Lab 2 : Managing Infrastructure as Code (IaC)

Section 1: Authenticating to AWS on the Command Line

Objectif de l'étape : L'objectif est de créer des clés d'accès IAM. Elles permettent d'établir une connexion sécurisée entre l'ordinateur local et le compte AWS pour pouvoir gérer les ressources depuis un terminal local. Ensuite, il faut configurer les variables d'environnement sur la machine locale pour qu'elle utilise automatiquement les clés d'accès lors de l'exécution de commandes AWS.

Protocole suivi :

Se connecter à la console de gestion AWS et naviguer vers la console IAM.
Sélectionner l'utilisateur IAM dans la section Utilisateurs.
Accéder à l'onglet Information d'identification de sécurité (Security Credentials).
Créer une nouvelle clé d'accès en sélectionnant explicitement le cas d'utilisation : "Interface de ligne de commande (CLI)".
Fournir une description ("lab2" par exemple).
Récupérer et retenir l'ID de clé d'accès (Access Key ID) et la Clé d'accès secrète (Secret Access Key).

Définir les variables d'environnement dans le terminal à l'aide des commandes :

- export AWS_ACCESS_KEY_ID=<id de la clé d'accès>
- export AWS_SECRET_ACCESS_KEY_ID=<id de la clé secrète>

```
lubin@benoitl ~
$ aws configure list
NAME       : VALUE                    : TYPE             : LOCATION
profile    : <not set>                : None             : None
access_key : ****************O6TF     : env              :
secret_key : ****************tmN1     : env              :
region     : <not set>                : None             : None
```

Explication : L'ID de clé d'accès sert d'identifiant public et la clé secrète de mot de passe. Elles servent à s'authentifier auprès des services AWS depuis le terminal. En exportant les clés dans le terminal, les outils comme l'AWS CLI ou OpenTofu peuvent authentifier les requêtes envoyées au cloud AWS sans qu'on ait à saisir les clés à chaque action.

Résumé (ce que j'ai retenu) :

Nous avons créé des clés d'accès IAM et avons configuré le terminal avec ces clés pour pouvoir envoyer des requêtes au cloud AWS avec authentification automatique.

Section 2: Deploying an EC2 Instance Using a Bash Script

Objectif de l'étape : Automatiser le déploiement d'une infrastructure AWS en utilisant des scripts "Ad hoc".

Protocole suivi :

Créer le répertoire devops_base/lab2/scripts/bash/ et s'y rendre.
Créer le script user-data.sh avec le contenu indiqué dans le sujet du lab.
Créer le script principal deploy-ec2-instance.sh avec le contenu indiqué dans le sujet du lab.
Rendre le script exécutable : chmod u+x deploy-ec2-instance.sh (commande linux).
Modifier le contenu du script :

- Utiliser le lien GitHub suivant : https://raw.githubusercontent.com/BTajini/devops-base/0389423c39cae0c094c394824f2e2cb88c0f763c/td1/sample-app/app.js
- Remplacer le port 80 par le port 8080 utilisé par l'app pour écouter.

Supprimer le groupe de sécurité préexistant avant chaque exécution du script.
Exécuter le script principal ./deploy-ec2-instance.sh.
Vérifier dans un navigateur que le site est opérationnel.
Résilier les instances après utilisation.

Explication : Le script "User Data" permet de définir les instructions que l'instance EC2 doit exécuter automatiquement lors de son premier démarrage, comme l'installation des dépendances. nohup permet à l'application de continuer à s'exécuter même après avoir terminé l'exécution du script. Le script deploy-ec2-instance.sh permet d'orchestrer la création du groupe de sécurité et de l'instance EC2 via l'interface de ligne de commande AWS CLI. Le groupe de sécurité permet de contrôler le trafic réseau entrant et sortant de l'instance.

Exercise 1: What happens if you run the script a second time? Try it and observe the output. Explain why this happens.

On ne peut pas déployer plusieurs instances juste en relançant le script, car AWS n'autorise pas deux Security Groups avec le même nom dans la même VPC. Le script n'est pas idempotent.

Exercise 2: Modify the script to deploy multiple EC2 instances. How would you adjust the script to handle this?

Le script actuel déploie une seule instance EC2. Pour gérer le déploiement de plusieurs instances, il suffit d'ajouter un paramètre de nombre d'instances (par exemple INSTANCE_COUNT) et d'utiliser l'option --count de la commande aws ec2 run-instances, ou bien une boucle pour lancer plusieurs instances. Le script doit ensuite attendre l'état running pour l'ensemble des instances et récupérer leurs adresses IP publiques.

Résumé (ce que nous avons retenu) :

Cette section a mis en pratique le premier type d'outil IaC : les scripts Ad hoc Bash. Utiliser un script Bash pour déployer une instance EC2 sur AWS est plus pratique que de configurer manuellement l'instance sur la console AWS. Mais le problème majeur est qu'après avoir exécuté le script une fois, il échouera une seconde fois car il tente de créer des ressources (groupe de sécurité) qui existent déjà avec un nom unique dans AWS. Les scripts Bash ne conservent pas de mémoire de l'infrastructure déployée, ils sont non idempotents.

