — blocs et arguments

```
# Un BLOC : type "nom_local" "identifiant" { ... }
resource "libvirt_domain" "vm" {     # type, label1, label2
  name   = "demo-vm"                 # argument (=)
  memory = 1024                      # nombre
  vcpu   = 2

  disk {                             # bloc imbriqu\'e
    volume_id = libvirt_volume.disk.id
  }

  network_interface {                # second bloc imbriqu\'e
    network_name = "default"
  }
}
```

- **Identifiants** : `type.label_local.attribut` (`libvirt_domain.vm.id`)
- **Types** : string, number, bool, list, set, map, object, tuple
- **Commentaires** : # ou // (mono-ligne), /* */ (multi)

TP02 `tp02-premier-projet/` — Première VM Debian

Objectif pédagogique

Créer une VM Debian via le provider libvirt ; découvrir `terraform{}`, `provider{}`, `resource{}`, variables et outputs.

Ce qui sera appris :

- Structure d'un projet `.tf`
- Workflow `init` → `plan` → `apply` → `destroy`
- Déclaration de variables et validations
- Références croisées entre ressources

Fichiers spécifiques : main.tf, variables.tf, outputs.tf

Scripts du TP : `tp02-do-run.sh` · `tp02-do-reset.sh` · `tp02-do-check.sh`

HCL — HashiCorp Configuration Language

- **Créé par HashiCorp** en 2014 pour Terraform
- Objectif : un *format de configuration* à la fois **lisible par l'humain** et **structuré pour la machine**
- Compromis entre JSON (machine) et YAML (humain) : moins ambigu, plus déclaratif
- **Spécification ouverte** : `https://github.com/hashicorp/hcl` (Mozilla Public License 2.0)
- Utilisé par tout l'écosystème *infrastructure-as-code* :

- **Terraform / OpenTofu** : provisionning IaC
- **Packer** : baking d'images
- **Vault / OpenBao** : policies de secrets
- **Consul** : configuration de services
- **Nomad** : descriptions de jobs
- **Waypoint** : pipelines de déploiement

Références disponibles dans le repo

- `labs/hcl/index.html` — Doc HCL technique en français (11 chapitres)
- `ebooks/HCL-Language-Specification-Official.pdf` (38 p.) — spec officielle

HCL vs Terraform/OpenTofu — ne pas confondre !

Distinction fondamentale

HCL définit une **forme d'écriture** : blocs, arguments, collections, chaînes, templates, expressions. Le langage est *agnostique au domaine*.

Les blocs `resource`, `provider`, `module`, `variable`, `terraform{}`... appartiennent au **schéma Terraform/OpenTofu**, pas au langage HCL lui-même.

Ce que HCL définit

- Syntaxe lexicale (tokens, identifiants)
- Structure : blocs, attributs, body
- Littéraux : number, string, bool, null
- Expressions et opérateurs
- Templates et interpolation
- Système de types `cty`
- Fonctions standards (stdlib)

Ce que OpenTofu/TF ajoute

- Schéma : types de blocs autorisés (`resource`, `variable`...)
- Sémantique : comportement de chaque bloc
- Providers : extensions externes
- Cycle de vie : `plan` / `apply` / `destroy`
- State : tracking de l'infra créée
- Fonctions métier (`templatefile`, `cidrsubnet`)

Analogie

HCL est à OpenTofu ce que JSON est à Kubernetes. JSON définit la syntaxe ; Kubernetes ajoute le schéma (Pod, Service, Deployment...) et la sémantique. Packer, Vault, Nomad, Consul utilisent *le même HCL* mais chacun avec son propre schéma.

HCL1 vs HCL2 — une refonte complète

HCL1 (2014–2018)

- Format `key = value` basique
- Interpolation *string-only* : `"${var.x}"`
- **Pas** d'expressions, pas de boucles, pas de conditional
- Pas de type system formel (tout est string)
- Pas d'équivalence JSON officielle
- Utilisé par Terraform 0.6 à 0.11
- Packer JSON-only à l'époque (`.json`)

HCL2 (2018+)

- Refonte complète : parser, AST, évaluation
- Expressions *first-class* (n'importe où)
- `for`, `dynamic`, conditional, splat, locals
- Système de types statique `cty`
- Équivalence JSON native (`.tf.json`)
- Templates avancés (`%{if/for}`)
- Adopté Terraform 0.12 (2019), Packer 1.5 (2020)

Aujourd'hui

Tout le monde est en HCL2. Quand on dit « HCL», on parle d'HCL2. HCL1 est mort en même temps que Terraform 0.11 (fin de support 2020).

HCL1 vs HCL2 — exemples comparatifs

HCL1 (Terraform 0.11) — limité

```
resource "aws_instance" "web" {
  count         = "${var.enable ? 1 : 0}"     # interpolation forcee
  ami           = "${var.ami}"                # tout entre "${...}"
  instance_type = "${lookup(var.sizes, var.env)}"  # acces map awkward
  # Pas de for_each, pas de dynamic block, pas de splat moderne
}
```

HCL2 (Terraform 0.12+, OpenTofu, Packer 1.5+) — expressif

```
resource "aws_instance" "web" {
  for_each      = var.servers              # for_each sur map / set
  ami           = var.ami                  # plus de "${}" obligatoire
  instance_type = var.sizes[var.env]       # acces map natif
  tags          = merge(local.tags, { Name = each.key })

  dynamic "ebs_block_device" {             # bloc dynamique
    for_each = each.value.disks
    content {
      device_name = ebs_block_device.value.name
      volume_size = ebs_block_device.value.size
    }
  }
}
```

HCL2 — qui l'utilise aujourd'hui ?

Réponse courte

Tout l'écosystème HashiCorp moderne, plus OpenTofu.

Adoption définitive

- **Terraform** 0.12+ (juin 2019)
- **OpenTofu** : 100% HCL2 dès 1.6.0 (hérité du fork)
- **Packer** 1.5+ (décembre 2019)
- **Vault** : policies en HCL2
- **Consul** : configuration en HCL2
- **Nomad** 1.0+ : jobs en HCL2
- **Waypoint** : pipelines en HCL2

Spécificités par produit

- **Packer** : supporte encore `.json` (legacy) mais le format moderne est `.pkr.hcl` (HCL2)
- **OpenTofu** ajoute des extensions *exclusives* (early eval, encryption, ephemerality)
- **Nomad** : ses jobs HCL2 supportent `multiline` moins riches que TF
- **Vault** policies : sous-ensemble (pas de variables, pas d'expressions complexes)

Règle

Si vous écrivez du HCL pour un produit récent ($\geq$ 2020), c'est du HCL2.

HCL vs autres formats de configuration

Critère	HCL	YAML	JSON	TOML
Cible	**IaC + config**	**Config + déploiement**	**API/données**	**Config simple**
Lisibilité	**Bonne**	**Bonne (indentation)**	**Moyenne**	**Bonne**
Commentaires	**#, //, /* */**	**#**	**Aucun (officiel)**	**#**
Strings multi-l.	**«EOT heredoc**	**| block**	**Non**	**Triple quote**
Types complexes	**Riche (any, object, tuple)**	**Implicite**	**Très basique**	**Limité**
Templates / interp.	**${var.x} natif**	**Via outils tiers**	**Non**	**Non**
Boucles / conditions	**for, if, dynamic**	**Non**	**Non**	**Non**
Schema typing	**Statique cty**	**Aucun**	**JSON Schema**	**Aucun**
Ambigüité indentation	***N/A* (blocs)**	**Problématique (YAML 1.1/1.2)**	**N/A**	**N/A**
Équivalent JSON	**Oui (.tf.json)**	**Oui**	**–**	**Approximatif**

Force d'HCL

Le seul format **de configuration** qui supporte nativement les **expressions**, **templates**, **boucles** et un **système de types statique**, tout en restant lisible. C'est ce qui le rend dominant en IaC.

HCL — Information Model

Un fichier HCL est composé de **blocs** et d'**attributs**.

