

Définition de Terraform :

"Outil d'Infrastructure as Code (IaC) développé par HashiCorp pour provisionner et gérer des infrastructures cloud de manière déclarative."

L'Infrastructure as Code (IaC) est une pratique DevOps qui consiste à **définir, gérer et provisionner l'infrastructure informatique (serveurs, réseaux, bases de données, etc.) via du code**, plutôt que manuellement.

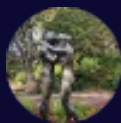


🔧 Outils d'IaC Populaires

- **Terraform** (HashiCorp) : Déploiement multi-cloud "*déclaratif*" (vous décrivez le résultat souhaité).
- **Ansible** : Gestion de configuration "*impérative*" (vous décrivez les étapes).
- **AWS CloudFormation** : Spécifique à AWS.
- **Pulumi** : IaC avec des vrais langages (Python, JavaScript).

🚀 Pourquoi utiliser l'IaC ?

1. **Automatisation** : Plus besoin de configurer manuellement les serveurs.
2. **Reproductibilité** : Le même code produit la même infrastructure à chaque fois.
3. **Collaboration** : Le code peut être versionné avec Git (ex : GitHub).
4. **Éviter les erreurs humaines** : Fini les oublis de configuration !
5. **Scalabilité** : Déployer 1 serveur ou 1000 avec le même code.



1. Création d'une VM Debian 12 sans interface graphique

- Crée un **utilisateur standard** (ex : `terraform`) pendant l'installation.

```
Last login: Mon Jul 21 14:25:34 2025  
terraform@terraform:~$
```

🎯 Objectif du Job 1 :

Avant d'automatiser quoi que ce soit avec Terraform, on a besoin d'une **base propre et fonctionnelle** sur laquelle s'appuyer.

Ce job consiste à **créer une VM Debian 12 minimaliste**, qui servira de **template** pour toutes les futures VMs que Terraform va déployer.

Cela permet d'avoir une **VM de référence** avec tous les éléments de base déjà installés (SSH, sudo, outils VMware), ce qui évite de tout refaire à chaque déploiement.

Installation d'Open-VM-Tools

```
terraform@terraformserveur:~$ sudo apt install open-vm-tools -y
```

PC hôte (machine physique avec VMware Workstation)

Repérer le chemin complet vers le fichier `.vmx` de la VM



Working Directory

Suspend and snapshot files will be stored here.

/home/laurent/vmware/Terraform

Browse...

Retrouver manuellement le chemin sur le disque

```
laurent@debian:~$ sudo find /home -name "*.vmx"
```

```
/home/laurent/vmware/Terraform/Terraform.vmx
```

➡ Ce fichier `.vmx` va servir de base (template) pour que Terraform puisse cloner automatiquement des VMs identiques.



HashiCorp

Terraform

1. Créer le dossier de projet

```
laurent@debian:~$ mkdir ~/terraform_debian_lab  
cd ~/terraform_debian_lab  
laurent@debian:~/terraform_debian_lab$
```

2. Télécharger et installer Terraform (si ce n'est pas déjà fait)

```
laurent@debian:~/terraform_debian_lab$ sudo apt update  
sudo apt install unzip curl -y  
curl -fsSL https://releases.hashicorp.com/terraform/1.8.4/terraform_1.8.4_linux_amd64.zip -o terraform.zip  
unzip terraform.zip  
sudo mv terraform /usr/local/bin/  
terraform -version
```

Your version of Terraform is out of date! The latest version is 1.12.2. You can update by downloading from <https://www.terraform.io/downloads.html>

4. Initialiser Terraform

```
laurent@debian:~/terraform_debian_lab$ cd ~/terraform_debian_lab  
terraform init
```

Terraform has been successfully initialized!

```
laurent@debian:~$ vmrest  
[AppLoader] Use shipped Linux kernel AIO access library.  
An up-to-date "libaio" or "libaio1" package from your system is preferred.  
VMware Workstation REST API  
Copyright (C) 2018-2025 Broadcom.  
All Rights Reserved  
  
vmrest 1.3.1 build-24583834  
-  
Using the VMware Workstation UI while API calls are in progress is not recommend  
ed and may yield unexpected results.  
-  
Serving HTTP on 127.0.0.1:8697  
-  
Press Ctrl+C to stop.
```

✓ 4. Initialiser le projet Terraform

```
laurent@debian:~/terraform_debian_lab$ vmrest -C
```

```
laurent@debian:~/terraform_debian_lab$ vmrest
[AppLoader] Use shipped Linux kernel AIO access library.
An up-to-date "libaio" or "libaio1" package from your system is preferred.
VMware Workstation REST API
Copyright (C) 2018-2025 Broadcom.
All Rights Reserved

vmrest 1.3.1 build-24583834
-
Using the VMware Workstation UI while API calls are in progress is not recommended and may yield unexpected results.
-
Serving HTTP on 127.0.0.1:8697
-
Press Ctrl+C to stop.
```

Démarrer le serveur
API REST et le laisser
fonctionner pour faire le
clonage des VM

PC hôte (machine physique avec VMware Workstation)

✳️ Détail de la commande :

`vmrest` : lance le serveur d'API REST intégré à VMware Workstation.

