TP IAC 02 (version étudiant)

TP IAC

Consignes **2**
- Fonctionnement du TP 2
- Evaluation 2
- Environnement de travail 2
- Si vous êtes bloqués 3

Terraform **3**
- Initialisation 3
  - Accès aux différents credentials 3
  - Connexion à la console et restriction du compte 4
  - Un mot sur les clés d'API (Access key) 6
  - Répertoire de travail 6
  - Export des credentials dans l'environnement shell 7
  - Terraform init 8
- Créer (et détruire) votre première instance 9
  - Problématique du compte AWS partagé 9
  - Créons notre première VM 10
- Comprendre la notion de stateFile 12
- Comprendre la notion de plan 15
- Multiplier les ressources et premier aperçu des variables 17
- Les variables dans terraform 17
  - input vars 18
  - output values 19
- Providers, ressources et datasources 20
  - Providers 20
  - Ressources 21
  - Datasources 21
- La notion de lifecycle et éviter les modifications et les dépendances 22
  - lifecycle 22
  - Dépendances 23
- Notion de modules et éléments avancés 23
- Connexion SSH 24
- Un dernier mot sur les stateFiles (et le multi-environnement) 26
  - Stockage du tfstate 26
  - Multi-environnement 27

Ansible **28**
- Premier test sur localhost 28
- Un test sur les instances distantes 29
  - Sans inventaire 29
  - Avec inventaire et configuration 29
- Variables 30
- Documentation et aide sur les modules 32

Playbooks 32
Choix du chemin d'apprentissage 34
Rôles 34
Organisation du role 34
Conditions when 35
Copie de fichiers, templating jinja et manipulation 36
Gestion des boucles dans les tâches 37
Déléguer l'exécution des tâches à un host particulier 38
Utilisation des tag pour conditionner l'exécution 38
Retry sur les tasks 39
Import et inclusion de tasks 39
Exercices divers 39
Autres fonctionnalités intéressantes 40
Stratégies de déploiement 40
Inventaire dynamique 41
Handler 42
ansible-galaxy 42
Un exemple concret : l'application demoboard 43
Architecture globale de l'application 43
Création de l'infrastructure avec terraform 44
Configuration des serveurs avec Ansible 45
Travaux d'optimisation - exercices 49
FACILE 50
Terraform 50
Ansible 50
Terraform + Ansible 51
Terraform 51
Ansible 52
Terraform + Ansible 52
DIFFICILE 53
Ansible 53
Terraform + Ansible 53
!! Suppression de l'ensemble des ressources CLOUD 54

Consignes

Vous aurez besoin pour ce TP d'une machine de travail avec terraform et ansible.
Il est possible d'installer ces éléments sur une machine personnelle ou créer une VM avec ces éléments. Toutefois, un tel setup pourrait quasiment faire l'objet d'un TP en soi.
Vous aurez donc à votre disposition une VM dans le CLOUD pré configurée avec tous les éléments nécessaires. Les pré-requis étant :

- Disposer d'un accès à INTERNET (en cas de problème - très rare - avec la connexion de l'école, une solution de secours avec l'utilisation d'un smartphone en mode partage de connexion 4G peut se révéler utile)
- Un navigateur html5 moderne (edge, chrome, firefox, …)
- éventuellement (non nécessaire) pouvoir lancer / installer un client RDP (natif sur windows)
- éventuellement (non nécessaire) pouvoir faire directement du SSH vers cette machine

Fonctionnement du TP

Vous devez réaliser le TP dans l'ordre, vous pourriez sinon manquer des étapes et informations nécessaires pour la suite

Evaluation

Durant le TP, vous devrez répondre à des questions et effectuer des actions (encadrés en couleur).

Notez dans un fichier de travail les réponses (n'hésitez pas à copier le résultat des commandes et/ou prendre des captures d'écran). Ceci vous permettra de partager avec le groupe vos réflexions au fur et à mesure du TP.

Cette étape est facultative mais répondre aux questions est la meilleure façon de valider votre compréhension de chaque élément.

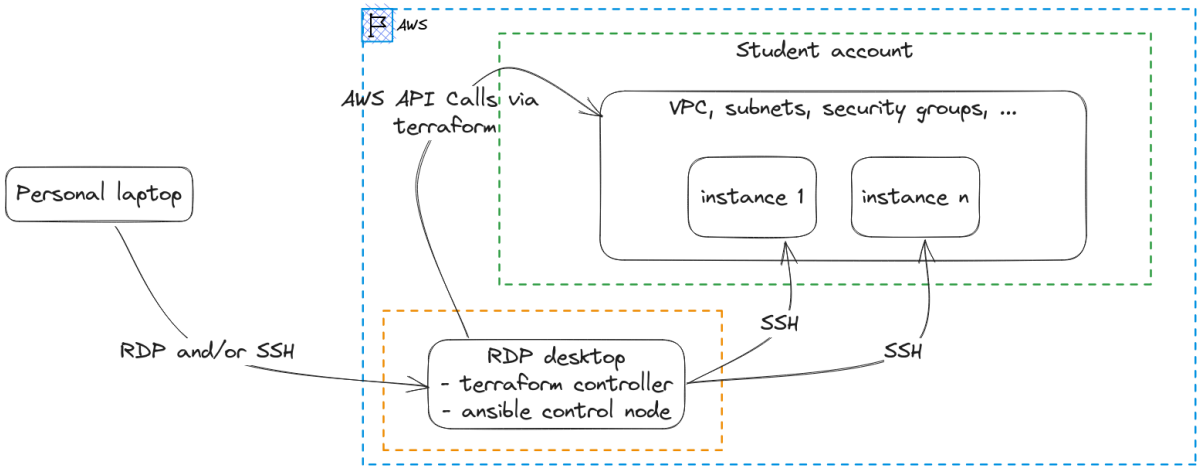
Note pour les étudiants du MSASI : l'évaluation finale du TP se fait ultérieurement via un QCM en ligne sur EDUNAO et n'est pas directement lié à ces questions.

Dans les questions (cadres en jaune) il est parfois indiqué BONUS : ces questions sont souvent assez/très difficiles et demandent du temps de recherche. N'hésitez pas à les sauter (vous y reviendrez plus tard s'il vous reste du temps), il s'agit d'un challenge ou d'une façon d'aller plus loin pour les étudiants déjà familiers avec le thème.

Environnement de travail

Pour réaliser ce TP, vous disposerez d'un environnement de travail complet dans le CLOUD.

- Tout d'abord une machine de travail (bureau RDP) par étudiant qui contient l'ensemble des outils nécessaires (terraform, ansible, vscode, …)
- un user AWS IAM (au sein d'un compte AWS partagé avec les autres étudiants) vous permettant d'y créer des ressources d'infrastructures (réseau, vms, …)



L'accès à la VM se fait de multiples manières :

- *Portail HTTPS guacamole : https://access.tpcsonline.org login/mdp de type vmXX (le formateur vous attribuera le numéro - identique au numéro de la VM)*
    - *Le raccourci ctrl + shift + alt vous permet de transférer des fichiers depuis et vers votre machine personnelle si besoin*
- *Accès direct en RDP ou SSH sur vmXX.tpcsonline.org (user/pwd = vmXX)*

Si vous êtes bloqués

Le TP peut être difficile ou sa structure peut ne pas correspondre à votre chemin de pensée. Il est normal de pouvoir se sentir un peu perdu, pas de panique ! Demander de l'aide au formateur, il est disponible pour vous.

N'hésitez pas également à remonter toute erreur ou formulation qui vous paraît peu claire. Le formateur fera évoluer le document pour les prochaines sessions.

Enfin, rappelez-vous que vous êtes là pour apprendre par la pratique, n'hésitez pas à essayer, il s'agit d'environnements de test, profitez-en. Vous travaillez sur un sujet nouveau avec peu de temps, il est tout à fait normal de ne pas comprendre ou ne pas connaître certains concepts.

Terraform

Initialisation

Accès aux différents credentials

En plus de la VM qui vous est fournie comme controller terraform et ansible pour réaliser le TP, vous disposerez d'un compte AWS IAM pour créer et manager des ressources.

L'URL de login, le username (vmXX) et le password associés pour se connecter à la console AWS sont disponibles dans un fichier qui a été généré pour vous sur la VM.
Ce fichier contient également les clés d'accès à l'API aws qui seront nécessaires pour terraform.