- **Body** : conteneur racine ou contenu d'un bloc
- **Block** : `type "label1" "label2" { ...}`
  - Type : identifiant (`resource`, `variable`...)
  - Labels : 0 à N strings entre guillemets (selon le *schema*)
  - Body : nouvel ensemble d'attributs et de blocs imbriqués
- **Attribute** : `name = expression`
  - Identifier = Expression
  - Évalué à une valeur typée (string, number, bool, list...)
- **Expression** : littéral, référence, opération, appel de fonction...

Schéma d'extraction

L'application hôte (OpenTofu/Packer/Vault...) définit un *schema* qui dit quels types de blocs et quels attributs sont valides. HCL valide la syntaxe ; l'application valide la sémantique.

HCL — Structure lexicale

Tokens

- Identifiants : `[a-zA-Z_][a-zA-Z0-9_]*`
- Littéraux numeriques : 42, 3.14, 1e6
- Strings : `"..."` ou `«EOT ...EOT`
- Booleens : `true`, `false`
- Nul : `null`
- Opérateurs : `+ - * / % && || ! == != < > <= >= ?: ...`

Commentaires (3 formes)

```
# Style shell (mono-ligne)

// Style C (mono-ligne)

/* Multi-ligne
   sur plusieurs
   lignes */
```

Espacement

- Pas significatif (sauf dans strings)
- Nouvelle ligne sépare les attributs/blocs
- Tab ou espaces : indifférent (`tofu fmt` normalise)

HCL — Strings, templates et heredocs

```
# String simple (quoted)
name = "linuxmaine"

# Interpolation : ${...}
greeting = "Bonjour ${var.user.name}, env=${var.env}"

# Heredoc : multi-ligne, indentation conservee
config = <<EOT
server {
  listen 80;
  server_name ${var.domain};
}
EOT

# Indented heredoc : <<- supprime l'indentation commune
script = <<-BASH
  #!/bin/bash
  echo "Deploying to ${var.env}"
  systemctl restart nginx
BASH

# Directives template : %{ ... }
template = <<-EOT
%{ for user in var.users ~}
  user: ${user.name} (${user.role})
%{ endfor ~}
EOT
```

Les tildes ~ suppriment le saut de ligne adjacent.

HCL — Système de types (cty)

Le moteur d'évaluation HCL utilise la bibliothèque `cty`.

Primitifs

- `string` : `"..."`
- `number` : 42, 3.14
- `bool` : true / false
- `null` : absence

Collections homogènes

- `list(T)` : ordonnée, indexable
- `set(T)` : non-ordonnée, unique
- `map(T)` : clés string → T

Structurels (hétérogènes)

- `object({a=number, b=string})`
- `tuple([number, string, bool])`

Spéciaux

- `any` : type dynamique
- `dynamic` : type inféré à l'éval
- *ephemeral* (OpenTofu 1.10+) : valeur transitoire

```
variable "users" {
  type = list(object({
    name  = string
    admin = optional(bool, false)
    quota = optional(number)
  }))
}
```

HCL — Opérateurs

Arithmétiques

- `+ - * /` : binaire
- `%` : modulo
- `-x` : négation

Comparaison

- `== !=`
- `< > <= >=`

Logiques

- `&&` (AND), `||` (OR), `!` (NOT)

Conditional (ternaire)

```
size = var.env == "prod" ? "t3.large" : "t3.micro"
```

Splat (accès multi-ressource)

```
ips = aws_instance.web[*].public_ip
# = [aws_instance.web[0].public_ip,
#    aws_instance.web[1].public_ip, ...]
```

Accès indexe

```
first = var.list[0]
val   = var.map["key"]
attr  = var.obj.field
```

HCL — Expressions for et dynamic

For sur liste :

```
upper_names = [for u in var.users : upper(u.name)]
admins      = [for u in var.users : u.name if u.admin]
```

For sur object/map (=> produit une map) :

```
user_quota = { for u in var.users : u.name => u.quota
               if u.quota != null }
```

Dynamic block : génération de sous-blocs en boucle :

```
resource "aws_security_group" "web" {
  dynamic "ingress" {
    for_each = var.allowed_ports
    iterator = port
    content {
      from_port   = port.value
      to_port     = port.value
      protocol    = "tcp"
      cidr_blocks = ["0.0.0.0/0"]
    }
  }
}
```

HCL — Références et chemins

```
# Variables
var.region                    # variable d'entr\'ee
local.tags                    # local computed

# Ressources et data
aws_instance.web.id           # attribut d'une resource
data.aws_ami.debian.id        # data source
module.network.vpc_id         # output d'un module

# Iterators (count / for_each / dynamic)
count.index                   # 0..n-1 (avec count)
each.key                      # cle (avec for_each)
each.value                    # valeur (avec for_each)

# Paths
path.module                   # chemin du module en cours
path.root                     # chemin du module racine
path.cwd                      # current working dir

# Terraform context
terraform.workspace           # workspace courant
```

HCL — Native Syntax vs JSON Syntax

HCL est **bi-syntaxe** : tout fichier `.tf` a un équivalent `.tf.json` (utile pour la génération programmatique).

Native (`main.tf`)

```
resource "aws_instance" "web" {
  ami           = "ami-123"
  instance_type = "t3.micro"
  tags = {
    Name = "web"
  }
}
```

JSON (`main.tf.json`)

```
{
  "resource": {
    "aws_instance": {
      "web": {
        "ami": "ami-123",
        "instance_type": "t3.micro",
        "tags": { "Name": "web" }
      }
    }
  }
}
```

Cas d'usage JSON

Génération depuis Python/Go/TS (CDK, terragrunt), pipeline où l'on assemble du HCL *à la volée*. La syntaxe native reste préférable pour le code maintenu à la main.

HCL — Standard library de fonctions (1/2)

Catégorie	Fonctions principales
Strings	`format, formatlist, join, split, substr, replace, regex, regexall, lower, upper, title, trim, trimprefix, trimsuffix, trimspace, chomp, indent`
Nombres	`min, max, abs, ceil, floor, parseint, pow, log, signum`
Collections	`length, contains, distinct, sort, reverse, slice, flatten, chunklist, coalescelist, compact, concat, element, index, range, setproduct, zipmap`
Maps/Objects	`keys, values, lookup, merge, setunion, setintersection, setsubtract, setsymmetricdifference`
Encoding	`jsonencode, jsondecode, yamlencode, yamldecode, csvdecode, base64encode, base64decode, base64gzip, urlencode, textencodebase64`
Filesystem	`file, fileexists, fileset, filebase64, templatefile, templatestring, abspath, dirname, basename, pathexpand`
Date & Time	`timestamp, formatdate, timeadd, plantimestamp`
Hash / Crypto	`md5, sha1, sha256, sha512, uuid, uuidv5, bcrypt, rsadecrypt`
Network / CIDR	`cidrhost, cidrnetmask, cidrsubnet, cidrsubnets, cidrcontains, cidrrange`
Type / Conv.	`tostring, tonumber, tobool, tolist, toset, tomap, can, try, type, defaults, nonsensitive, sensitive, issensitive`

HCL — Standard library de fonctions (2/2 exemples)

```
# Strings et templating
locals {
  greeting = format("Hello %s v%d", var.name, var.version)
  hostnames = formatlist("web-%02d", range(var.count))
}

# Collections
locals {
  unique   = distinct(["a", "b", "a", "c"])     # ["a","b","c"]
  merged   = merge({a=1}, {b=2, a=10})          # {a=10, b=2}
  filtered = [for x in var.list : x if x > 10]
}

# File + template
locals {
  config = templatefile("${path.module}/nginx.conf.tpl", {
    port  = var.port
    hosts = var.upstreams
  })
  cert = file("${path.module}/ca.pem")
}

# CIDR
locals {
  subnet_a = cidrsubnet(var.vpc_cidr, 8, 0)   # 10.0.0.0/24
  subnet_b = cidrsubnet(var.vpc_cidr, 8, 1)   # 10.0.1.0/24
  host_10  = cidrhost(local.subnet_a, 10)     # 10.0.0.10
}

# Try / Can (gestion d'erreurs d'expression)
locals {
  port = try(jsondecode(file("conf.json")).port, 8080)
  ok   = can(regex("^web-", var.name))
}
```