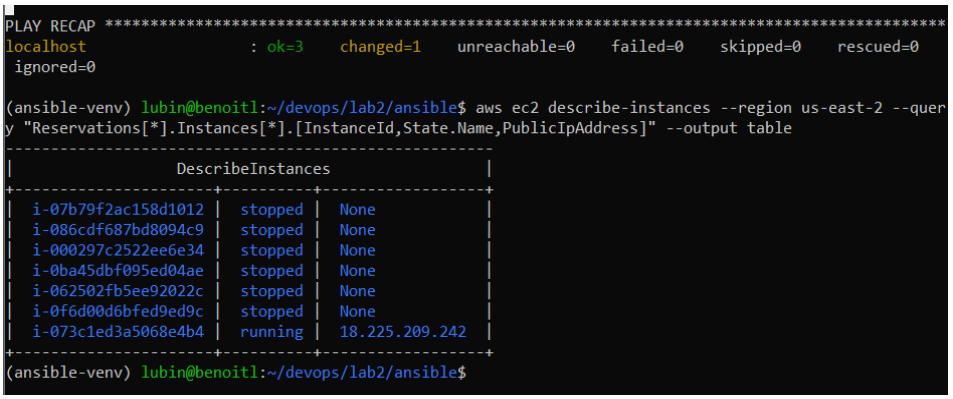
Section 3: Deploying an EC2 Instance Using Ansible

Objectif de l'étape : Utiliser un outil de gestion de configuration (Ansible) pour créer les ressources AWS (instances, groupe de sécurité et clés SSH).

Protocole suivi :

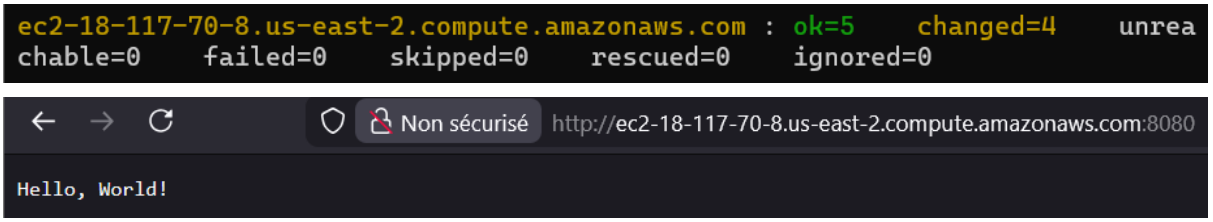
(Nous avons utilisé l'environnement WSL à la place de Cygwin sur VS Code à partir de ce moment pour des raisons de compatibilité et de simplicité)

1/ Création des ressources AWS
Créer le fichier de playbook create_ec2_instance_playbook.yml avec le contenu fourni par le sujet du lab.
Vérifier que l'AMI est correct avec la commande aws ec2 describe-images et les filtres nécessaires (ID de l'image, région, requête sur le nom et la plateforme), en regardant la compatibilité avec l'architecture (x86_64) et l'OS (Amazon Linux 2023).
Configurer l'instance avec le type t3.micro pour garantir l'éligibilité au niveau gratuit d'AWS. En effet, la commande aws ec2 describe-instance-types avec les filtres de région et free-tier-eligible fournit les types d'instances possibles.
Lancer la création de l'infrastructure et le déploiement avec la commande : ansible-playbook -v create_ec2_instance_playbook.yml



2/ Configuration de l'application sur l'instance
Configurer un inventaire dynamique avec inventory.aws_ec2.yml avec le contenu fourni par le sujet du lab.
Définir les variables de connexion dans group_vars/ch2_instances.yml avec le contenu fourni par le sujet du lab (utilisateur ec2-user, clé SSH).
Créer un rôle Ansible (sample-app) comprenant les tâches d'installation de Node.js, la copie du fichier app.js et le lancement de l'application avec nohup. Autrement dit, créer les fichiers : roles/sample-app/files/app.js en reprenant l'app du dépôt GitHub (cf section 2), et roles/sample-app/tasks/main.yml avec le contenu fourni par le sujet du lab.

Créer le playbook de configuration configure_sample_app_playbook.yml avec le contenu fourni par le sujet du lab.
Après avoir lancé le playbook de création de l'infrastructure, vérifier son état :
aws ec2 describe-instances --region us-east-2 --query "Reservations[*].Instances[*].[InstanceId, State.Name, PublicDnsName]" --output table
Lancer le playbook de configuration de l'application :
ansible-playbook -v -i inventory.aws_ec2.yml configure_sample_app_playbook.yml



Explication : Le playbook de création create_ec2_instance_playbook.yml utilise des modules Ansible spécialisés pour déclarer l'état souhaité des ressources AWS, contrairement au script Bash de la section précédente. Il crée un groupe de sécurité ouvrant les ports 8080 et 22, génère la paire de clés SSH pour l'accès et lance l'instance en lui donnant un tag. L'inventaire dynamique inventory.aws_ec2.yml regroupe toutes les instances possédant le tag généré précédemment, dans un seul groupe. Les variables de groupe group_vars/ch2_instances.yml définissent les paramètres de connexion SSH. Les informations de connexion sont ainsi séparées de la logique du déploiement. Le playbook de configuration configure_sample_app_playbook.yml orchestre le déploiement.