- `-C` : signifie "**Credential**", elle force la **création ou la modification d'un compte utilisateur** pour accéder à l'API REST.

The screenshot shows the VMware Workstation REST API Explorer interface. The browser address bar displays the URL `127.0.0.1:8697/#!/VM_Management/getAllVMs`. The interface includes a search bar with the text "Rechercher". Below the header, there is a "TRY IT OUT!" button and a "Hide Response" link. The "Curl" section shows the command: `curl 'http://127.0.0.1:8697/api/vms' -X GET --header 'Accept: application/vnd.vmware.vmw.rest-v1+json' --header 'Authorization: Basic dGVyc`. The "Request URL" section shows `http://127.0.0.1:8697/api/vms`. The "Response Body" section displays a JSON array of VM objects:

```
{
  "id": "5FI9I75SC58IB8LI73N31Q34GS3L3DUN",
  "path": "/home/laurent/vmware/Clone of Client/Clone of Client.vmx"
},
{
  "id": "BH006TEJ4K8MUSSB008DRDHH26J4HJN4",
  "path": "/home/laurent/vmware/Clone of client_laurent_DNS/Clone of client_laurent_DNS.vmx"
},
{
  "id": "66KR8ELLN6I1M8MSOPFIV9M5JLK3FVNU",
  "path": "/home/laurent/vmware/Debian 12.x 64-bit (2)/Debian 12.x 64-bit (2).vmx"
},
}
```

✓ main.tf

clonage de la VM

```
GNU nano 7.2
terraform {
  required_providers {
    null = {
      source = "hashicorp/null"
      version = "~> 3.0"
    }
  }
}

provider "null" {}

resource "null_resource" "clone_vm" {
  provisioner "local-exec" {
    command = "bash ./scripts/clone_vm.sh ${var.vm_path} ${var.vm_name}"
  }
}
```

✓ variables.tf

Tu dois maintenant définir les deux variables utilisées (`vm_path` et `vm_name`) :

```
GNU nano 7.2
variable "vm_path" {
  description = "Chemin vers la VM source à cloner"
  type        = string
  default     = "/home/laurent/vmware/Terraform/Terraform.vmx"
}

variable "vm_name" {
  description = "Nom de la nouvelle VM clonée"
  type        = string
  default     = "Clone_Terraform_VM"
}
```

✓ scripts/clone_vm.sh

```
laurent@debian:~/terraform_debian_lab/scripts$ cat clone_vm1.sh
#
#!/bin/bash
SOURCE_VM="$1"
NEW_VM_NAME="$2"
VM_DIR=$(dirname "$SOURCE_VM")

cp -r "$VM_DIR" "${VM_DIR}_${NEW_VM_NAME}"
echo "Clonage terminé vers ${VM_DIR}_${NEW_VM_NAME}"
```

✓ Rendez le script exécutable :

laurent@debian:~/terraform_debian_lab/scripts\$ sudo chmod +x scripts/clone_vm.sh




```
laurent@debian:~/terraform_debian_lab$ terraform init
```

```
Initializing the backend...
```

```
Initializing provider plugins...
```

```
- Reusing previous version of hashicorp/null from the dependency lock file  
- Using previously-installed hashicorp/null v3.2.4
```

```
Terraform has been successfully initialized!
```

◆ terraform init

- **But** : initialise le projet Terraform.
- **Fait quoi ?** : télécharge les plugins nécessaires (providers), prépare le dossier `.terraform/`.

✚ **À faire une seule fois** au début (ou quand tu ajoutes un nouveau provider).

◆ terraform plan

- **But** : montre ce que Terraform va faire.
- **Fait quoi ?** : compare le code `.tf` avec ce qui est déjà déployé, et affiche les actions prévues (ajouter, modifier, supprimer).