HCL — Précédence des opérateurs

Priorité	Opérateurs	Associativité
1 (haute)	`!`, `-` **(unaire)**	**droite**
2	`*`, `/`, `%`	**gauche**
3	`+`, `-` **(binaire)**	**gauche**
4	`>`, `>=`, `<`, `<=`	**gauche**
5	`==`, `!=`	**gauche**
6	`&&`	**gauche**
7	`||`	**gauche**
8 (basse)	`?` `:` **(ternaire conditional)**	**droite**

```
# Sans parentheses : ambigu\"it\'e ?
x = var.enable && var.env == "prod" ? "L" : "S"
# Equivalent :
x = (var.enable && (var.env == "prod")) ? "L" : "S"

# En cas de doute, parenther !
size = (var.replicas > 3 || var.tier == "premium") ? "large" : "small"
```

HCL — Outils du parser et de la chaîne

- **Bibliothèque officielle (Go)** : `github.com/hashicorp/hcl`
  - `hclparse` : parser générique
  - `hclsyntax` : syntaxe native
  - `json` : syntaxe JSON
  - `hclwrite` : génération programmatique
- **Type system** : `github.com/zclconf/go-cty` (types `cty.Value`)
- **Formatters** :
  - `tofu fmt` / `terraform fmt` : spécifique TF/OpenTofu
  - `hclfmt` : outil générique HCL
  - `tflint` : lint avancé (compatible OpenTofu)
- **Bindings non-Go** :
  - Python : `python-hcl2` (sous-set HCL2)
  - Rust : `hcl-rs`
  - TypeScript : `cdktf` (CDK for Terraform) qui synthétise du HCL/JSON

Pour aller plus loin

Spécification complète : `ebooks/HCL-Language-Specification-Official.pdf`

HCL — Bonnes pratiques

- **Toujours** `tofu fmt -recursive` avant commit (pre-commit hook)
- Préférer les `locals` aux répétitions d'expressions
- Déclarer *toutes* les variables (même avec `default`) pour rendre l'interface explicite
- Utiliser `validation` sur les variables d'entrée
- `sensitive = true` sur les outputs et variables qui sont des secrets
- Splitter en fichiers : `main.tf`, `variables.tf`, `outputs.tf`, `versions.tf`, `providers.tf`, `locals.tf`
- Versionner `.terraform.lock.hcl` (lockfile des providers)
- **Ne pas** versionner `*.tfstate`, `*.tfvars` (sauf `example.tfvars`)
- Préférer `for_each` à `count` dès qu'on a une clé nommée
- `moved {}` et `removed {}` plutôt que `state mv/rm` en CLI

Providers : configuration et alias

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

provider "aws" {
  region = "eu-west-3"
}

# Provider alias : second compte / r\'egion
provider "aws" {
  alias  = "us"
  region = "us-east-1"
}

resource "aws_s3_bucket" "logs_us" {
  provider = aws.us
  bucket   = "my-logs-us-east-1"
}
```

Les providers se chargent depuis `registry.opentofu.org` (mirror) ou un registry privé.

Providers officiels OpenTofu — panorama

Catégorie	Providers majeurs	Source
Hyperscalers cloud	aws, google, azurerm, oci, alicloud, ibm	hashicorp/*, oracle/oci
Clouds européens	ovh, scaleway, hcloud **(Hetzner)**, upcloud, exoscale	ovh/ovh, scaleway/scaleway, hetznercloud/hcloud
Clouds abordables	digitalocean, vultr, linode **(Akamai)**	digitalocean/digitalocean, vultr/vultr, linode/linode
DNS / CDN / WAF	cloudflare, dnsimple, gandi, nsi	cloudflare/cloudflare, dnsimple/dnsimple
Secrets / HSM	vault, openbao, aws-kms	hashicorp/vault, openbao/openbao
Container / Orchestr.	kubernetes, helm, nomad, docker, kind	hashicorp/kubernetes, hashicorp/helm
CI/CD / SCM	gitlab, github, gitea, bitbucket	gitlabhq/gitlab, integrations/github
Observabilité	grafana, datadog, newrelic, pagerduty	grafana/grafana, datadog/datadog
Self-hosted	libvirt **(KVM)**, proxmox, vsphere, xen	dmacvicar/libvirt, Telmate/proxmox
Généraux	local, null, random, tls, http, external	hashicorp/*
Bases de données	postgresql, mysql, mongodb, redis	cyrilgdn/postgresql, ...

Statistiques

`registry.opentofu.org` héberge **plus de 3000 providers**. Plus de 90% sont des forks/mirrors des providers Terraform compatibles.

Créer son propre provider OpenTofu (1/2 : principes)

Pourquoi créer un provider custom ?

- Intégrer un service interne (API maison, ERP, intranet)
- Manipuler une ressource non couverte par les providers existants
- Tester / mocker une API en local
- Compléter un provider existant avec un comportement métier

Boite à outils officielle

- `terraform-plugin-framework` (Go) — framework moderne (recommandé)
- `terraform-plugin-sdk` (Go, légacy) — SDK historique TF
- `terraform-plugin-docs` — génération auto de la doc
- Protocol gRPC v5 ou v6 (compatible OpenTofu et Terraform)

Anatomie d'un provider

- *Schema* : types de ressources et data sources exposés
- *CRUD* pour chaque resource : Create, Read, Update, Delete, ImportState
- *Configure* : authentification et options globales
- *Validation* : pré-checks sur les attributs

Créer son propre provider OpenTofu (2/2 : workflow)

Workflow d'élévage

```
# 1. Scaffold via le template officiel HashiCorp
git clone https://github.com/hashicorp/terraform-provider-scaffolding-framework \
          terraform-provider-myservice
cd terraform-provider-myservice
go mod edit -module github.com/myorg/terraform-provider-myservice

# 2. Implementer en Go (provider.go, resources/, data_sources/)
# Editer : internal/provider/example_resource.go etc.

# 3. Compiler et installer localement
make install                # binaire dans ~/.terraform.d/plugins/
# ou : go build -o terraform-provider-myservice

# 4. Tester avec un projet OpenTofu
cat > main.tf <<HCL
terraform {
  required_providers {
    myservice = {
      source = "myorg.local/internal/myservice"   # nom complet
      version = "0.1.0"
    }
  }
}
HCL
tofu init -plugin-dir=$HOME/.terraform.d/plugins
tofu apply

# 5. Publier (optionnel)
# - Registry public : github.com/opentofu/registry/ (PR public)
# - Registry priv|'e : artifactory, nexus, mirror local
```

Provider custom — exemple minimal (Go)

```go
// internal/provider/example_resource.go (extrait simplifié)
type exampleResource struct{}

func (r *exampleResource) Schema(_ context.Context,
    _ resource.SchemaRequest, resp *resource.SchemaResponse) {
  resp.Schema = schema.Schema{
    Attributes: map[string]schema.Attribute{
      "id":   schema.StringAttribute{Computed: true},
      "name": schema.StringAttribute{Required: true},
    },
  }
}

func (r *exampleResource) Create(ctx context.Context,
    req resource.CreateRequest, resp *resource.CreateResponse) {
  // 1. Decoder l'input HCL
  var plan ExampleModel
  resp.Diagnostics.Append(req.Plan.Get(ctx, &plan)...)

  // 2. Appeler l'API (HTTP, DB, etc.)
  id, err := callMyApiCreate(plan.Name.ValueString())
  // ... gestion erreur ...