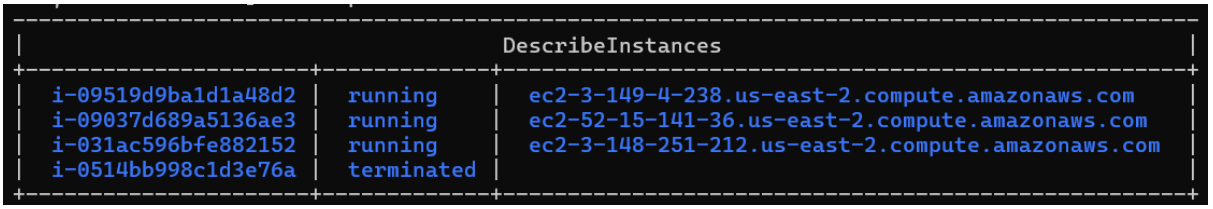
Exercise 3: What happens if you run the configuration playbook a second time? Observe and explain.

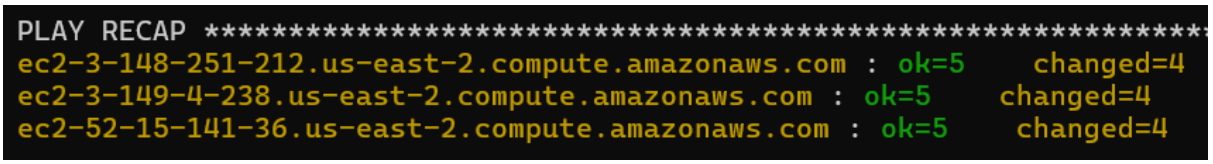
Seules les tâches non idempotentes ont changé l'état (changed: true). Ansible est conçu pour être idempotent, mais cela dépend de la façon dont les tâches sont écrites. Les modules natifs d'Ansible (yum, copy, template) sont généralement idempotents, et les commandes shell brutes (nohup node app.js &) ne le sont pas.

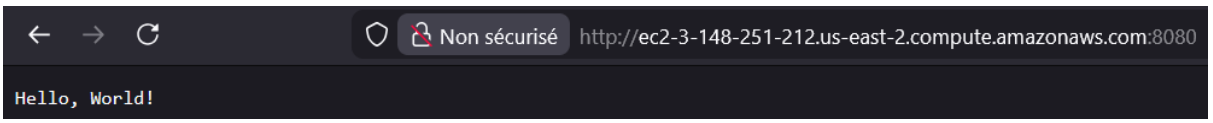
- App Node packages to yum (non idempotente) : relance le script sans impact majeur
- Install Node.js (idempodente) : Ne fait rien si Node.js est déjà installé.
- Copy sample app (idempotente) : Ne copie pas le fichier s'il est déjà présent et inchangé
- Start sample app (non idempotente) : Essaie de relancer l'application (erreur EADDRINUSE).

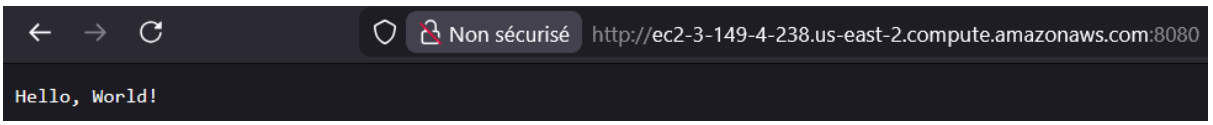
Exercise 4: Modify the playbook to deploy and configure multiple EC2 instances. How would you adjust the playbook and inventory?

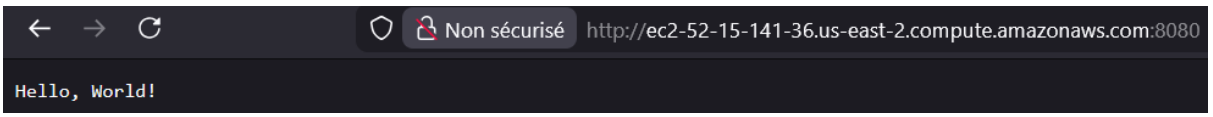
Dans create_ec2_instance_playbook.yml, il suffit d'ajouter un paramètre count dans amazon.aws.ec2_instance. Par exemple, "count: 3" crée 3 instances EC2, chacune avec une ip publique différente.











Les instances EC2 sont résiliées via la commande terminate-instances. Une fois les instances supprimées, la key pair et le security group associés ont été supprimés pour éviter d'être facturés par AWS inutilement.

Résumé (ce que nous avons retenu) :

L'outil Ansible pour déployer une instance EC2 sur AWS permet de séparer la création du serveur (provisionnement) de son paramétrage logiciel (configuration). Utiliser des inventaires dynamiques et des rôles rend l'infra plus facile à maintenir et à faire évoluer que les scripts manuels. Ansible garantit que si les fichiers sont relancés, le résultat final sera toujours identique et stable (idempotence).

Section 4: Creating a VM Image Using Packer

Objectif de l'étape : L'objectif ici est d'utiliser Packer, un outil de "server templating", pour créer une Amazon Machine Image (AMI). Contrairement à la section précédente où l'on configurait une instance existante, Packer permet de créer une image où l'application est déjà préinstallée.