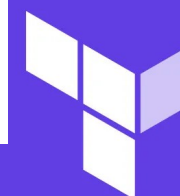
✚ **Aucune modification réelle**, c'est une **simulation**.

```
laurent@debian:~/terraform_debian_lab$ terraform plan
```

```
null_resource.clone_vm: Refreshing state... [id=6664252717971100550]
```

```
No changes. Your infrastructure matches the configuration.
```

```
Terraform has compared your real infrastructure against your configuration and found no differences, so no changes are needed.
```



HashiCorp

Terraform

```
laurent@debian:~/terraform_debian_lab$ terraform apply

Terraform used the selected providers to generate the following execution plan. Resource actions are indicated with the following symbols:
+ create

Terraform will perform the following actions:

# null_resource.clone_vm will be created
+ resource "null_resource" "clone_vm" {
  + id = (known after apply)
}
```

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

Do you want to perform these actions?
Terraform will perform the actions described above.
Only 'yes' will be accepted to approve.

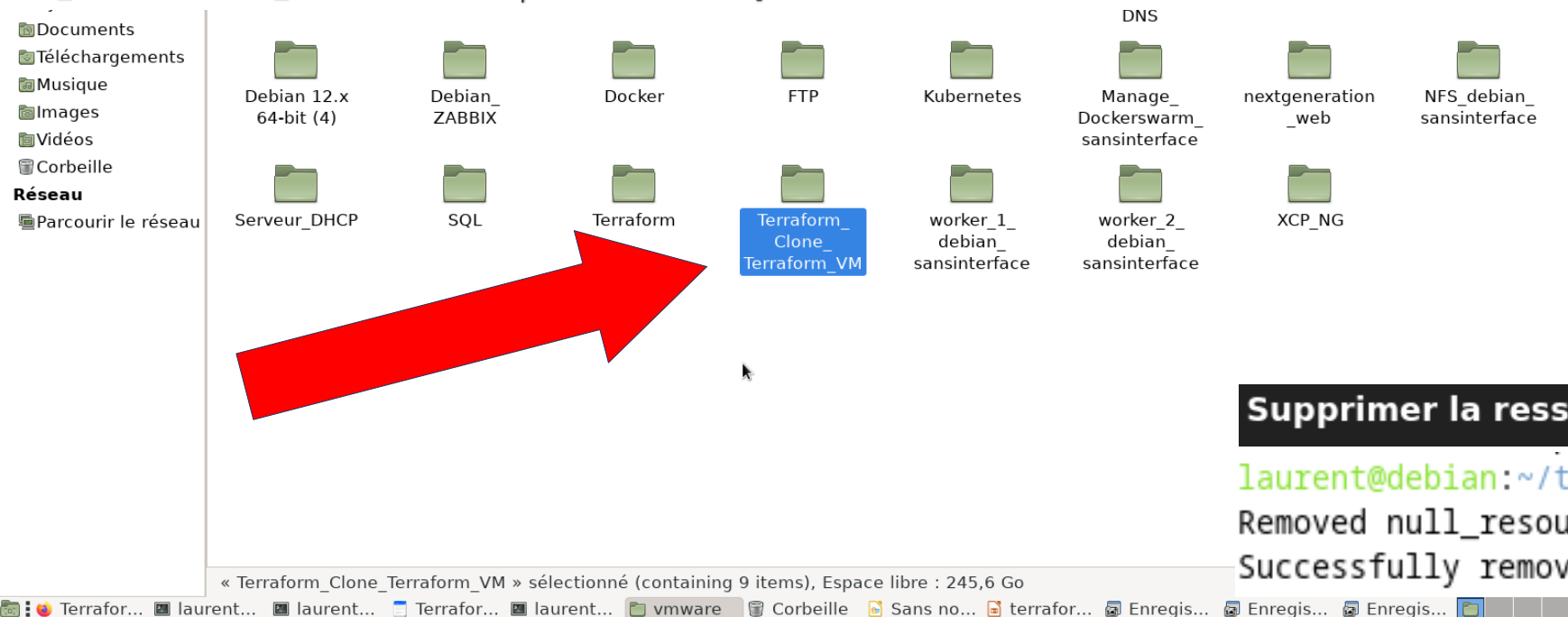
Enter a value: yes

◆ terraform apply

- **But** : applique les changements.
- **Fait quoi ?** : exécute réellement les actions prévues par plan (création, modification ou suppression de ressources).