  // 3. Stocker dans le state
  plan.ID = types.StringValue(id)
  resp.Diagnostics.Append(resp.State.Set(ctx, plan)...)
}
```

Pour aller plus loin

Tutoriel officiel : `https://opentofu.org/docs/plugin/framework/`
Exemples : `https://github.com/opentofu/opentofu/tree/main/internal/providers`

Variables, locals et outputs

```
variable "env" {
  type        = string
  default     = "dev"
  validation {
    condition     = contains(["dev","staging","prod"], var.env)
    error_message = "env doit etre dev, staging ou prod"
  }
}

locals {
  prefix   = "lm-${var.env}"
  tags     = { Project = "linuxmaine", Env = var.env }
  is_prod  = var.env == "prod"
}

output "bucket_name" {
  value       = aws_s3_bucket.logs.bucket
  description = "Nom du bucket S3 cr\'e\'e"
  sensitive   = false
}
```

Data sources : lire l'existant

```
# Lire une AMI d\'ej\`a publi\'ee
data "aws_ami" "debian" {
  most_recent = true
  owners      = ["136693071363"]   # Debian
  filter {
    name   = "name"
    values = ["debian-12-amd64-*"]
  }
}

# Utilisation dans une ressource
resource "aws_instance" "web" {
  ami           = data.aws_ami.debian.id
  instance_type = "t3.micro"
}
```

Les data sont en lecture seule : pas de modification ni de création.

Expressions : count, for_each, conditional

```
# count : meme ressource n fois (index num\'erique)
resource "aws_instance" "workers" {
  count         = var.worker_count
  ami           = data.aws_ami.debian.id
  instance_type = "t3.small"
  tags = { Name = "worker-${count.index}" }
}

# for_each : map ou set, index nomm\'e
resource "aws_iam_user" "team" {
  for_each = toset(["alice","bob","charlie"])
  name     = each.value
}

# conditional
locals {
  size = var.env == "prod" ? "t3.large" : "t3.micro"
}
```

Expressions avancées : for, splat, dynamic

```
# for expression sur une map
locals {
  user_names = [for u in var.users : u.name if u.active]
  user_map   = { for u in var.users : u.id => u.name }
}

# Splat sur une liste de ressources (count ou for_each)
output "all_ips" {
  value = aws_instance.workers[*].public_ip
}

# Bloc dynamic : g\'en\'erer des sous-blocs en boucle
resource "aws_security_group" "web" {
  name = "web"
  dynamic "ingress" {
    for_each = var.allowed_ports
    content {
      from_port = ingress.value
      to_port   = ingress.value
      protocol  = "tcp"
      cidr_blocks = ["0.0.0.0/0"]
    }
  }
}
```

Meta-arguments : lifecycle, depends_on, provisioner

```
resource "aws_instance" "web" {
  ami           = data.aws_ami.debian.id
  instance_type = "t3.micro"

  lifecycle {
    create_before_destroy = true        # bleu/vert
    prevent_destroy       = false       # protege une ressource critique
    ignore_changes        = [ tags ]    # n'observe pas certains attributs
    replace_triggered_by  = [ aws_security_group.web.id ]   # re-cr\'eation li\'ee
  }

  depends_on = [ aws_iam_role.web_role ]    # d\'ependance implicite manquante

  provisioner "remote-exec" {
    inline = [ "echo prov done" ]
    connection { type = "ssh"; user = "admin"; host = self.public_ip }
  }
}
```

Fonctions HCL — les indispensables

Catégorie	Fonctions courantes
Strings	upper, lower, trim, replace, format, regex, regexall, split, join, substr
Numbers	min, max, abs, ceil, floor, parseint, pow, log
Collections	length, contains, distinct, sort, reverse, slice, flatten, merge, lookup
Map/Object	keys, values, zipmap, setunion, setintersection, setsubtract
Encoding	jsonencode, jsondecode, yamlencode, yamldecode, base64encode, base64decode
File	file, fileexists, fileset, templatefile, filebase64
Date	timestamp, formatdate, timeadd
Hash/Crypto	md5, sha1, sha256, sha512, uuid, bcrypt
Network	cidrhost, cidrnetmask, cidrsubnet, cidrrange
Type	tostring, tonumber, tobool, tolist, toset, tomap, can, try, type

```
locals {
  config = jsondecode(file("${path.module}/config.json"))
  ipv4   = cidrhost(var.vpc_cidr, 10)              # 10.0.0.10
  rendered = templatefile("init.tpl", {name="web"})
}
```

Checks et postconditions ($\geq$ 1.5)

```
# Check block : assertions au plan/apply, n'\'echoue pas le run
check "https_endpoint" {
  data "http" "status" {
    url = "https://${aws_lb.web.dns_name}/health"
  }
  assert {
    condition     = data.http.status.status_code == 200
    error_message = "Le health check renvoie ${data.http.status.status_code}"
  }
}

# Preconditions / postconditions dans une resource ou data
resource "aws_instance" "web" {
  ami           = var.ami_id
  instance_type = "t3.micro"
  lifecycle {
    precondition {
      condition     = data.aws_ami.this.architecture == "x86_64"
      error_message = "Architecture incorrecte"
    }
  }
}
```

Modules : composition réutilisable

```
# Module LOCAL
module "network" {
  source = "./modules/network"
  cidr   = "10.0.0.0/16"
}

# Module depuis le REGISTRY OpenTofu
module "vpc" {
  source  = "registry.opentofu.org/terraform-aws-modules/vpc/aws"
  version = "5.1.0"
}

# Module GIT (commit ou tag)
module "app" {
  source = "git::https://github.com/org/module-app.git?ref=v2.0"
}

# Acc\'es aux outputs du module
output "vpc_id" {
  value = module.vpc.vpc_id
}
```

State : backends et locking

```
# Backend local (par d\'efaut)
terraform { backend "local" { path = "terraform.tfstate" } }

# Backend S3 (multi-utilisateur, lock via DynamoDB ou S3 Object Lock)
terraform {
  backend "s3" {
    bucket         = "tfstate-linuxmaine"
    key            = "infra/prod/terraform.tfstate"
    region         = "eu-west-3"
    dynamodb_table = "tflock"
    encrypt        = true
  }
}

# Backend GCS
terraform { backend "gcs" { bucket = "tfstate-lm"; prefix = "infra" } }

# Backend HTTP (GitLab Managed State)
terraform {
  backend "http" {
    address = "https://gitlab.com/api/v4/projects/123/terraform/state/prod"
  }
}
```

Import : adopter une ressource existante

Forme moderne (bloc, depuis 1.5) :

```
import {
  to = aws_s3_bucket.legacy
  id = "my-legacy-bucket-2019"
}

# OpenTofu g\'en\'ere le bloc resource si on lance :
# tofu plan -generate-config-out=generated.tf
```

Forme historique (CLI) :

```
tofu import aws_instance.web i-0123456789abcdef0
```

Bénéfice

Permet de **reprendre la main** sur une infra créée à la main, sans `destroy`.

Refactoring : moved et removed

```
# Renommer une ressource SANS la re-cr\'eer
moved {
  from = aws_instance.web_old
  to   = aws_instance.web
}