Observez le contenu du fichier soit en utilisant l'éditeur vscode qui a dû s'ouvrir au démarrage soit depuis un terminal avec la commande `cat /home/vmXX/tpcs-iac/.env`

Le contenu devrait ressembler à cela :

```
# aws console login URL :
https://tpiac.signin.aws.amazon.com/console/
# aws console username : "vmXX"
# aws console password : "<motDePasse>"
# TP IaC AWS credentials are configured in the default AWS CLI
profile:
# - /home/{{ ansible_hostname }}/.aws/credentials
# - /home/{{ ansible_hostname }}/.aws/config
#
# They are kept here as a memo for the TP:
# aws_access_key_id = "<chaîneDeCaractères>"
# aws_secret_access_key = "<chaîneDeCaractères>"
# region = "<eu-*******>"

# These Terraform variables are also written to
demoboard/terraform/tpiac.auto.tfvars.
# Sourcing this file is only kept as a compatibility helper.
export TF_VAR_ami="<chaîneDeCaractères>"
export TF_VAR_type=t3.micro
export TF_VAR_instance_details='"<chaîneDeCaractères>"'
export TF_VAR_ssh_key_public=$(cat /home/{{ ansible_hostname
}}/tpcs-iac/demoboard/terraform/tp-iac.pub)
export
TF_VAR_demoboard_aws_zones='["eu-*****a","eu-*****b","eu-****c"]'
```

Les 3 premières lignes sont les infos pour se connecter à la console
Les lignes suivantes sont pour l'accès à terraform avec des clés d'API (plus d'informations dans les paragraphes suivants)
L'avant dernière ligne est une clé SSH qui a déjà été créée pour vous pour la partie finale du TP (nous y reviendrons plus tard, vous pouvez ignorer ce point pour le moment)

En réalité, ce fichier est surtout pour information, il n'est plus impératif de le sourcer dans l'environnement. Il va essentiellement vous servir pour vous connecter à la console (les 3 premières lignes)

Connexion à la console et restriction du compte

Commencez par vous connecter à la console AWS avec un browser.

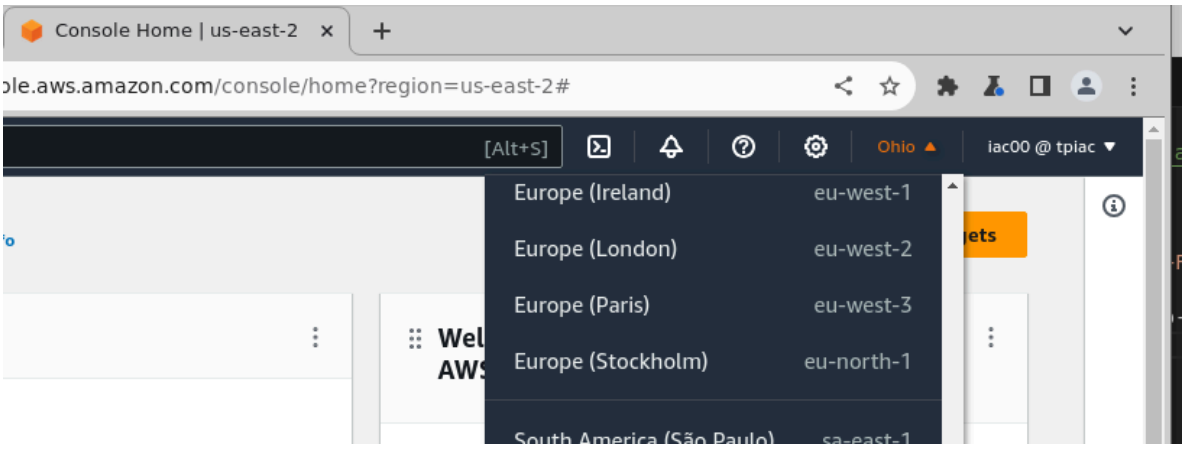
Vous pouvez le faire depuis la VM de travail directement (dans le menu de démarrage, cherchez le sous-menu internet) ou bien sûr depuis votre propre laptop.

Utilisez l'URL, le user et le mot de passe des premières lignes. Si vous avez un problème de connexion à la console ou si certaines lignes sont vides et ne contiennent pas de valeurs, veuillez contacter le formateur.

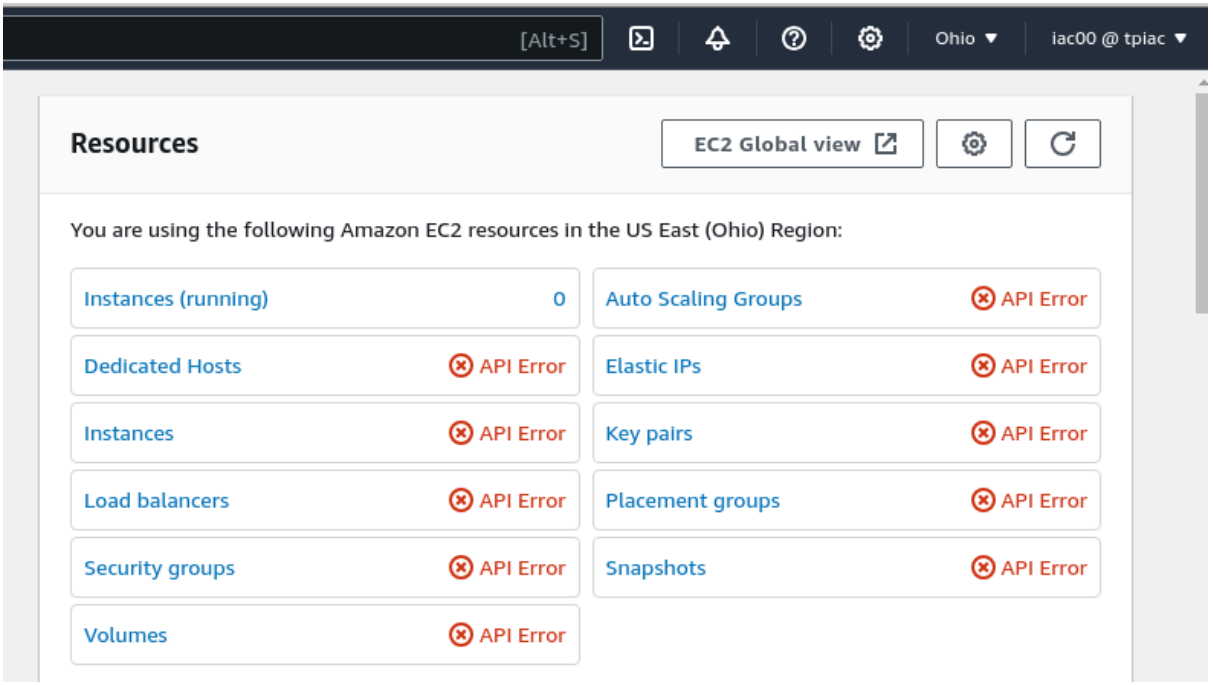
Le compte est volontairement restreint et ne vous permettra pas de manager tout type de ressources AWS ni d'utiliser différentes régions (les droits sont bien sûr suffisants pour réaliser l'ensemble du TP).
Vous serez limités à :

- Uniquement une région d'europe différente de paris (par exemple eu-south-2)
    - **/!** Cette région vous est propre et est disponible dans le fichier .env
- L'ensemble des ressources EC2
- L'ensemble des ressources VPC
- L'ensemble des ressources pour le LoadBalancing
- Management des access keys (clés API) pour le user
- L'utilisation de compute-optimizer et DescribeAlarms de Cloudwatch (nécessaire pour éviter des erreurs dans la consultation de la console sur les instances)

Pensez donc bien à **sélectionnez la région qui vous est attribuée dans le fichier .env** sinon vous aurez des erreurs ou simplement pas d'affichage de vos ressources



Il est normal que des erreurs s'affichent si vous tentez de consulter ou créer des ressources sur d'autres régions où vous n'avez pas de droits. Exemple si vous consultez EC2 sur une région différente de celle qui vous est affectée.



En conclusion, limitez-vous à la création des ressources nécessaires au TP **et surtout supprimez l'ensemble des ressources en fin de TP** (et quand c'est possible au fur et à mesure des exercices).

Un mot sur les clés d'API (Access key)

Pour appeler l'API de AWS, terraform a besoin d'une clé d'API car il ne peut pas utiliser le compte avec un user/password.

La génération d'une clé pour le compte a donc été faite pour vous, pour votre information si vous souhaitez un jour utiliser vous mêmes terraform avec un compte AWS.