✖ Il demande confirmation **sauf si** tu utilises `terraform apply -auto-approve`.

```
null_resource.clone_vm: Creating...
null_resource.clone_vm: Provisioning with 'local-exec'...
null_resource.clone_vm (local-exec): Executing: ["/bin/sh" "-c" "bash ./scripts/clone_vm.sh /home/laurent/vmware/Terraform/Terraform.vmx Clone_Terraform_VM"]
null_resource.clone_vm: Still creating... [10s elapsed]
null_resource.clone_vm (local-exec): Clonage terminé vers /home/laurent/vmware/Terraform_Clone_Terraform_VM
null_resource.clone_vm: Creation complete after 14s [id=6664252717971100550]
```



```
# Initialisez à nouveau
terraform init

# Vérifiez le plan
terraform plan

# Appliquez
terraform apply
```

Supprimer la ressource du fichier d'état Terraform

```
laurent@debian:~/terraform_debian_lab$ terraform state rm null_resource.clone_vm
Removed null_resource.clone_vm
Successfully removed 1 resource instance(s).
```

AMELIORATION ET GESTION DU CYCLE DE VIE

1. Utilisation avancée des variables (variables.tf)

```
# Variables obligatoires avec validation
variable "vm_template_path" {
  type      = string
  description = "Chemin absolu du template .vmx"
  validation {
    condition     = fileexists(var.vm_template_path)
    error_message = "Le fichier .vmx spécifié n'existe pas"
  }
}
```

```
# Variables optionnelles avec valeurs par défaut
variable "vm_settings" {
  type = object({
    cpus      = number
    memory    = number
    network   = string
  })
  default = {
    cpus      = 2
    memory    = 2048 # MB
    network   = "NAT"
  }
}
```

```
# Variables sensibles (marquées comme telles)
variable "provisioning_ssh_key" {
  type      = string
  sensitive = true
}
```

2. Gestion des modifications (main.tf)

```
# Configuration modulaire avec lifecycle
resource "vmware_vm" "debian_vm" {
  name      = "debian-${formatdate("YYYYMMDD", timestamp())}"
  template  = var.vm_template_path
  cpus      = var.vm_settings.cpus
  memory    = var.vm_settings.memory

  network_interface {
    network = var.vm_settings.network
  }

  lifecycle {
    ignore_changes = [
      # Empêche la recreation si modification manuelle
      annotation,
      network_interface[0].mac_address
    ]
    create_before_destroy = true # Zero Downtime
  }
}
```

```
# Provisioning conditionnel
provisioner "remote-exec" {
  when      = create
  inline    = ["sudo apt update && sudo apt upgrade -y"]
}
}
```


3. Destruction contrôlée (`destroy.tf`)

```
# Nettoyage avant destruction
resource "null_resource" "pre_destroy" {
  triggers = {
    vm_id = vmware_vm.debian_vm.id
  }

  provisioner "local-exec" {
    when      = destroy
    command = <<EOT
      vmrun stop ${self.triggers.vm_id}
      sleep 10
      vmrun deleteVM ${self.triggers.vm_id}
    EOT
  }
}
```

```
# Sortie utile pour le CI/CD
output "destruction_clean" {
  value      = "Resources marked for clean deletion"
  description = "Confirme que les hooks de destruction sont configurés"
}
```

Workflow d'exécution

```
# Initialisation
terraform init -upgrade

# Planification avec variables
terraform plan -var="vm_template_path=/path/to/debian.vmx" -out=tfplan

# Application sécurisée
terraform apply tfplan

# Destruction propre
terraform destroy -auto-approve
```

Bonnes pratiques implémentées :

1. **Validation des entrées** : Vérification du fichier .vmx
2. **Sécurité** : Marquage des variables sensibles
3. **Stabilité** : `lifecycle` pour éviter les créations intempestives
4. **Nettoyage** : Arrêt propre de la VM avant suppression
5. **Idempotence** : Provisioning conditionnel avec `when=create`

Cette configuration offre une gestion professionnelle du cycle de vie tout en restant compatible avec vos exigences initiales.

Laboratoire de Cybersécurité simple

main.tf

```
GNU nano 7.2
terraform {
  required_providers {
    null = {
      source  = "hashicorp/null"
      version = "~> 3.0"
    }
  }
}

provider "null" {}

# Clone VM 1 - Kali-like
resource "null_resource" "clone_kali_like" {
  provisioner "local-exec" {
    command = "bash ./scripts/clone_vm.sh ${var.vm_path} kali-like-vm"
  }

  provisioner "remote-exec" {
    connection {
      type      = "ssh"
      user      = "laurent"
      private_key = file("~/ssh/id_rsa")
      host      = var.kali_ip
    }

    inline = [
      "sudo apt update",
      "sudo apt install -y nmap masscan"
    ]
  }

  depends_on = [null_resource.clone_kali_like]
}

# Clone VM 2 - Victim
resource "null_resource" "clone_victim" {
  provisioner "local-exec" {
```

✓ Objectif :

Créer un fichier `main.tf` qui :

- déploie **2 VMs Debian 12** (une « attaquante », une « victime »),
- installe automatiquement quelques outils/services via `remote-exec`,
- place les VMs dans le **même réseau** (ex : `VMnet8` en NAT local),
- permet l'interaction réseau entre les deux.

main.tf

Contient le cœur de la configuration Terraform (provider, ressource `vsphere_virtual_machine`, provisioners...)