# Sortir une ressource du code SANS la d\'etruire (depuis 1.7)
removed {
  from = aws_s3_bucket.legacy
  lifecycle {
    destroy = false
  }
}
```

- `moved` : indispensable lors de refactor (renommage, modularisation)
- `removed` : retire du state sans appeler l'API
- Alternative au `state mv` CLI, mais déclarative et versionnée

Compatibility settings et garanties OpenTofu

- `required_version` : pin du tooling OpenTofu (ex. `">= 1.6, < 2.0"`)
- `required_providers` : versions des providers
- **Compatibility Promises** : OpenTofu garantit la stabilité de l'interface CLI/HCL pour les versions 1.x
- Migrations cassantes annoncées dans le *Upgrade Guide*
- **Ephemerality** (depuis 1.10) : valeurs *transitoires* non persistées dans le state (secrets de session)

Ephemerality

Utile pour créer un mot de passe temporaire, le passer à un provider, sans le mémoriser dans le state ni l'output.

TP03 tp03-multi-vm/ — for_each

Objectif pédagogique

Provisionner 3 VMs avec des configurations différentes via `for_each`.

- Variable `vms` : map de configurations (type, mémoire, disque)
- Une seule `resource libvirt_domain` avec `for_each = var.vms`
- Accès aux IPs en sortie via `values()[*].network_interface...`

Scripts du TP : `tp03-do-run.sh` · `tp03-do-reset.sh` · `tp03-do-check.sh`

TP04 `tp04-modules/` — Modules locaux et registry

Objectif pédagogique

Créer un module `vm-debian` et l'appeler 3 fois ; comprendre la composition.

- Sous-répertoire `modules/vm-debian/`
- Variables d'entrée du module (nom, ressources)
- Outputs du module remontés au projet racine
- Versioning : `source = "./modules/..."` vs `source = "git::..."`

Scripts du TP : `tp04-do-run.sh` · `tp04-do-reset.sh` · `tp04-do-check.sh`

TP05 tp05-state/ — State management

Objectif pédagogique

Manipuler le state : `list`, `show`, `mv`, `rm`, `pull`, `push`.

- `tofu state list` — inventaire des ressources
- `tofu state mv` — renommer dans le state (après un `moved`)
- Sauvegarde et restauration via `pull`/`push`
- Backup auto `terraform.tfstate.backup`

Scripts du TP : `tp05-do-run.sh` · `tp05-do-reset.sh` · `tp05-do-check.sh`

TP06 `tp06-provisioners/` — Provisioners

Objectif pédagogique

Utiliser `remote-exec`, `local-exec`, `file` ; comprendre les limites.

- Provisioners = "*escape hatch*" : utilisation **exceptionnelle**
- Préférer **cloud-init** ou **Ansible** après OpenTofu
- `when = destroy` : action au destroy

Scripts du TP : `tp06-do-run.sh` · `tp06-do-reset.sh` · `tp06-do-check.sh`

TP07 tp07-cloud-init/ — Cloud-init et template

Objectif pédagogique

Paramétrer une VM via `templatefile` et un user-data cloud-init.

- Création d'un `cloud-init.yml.tpl`
- Injection des variables par `templatefile(..., {...})`
- Lecture du résultat par cloud-init au premier boot

Scripts du TP : `tp07-do-run.sh` · `tp07-do-reset.sh` · `tp07-do-check.sh`

TP08 tp08-variables-avancees/ — Validations et types complexes

Objectif

Variables typées (object, list of object), validations, valeurs sensibles.

- Types composés : `list(object({...}))`
- `validation` : conditions sur `var.<name>`
- `sensitive = true` : masquage des outputs

Scripts du TP : `tp08-do-run.sh` · `tp08-do-reset.sh` · `tp08-do-check.sh`

TP09 tp09-conditions/ — Conditions et expressions for

Objectif

Brancher la création de ressources via `count`/`for_each` conditionnel.

- `count = var.enable ? 1 : 0`
- Expressions `for` et `if`
- Locals calculés pour rendre le code lisible

Scripts du TP : `tp09-do-run.sh` · `tp09-do-reset.sh` · `tp09-do-check.sh`

TP10 tp10-outputs-avances/ — Outputs et dépendances

Objectif

Maitriser les outputs : sensitive, precondition, depends_on.

- `sensitive = true` : vérification au plan
- Sortie d'une map / object complet
- `depends_on` sur un output (pour séquencer)

Scripts du TP : `tp10-do-run.sh` · `tp10-do-reset.sh` · `tp10-do-check.sh`

TP11 `tp11-fonctions/` — Fonctions HCL

Objectif

Manipuler strings, collections, files, encoding, cidr.

- `templatefile`, `jsondecode`, `cidrhost`, `cidrsubnet`
- `can()`, `try()` : gestion des erreurs d'expression
- `tofu console` : pratiquer en REPL

Scripts du TP : `tp11-do-run.sh` · `tp11-do-reset.sh` · `tp11-do-check.sh`

State Encryption (exclusivité OpenTofu)

```
terraform {
  encryption {
    key_provider "pbkdf2" "main" {
      passphrase = var.state_passphrase
    }
    method "aes_gcm" "main" {
      keys = key_provider.pbkdf2.main
    }
    state { method = method.aes_gcm.main }
    plan  { method = method.aes_gcm.main }
  }
}
```

Key providers supportés : `pbkdf2`, `aws_kms`, `gcp_kms`, `openbao`.
Méthodes : `aes_gcm`, `unencrypted` (migration).

Early Variable Evaluation (exclusivité OpenTofu)

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

terraform {
  backend "s3" {
    bucket = "tfstate-${var.env}"
    key    = "infra/${var.env}/state.tfstate"
    region = "eu-west-3"
  }
}
```

Avant OpenTofu 1.7

Le bloc `terraform{}` était évalué *avant* les variables. Il fallait passer par `-backend-config=*.hcl` pour chaque environnement.

TP12 `tp12-state-encryption/` — Chiffrement PBKDF2

Objectif

Activer le chiffrement natif du state et vérifier que le `terraform.tfstate` est illisible sans la passphrase.

Scripts du TP : `tp12-do-run.sh` · `tp12-do-reset.sh` · `tp12-do-check.sh`

TP13 tp13-backend-dynamique/ — Early eval

Objectif

Paramétrer dynamiquement le backend selon la variable `env`.

Scripts du TP : `tp13-do-run.sh` · `tp13-do-reset.sh` · `tp13-do-check.sh`

AWS — VPC + EC2

```
terraform {
  required_providers { aws = { source = "hashicorp/aws", version = "~> 5.0" } }
}
provider "aws" { region = "eu-west-3" }

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

resource "aws_subnet" "public" {
  vpc_id            = aws_vpc.main.id
  cidr_block        = "10.0.1.0/24"
  availability_zone = "eu-west-3a"
}

resource "aws_instance" "web" {
  ami                    = data.aws_ami.debian.id
  instance_type          = "t3.micro"
  subnet_id              = aws_subnet.public.id
  vpc_security_group_ids = [aws_security_group.web.id]
  tags = { Name = "web-${var.env}" }
}
```

GCP — VPC + Compute

```
terraform {
  required_providers { google = { source = "hashicorp/google", version = "~> 5.0" } }
}
provider "google" {
  project = "linuxmaine-formation"
  region  = "europe-west9"
}

resource "google_compute_network" "vpc" {
  name                    = "lm-vpc"
  auto_create_subnetworks = false
}

resource "google_compute_instance" "web" {
  name         = "web-vm"
  machine_type = "e2-small"
  zone         = "europe-west9-a"
  boot_disk { initialize_params { image = "debian-cloud/debian-12" } }
  network_interface {
    network = google_compute_network.vpc.id
    access_config {}
  }
}
```

Azure — Resource Group + VM

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

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

resource "azurerm_linux_virtual_machine" "web" {
  name                = "vm-web"
  resource_group_name = azurerm_resource_group.rg.name
  location            = azurerm_resource_group.rg.location
  size                = "Standard_B1s"
  admin_username      = "tofu"
  network_interface_ids = [azurerm_network_interface.nic.id]
  os_disk { caching = "ReadWrite", storage_account_type = "Standard_LRS" }
  source_image_reference {
    publisher = "Debian"; offer = "debian-12"; sku = "12"; version = "latest"
  }
}
```

OVH — Cloud Project + DNS

```
terraform {
  required_providers {
    ovh = { source = "ovh/ovh", version = "~> 0.50" }
  }
}
provider "ovh" {
  endpoint           = "ovh-eu"
  application_key    = var.ovh_app_key
  application_secret = var.ovh_app_secret
  consumer_key       = var.ovh_consumer_key
}

# Cloud instance (Public Cloud)
resource "ovh_cloud_project_instance" "vm" {
  service_name = var.ovh_project
  region       = "GRA9"
  flavor_name  = "b3-8"
  image_name   = "Debian 12"
}

# DNS : enregistrement A
resource "ovh_domain_zone_record" "www" {
  zone      = "linuxmaine.org"
  subdomain = "www"
  fieldtype = "A"
  target    = ovh_cloud_project_instance.vm.public_ip
  ttl       = 300
}
```

Hetzner / Scaleway / DigitalOcean

Tous les providers majeurs sont sur `registry.opentofu.org` :

```
# Hetzner Cloud
provider "hcloud" { token = var.hcloud_token }
resource "hcloud_server" "web" {
  name = "web"; image = "debian-12"; server_type = "cx21"; location = "fsn1"
}

# Scaleway
provider "scaleway" { region = "fr-par"; zone = "fr-par-1" }
resource "scaleway_instance_server" "web" {
  type = "DEV1-S"; image = "debian_bookworm"
}