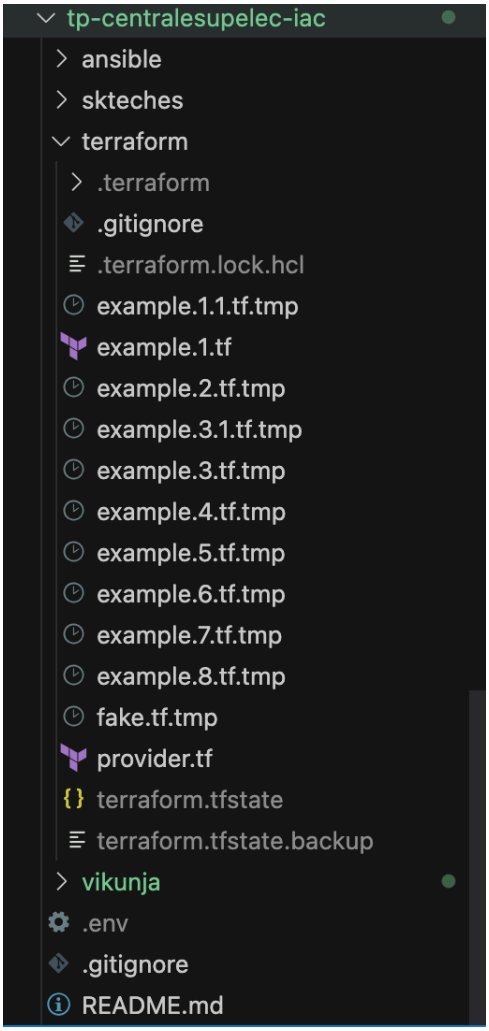
> *Cliquer sur le nom de l'utilisateur en haut à droite, puis sélectionnez "identification et paramètres de sécurité" (ou security credentials) dans le menu*
>
> *Dans la page à laquelle vous accédez, aller à la section clé d'accès (Access keys) et cliquer sur créer une clé d'accès.*
> *S'il y a déjà des clés existantes vous pouvez les supprimer car il n'est pas possible de créer plus de deux clés d'accès sur un compte.*
> *Pour supprimer une clé, il faut qu'elle soit désactivée d'abord (il faut donc faire les deux opérations)*
>
> *Copier bien dans un fichier la clé secrète car elle ne s'affiche qu'une seule fois, vous ne pourrez pas y revenir via la console (en cas de perte, il faudra supprimer et recréer une nouvelle clé)*

Répertoire de travail

Dans le home directory (/home/vmXX) de la VM de travail, vous avez un répertoire tpcs-iac qui contient un clone git du repo des exercices. https://github.com/seb54000/tpcs-iac

Pour les premières parties (hors demoboard), vous pouvez utiliser directement les fichiers et répertoires pour vous faciliter la tâche ou préférez repartir de zéro dans un répertoire vierge où vous créerez petit à petit la même structure.
Si vous choisissez la seconde solution, n'hésitez pas à jeter un oeil de temps en temps aux fichiers d'origine

Voici un exemple de l'arborescence de départ. Sur votre machine, vous ne devriez pas voir les éléments grisés ci-dessous. En effet, ce sont des fichiers qui sont ignorés par git et donc jamais envoyés sur le repository.



Export des credentials dans l'environnement shell

Terraform aura donc besoin des credentials pour se connecter, on peut les mettre dans les fichiers terraform mais c'est une mauvaise pratique de sécurité puisque ces fichiers ont normalement vocation à être stockés dans un repository de git partagé (et parfois public).

On peut en revanche, charger des variables d'environnement dans le shell courant qui seront accessibles au binaire terraform.

C’est pour cela que le fichier .env a été généré (si vous aviez récupérer vos clés vous-mêmes, il faudrait créer un fichier sur le même principe que notre .env ou renseigner ces clés dans un profil AWS, cf. ci-dessous)

Vous pourriez donc sourcer le .env (`source .env`) pour charger les variables et même ajouter une ligne à la fin du $HOME/.bashrc de votre user pour charger ces variables à chaque démarrage d’un shell.
C’est inutile car le provider terraform (cf. plus bas) est également capable de lire automatiquement les configuration et default profile AWS. C’est ce qui va être utilisé automatiquement car l'environnement $HOME/.aws a été configuré pour vous (vous pouvez regarder les fichiers si besoin (nous utilisons des profils car pour certains TP, nous utilisons aussi eks kubernetes qui nécessite d’autres clés d’API)

Terraform init

Placez-vous maintenant dans le répertoire terraform sous le répertoire de travail (ou créer un répertoire terraform si vous décidez de travailler en dehors du répertoire fourni)

Vérifier qu’il existe bien un fichier provider.tf sinon créer le avec comme contenu :
`provider "aws" {}`

Depuis un shell dans le répertoire terraform, lancez la commande : `terraform init`
Vous devriez voir Terraform télécharger le provider aws.

Vous voilà prêt à faire du terraform !

Attention, pour la suite de la partie terraform, si vous travaillez dans le répertoire fourni, vous remarquerez qu’il y a toute une série de fichiers .tf.tmp

En fait, terraform interprète l’ensemble des fichiers .tf du répertoire courant. Comme il s’agit d’exercices différents, ils sont suffixés en .tmp pour être ignorés.

Quand vous êtes sur un exercice correspondant au fichier example.2.tf par exemple, cela signifie que vous devez supprimer le suffixe .tmp et par contre ajouter le suffixe .tmp au fichier sur lequel vous aviez fait l’exercice précédent.

En cas de souci sur le terraform init ou de doutes ou soucis ultérieurement sur les credentials, vous pouvez faire quelques tests pour être sûr que vous utilisez bien le aws default profile, depuis un shell :

```
aws configure list
aws sts get-caller-identity
terraform providers
```

Créer (et détruire) votre première instance

Problématique du compte AWS partagé

Nous utilisons un seul compte racine AWS qui va contenir tous les user IAM et l'ensemble de toutes les ressources que vous allez créer.

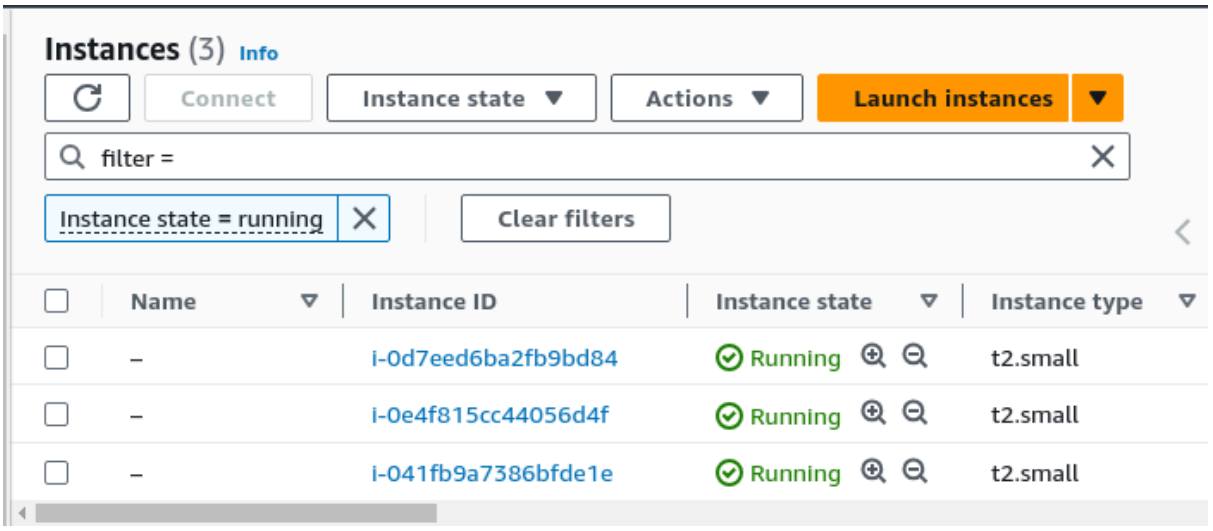
Il est évidemment problématique de se repérer si l'on ne met pas de tag (balise ou étiquette) sur nos ressources. Tous les exemples de code que nous allons utiliser contiennent dès le départ un tag nommé "filter" et dont la valeur sera tpiacXX (XX le numéro de votre user/vm)

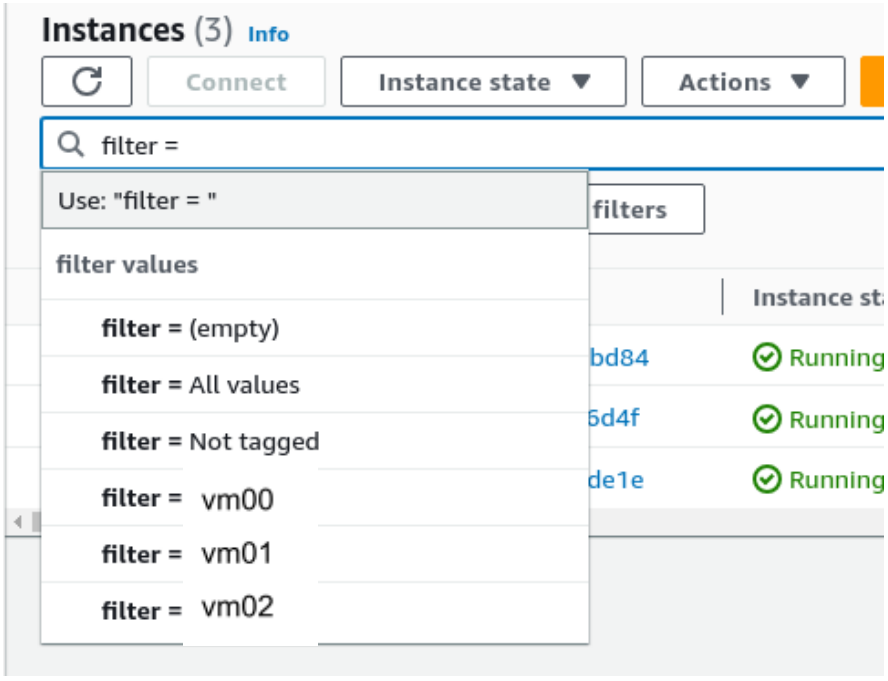
D'un point de vue du code, ignorez dans un premier temps la façon dont cela est construit, il s'agira toujours d'une ligne de type `filter = chomp(file("/etc/hostname"))` (nous parlerons de son fonctionnement dans un paragraphe ultérieur).