✓ Ce qu'il faut utiliser

Pour **VMware Workstation**, le provider à utiliser est :

[Copier](#) [Modifier](#)

terraform-provider-vmware

Mais **il n'existe pas officiellement** un provider officiel de VMware pour VMware Workstation. Il faut donc passer par une solution **non-officielle** ou **via des images déjà préparées** avec un `provisioner` local (type `virtualbox`, ou par commande `vmrun`, ou via `remote-exec` sur SSH).

Mais il existe une alternative à base de `null_resource` + `vmrun` + `vmrest`, combinée à un script local pour cloner une VM existante (fichier `.vmx`) et la personnaliser.

terraform

main.tf

```
provisioner "remote-exec" {
  connection {
    type      = "ssh"
    user      = "laurent"
    private_key = file("~/ssh/id_rsa")
    host      = var.victim_ip
  }

  inline = [
    "sudo apt update",
    "sudo apt install -y python3",
    "nohup python3 -m http.server 8080 &"
  ]
}

depends_on = [null_resource.clone_victim]
```

Laboratoire de Cybersécurité simple

variables.tf

```
GNU nano 7.2
variable "vsphere_user" {
  description = "Nom d'utilisateur pour vSphere"
  type        = string
}

variable "vsphere_password" {
  description = "Mot de passe pour vSphere"
  type        = string
  sensitive   = true
}

variable "vsphere_server" {
  description = "Adresse IP ou nom du serveur vSphere"
  type        = string
}

variable "datacenter_name" {
  description = "Nom du datacenter vSphere"
  type        = string
}

variable "datastore_name" {
  description = "Nom du datastore vSphere"
  type        = string
}

variable "vm_template" {
  description = "Nom du template de VM à cloner"
  type        = string
}

variable "vm_name" {
  description = "Nom de la nouvelle VM clonée"
  type        = string
  default     = "Clone_Terraform_VM"
```

terraform.tfvars

```
GNU nano 7.2
vsphere_user      = "terraform"
vsphere_password  = "@Lo29031974"
vsphere_server    = "192.168.1.10"

datacenter_name   = "Datacenter"
datastore_name    = "datastore1"

vm_template       = "Template-Debian"
network_name      = "VM Network"
```

terraform.tfvars

Contient les **valeurs concrètes** à affecter aux variables définies dans
variables.tf

variables.tf

Déclare toutes les variables utilisées dans main.tf



hiCoro

variables.tf


```
variable "vm_name" {  
  description = "Nom de la nouvelle VM clonée"  
  type        = string  
  default     = "Clone_Terraform_VM"  
}  
  
variable "vm_path" {  
  description = "Chemin vers la VM source à cloner (VMware Workstation par exemple)"  
  type        = string  
  default     = "/home/laurent/vmware/Terraform/Terraform.vmx"  
}  
  
variable "network_name" {  
  description = "Nom du réseau à utiliser pour la VM"  
  type        = string  
}
```

 Ce projet est un échec !!!