# DigitalOcean
provider "digitalocean" { token = var.do_token }
resource "digitalocean_droplet" "web" {
  image = "debian-12-x64"; name = "web"; region = "ams3"; size = "s-1vcpu-1gb"
}
```

Clouds supportés par OpenTofu — Synthèse

Cloud / Service	Provider	Source	Catégorie	Note
Amazon AWS	aws	hashicorp/aws	**Hyperscaler**	**Le plus complet (1000+ resources)**
Google Cloud (GCP)	google	hashicorp/google	**Hyperscaler**	**Tres mature**
Microsoft Azure	azurerm	hashicorp/azurerm	**Hyperscaler**	**+ azuread pour Entra ID**
Oracle Cloud (OCI)	oci	oracle/oci	**Hyperscaler**	**Officiel Oracle**
Alibaba Cloud	alicloud	aliyun/alicloud	**Hyperscaler**	**Officiel Aliyun**
IBM Cloud	ibm	ibm-cloud/ibm	**Hyperscaler**	**+ Power Systems**
OVHcloud	ovh	ovh/ovh	**Cloud européen**	**Public Cloud / DNS / Bare Metal**
Scaleway	scaleway	scaleway/scaleway	**Cloud européen**	**Officiel**
Hetzner Cloud	hcloud	hetznercloud/hcloud	**Cloud européen**	**Tres bon rapport prix**
DigitalOcean	digitalocean	digitalocean/digitalocean	**Cloud abord.**	**Simple**
Vultr	vultr	vultr/vultr	**Cloud abord.**	**30+ locations**
Linode (Akamai)	linode	linode/linode	**Cloud abord.**	**Officiel**
UpCloud	upcloud	UpCloudLtd/upcloud	**Cloud européen**	**Nordique**
Exoscale	exoscale	exoscale/exoscale	**Cloud européen**	**Suisse**
Cloudflare	cloudflare	cloudflare/cloudflare	**DNS / CDN / WAF**	**Pages, R2, Workers**
DNSimple, Gandi, Nic.fr	**multi**	dnsimple, gandi/gandi	**DNS**	**Multi-registrar**
Vault / OpenBao	vault, openbao	hashicorp/vault, openbao/openbao	**Secrets**	**Secrets stores**
Kubernetes	kubernetes	hashicorp/kubernetes	**Orchestration**	**+ helm pour les charts**
Proxmox	proxmox	Telmate/proxmox	**Self-host**	**PVE 7/8**
libvirt (KVM)	libvirt	dmacvicar/libvirt	**Self-host**	**Local QEMU**

Liste complète (3000+ providers) : https://registry.opentofu.org

TP27 `tp27-aws/` — AWS : VPC + EC2 + Security Group

Objectif pédagogique

Déployer une instance EC2 dans un VPC dédié, avec un security group HTTP/SSH.

- Provider `hashicorp/aws`, region `eu-west-3`
- Ressources : VPC, Internet Gateway, Subnet, Route Table, SG, EC2
- Variable d'auth : `AWS_PROFILE` (ou `AWS_ACCESS_KEY_ID`)
- Outputs : `public_ip`, `instance_id`, commande SSH

Scripts du TP : `tp27-do-run.sh` · `tp27-do-reset.sh` · `tp27-do-check.sh`

TP28 tp28-gcp/ — GCP : Compute Engine + VPC

Objectif pédagogique

Déployer une VM Compute Engine dans un VPC custom avec firewall ouvert.

- Provider `hashicorp/google`, region `europe-west9` (Paris)
- Ressources : Network, Subnet, Firewall, Instance Debian 12
- Auth via `GOOGLE_APPLICATION_CREDENTIALS` ou `gcloud auth`
- Outputs : `nat_ip`, `instance_self_link`

Scripts du TP : `tp28-do-run.sh` · `tp28-do-reset.sh` · `tp28-do-check.sh`

TP29 tp29-azure/ — Azure : Resource Group + Linux VM

Objectif pédagogique

Déployer une VM Debian dans un Resource Group avec VNet + NIC + Public IP.

- Provider `hashicorp/azurerm`, location `France Central`
- Ressources : RG, VNet, Subnet, NSG, NIC, Public IP, Linux VM
- Auth via `az login` ou Service Principal (`ARM_*`)
- Outputs : `public_ip`, `vm_id`

Scripts du TP : `tp29-do-run.sh` · `tp29-do-reset.sh` · `tp29-do-check.sh`

TP30 tp30-ovh-public-cloud/ — OVH : Public Cloud Instance

Objectif pédagogique

Déployer une instance dans le Public Cloud OVH (region GRA9, Paris).

- Provider `ovh/ovh`, endpoint `ovh-eu`
- Ressources : `ovh_cloud_project_instance`, `ovh_cloud_project_network_private`
- Auth via `OVH_*` (app_key / app_secret / consumer_key)
- Outputs : `public_ip`, `instance_id`

Scripts du TP : `tp30-do-run.sh` · `tp30-do-reset.sh` · `tp30-do-check.sh`

TP31 tp31-scaleway/ — Scaleway : Instance

Objectif pédagogique

Déployer une instance DEV1 sous Scaleway (zone `fr-par-1`).

- Provider `scaleway/scaleway`
- Ressources : `scaleway_instance_server`, `scaleway_instance_ip`, `scaleway_instance_security_group`
- Auth via `SCW_ACCESS_KEY` / `SCW_SECRET_KEY` / `SCW_DEFAULT_PROJECT_ID`
- Outputs : `ip_address`, `server_id`

Scripts du TP : `tp31-do-run.sh` · `tp31-do-reset.sh` · `tp31-do-check.sh`

TP32 `tp32-hetzner/` — Hetzner Cloud : Server + Firewall

Objectif pédagogique

Déployer un serveur `cx22` + firewall dans la région `fsn1` (Falkenstein, DE).

- Provider `hetznercloud/hcloud`
- Ressources : `hcloud_server`, `hcloud_firewall`, `hcloud_ssh_key`
- Auth via `HCLOUD_TOKEN`
- Outputs : `ipv4`, `ipv6`

Scripts du TP : `tp32-do-run.sh` · `tp32-do-reset.sh` · `tp32-do-check.sh`

TP33 tp33-digitalocean/ — DigitalOcean : Droplet + Reserved IP

Objectif pédagogique

Déployer un Droplet `s-1vcpu-1gb` avec IP réservée à Amsterdam.

- Provider `digitalocean/digitalocean`
- Ressources : `digitalocean_droplet`, `digitalocean_reserved_ip`, `digitalocean_firewall`
- Auth via `DIGITALOCEAN_TOKEN`
- Outputs : `ipv4`, `reserved_ip`

Scripts du TP : `tp33-do-run.sh` · `tp33-do-reset.sh` · `tp33-do-check.sh`

TP34 `tp34-cloudflare/` — Cloudflare : DNS + Page Rules

Objectif pédagogique

Gérer enregistrements DNS + Page Rules + WAF d'une zone Cloudflare.

- Provider `cloudflare/cloudflare` (v5)
- Ressources : `cloudflare_record` (A, AAAA, CNAME, MX), `cloudflare_page_rule`
- Auth via `CLOUDFLARE_API_TOKEN` (scope `Zone:Edit`)
- Outputs : `record_ids`, FQDN

Scripts du TP : `tp34-do-run.sh` · `tp34-do-reset.sh` · `tp34-do-check.sh`

OpenTofu vs Ansible — positionnement

Deux outils, deux rôles distincts

- **OpenTofu** : provisionne l'**infrastructure** (VPC, VM, DNS, IAM...) — déclaratif, multi-cloud
- **Ansible** : configure l'**intérieur** des serveurs (paquets, fichiers, services) — procédural, agentless

Complémentaires plus que concurrents

- *Day 0* : OpenTofu crée les serveurs
- *Day 1* : Ansible les configure
- *Day 2+* : Ansible maintient ; OpenTofu refait l'infra

Règle simple

Murs (immuables) → OpenTofu. *Intérieur* (configurable) → Ansible.

Workflow OpenTofu + Ansible

Répartition des rôles

- **OpenTofu** : crée VPC, sous-réseaux, VMs, security groups, DNS, IAM
- **Ansible** : installe paquets, déploie applications, configure services

- OpenTofu génère un **inventory dynamique** pour Ansible
- Couplage par `local-exec` ou `terraform output -json` → inventaire
- Alternative moderne : **collections Ansible** qui consomment le state OpenTofu

Inventaire Ansible à partir d'OpenTofu