Protocole suivi :

(Nous avons créé un environnement virtuel pour Packer sur WSL)
Créer la structure de répertoires devops_base/lab2/scripts/packer/ et y copier le fichier app.js.
Créer le fichier de configuration sample-app.pkr.hcl.
Dans le template, définir le type d'instance sur t3.micro pour assurer l'éligibilité au "Free Tier" d'AWS.
Initialiser Packer pour télécharger les plugins requis : packer init sample-app.pkr.hcl
Lancer la création de l'image : packer build sample-app.pkr.hcl

```
==> Wait completed after 4 minutes 9 seconds

==> Builds finished. The artifacts of successful builds are:
--> amazon-ebs.amazon_linux: AMIs were created:
us-east-2: ami-0b3684a598f153d83

(packer-venv) lubin@benoitl:~/devops/lab2/packer$
```

```
----------------------------------------------------------------------------------
|                                 DescribeImages                                 |
+------------------------+---------------------------------------------------------+
|  ami-0b3684a598f153d83 |  sample-app-packer-870a23cd-a5f2-4600-b5b0-aa2f7a4328a5  |
+------------------------+---------------------------------------------------------+
```

Explication : Le fichier .pkr.hcl permet de configurer l'espace de travail local et définir les spécifications de l'image à construire. Lors du build, Packer effectue plusieurs actions automatiques : il lance une instance EC2 temporaire, y transfère le fichier app.js (Il copie app.js depuis le WSL local .../packer/app.js vers /home/ec2-user/app.js sur l'instance temporaire Packer), puis installe Node.js. Il crée ensuite une AMI à partir de cette instance et supprime l'instance temporaire pour éviter les coûts inutiles.

Exercise 5: What happens if you run packer build a second time? Explain why.
Hint: Consider the uniqueness of the ami_name .

Lors d'une seconde exécution de packer build, une nouvelle AMI est créée. Cela s'explique par l'utilisation de ${uuidv4()} dans le paramètre ami_name, qui garantit un nom unique à chaque build. AWS considère donc chaque exécution comme une image distincte.

Exercise 6: Modify the Packer template to create images for another cloud provider or for local use (e.g., VirtualBox)

- Ajouter le plugin Virtual Box
- Ajouter une source VirtualBox
- Étendre la section build

Le builder VirtualBox est ajouté dans le template Packer.

Supprimer les AMI enregistrées après utilisation (à la fin du lab).

Résumé (ce que nous avons retenu) :

Packer permet de créer des images immuables. Une fois l'AMI générée, on peut déployer autant de serveurs que nécessaire sans avoir à réinstaller les logiciels à chaque fois, ce qui rend le déploiement rapide et fiable.

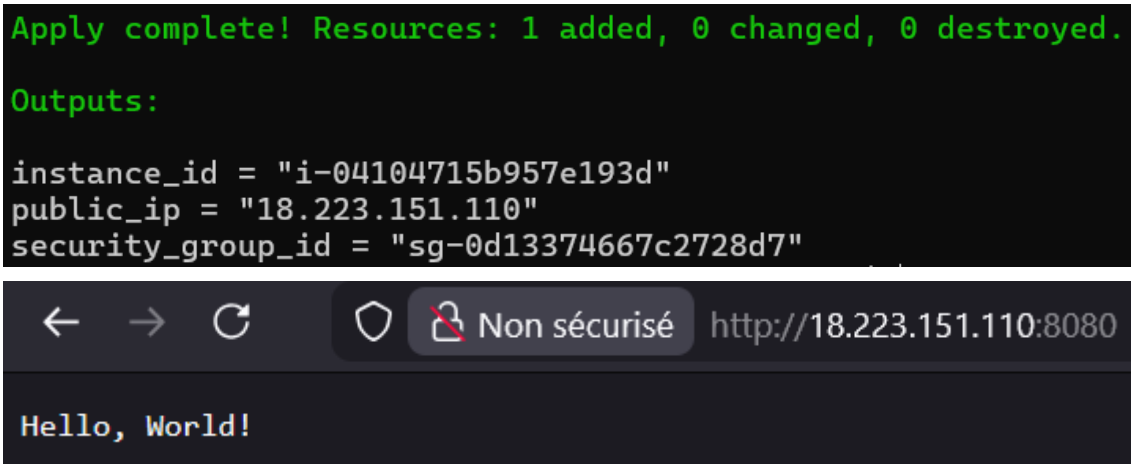
Section 5: Deploying, Updating, and Destroying an EC2 Instance Using OpenTofu

Objectif de l'étape : Découvrir et utiliser OpenTofu, un outil de provisionnement pour automatiser le cycle de vie d'une infrastructure cloud (ressources AWS).