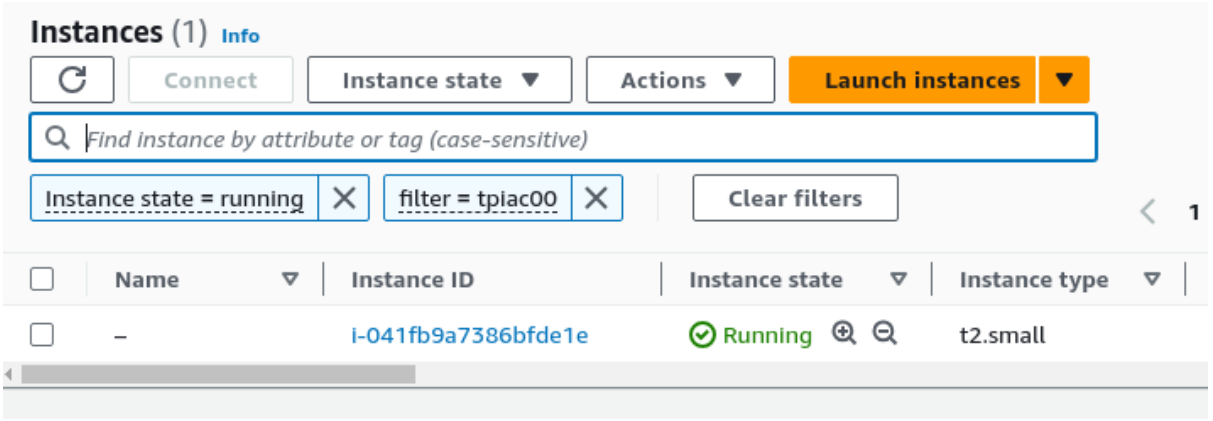
L'important pour vous est que ces lignes soient présentes et lorsque vous devrez constater le fonctionnement dans la console AWS, vous devrez ajouter le filtre pour ne pas être perturbés par les travaux de vos collègues.

Il suffit pour cela de taper filter dans la barre de recherche, il devrait vous proposer un type = et ensuite une liste des valeurs, vous verrez vmXX, il suffit de cliquer dessus. à partir de là, l'affichage est filtré sur les ressources ayant un tag filter=vmXX

Voici un exemple sur la console EC2







Créons notre première VM

Les références (ID) des images AWS pour créer une VM (instance) sont spécifiques par région. L'image ID de ubuntu noble pour amd64 qui correspond à votre région est renseignée dans le fichier .env (en commentaire : ami-id).

Vous devrez utiliser cet **AMI-ID** dans tout le reste du TP (à modifier dans les exemples du document **si besoin - dans les versions récentes du TP, la substitution de l'AMI-ID peut avoir été faite automatiquement**)

Créer simplement un fichier instances.tf *(ou utilisez le fichier example.1.tf si vous utiliser le répertoire git, ces fichiers ont déjà été mis à jour avec le bon AMI-ID)* au même niveau dans l'arborescence que votre provider.tf avec le contenu suivant

```
resource "aws_instance" "test-instance" {
  ami             = "<AMI-ID>"
  instance_type = "t3.micro"
```

```
  tags = {
    filter = chomp(file("/etc/hostname"))
  }
}
```

Tapez ensuite la commande `terraform apply`

Vous devriez voir un résumé de ce que terraform propose de faire (la création d'une instance) et une demande de confirmation : tapez `yes`

Si vous avez une erreur indiquant que l'ami n'existe pas, vérifiez l'ID de l'ami (il s'agit de l'image de la VM, ici une ubuntu). Les ami ID sont spécifiques à une région AWS, vérifier que l'ID AMI renseigné dans vos fichiers correspond bien à la région qui est affectée à votre user AWS.
Vous pouvez rechercher les ID ami pour ubuntu ici :
https://cloud-images.ubuntu.com/locator/ec2/

Une fois la création de votre instance (VM) terminée, connectez-vous sur la console AWS pour vérifier qu'elle existe bien (dans le menu EC2, vérifier bien que la bonne région est sélectionnée)
Vous devriez voir votre VM (il ne doit pas y avoir d'autres ressources existantes sur le compte - revoyez les captures d'écran plus haut pour positioner les fitlres si vous ne l'avez pas fait) et celle-ci n'aura aucun nom mais uniquement un UID

Nous allons donc ajouter un nom à votre VM, modifier le code de instances.tf *(ou utilisez example.1.1.tf si vous utiliser le répertoire git)*

```
resource "aws_instance" "test-instance" {
  ami             = "<AMI-ID>"
  instance_type = "t3.micro"
  tags = {
    Name = "my-test-vm"
    filter = chomp(file("/etc/hostname"))
  }
}
```

Lancer à nouveau la commande `terraform apply`
Cette fois terraform devrait vous indiquer uniquement une modification et vous la montrer, vous pouvez toujours valider avec `yes`
Retournez dans la console et rafraîchissez la vue si nécessaire, votre VM a désormais un nom dans la console.

TQ01 :
Prenez une capture d'écran de la console EC2 montrant votre VM avec le nom que vous avez choisi via le code.

BONUS : Pouvez-vous lister les fichiers que terraform a créé automatiquement (lors des phases de init et apply) ? Pensez à regarder les fichiers cachés également (ls -a)

Nous allons supprimer l'ensemble de l'infrastructure que nous gérons via terraform (une seule VM pour l'instant mais ce serait identique si nous en avions mille).
Exécutez la commande `terraform destroy`
Terraform doit vous lister les ressources qu'il va supprimer. Confirmer et valider dans la console AWS que la ressource a disparu (état résilié).

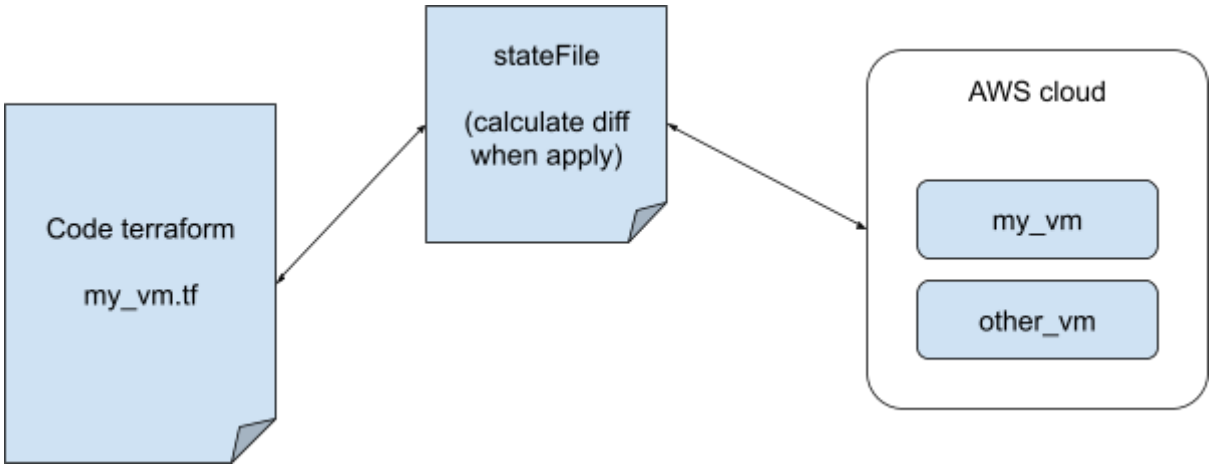
Vous ne devriez pas en avoir besoin durant le TP mais au cas où et par curiosité, sachez que si vous voulez afficher beaucoup plus de log (mode verbeux) lors de l'exécution de terraform, il faut setter une variable d'environnement, voici un exemple

`TF_LOG=DEBUG terraform destroy`
https://developer.hashicorp.com/terraform/internals/debugging

Comprendre la notion de stateFile

Terraform conserve une vue de l'état courant de l'infrastructure dans un fichier tfstate. Dans notre cas et à titre de simplification, ce fichier est stocké localement.

Vous pouvez le voir à la racine de votre répertoire de travail terraform sous le nom terraform.tfstate
Le fichier stateFile permet à terraform de calculer la différence qui existe entre l'état désiré (votre code) et l'état courant du système (ce que l'on voit via l'API AWS ou dans la console AWS)
Si l'on modifie manuellement des éléments via l'API ou la console AWS, ils ne sont pas encore connus de terraform (pas dans le tfstate). De la même manière, si l'on modifie notre code, tant que nous n'avons pas fait d'apply, ses éléments ne sont pas encore créés sur AWS et dans le stateFile.



Nous allons tenter d'illustrer tout cela dans les exercices suivants.

Refaites un `terraform apply` pour créer à nouveau la machine virtuelle. Vous pouvez ensuite jeter un œil au fichier tfstate, il est au format json et va contenir tous les détails de la VM telle qu'elle est sur le cloud provider.
Rechercher le tag Name dans le fichier

TQ02 :
Quelle est la valeur du tag Name dans le tfstate ?
A quoi correspond-elle ?