```
# outputs.tf : exporter l'inventaire en YAML
output "ansible_inventory" {
  value = <<-EOT
    all:
      hosts:
        %{ for k, v in libvirt_domain.vm ~}
        ${k}:
          ansible_host: ${v.network_interface[0].addresses[0]}
          ansible_user: tofu
        %{ endfor ~}
  EOT
}
```

```
# Générer puis lancer
tofu output -raw ansible_inventory > inventory.yml
ansible -i inventory.yml all -m ping
ansible-playbook -i inventory.yml site.yml
```

Inventaire dynamique : plugin `cloud.terraform`

- Collection officielle Ansible `cloud.terraform`
- Lit directement le `terraform.tfstate` (ou via API)
- Plus de fichier d'inventaire à maintenir

```
# inventory.tofu.yml
---
plugin: cloud.terraform.terraform_state
backend_type: local
backend_config:
  path: /home/tofu/projet/terraform.tfstate
search_child_modules: true
```

```
ansible-galaxy collection install cloud.terraform
ansible -i inventory.tofu.yml all -m ping
```

TP17 `tp17-lamp-infra/` — Infra LAMP de bout en bout

Objectif

Provisionner une infra LAMP : VM + réseau + DNS via OpenTofu, puis configurer LAMP via Ansible.

- Création VM Debian + ouverture port 80/443 (libvirt + iptables)
- Inventaire Ansible généré par OpenTofu
- Playbook Ansible installe Apache + MariaDB + PHP

Scripts du TP : `tp17-do-run.sh` · `tp17-do-reset.sh` · `tp17-do-check.sh`

TP18 `tp18-hardening/` — Durcissement avec OpenTofu

Objectif

Paramétrer le hardening dès la création : SSH, firewall, audit.

- Variables dédiées au hardening
- Cloud-init avec `ufw` et SSH durci
- Lien avec scan `lynis` / `openscap`

Scripts du TP : `tp18-do-run.sh` · `tp18-do-reset.sh` · `tp18-do-check.sh`

TP14 tp14-secrets/ — Gestion des secrets

Objectif

Éviter les secrets en clair : `TF_VAR_xxx`, `sensitive=true`, KMS.

- Variable d'env. `TF_VAR_db_password`
- Provider `vault` ou `aws_secretsmanager_secret`
- Chiffrement du state (cf. TP12)

Scripts du TP : `tp14-do-run.sh` · `tp14-do-reset.sh` · `tp14-do-check.sh`

TP15 tp15-debugging/ — Debugging

Objectif

Utiliser `TF_LOG`, `tofu console`, `tofu graph`.

- `TF_LOG=TRACE tofu plan 2> trace.log`
- `tofu graph -type=plan | dot -Tpng`
- Analyse du `terraform.tfstate.backup`

Scripts du TP : `tp15-do-run.sh` · `tp15-do-reset.sh` · `tp15-do-check.sh`

TP16 tp16-testing/ — Tests natifs tofu test

```
# tests/network.tftest.hcl
run "subnet_count" {
  command = plan
  variables { subnet_count = 3 }
  assert {
    condition     = length(libvirt_network.lab) == 3
    error_message = "Doit creer 3 reseaux"
  }
}
```

```
tofu test                  # run all
tofu test -junit-xml=results.xml
```

Scripts du TP : tp16-do-run.sh · tp16-do-reset.sh · tp16-do-check.sh

Pipeline GitLab CI : .gitlab-ci.yml

```
stages: [validate, plan, apply]

variables:
  TF_ROOT: "${CI_PROJECT_DIR}/infra"
  TF_STATE_NAME: "${CI_PROJECT_NAME}-${CI_COMMIT_REF_SLUG}"

image: ghcr.io/opentofu/opentofu:1.10.6

default:
  cache: { paths: [.terraform] }
  before_script: [ cd $TF_ROOT, tofu init ]

validate:
  stage: validate
  script: [ tofu fmt -check, tofu validate ]

plan:
  stage: plan
  script: [ tofu plan -out=tf.plan, tofu show -json tf.plan > plan.json ]
  artifacts: { paths: [infra/tf.plan, infra/plan.json] }

apply:
  stage: apply
  script: [ tofu apply -auto-approve tf.plan ]
  when: manual
  only: [ main ]
```

TP20 `tp20-multi-env/` — Multi-environnement (workspaces)

Objectif

Trois environnements : `dev`, `staging`, `prod`.

- `tofu workspace new staging`
- Variable `env = terraform.workspace`
- State séparé par environnement

Scripts du TP : `tp20-do-run.sh` · `tp20-do-reset.sh` · `tp20-do-check.sh`

TP21 `tp21-infra-complete/` — Infrastructure complète

Objectif

Synthèse : modules, multi-env, secrets, encryption, tests, CI.

- Structure modulaire (`network`, `compute`, `database`)
- Backend chiffré + early eval
- Pipeline GitLab CI complet
- Doc `terraform-docs`

Scripts du TP : `tp21-do-run.sh` · `tp21-do-reset.sh` · `tp21-do-check.sh`

TP22 `tp22-ovh-dns/` — DNS OVH (linuxmaine.org)

Objectif

Mettre à jour un enregistrement A du domaine `linuxmaine.org` via le provider `ovh`.

- Création d'un *API token* OVH (Application key/secret/consumer)
- Ressource `ovh_domain_zone_record`
- Mise à jour idempotente, refresh du cache DNS via `ovh_domain_zone`

Scripts du TP : `tp22-do-run.sh` · `tp22-do-reset.sh` · `tp22-do-check.sh`

Qualité et tests

Lint et sécurité

- **tflint** : linter HCL (compatible OpenTofu)
- **tfsec** : scanner sécurité (Aqua Security)
- **checkov** : compliance multi-framework
- **tofu fmt** / **tofu validate**

Tests

- **Terratest** : Go (compatible OpenTofu)
- **tofu test** : tests natifs
- **terraform-docs** : doc auto
- **infracost** : coûts cloud
- **OPA** : Policy as Code

Accélérateurs de performance

- **Parallelism** : `tofu apply -parallelism=20`
- **Refresh skip** : `tofu plan -refresh=false`
- **Target** : `tofu apply -target=module.web`
- **State splitting** : un state par couche (réseau / compute / app)
- **Provider cache** : `TF_PLUGIN_CACHE_DIR=~/.terraform.d/plugin-cache`
- **Lock file** : `tofu init -lockfile=readonly` en CI

Annexe — Lexique (1/3)

IaC *Infrastructure as Code* — infra décrite en fichiers versionnés

API Interface de programmation entre logiciels

CLI Interface en ligne de commande

HCL HashiCorp Configuration Language — DSL HashiCorp (utilisé par TF/OpenTofu/Packer)

JSON JavaScript Object Notation — format de données

YAML YAML Ain't Markup Language — format de données lisible

Idempotence N exécutions = même résultat qu'une seule

Déclaratif Décrire l'état voulu (vs impératif : les étapes)

Provider Plugin pour un service (AWS, libvirt, OVH...)

Resource Élément d'infra géré (VM, réseau, DNS...)

Data source Lecture seule d'une ressource existante

State Fichier JSON de l'état réel suivi par OpenTofu

Backend Stockage du state (local, S3, GCS, HTTP, Postgres...)

Lock Verrou pour éviter les modifications concurrentes du state

Annexe — Lexique (2/3)

Module Bloc réutilisable de configuration HCL
Variable Paramètre d'entrée d'un projet ou d'un module
Output Valeur exposée après `apply`
Workspace Environnement isolé (state séparé)
Plan Prévisualisation des changements
Apply Application des changements
Destroy Suppression de toute l'infrastructure gérée
Refresh Synchronisation du state avec l'état réel
Import Adoption d'une ressource existante dans le state
Drift Écart entre le state et la réalité (modification hors OpenTofu)
Lockfile `.terraform.lock.hcl` : versions gélées des providers
Provisioner Action impérative après création (`remote-exec...`)
Backend dynamique Backend dont la clé dépend d'une variable (OpenTofu)
State Encryption Chiffrement natif du state (OpenTofu 1.7+)
Early Eval Évaluation anticipée des variables dans `terraform{}` (OpenTofu 1.7+)
Ephemerality Valeurs transitoires non persistées dans le state (OpenTofu 1.10+)

Annexe — Lexique (3/3)

PBKDF2 Dérivation de clé à partir d'une passphrase

AES-GCM Chiffrement symétrique authentifié

KMS Key Management Service (AWS / GCP / Azure)

OpenBao Fork open source de HashiCorp Vault (Linux Foundation)

Registry Dépôt de modules et providers

Manifeste OpenTofu Document fondateur du fork (septembre 2023)

MPL 2.0 Mozilla Public License — véritablement libre

BSL 1.1 Business Source License — source-available, restrictions commerciales

SAST Static Application Security Testing (scan du code)

DAST Dynamic Application Security Testing (scan en runtime)

SCA Software Composition Analysis (scan des dépendances)

SBOM Software Bill of Materials

Shift-Left Décaler la sécurité *vers la gauche* du cycle (dès le code)

Packer Outil HashiCorp de *baking* d'images machines (BSL 1.1)

Cloud-init Standard de configuration initiale d'une VM cloud

Annexe — Ouvrages recommandés (FR)

- *Terraform : Infrastructure as Code* — Yevgeniy Brikman (trad. fr.) – O'Reilly, 2024
- *L'infrastructure as code avec Terraform* — Pierre Mavro – ENI, 2023
- *DevOps - Faire de la sécurité un atout pour la transformation digitale* — Pierre-Henri Lavallée – Eyrolles, 2022
- *Débuter avec l'Infrastructure as Code* — Mickael Roger (Hackvirt), Pierre Tibulle – Hackvirt, 2024
- *Cloud Computing : approche par les certifications* — Lokesh Vij – ENI, 2023
- *Le Guide DevOps pour les Nuls* (Hewlett-Packard Enterprise Edition) – 2021 (gratuit)
- *Ansible : Gérer votre infrastructure avec un outil simple et efficace* — Daniel Hagimont, Boris Tachan – ENI, 2022

Voir aussi les ebooks PDF inclus dans le répertoire `ebooks/`.

Annexe — Books in English

- *Terraform: Up & Running* (3rd ed.) — Yevgeniy Brikman — O'Reilly, 2022 (s'applique à OpenTofu)
- *Infrastructure as Code* (3rd ed.) — Kief Morris — O'Reilly, 2024
- *Terraform Cookbook* (2nd ed.) — Mikael Krief — Packt, 2023
- *OpenTofu Cookbook* — Mikael Krief — Packt, 2025
- *Pulumi for Cloud Architects* — Christian Nunciato — O'Reilly, 2024
- *The Phoenix Project* — Gene Kim et al. — IT Revolution, 2013 (culture DevOps)
- *The DevOps Handbook* — Gene Kim et al. — IT Revolution, 2nd ed. 2021
- *Cloud Native Infrastructure* — Justin Garrison & Kris Nova — O'Reilly, 2017

Annexe — Refcard .gitlab-ci.yml