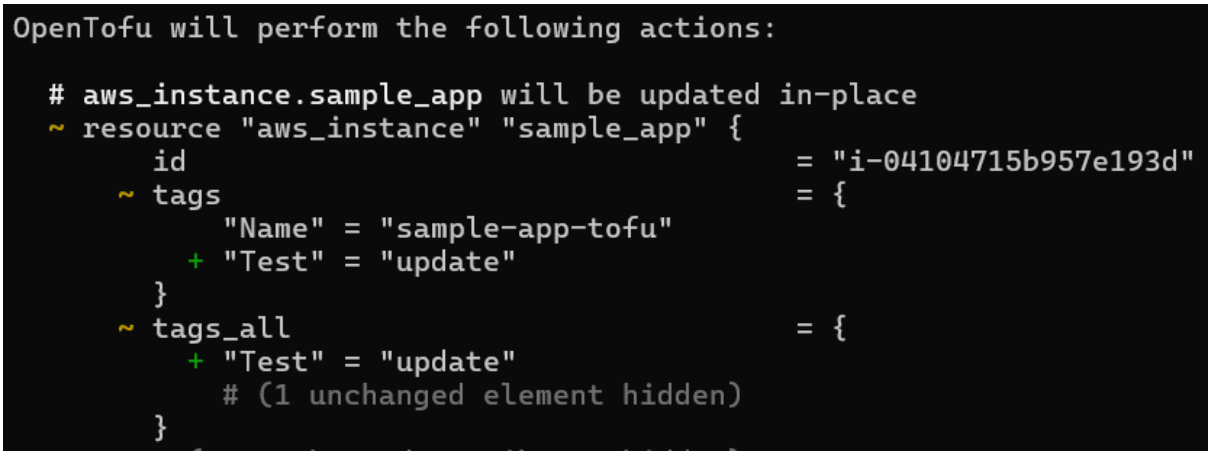
Protocole suivi :

1/ Préparation des fichiers
(Nous avons utilisé l'environnement local WSL et installé les dépendances.)
Créer le répertoire devops_base/lab2/scripts/tofu/ec2-instance/ et s'y rendre.
Créer le fichier main.tf pour définir le fournisseur (AWS), le groupe de sécurité (port 8080) et l'instance EC2. Définir le type d'instance en t3.micro comme dans les sections précédentes.
Créer variables.tf pour paramétrer l'ID de l'AMI et outputs.tf pour extraire l'IP publique après le déploiement.
Créer user-data.sh pour automatiser le lancement de l'application Node.js au démarrage.

2/ Initialisation et déploiement
Initialiser le répertoire de travail avec : tofu init
Appliquer la configuration avec : tofu apply
Saisir l'ID de l'AMI généré lors de la section 4 (Packer) dans le terminal.
Tester l'accès sur l'URL publique, sur le port 8080.



3/ Mise à jour de l'infrastructure sur place
Modifier main.tf pour ajouter un tag supplémentaire : Test = "update"
Relancer la commande : tofu apply



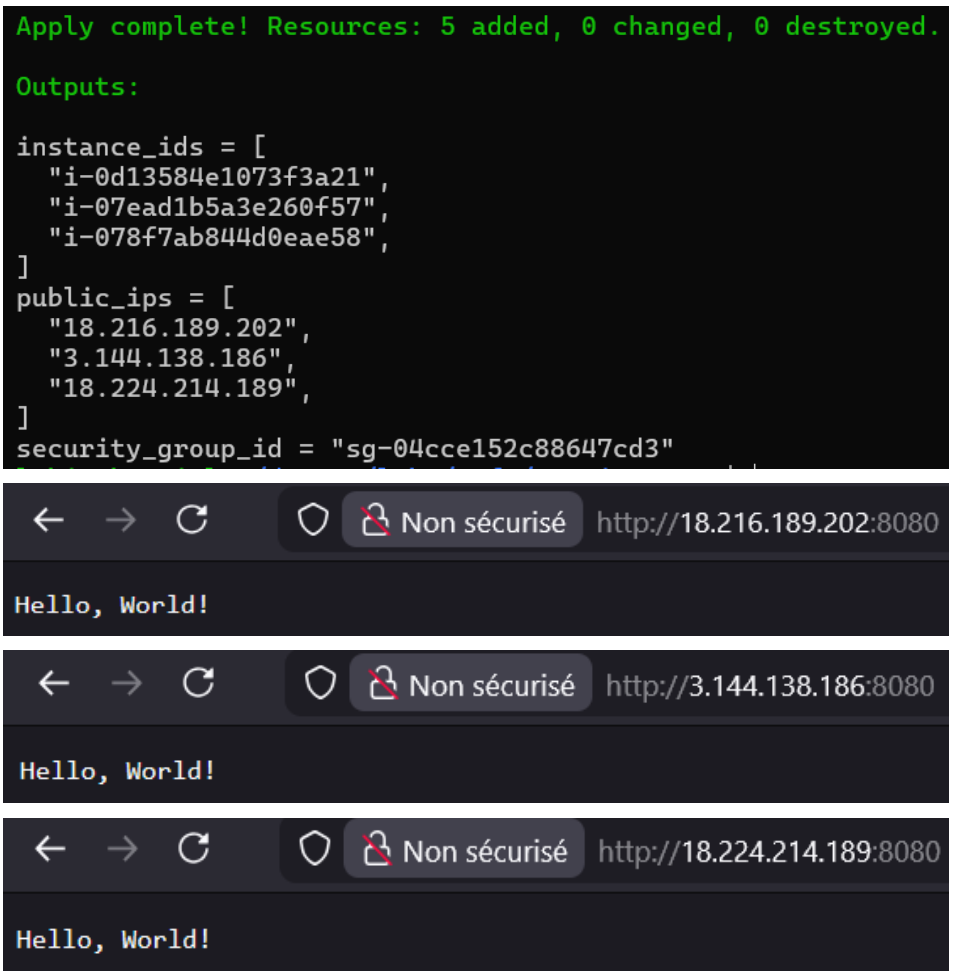
Explication : OpenTofu fonctionne en séparant la configuration en plusieurs fichiers pour une meilleure modularité. L'utilisation d'une variable pour l'AMI permet d'utiliser l'image créée précédemment avec Packer (cf section 4). tofu init télécharge les plugins nécessaires tandis que tofu apply compare l'état souhaité (le code) avec l'état réel du cloud et crée un fichier de "state" pour suivre les ressources déployées. Lorsqu'on met à jour l'infrastructure en modifiant le code pour ajouter un tag, et en faisant tofu apply, OpenTofu affiche un "plan" puis effectue la mise à jour sur place de l'infrastructure existante sur AWS sans interruption de service, sans la détruire pour en recréer une autre.