Afin de comprendre les mécanismes autour du stateFile qui est un élément central dans terraform, nous allons modifier au niveau du cloud provider une propriété de la VM.
Connectez vous sur la console AWS EC2 et modifier le nom de la VM (ajouter -modified au nom de l'instance)

Nous allons forcer terraform à mettre à jour son tfstate, exécuter `terraform refresh`

TQ03 :
Après l'exécution, quelle est la valeur du Tag Name dans le fichier tfstate ?
A quoi correspond-elle ? (c'est bien la même question que juste avant, constatez-vous une différence dans la réponse ?)

Nous allons à nouveau réaliser un `terraform apply`
N'appliquez pas les modifications si terraform vous en propose.

TQ04 :
Que se passe-t-il ?
Quelle est la configuration qui a la priorité dans un terraform apply entre l'état courant (tfstate) et l'état désiré (le code terraform que nous avons écrit) ?

Est-ce qu'il faut systématiquement lancer un terraform refresh avant un apply (aidez-vous des logs de vos précédentes exécutions de terraform apply)

Pour la suite des exercices, vous devez toujours avoir un nom de VM au niveau de la console du cloud provider différent de celle indiquée dans votre code (avec le suffixe -modified).

Nous allons maintenant modifier la vision de terraform de l'état courant du cloud provider.
Oui, nous allons modifier le stateFile.

Bien qu'il soit possible d'éditer le fichier manuellement, c'est une pratique très peu recommandable qui ne devrait être utilisée qu'en cas de dernier ressort (et vous avez intérêt à disposer d'un backup).

Nous allons plutôt utiliser des commandes disponibles dans terraform pour manipuler le stateFile. (même avec ces commandes, dans un environnement de production, il est recommandé de disposer de backups, on ne soulignera jamais assez le niveau de criticité du stateFile)

Commencez par lister les ressources du tfstate (vous pouvez utiliser show pour voir le détail)

```
terraform state list
terraform state show <resource_reference>
```

TQ05 :
Que voyez-vous dans la liste des ressources ?

Quel est l'id de la ressource (vm instance) quand vous réalisez la commande state show ? (notez le bien, nous en avons besoin ultérieurement)
Le tag Name est-il bien présent avec le même nom que sur la console du cloud provider et différent de ce que vous avez dans votre code ?

Nous allons maintenant supprimer du tfstate la référence à notre VM

```
terraform state rm <resource_reference>
```

TQ06 :
Quel résultat si vous relancez un state list ?
Que voyez-vous dans le fichier tfstate si vous l'éditez ?

Que voyez-vous au niveau de la console du cloud provider ?

Faites maintenant un `terraform apply` à nouveau et accepter les modifications proposées

TQ07 :
Qu'est-il en train de se passer ? Pourquoi Terraform recrée une nouvelle VM ?

Que voyez-vous dans la console du cloud provider ? Poster également une capture d'écran de la vue des instances.

Nous allons essayer d'importer l'ancienne VM (instance) que nous avions supprimée du tfstate (celle où nous avions noté l'ID précieusement)
Appelons cette ressource imported-instance par exemple

```
terraform import aws_instance.imported-instance <vm-id>
```

TQ08 :
Qu'avez-vous comme retour de terraform ? Que faudrait-il faire pour importer la ressource avec imported-instance comme nom ?

Nous allons utiliser une autre approche. Nous allons tenter d'importer l'ancienne VM dans la ressource tfstate correspondant à notre code : test-instance
`terraform import aws_instance.test-instance <vm-id>`

TQ09 :
Qu'avez-vous comme retour de terraform ? Que faudrait-il faire pour importer la ressource avec test-instance comme nom ?

Vous allez maintenant détruire la VM nouvellement créée avec un simple terraform destroy
`terraform destroy -auto-approve`
Et retenter un import
`terraform import aws_instance.test-instance <vm-id>`

TQ10 :
A quoi sert l'option -auto-approve ? Est-ce une bonne idée d'utiliser ce type d'option ? Dans quel cas est-ce que ce type d'option est indispensable ?

Qu'avez-vous comme retour de terraform pour l'import ?

Utilisez à nouveau la commande `state show <resource_reference>` , quelle est la valeur du tag Name ? Est-ce que cela vous paraît normal ?

Nous avons donc finaliser l'import de notre ancienne VM dans le tfstate. Tout est rentré dans l'ordre, nous pouvons à nouveau manager cette ressource avec terraform !

TQ11 :
Que faut-il lancer comme commande pour vérifier et éventuellement réconcilier l'état désiré (le code) avec l'état actuel que nous avons importé du cloud provider

Réaliser cette commande et indiquer ici le résultat de cette dernière. Est-ce que vous vous attendiez à cela ?

Comprendre la notion de plan

Tout d'abord, parlons d'une commande terraform qui permet de valider que vos fichiers n'ont pas de problèmes de syntaxe. il s'agit de `terraform validate`

Cette commande va s'assurer que tout votre code est cohérent et ne comporte pas d'erreurs. C'est une bonne pratique de le lancer en premier. Surtout, dans une usine logicielle, on va utiliser la CI/CD pour lancer ce type de commande.
On peut même utiliser un hook git precommit pour éviter d'autoriser un push de code dont la syntaxe est incorrect (pour les curieux, vous pouvez vous référer à ce type d'article : https://jamescook.dev/pre-commit-for-terraform#heading-installing-pre-commit)

Pour la suite des exercices, vous pouvez l'utiliser mais c'est facultatif, dans tous les cas, lors d'un apply ou autre commande, si votre code comporte une erreur, il ne sera pas exécuté.

Nous l'avons vu, un intérêt de terraform est de proposer une vue des modifications à réaliser par rapport à l'état courant pour atteindre l'état désiré. On parle de plan (qui est réalisé automatiquement quand on fait un apply).

Mais on peut lancer un plan et sauvegarder son résultat pour ne l'appliquer que plus tard. C'est souvent ce qui est fait en CI/CD pour permettre une éventuelle revue du plan avant apply afin d'éviter une erreur (on ajoutera alors des stages de validation manuelle dans le pipeline)

Vous pouvez vous représenter un plan comme un script qui contiendrait toutes les actions à exécuter pour réconcilier l'état désiré et l'état courant au moment où la commande plan a été lancée. Par exemple : delete vm avec id=xxx ; ajouter un tag sur vm avec id=yyy ; ajouter une nouvelle vm avec id=zzz

Nous allons faire une série d'actions pour illustrer comment cela fonctionne :

- Faites un `terraform apply` de la dernière configuration pour disposer d'une VM avec un tag name standard.
- Connectez-vous sur la console AWS et modifier à nouveau le nom de la VM (-modified)
- Faites maintenant un plan dont vous sauvegardez le résultat
  - `terraform plan -out myplan`
- Faites à nouveau un `terraform apply` standard pour revenir à la VM avec le name initial de votre code
- Faites maintenant un show du plan pour voir ce qu'il va vouloir réaliser
  - `terraform show myplan`
- Appliquer maintenant ce plan
  - `terraform apply myplan`

TQ12 :
Quel est le type de contenu du fichier de plan myplan ? Est-ce que vous pouvez facilement l'interpréter ?

Quel est le résultat de la commande show ? Trouvez-vous cela normal, comment l'expliquez-vous ?

BONUS : Pouvez-vous lister la signification des couleurs et actions du plan et expliquer la notion de replace ou in-place ? (https://developer.hashicorp.com/terraform/cli/state/taint)

Quel est le résultat du apply ?

Faisons un second exercice un peu différent :

- Commencer par un `terraform destroy` pour partir de zéro
- Faites un plan que vous sauvegardez
  - `terraform plan -out myplan`
- Appliquer le plan une première fois (ou réaliser un apply)

    - `terraform apply   (terraform apply myplan)`
- Appliquer le plan une seconde fois
    - `terraform apply myplan`

TQ13 :
Que se passe-t-il au deuxième apply du plan ? Avez-vous une idée d'explication ? Est-il intéressant d'utiliser cette fonction plan sauvegardé et appliqué plus tard lorsque l'on travaille à plusieurs ?

Faites un `terraform destroy` en fin d'exercice

Multiplier les ressources et premier aperçu des variables

Si vous voulez ajouter une seconde VM totalement identique à la première, vous pouvez réécrire un second bloc de code ou utiliser "count"
(ou utilisez example.2.tf si vous utiliser le répertoire git)

```
resource "aws_instance" "test-instance" {
  count = 2

  ami             = "<AMI-ID>"
  instance_type = "t3.micro"

  tags = {
    Name = "my-test-vm-${count.index}"
    # Name = "${format("my-test-vm-%03d", count.index + 1)}"
    filter = chomp(file("/etc/hostname"))
  }
}
```

Mettez à jour votre code avec les éléments ci-dessus. Terraform va automatiquement peupler une variable count.index que vous pouvez utiliser dans votre bloc comme dans l'exemple.