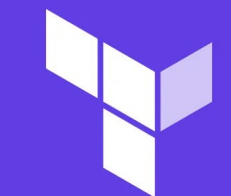
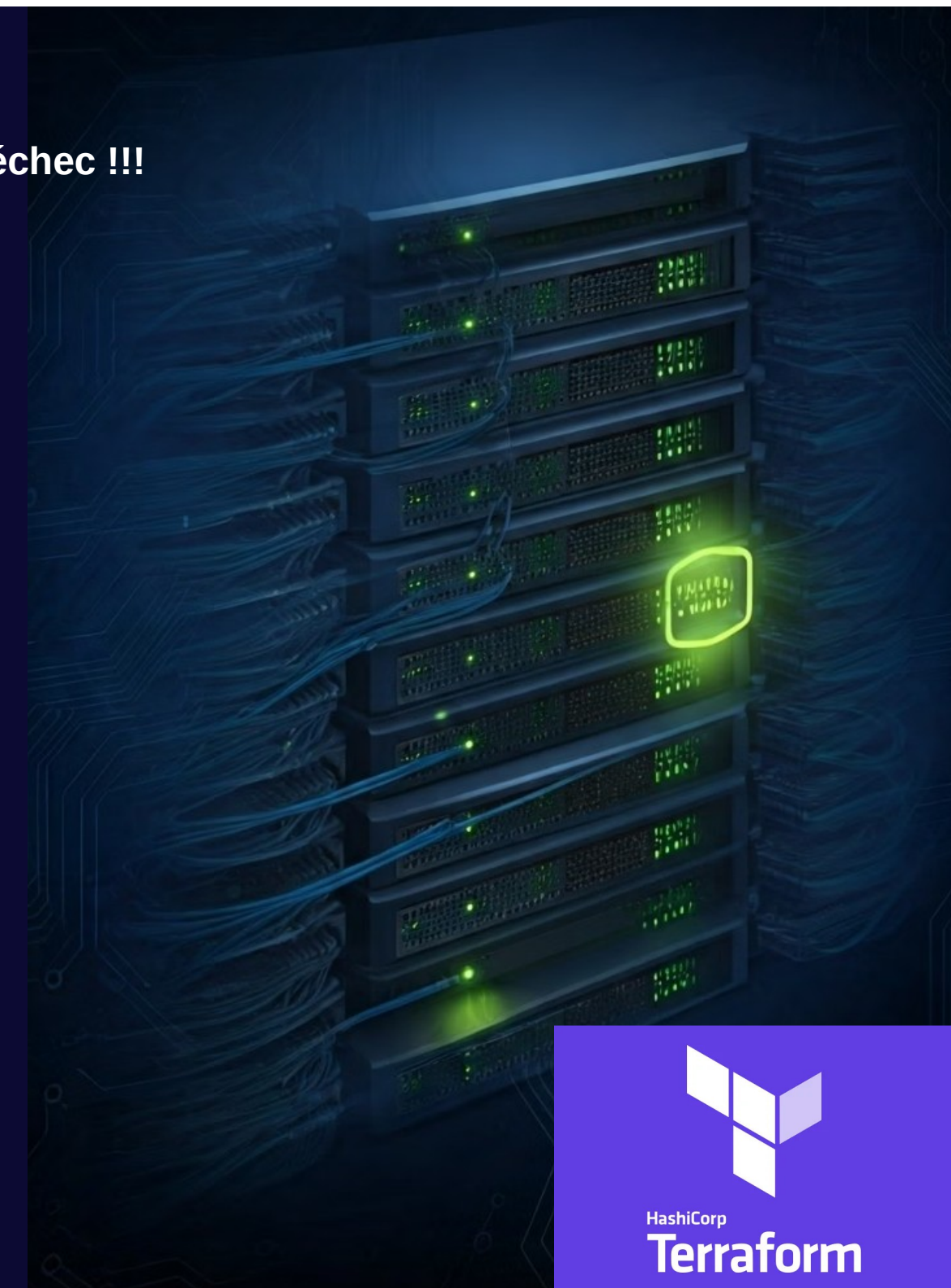
Nouveau projet !

Structure de ton projet Terraform

bash

 Copier  Modifier

```
terraform_debian_lab/  
|  
├─ main.tf  
├─ variables.tf  
├─ terraform.tfvars (optionnel)  
└─ debian12.vmx (le fichier source à cloner, base Debian 12)
```