Exercise 7: What happens if you run tofu apply after the resources have been destroyed? Explain the behavior.

tofu apply est idempotent : il fait en sorte que l'infrastructure corresponde exactement à la configuration. Après une destruction, il recrée les ressources manquantes pour atteindre l'état souhaité.

Exercise 8: How would you modify the OpenTofu code to deploy multiple EC2 instances? Implement this change using a loop or by defining multiple resources.

Comme pour la section 3 avec Ansible, on peut utiliser count (boucle simple) pour créer plusieurs instances. Ajouter count: 3 dans resource "aws_instance" "sample_app" de main.tf. Il est aussi nécessaire de lister tous les IDs et IPs dans outputs.tf car aws_instance.sample_app est maintenant un tableau de ressources.



Détruire les instances : tofu destroy

Résumé (ce que nous avons retenu) :

OpenTofu (fork open-source de Terraform) est un outil de provisionnement de l'infrastructure. Il est surtout utilisé pour orchestrer les machines virtuelles et déclarer/gérer l'existence des ressources cloud (instances, groupes de sécurité). OpenTofu sait exactement ce qui est déployé et peut appliquer des modifications sur place de manière intelligente.

Nous avons déployé, mis à jour et détruit une instance EC2 AWS à partir d'une AMI Packer via OpenTofu. Cette section a permis d'appliquer les concepts d'Infrastructure as Code (IaC) et de tester la gestion de l'état de l'infrastructure.

Section 6: Deploying an EC2 Instance Using an OpenTofu Module

Objectif de l'étape : Organiser plus proprement le code de la configuration OpenTofu en le rendant modulaire.

Protocole suivi :

1/ Création de la structure modulaire
Créer un répertoire devops_base/lab2/scripts/tofu/live/sample-app/ et s'y rendre.
Créer un fichier main.tf dans ce répertoire, pour la configuration racine (root module).
Modifier le fichier pour appeler les modules sample_app_1 et sample_app_2 en pointant vers la source relative ../../modules/ec2-instance
Modifier l'ID d'AMI pour utiliser l'AMI créée par Packer (cf section 4).

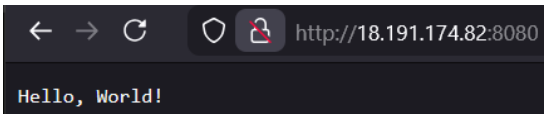
2/ Paramétrage du module
Modifier devops_base/lab2/scripts/tofu/modules/ec2-instance/variables.tf pour ajouter une variable name. Configurer le type d'instance en t3.micro comme précédemment.

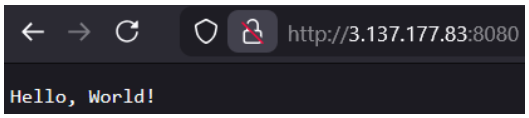
3/ Initialisation et déploiement
Configurer les identifiants si nécessaire : aws configure
Se rendre dans devops_base/lab2/scripts/tofu/live/sample-app/
Lancer tofu init et tofu apply.

```
Apply complete! Resources: 6 added, 0 changed, 0 destroyed.

Outputs:

sample_app_1_instance_id = "i-0e9da4c0a03facff2"
sample_app_1_public_ip = "18.191.174.82"
sample_app_2_instance_id = "i-0f315a800217338f5"
sample_app_2_public_ip = "3.137.177.83"
```





Explication : La structure du code est :

```
├── modules/
│   └── ec2-instance/  # Module réutilisable pour créer une instance EC2
│       ├── main.tf    # Définition des ressources (SG, instance, etc.)
│       ├── variables.tf # Variables d'entrée du module
│       └── user-data.sh # Script de démarrage de l'instance
└── live/
```

```
└── sample-app/    # Configuration "root" qui utilise le module
    └── main.tf    # Appel du module avec des valeurs spécifiques
```

La configuration statique du code a été transformée en organisant les fichiers dans un répertoire modules (code générique) et un répertoire live (code spécifique à un environnement). Un module agit comme un modèle (template). Cette séparation des ressources permet d'instancier plusieurs serveurs identiques, ou légèrement différents, simplement en appelant le module plusieurs fois dans le dossier live. Les variables du fichier variables.tf permettent d'éviter que OpenTofu crée des ressources avec le même nom pour chaque appel du module. La commande tofu init permet à OpenTofu de découvrir et d'indexer les modules locaux spécifiés dans le code, ce qui permettra de créer deux instances distinctes avec leurs propres adresses IP publiques.

Exercise 9: Modify the module to accept additional parameters like instance_type and port. Update the root module to pass these parameters.