TQ14 :
faites un terraform apply et jouer un peu en modifiant la valeur de count (en restant dans des limites raisonnables - ie. moins de 10). Est-il facile d'ajouter et de supprimer des instances ?

Comment fonctionne et quel est l'intérêt de la ligne commentée pour le tag ? Vous pouvez bien sûr la dé commenter (et commenter l'autre) pour tester avec un apply.

Vous pouvez faire un `terraform destroy` en fin d'exercice

Les variables dans terraform

Terraform propose 3 grandes familles de variables :

1. local values : qui sont un peu comme les variables locales/internes d'une fonction et ne peuvent être modifiées lors de l'exécution du code

2. input vars : qui sont comme les arguments d'une fonction. On va les définir et pouvoir passer des valeurs spécifiques lors de l'exécution
3. output vars : qui sont les valeurs de sortie de l'exécution que l'on souhaite conserver pour consultation ultérieure facilitée

Nous n'allons pas illustrer les local values qui n'ont pas de difficulté particulière et sont moins utilisées, vous pouvez vous reporter à la documentation si vous êtes curieux (https://developer.hashicorp.com/terraform/language/values/locals)

input vars

Les input vars sont les plus utilisés et vont permettre de réutiliser du code pour des besoins différents ou en changeant des paramètres facilement. On peut bien sûr également les utiliser comme des variables "internes".
Copier le code suivant dans un fichier ou utilisez le fichier example.3.tf.tmp

```
resource "aws_instance" "test-instance" {
  count = var.instance_count

  ami             = var.ami
  instance_type   = "${var.type}"

  tags = {
    Name = "my-test-vm-${count.index}"
    filter = chomp(file("/etc/hostname"))
  }
}

variable "instance_count" {
  type = number
  default = 2
}
variable "ami" {
  type = string
  description = "AMI reference ID"
  default = "<AMI-ID>"
}
variable "type" {
  type = string
}
```

Vous voyez que les variables sont déclarées directement dans le même fichier. On peut bien sûr (et c'est une bonne pratique) déclarer toutes les variables dans un fichier unique séparé (qu'on peut appeler variables.tf par exemple).

Les variables ont un type (obligatoire) et d'autres champs facultatifs. On voit que l'on peut assigner une valeur par défaut qui sera prise en compte si aucune valeur particulière n'est fournie au moment de l'exécution.

Il est possible de passer des valeurs pour les variables de différentes manières :

- Lors de l'exécution en entrant une valeur demandée par le prompt
- Sur la ligne de commande, par exemple
  - `terraform apply -var type="t3.small"`
- Via les variables de l'environnement shell en utilisant le préfixe TF_VAR_
  - `export TF_VAR_type="t3.micro"`
- En utilisant un fichier externe dans lequel on indiquera les valeurs
  - contenu : `type = "t3.nano"`
  - Si le fichier s'appelle : `terraform.tfvars` ou si le suffixe est `.auto.tfvars` il sera chargé automatiquement
  - Sinon, il faudra demander explicitement à le charger : `terraform apply -var-file=myvarsvalues.tfvars`

Tester les différentes possibilités de passer des valeurs de variables (en mettant des valeurs différentes) et réaliser les apply. Commencez bien sûr par ne rien donner pour expérimenter le mode prompt

TQ15 :
Si la valeur est définie dans TF_VAR_ via l'environnement et également via un fichier de variable et également sur la ligne de commande, quels sont les ordres de précédence ?

Pourquoi ne pas appeler la variable du simple nom de count au lieu de instance_count ?

BONUS : A l'aide de la documentation sur la notion de règles de validation, revoyez votre déclaration de variables pour ajouter une vérification empêchant que le nombre d'instances soit supérieur à 10 Vms

Nous avons vu la déclaration de 3 variables distinctes, vous pouvez regarder le fichier example.3.1.tf.tmp pour voir une façon différente de déclarer les variables.

TQ16 :
Quel intérêt de déclarer les variables ainsi ?

BONUS : comment déclarer des valeurs pour ces variables ?

output values

Les outputs permettent d'afficher en fin d'exécution les valeurs et propriétés de certaines ressources.
Cela peut également permettre de partager des informations et références entre différents projets terraform

Il faut donc déclarer des output values pour qu'elles soient ensuite peuplées par terraform à l'exécution.
Vous pouvez étudier le contenu du fichier `example.4.tf.tmp` et exécutez un `terraform apply` dessus

Vous pouvez ensuite accéder aux terraform output via la ligne de commande (si vous voulez réaliser des opérations de sélection plus complexes, il vous faudra utiliser l'outil jq pour parser la sortie en mode json

```
terraform output private_ip
terraform output -json instances | jq -r '.[].private_dns[0]'
```

TQ17 :
Qu'obtenez-vous comme sortie sur les commandes output ?

Donner un exemple de code permettant de capturer dans un output le nombre de core cpu de l'instance

Souvenez-vous que pour des opérations simples vous pouvez également utiliser terraform state show pour obtenir des informations depuis le tfstate sans avoir besoin de passer par les outputs. Vous pouvez aussi utiliser terraform show -json pour printer l'ensemble du state et le parser avec jq.

Faites un `terraform destroy` en fin d'exercice

Providers, ressources et datasources

Dans tous nos exemples, nous avons essentiellement créé des VMs (instances) sur le CLOUD provider AWS.
Mais bien sûr, terraform ne se limite pas à cela. Il est temps de parler rapidement des notions de providers, ressources et datasources.

Providers

Un provider est une sorte de driver permettant à terraform de s'adresser à un service que l'on peut piloter as code. Il peut s'agir d'un CLOUD provider, mais aussi d'un service SaaS ou de tout autre service pilotable par API.

La liste des providers est longue et s'étoffe continuellement :
https://registry.terraform.io/browse/providers

Cela va bien sûr de AWS, GCP, AZURE, à des composants comme les LoadBalancer de F5networks BigIP en passant par des outils de CI/CD comme GitLab par exemple.
On peut s'y perdre un peu et surtout (nous ne reparlerons mais c'est là qu'on va voir un recoupement avec les autres outils d'infrastructure as code et notamment ansible)

TQ18 :
Pensez à un service, logiciel ou une infrastructure que vous avez utilisé sur un projet récent et cherchez s'il existe comme provider. Indiquez la référence que vous l'ayez trouver ou non.

Ressources

Ensuite chaque provider a des ressources qu'il peut créer. En sélectionnant un provider, on va pouvoir obtenir la documentation de toutes les ressources qu'il peut gérer.
Evidemment, pour un provider comme AWS la liste est impressionnante :
https://registry.terraform.io/providers/hashicorp/aws/latest/docs

Et chaque ressources a un certain nombre d'options et de propriétés qu'il faut obligatoirement renseigner pour la création (ainsi que des facultatives), regardez la documentation de aws_instance :
https://registry.terraform.io/providers/hashicorp/aws/latest/docs/resources/instance

Vous voyez aussi la notion d'attributes dans la documentation d'une ressource pour indiquer les informations qui seront disponibles après création (et que l'on pourra capturer dans un output par exemple)

TQ19 :
En consultant la documentation, est-ce que vous pouvez indiquer comment spécifier que le disque de démarrage attaché par défaut à l'instance soit d'une taille de 50 Go ?

BONUS : Si vous aviez trouvé un service avec un provider terraform (questions précédentes), est-ce que vous avez trouvé les ressources que vous souhaitez configurer idéalement ? Pouvez-vous donner quelques exemples ?

Datasources

Les datasources vont permettre de référencer et intégrer des informations sur des éléments externes à notre code terraform.
Le cas le plus standard est de vouloir utiliser des références à des ressources qui sont déjà créées (potentiellement par d'autres pans de code).

On va par exemple gérer toute la partie réseau d'une infrastructure avec du code et un repository dédié car ce code va changer sans doute beaucoup moins souvent. On n'a pas non plus envie de lancer des apply sur cette base de code si rien ne change (pour éviter des problèmes de perfs mais aussi un bug qui casserait notre infrastructure alors qu'on ne souhaite rien changer)

On a donc un autre repository avec le code pour créer nos VMs sur ce réseau et là, il va changer beaucoup plus et on va détruire et recréer nos vms. Mais on doit le faire dans le réseau qu'on a construit avec l'autre code. On va alors utiliser les datasources pour obtenir les références et les utiliser dans notre code.

Le fichier `example.5.tf.tmp` va nous donner une illustration très simple. On veut nommer notre instance avec l'ID du VPC (le réseau privé virtuel) par défaut disponible dans notre compte. (C'est un exemple un peu hors sol mais qui fait appel à une ressource déjà existante)

TQ20 :

Faites un apply du fichier, que se passe-t-il ? Quel est le nom de votre instance ?