```
stages: [ validate, security, plan, apply ]

include:
  - template: Security/SAST.gitlab-ci.yml

variables:
  TF_VERSION: "1.10.6"

default:
  image: ghcr.io/opentofu/opentofu:${TF_VERSION}
  cache:
    key: { files: [.terraform.lock.hcl] }
    paths: [.terraform/]
  before_script:
    - tofu init -input=false

fmt:        { stage: validate, script: [tofu fmt -check -recursive] }
validate:   { stage: validate, script: [tofu validate] }
tfsec:      { stage: security, script: [tfsec --no-color .] }
checkov:    { stage: security, script: [checkov -d . --quiet] }
plan:
  stage: plan
  script: [tofu plan -out=tf.plan]
  artifacts: { paths: [tf.plan], expire_in: 1 week }
apply:
  stage: apply
  script: [tofu apply -auto-approve tf.plan]
  when: manual
  only: [main]
```

Annexe — Refcard .gitignore

```
# OpenTofu / Terraform
.terraform/
.terraform.lock.hcl    # OUI a versionner (NE PAS gitignorer)
*.tfstate
*.tfstate.*
*.tfplan
tf.plan
override.tf
override.tf.json
*_override.tf
*_override.tf.json
crash.log
crash.*.log
.terraformrc
terraform.rc
.tofurc

# Secrets (NE JAMAIS commit)
*.tfvars
!example.tfvars
*.auto.tfvars
.env

# Editeurs
.vscode/
.idea/
*.swp

# Cache providers (souvent ignor|'e)
.terraform.d/
```

Annexe — Refcard `terraform.tfvars` et `pre-commit`

```
# .pre-commit-config.yaml
repos:
  - repo: https://github.com/antonbabenko/pre-commit-terraform
    rev: v1.96.1
    hooks:
      - id: terraform_fmt
      - id: terraform_validate
      - id: terraform_tflint
      - id: terraform_tfsec
      - id: terraform_docs
        args: [--args=--config=.terraform-docs.yml]
  - repo: https://github.com/pre-commit/pre-commit-hooks
    rev: v5.0.0
    hooks:
      - id: trailing-whitespace
      - id: end-of-file-fixer
      - id: check-yaml
```

Activation

`pre-commit install` après `pip install pre-commit`.

Annexe — Liens officiels et ressources

Documentation locale (dans le dépôt)

- `labs/hcl/index.html` — Doc HCL technique en français (11 pages)
- `labs/opentofu/index.html` — Doc OpenTofu technique en français (20+ pages)
- `ebooks/HCL-Language-Specification-Official.pdf` — Spec HCL officielle (38 p.)
- `ebooks/OpenTofu Language Documentation.pdf` — Doc OpenTofu officielle
- `Readme.opentofu.install.md` & `Readme.opentofu.refcard.md`

Liens officiels OpenTofu

- **Site officiel** : `https://opentofu.org`
- **Documentation** : `https://opentofu.org/docs`
- **Langage** : `https://opentofu.org/docs/language/`
- **CLI** : `https://opentofu.org/docs/cli/`
- **Registry** : `https://registry.opentofu.org`
- **GitHub** : `https://github.com/opentofu/opentofu`
- **Migration TF → OpenTofu** : `https://opentofu.org/docs/intro/migration/`
- **Slack OpenTofu** : `https://opentofu.org/slack`
- **Manifeste OpenTofu** : `https://opentofu.org/manifesto/`

Liens officiels HCL

- **Spec HCL** : `https://github.com/hashicorp/hcl`
- **Stéphane Robert (FR)** : `https://blog.stephane-robert.info/docs/developper/autres-langages/hcl/`

Annexe — Chaines YouTube et blogs

Francophones

- **Stéphane Robert** : `https://blog.stephane-robert.info` (tutoriels TF/OpenTofu/Ansible)
- **Xavki YouTube**: `https://www.youtube.com/@xavki` (DevOps, IaC, K8s)
- **Cocadmin YouTube**: `https://www.youtube.com/@cocadmin`
- **Une Tech YouTube**: `https://www.youtube.com/@unetech`
- **It-Connect** : `https://www.it-connect.fr/`
- **Le Coin DevOps YouTube**: podcast et tutoriels

Anglophones

- **HashiCorp YouTube**: conférences HashiConf
- **OpenTofu YouTube**: workshops officiels
- **Spacelift Blog** : `https://spacelift.io/blog`
- **env0 Blog** : `https://www.env0.com/blog`
- **The Cloud Resume Challenge**
- **TechWorld with Nana YouTube**

Annexe — Refcard rapide OpenTofu

Workflow

- `tofu init [-upgrade]`
- `tofu fmt -recursive`
- `tofu validate`
- `tofu plan -out=tf.plan`
- `tofu apply tf.plan`
- `tofu destroy`
- `tofu output [-json]`
- `tofu console`

State / refactor

- `tofu state list/show/mv/rm`
- `tofu workspace new/select/list`
- `tofu import X ID`
- `tofu refresh` ($\rightarrow$ `apply -refresh-only`)
- `tofu providers schema -json`
- `tofu test [-junit-xml=*]`
- `tofu graph`
- `tofu force-unlock UUID`

Refcard détaillée : `Readme.opentofu.refcard.md` **à la racine du dépôt.**

Remerciements

Merci !

Atelier réalisé pour **Association LinuxMaine**

thierry.gayet@gmail.com

Questions ?