Ajouter les variables instance_type et http_port dans variables.tf pour ne plus dépendre de valeurs fixes.
Mettre à jour main.tf du module pour utiliser ces variables.
Mettre à jour le module racine (live/sample-app/main.tf) en passant les valeurs pour ces variables, par exemple instance_type = "t3.micro" et http_port = 8080

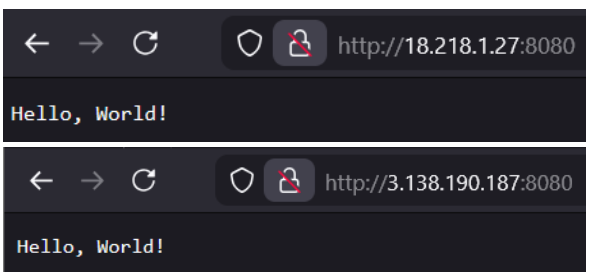
Exercise 10: Use OpenTofu's count or for_each to deploy multiple instances without duplicating code in the root module.

Modifier le module racine en utilisant l'argument count pour déployer plusieurs instances en une seule ligne de code. Cet argument transforme une ressource unique en un tableau de ressources.
Modifier outputs.tf (modules/ec2-instance/outputs.tf) de la même manière que pour l'exercice 8 avec la syntaxe [*] pour récupérer les IP et les ID de toutes les instances sous forme de liste.
De même pour les outputs du dossier live/sample-app/.
tofu destroy
tofu apply

```
Apply complete! Resources: 8 added, 0 changed, 0 destroyed.

Outputs:

sample_app_instance_ids = [
  [
    "i-063ef6c04e93ad988",
  ],
  [
    "i-087f278c861c7e751",
  ],
]
sample_app_public_ips = [
  [
    "3.138.190.187",
  ],
  [
    "18.218.1.27",
  ],
]
sample_app_security_group_ids = [
  "sg-0995cdb188ec6b825",
  "sg-0ee5e616892e6ff06",
]
```



Résumé (ce que nous avons retenu) :

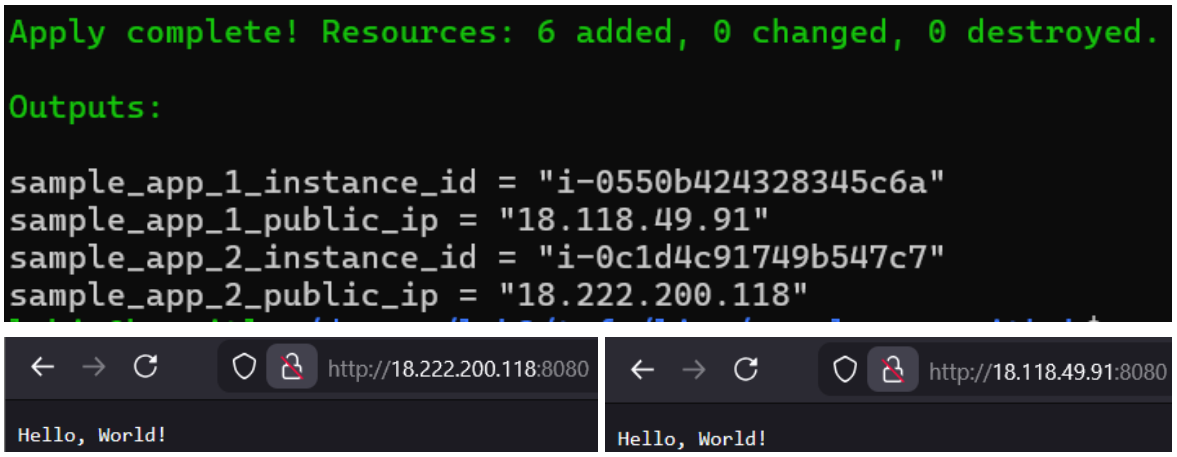
Dans cette section, nous apprenons à organiser le code OpenTofu avec des modules, afin de réutiliser le même code pour déployer plusieurs ressources AWS (ici des instances EC2). Cela permet d'éviter la duplication de code en utilisant un module EC2 réutilisable, puis l'appeler depuis un root module.

Section 7: Using OpenTofu Modules from GitHub

Objectif de l'étape : Reprendre la section précédente en explorant la réutilisation de code à l'échelle globale en décentralisation les modules d'infrastructure OpenTofu vers des dépôts distants.

Protocole suivi :

Modifier le fichier devops_base/lab2/scripts/tofu/live/sample-app-github/main.tf pour mettre à jour l'argument source du bloc module avec l'URL du dépôt GitHub.
Adapter les variables d'entrée en remplaçant ami_id par ami_name = "sample-app-packer-*" (nomenclature du code du dépôt) pour permettre au module de rechercher l'AMI dynamiquement.
Supprimer l'ancienne infrastructure locale si nécessaire : tofu destroy
Se placer dans tofu/live/sample-app-github/
Exécuter tofu init pour permettre à OpenTofu de cloner le module distant et déployer l'infra avec tofu apply.