BONUS : Pouvez-vous indiquer un exemple de code factice d'une data source où vous devriez retrouver un VPC dont le nom (tag Name) est "myvpc" mais qui n'est pas forcément celui par défaut

Faites un `terraform destroy` en fin d'exercice

Pour les curieux (et si vous êtes en avance), vous pouvez jeter un oeil sur ces ressources et datasources un peu particuliers qui peuvent répondre à des cas d'usages spécifiques (nous n'allons pas les développer ici par manque de temps)
https://registry.terraform.io/providers/hashicorp/template/latest/docs/data-sources/file
https://registry.terraform.io/providers/hashicorp/local/latest/docs/data-sources/file
https://registry.terraform.io/providers/hashicorp/null/latest/docs/resources/resource

Enfin sachez qu'il existe également la possibilité de lancer des scripts custom mais il doit s'agir vraiment d'actions de dernière chance quand on ne peut vraiment pas faire autrement On préférera nettement utiliser ansible pour réaliser ce type d'action où l'on disposera d'une grande quantité de bibliothèques pour réaliser des actions sur un OS
https://developer.hashicorp.com/terraform/language/resources/provisioners/syntax#provisioners-are-a-last-resort

La notion de lifecycle et éviter les modifications et les dépendances

Nous allons tout d'abord parler du sujet du lifecyle et en particulier le ignore_change. Ces propriétés permettent d'autoriser des modifications de certaines parties des ressources en dehors de terraform et d'ignorer les conflits.

lifecycle

Regarder le code du fichier `example.6.tf.tmp`, vous pouvez faire un `terraform apply` dessus
Allez ensuite dans la console du cloud provider et choisissez une des deux instances :

- éteignez la (status stopped)
- ajoutez aussi un tag (balise en français) depuis la console de type : test:unevaleur
- modifiez le tag Name en mettant une autre valeur au hasard

Et maintenant vous pouvez à nouveau appliquer un `terraform apply`

TQ21 :
Est-ce que les tags sont remis conformément à ce qui existe dans votre code terraform ? Pouvez-vous expliquer ce qui se passe ?

BONUS : Est-ce que la VM repasse au statut démarré ? Pourquoi est-il nécessaire de gérer le state de la machine spécifiquement ? Que donnerait un ignore_change sur le instance_state ?

Vous pouvez voir dans la documentation qu'il y a d'autres propriétés pour lifecycle, on peut citer notamment **`prevent_destroy`** qui permet d'éviter de supprimer une ressource. Dans ce cas, il est préférable de pouvoir gérer ce type de ressources avec un repository dédié pour éviter des problèmes si on veut fréquemment détruire l'ensemble d'une infrastructure (pour des tests par exemple). En effet, le prevent_destroy nécessite d'être supprimé ou commenté dans le code pour pouvoir effectivement réaliser le terraform destroy.

Faites un **`terraform destroy`** en fin d'exercice

Dépendances

Parlons des dépendances en observant le code du fichier **`example.7.tf.tmp`**, vous pouvez faire un **`terraform apply`** dessus

*Il est possible que vous ayez une erreur liée au provisionner local qui n'est pas encore référencé. Dans ce cas, il suffit de refaire **`terraform init`** un avant de relancer le apply.*

Commencez par observer le tag Name de la première instance (celle nommée first-in-code). Ensuite, regarder le code : le bloc sur le provisioner file est volontairement commenté car pour fonctionner réellement, il a besoin de paramètres de connexion (par exemple via SSH) pour pouvoir copier le fichier sur l'instance nouvellement créée.

Nous illustrons ici le fait qu'on utilise le provisioner local-exec pour générer un fichier de configuration localement qui sera ensuite uploadé sur l'instance. Comme le fichier de config généré n'est pas une ressource dont terraform a réellement connaissance, on utilise depends_on pour indiquer à terraform qui a besoin de telle ressource avant de créer telle autre.

TQ22 :
Suite au apply, est-ce que la ressource first-in-code est créée en premier ? Comment expliquez-vous ce qui se passe ? Vous pouvez utiliser la commande terraform graph pour éventuellement vous aider.

BONUS : avez-vous des idées d'autres useCase pour l'utilisation de depends_on ?

Vous pouvez faire un **`terraform destroy`** en fin d'exercice

Notion de modules et éléments avancés

Si vous n'êtes pas en avance, vous pouvez sauter cette section qui est uniquement informative et à laquelle vous pourrez revenir un autre jour pour creuser.

La durée du TP ne nous permet pas de parcourir toutes les possibilités de terraform. Sachez simplement qu'il existe une notion de modules que l'on peut développer pour pouvoir réutiliser des grands types de fonctions.

Par exemple, dans vos divers projets, vous voudrez toujours créer des instances avec des paramètres figés et certains que vous voulez modifier. Vous voudrez sans doute que ces

instances appartiennent toujours à certains réseaux ou modèles de réseaux que vous aurez définis ailleurs dans vos landing zones.

Les modules vous permettront de coder tout cela et de les appeler ensuite dans vos différents projets. L'utilisation des modules demande une gestion des variables plus complexes car ces dernières doivent être définies de façon cohérente entre les modules et le code qui les appelle.

Pour les curieux, vous pouvez bien sûr consulter la documentation sur les modules https://developer.hashicorp.com/terraform/language/modules/develop

Il existe bien sûr d'autres propriétés de terraform que l'on voudra potentiellement utiliser sur des gros projets. On peut citer les éléments suivants :

- For each, qui va permettre de réaliser des sortes de boucles (c'est tout de même moins simple et plus limité que dans un langage plus traditionnel) https://developer.hashicorp.com/terraform/language/meta-arguments/for_each
- Les expressions qui permettent de gérer les valeurs dans le code, l'expression la plus simple étant les interpolations de variables mais cela peut aller beaucoup plus loin, on peut citer
    - Strings and templates
    - For expressions
    - Dynamic blocks
    - qui sont assez fréquemment utilisés : https://developer.hashicorp.com/terraform/language/expressions
- Enfin, on peut citer les fonctions disponibles dans terraform pour réaliser des manipulations sur des valeurs, voici des exemples (presque) au hasard :
    - Lookup
    - formatdate
    - cidrsubnet
    - https://developer.hashicorp.com/terraform/language/functions

Puisque l'on parle des fonctions, vous pouvez maintenant mieux comprendre ce qui se cachait derrière notre filter dans les tags : `filter = chomp(file("/etc/hostname"))`

La fonction file va lire un fichier local au controller terraform (notre VM RDP) et en extraire le contenu. Le fichier /etc/hostname contient donc le hostname de la vm RDP (par exemple vm00) ainsi qu'un retour chariot en fin de ligne.
Nous utilisons donc par dessus la fonction chomp qui supprimer les retours chariots.

Nous aurions pu bien sûr utiliser une variable que l'on définit pour alimenter ce tag mais dans ce cas, il aurait fallu déclarer la définition de la variable dès le premier fichier et devoir expliquer ce concept.

Connexion SSH

Ce paragraphe est essentiel car il sera également utilisé pour les travaux sur Ansible. Si vous n'arrivez pas à faire fonctionner la connexion SSH, demandez de l'aide au formateur

avant de passer à la suite.

Maintenant que nous savons comment fonctionne terraform pour créer des ressources d'infrastructure, nous allons ultérieurement utiliser ansible pour les configurer et y installer des logiciels.

Que l'on souhaite réaliser ce type d'action avec ansible ou manuellement, nous aurons besoin de nous connecter aux VMs que nous approvisionnons.

Pour cela, la méthode la plus standard est la connexion SSH. Le serveur SSH est présent sur quasiment toutes les distributions Linux et les cloud providers (comme AWS) proposent directement dans leur provider terraform (via les mécanismes de cloud init) de configurer une clé SSH pour le user par défaut de l'OS.

Commencez par générer une clé SSH dans le répertoire courant où vous gérez vos fichiers terraform. Tapez la commande suivante et **n'ajoutez pas de passphrase de protection** lorsque cela vous sera proposé. (L'ajout d'une passphrase nécessiterait plus tard pour ansible d'utiliser ssh-agent, c'est possible mais peu pratique pour notre TP)

```
ssh-keygen -f ./tf-key
```

vous avez maintenant un fichier tf-key qui est la clé privée et un fichier tf-key.pub qui est la clé publique à la racine de votre répertoire.
Vous allez indiquer à terraform d'utiliser la clé publique pour l'autoriser à la connexion, observez comment on lit le fichier localement (datasource) que vous venez de créez pour ensuite le passer à la ressources aws_keypair qui sera elle même référencée dans les vms déployées.
Vous pouvez utiliser le fichier ***`example.8.tf.tmp`***

```
data "local_file" "sshkey" {
  filename = "${path.module}/tf-key.pub"
}

resource "aws_key_pair" "keypair_ansible_test" {
  key_name = "sshkey-ansible-test-${chomp(file("/etc/hostname"))}"
  public_key = "${data.local_file.sshkey.content}"
  tags = {
    filter = chomp(file("/etc/hostname"))
  }
}

resource "aws_instance" "ansible_instance" {
  count = 2
  ami              = "<AMI-ID>"   #
https://cloud-images.ubuntu.com/locator/ec2/
  instance_type = "t3.small"
  key_name                =
aws_key_pair.keypair_ansible_test.key_name
  vpc_security_group_ids =  [aws_security_group.ansible_test.id]
```

```
  tags = {
    Name = "first-ansible-vm-${count.index}"
    filter = chomp(file("/etc/hostname"))
  }
}

resource "aws_security_group" "ansible_test" {
  name        = "ansible_test-${chomp(file("/etc/hostname"))}"
  description = "ansible_test-${chomp(file("/etc/hostname"))}"
  ingress {
    from_port   = 22
    to_port     = 22
    protocol    = "TCP"
    cidr_blocks = ["0.0.0.0/0"]
  }
  tags = {
    Name        = "ansible_test-${chomp(file("/etc/hostname"))}"
    filter = chomp(file("/etc/hostname"))
  }
}
```

Lancez ensuite le provisionnement via terraform apply puis récupérez l'IP de la première instance et tentez de vous y connecter

```
terraform state show aws_instance.ansible_instance[0] | grep public_ip
```

```
ssh -i tf-key ubuntu@<instance_ip>
```

Vous pouvez également valider que vous arrivez à vous connecter sur la deuxième instance (ansible_instance[1])

TQ23 :
Est-ce que vous arrivez bien à vous connecter sur les deux instances ?
A quoi sert la ressource aws_security_group et pourquoi est-elle nécessaire et comment est-elle associée à l'instance ?