HashiCorp

Terraform

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

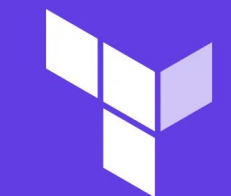
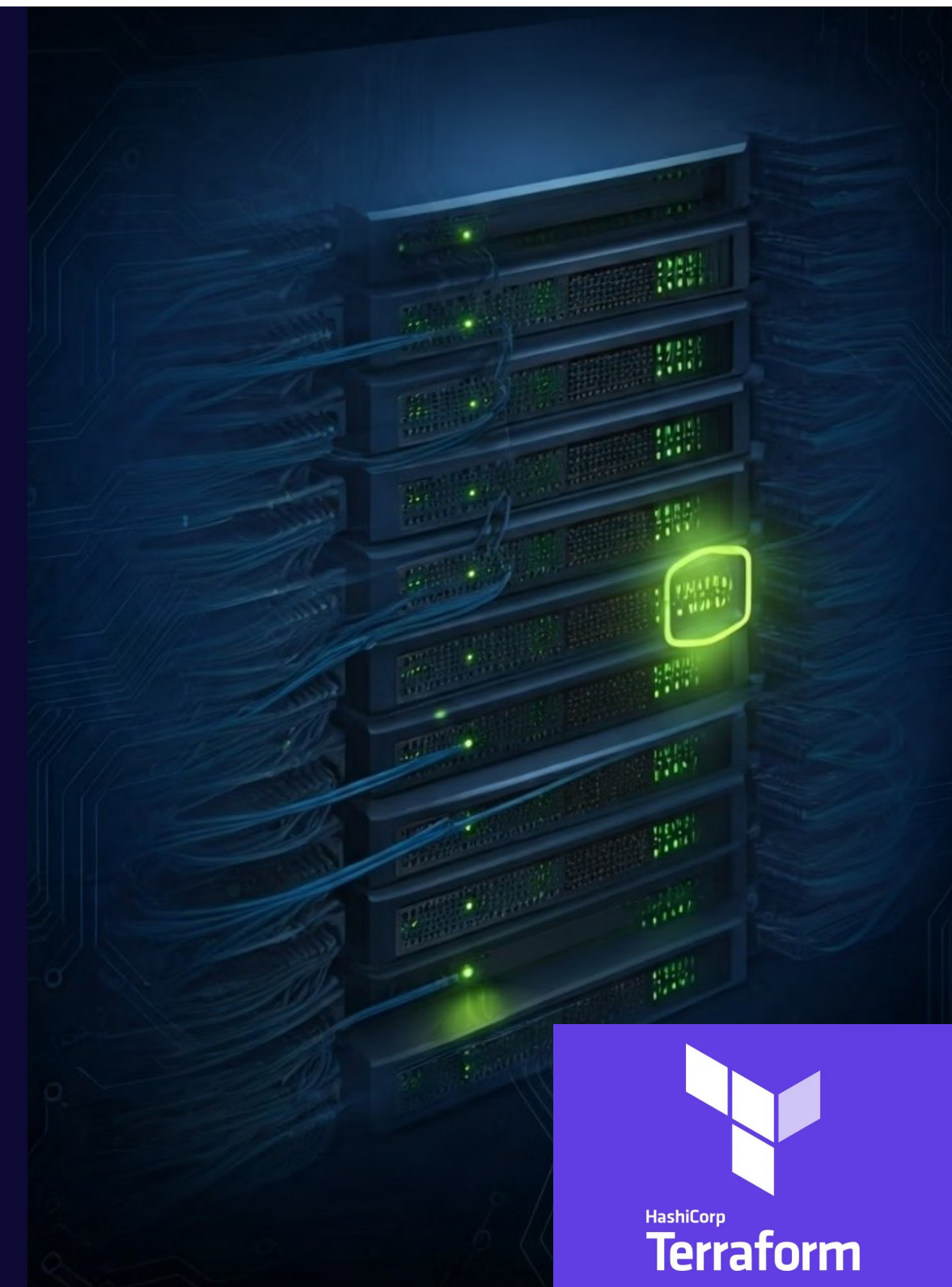
provider "null" {}

variable "vmx_path" {
  description = "Chemin vers la VMX source Terraform"
  type        = string
  default     = "/home/laurent/vmware/Terraform/Terraform.vmx"
}

variable "network" {
  description = "Nom du réseau VMware (ex: VMnet8)"
  type        = string
  default     = "VMnet8"
}

resource "null_resource" "create_kali_vm" {
  provisioner "local-exec" {
    command = <<EOT
curl -s -X POST http://127.0.0.1:8697/api/vms -H "Content-Type: application/json" -u 'laurent:MDP' -d '{
  "path": "${var.vmx_path}",
  "name": "kali-like-vm",
  "network_adapters": [{"type": "nat", "network": "${var.network}"}]
}'
EOT
  }
}

resource "null_resource" "create_victim_vm" {
  provisioner "local-exec" {
    command = <<EOT
curl -s -X POST http://127.0.0.1:8697/api/vms -H "Content-Type: application/json" -u 'laurent:ton_mdp' -d '{
  "path": "${var.vmx_path}",
  "name": "victim-vm",
  "network_adapters": [{"type": "nat", "network": "${var.network}"}]
}'
EOT
  }
}
```



HashiCorp
Terraform

variables.tf *

GNU nano 7.2

```
variable "vm_template_path" {
  default = "/home/laurent/vmware/Terraform/Terraform.vmx"
}

variable "vm_network" {
  default = "VMnet8"
}

variable "ssh_username" {
  default = "laurent"
}

variable "ssh_private_key" {
  default = "~/.ssh/id_rsa"
}
```

terraform.tfvars *

GNU nano 7.2

```
ssh_username = "laurent"
ssh_private_key = "~/.ssh/id_rsa"
```

```
provider "vmware" {
  vmrest_url = "http://127.0.0.1:8697"
  allow_unverified_ssl = true
}

resource "vmware_vmx" "kali" {
  name = "kali-like-vm"
  source_path = var.vm_template_path
  clone = true

  network_interface {
    network = var.vm_network
  }

  provisioner "remote-exec" {
    inline = [
      "sudo apt update",
      "sudo apt install -y nmap masscan"
    ]

    connection {
      type = "ssh"
      user = var.ssh_username
      private_key = file(var.ssh_private_key)
      host = self.ip_address
    }
  }
}

resource "vmware_vmx" "victim" {
  name = "victim-vm"
  source_path = var.vm_template_path
  clone = true