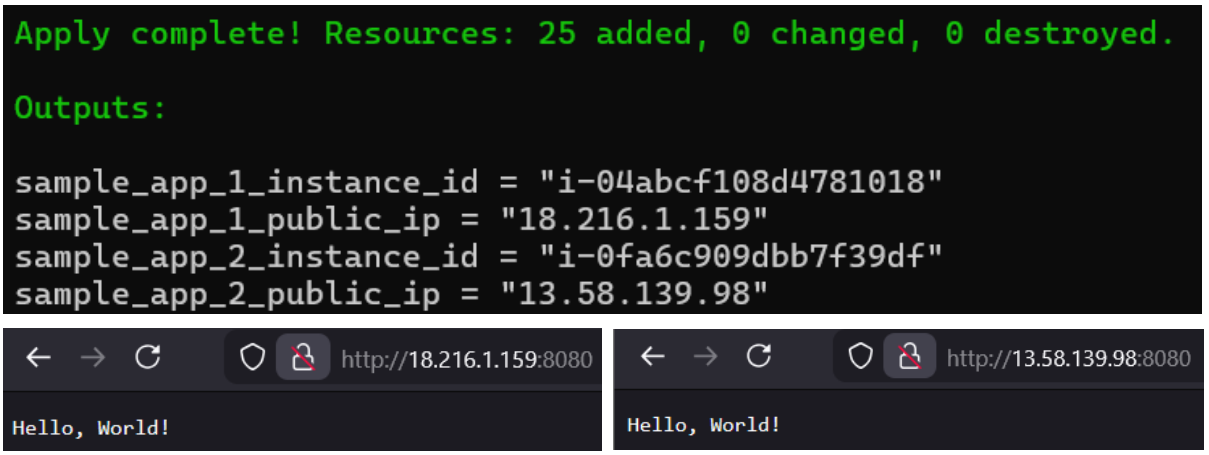
Explication : En pointant vers GitHub, OpenTofu ne cherche plus le code dans un dossier local, mais télécharge une copie du dépôt Git dans son répertoire de travail lors de l'initialisation. Cela permet à plusieurs développeurs d'utiliser la même source de vérité pour leurs déploiements.

Exercise 11: Explore the use of versioning with modules by specifying a specific Git tag or commit in the module source.

Pour sécuriser les déploiements en verrouillant le module sur une version spécifique, il suffit de rajouter ?ref=1.0.0 à la fin de l'URL source dans main.tf pour forcer d'utiliser ce tag même si d'autres versions sont publiées plus tard.
Mettre à jour la config locale : tofu init -upgrade
Appliquer la config : tofu apply

Exercise 12: Find an OpenTofu module in the Terraform Registry or another public repository and use it in your configuration.

Pour exploiter des modules validés par la communauté pour gagner en efficacité, on peut utiliser un module VPC public sur le Terraform Registry, comme par exemple terraform-aws-modules/vpc/aws
Au lieu de coder notre VPC nous-même, nous utilisons un module public existant pour créer un VPC complet (paramétré avec les sous-réseaux et passerelles NAT).



Une fois les tests terminés, détruire les ressources : tofu destroy

Explication : Sans précision de version, OpenTofu récupère généralement la branche principale. Le versionnement grâce à des tags Git garantit que l'infra reste stable même si le code du module distant évolue par la suite. L'utilisation de modules publics du Terraform Registry évite d'avoir à recoder les composants standards comme le réseau (VPC) pour les bases de données.

Résumé (ce que nous avons retenu) :

La section 7 permet d'apprendre à utiliser des modules OpenTofu hébergés sur GitHub pour simplifier le déploiement et la collaboration. Elle démontre que l'IaC devient réellement puissante lorsqu'elle est partagée et versionnée. L'utilisation de modules distants sur GitHub ou Registry permet de simplifier la collaboration entre équipes en assemblant des composants pré-existants et sécurisés.

Conclusion générale

Ce laboratoire illustre la transition essentielle entre une gestion manuelle de l'infrastructure et une approche entièrement définie par le code (Infrastructure as Code). À travers plusieurs outils complémentaires, nous avons observé une montée en maturité de l'automatisation.

Les scripts Bash constituent une première étape en permettant d'interagir avec AWS de manière reproductible. Toutefois, leur absence de gestion d'état les rend non idempotents et rapidement difficiles à maintenir.

Ansible introduit ensuite une approche déclarative de la configuration. Grâce aux playbooks, aux rôles et aux inventaires dynamiques, il garantit qu'une exécution répétée produit toujours le même résultat. L'infrastructure devient alors plus stable, prévisible et simple à faire évoluer.

Avec Packer, le paradigme change vers l'immuabilité. En générant des AMI contenant déjà l'application et ses dépendances, on réduit drastiquement le temps de déploiement tout en améliorant la fiabilité des environnements.

Enfin, OpenTofu permet d'orchestrer le cycle de vie complet des ressources cloud. La gestion du fichier d'état offre une vision précise de l'existant et autorise des mises à jour contrôlées, des créations ou des destructions propres. L'introduction des modules, qu'ils soient locaux ou distants via GitHub, favorise la réutilisation, la standardisation et la collaboration entre équipes.

Le principal enseignement du laboratoire est qu'une stratégie DevOps efficace repose sur la combinaison de ces outils : images immuables pour la rapidité, gestion de configuration pour la cohérence et orchestration déclarative pour la maîtrise globale de l'infrastructure. Ensemble, ils permettent de construire des déploiements fiables, reproductibles et évolutifs.