  network_interface {
    network = var.vm_network
  }

  provisioner "remote-exec" {
    inline = [
      "sudo apt update",
      "sudo apt install -y python3",
      "nohup python3 -m http.server 8080 &"
    ]
  }
}
```

main.tf

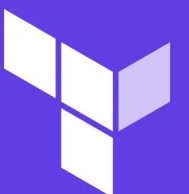
```
connection {
  type = "ssh"
  user = var.ssh_username
  private_key = file(var.ssh_private_key)
  host = self.ip_address
}
```

outputs.tf *

GNU nano 7.2

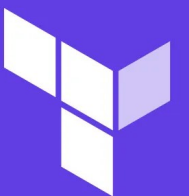
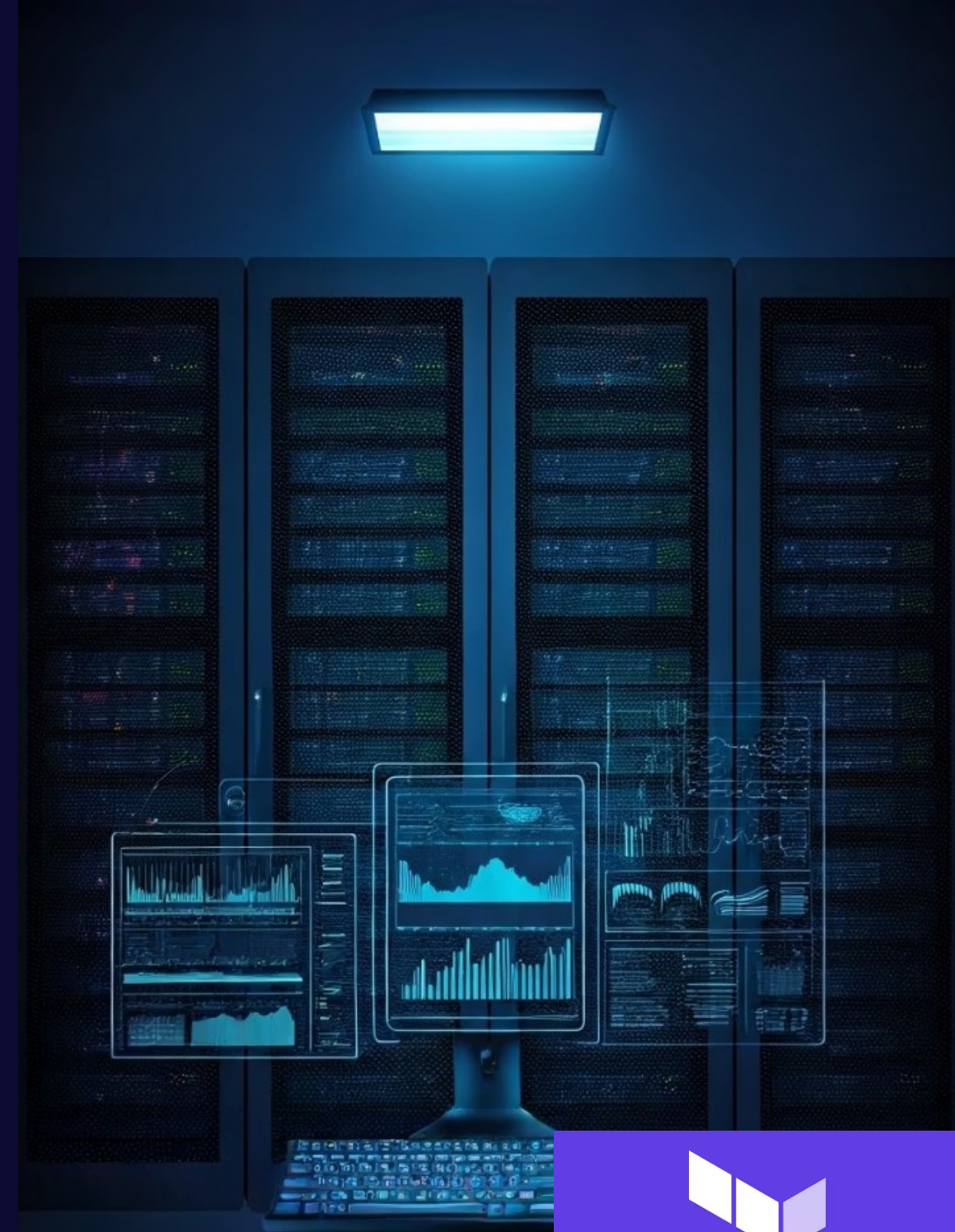
```
output "kali_ip" {
  value = vmware_vmx.kali.ip_address
}

output "victim_ip" {
  value = vmware_vmx.victim.ip_address
}
```



HashiCorp

Terraform



HashiCorp
Terraform