Un dernier mot sur les stateFiles (et le multi-environnement)

Si vous n'êtes pas en avance, vous pouvez sauter cette section qui est uniquement informative et à laquelle vous pourrez revenir un autre jour pour creuser.

Nous en avons parlé au début et réalisé des exercices, mais nous n'avons pas du tout parlé de deux points important :

- Où stocker le tfstate ?
- Comment gérer des environnements multiples (DEV et PROD par exemple) ?

Stockage du tfstate

Nous avons pour le moment stocké le tfstate en local de la machine où nous lançons notre code terraform. C'est parfait pour nos exercices et peut même être utilisé lorsque l'on est seul à travailler sur un projet.

En revanche, dès que l'on travaille à plusieurs, il est impératif de partager le stateFile au risque d'avoir de très gros problèmes.

TQ24 (BONUS) :
BONUS : Quel est le risque si j'ai deux développeurs qui partagent le même code terraform et lance chacun de leur côté en local des terraform apply et destroy ?

On peut donc utiliser des backends différents du modèle par défaut qui est le backend local : https://developer.hashicorp.com/terraform/language/settings/backends/local

Dans les backends alternatifs, on peut citer par exemple S3 qui va permettre de stocker le tfstate sur un bucket AWS (stockage objet) et utiliser une base DynamoDB chez AWS pour la gestion des locks.
https://developer.hashicorp.com/terraform/language/settings/backends/s3

On peut également citer le backend HTTP qui peut être implémenté en écrivant un serveur HTTP qui va prendre en charge les méthodes attendues (il existe même des projets qui font cela : https://github.com/mikalstill/junkcode/tree/master/terraform/remote_state mais surtout certains produits CI/CD comme Gitlab offre ce type de fonctionnalités : https://docs.gitlab.com/ee/user/infrastructure/iac/terraform_state.html

Nous n'avons pas le temps de tester ces configurations qui demandent des dépendances. **Retenez surtout que dans un déploiement en entreprise, il est essentiel de partager le tfstate** (tout en gérant la problématique des locks pour les accès multiples et de la sécurité, le tfstate peut contenir des informations sensibles)

Retenez également qu'il n'est pas du tout recommandé d'utiliser GIT pour gérer le stateFile. C'est une fausse bonne idée car cela impose de récupérer la dernière version avant de lancer un apply. Si cette opération est omise, c'est la catastrophe (de plus, tant que la nouvelle version n'est pas poussée sur le serveur GIT, un autre utilisateur peut aussi se servir du fichier, la gestion des locks n'étant pas possible puisque GIT est justement fait pour fonctionner en mode décentralisé)

Multi-environnement

Une question légitime concerne la manière de pouvoir gérer plusieurs environnements. Par exemple, comment utiliser mon code pour déployer une VM de type DEV, puis la même de type PROD.

- Je peux dupliquer le code
- Je peux utiliser les workspaces
- Je peux utiliser des tfstate différents

La duplication du code peut fonctionner pour quelque chose de minimaliste mais n'est évidemment pas une bonne solution et désastreuse à maintenir.

L'utilisation des workspaces permet de définir des espaces de travail (par exemple dev et prod) et d'utiliser une variable spécifique $workspace qu'on va pouvoir ajouter dans les Tag ou noms de nos ressources pour les différencier.
De plus, un fichier tfstate sera automatiquement disponible pour chaque workspace (un sous répertoire de terraform.tfstate.d) et on passe d'un workspace à l'autre avec de simples commandes (terraform workspace new|select|delete).
https://developer.hashicorp.com/terraform/cli/workspaces
C'est très intéressant et pratique mais avec quand même des risques d'erreurs (on ne voit pas toujours sur quel workspace on est) et on ne peut pas avoir des credentials différents pour gérer les backends tfstate suivant le workspace (qui peut être un pré-requis de sécurité pour la production par exemple)

Enfin, on peut gérer des tfstate différents avec des backends différents en utilisant des variables que l'on pourra également utiliser pour nommer les ressources suivant l'environnement.
Par exemple TF_VAR_env et on ajoutera ${env} en suffixe de nos tags Name.
C'est la solution la plus souple et sur un environnement large, on verra qu'il est de toute façon vite nécessaire de découper le code en plusieurs parties suivant leur cycle de vie (infrastructure réseau, serveurs, services de plus haut niveau) et donc manager des tfstate différents.
Un avantage et non des moindre sera de limiter le "blast radius" en cas d'erreur ou de malveillance, on ne détruira pas toute l'infrastructure mais seulement une partie et seulement sur un environnement particulier.

Ansible

Pour nos premiers tests avec ansible, nous allons utiliser le code terraform du paragraphe "Connexion SSH" de la partie terraform.

Ainsi, vous avez 2 instances (ou plus en modifiant le count) qui sont accessibles en SSH à l'aide de la clé SSH que vous avez générée.

Premier test sur localhost

Simplement pour valider que l'installation de ansible sur le desktop RDP fonctionne correctement, vous pouvez lancer des commandes ansible qui vont cibler le host local (et n'ont pas besoin de configuration particulière)

En cas de difficulté sur le copier/coller de certaines commandes issues du fichier PDF vers la VM (des caractères ou des sauts de lignes peuvent poser souci), vous retrouverez les commandes dans un fichier commands.txt à la racine du répertoire ansible.

Tapez la commande suivante
```
ansible localhost -a 'uptime'
```

Vous devriez avoir une sortie du type
```
localhost | CHANGED | rc=0 >>
 15:48:46 up  2:13,  1 user,  load average: 0.22, 0.66, 0.93
```
Vous pouvez tester une autre commande
```
ansible localhost -m shell -a 'ls -l'
```

Un test sur les instances distantes

Commencez par approvisionner via terraform les VMs de tests et la configuration SSH qui correspond au paragraphe “Connexion SSH” de la partie terraform.

Sans inventaire

Vous pouvez spécifier une liste de hosts ou IP directement dans la commande ansible

```
ansible all -i "15.236.186.186," -u ubuntu -m shell -a 'uptime' -e
ansible_ssh_private_key_file=./tf-key
```

Indiquez bien le chemin vers votre clé privée si vous n’êtes pas dans le même répertoire et surtout utilisez une adresse IP d’une de vos VM existantes.

AQ01 :
Est-ce que vous arrivez à faire fonctionner la commande et obtenir une sortie ?
(ATTENTION à la virgule dans la liste des IP dans l’option -i)
Est-ce que vous pouvez lancer la commande sur vos deux instances, quel est le résultat ?
A quoi sert l’option -u et pourquoi avoir indiqué ubuntu ?

Avec inventaire et configuration

Vous pouvez ensuite vous situer dans un répertoire ansible situé au même niveau que le répertoire terraform. Dans ce répertoire, vous allez créer un fichier inventory qui doit contenir les IP des deux premières instances que vous venez de créer.

```
[servers]
15.236.186.186
13.38.131.145
[group1]
15.236.186.186
[group2]
13.38.131.145
[all:vars]
ansible_user=ubuntu
ansible_ssh_private_key_file=../terraform/tf-key
```

Vous pouvez maintenant lancer les commandes ansible

```
ansible all -i inventory -m shell -a 'uptime'
ansible group1 -i inventory -m shell -a 'uptime'
```

Vous remarquez que nous avons reporté dans notre inventaire plusieurs fois les IP de nos machines 1 et 2 dans des groupes différents (elles sont toutes les deux dans le groupe servers et réparties dans le group1 et le group2)

AQ02 :
Quelle est la différence entre ces deux dernières commandes ?
A quoi cela peut-il servir par la suite ?

Si vous avez des problèmes avec l'inventaire ou que vous voulez valider qu'il fonctionne comme vous le souhaitez, vous pouvez utiliser le type de commande suivante :

```
ansible-inventory -i inventory --graph
ansible-inventory -i inventory --host <hostname ou IP du host dans l'inventaire>
```

AQ03 :
Que vous affiche la commande avec --host sur le host qui est dans le group1 ?

Vous pouvez également déclarer certaines directives de configurations dans un fichier ansible.cfg à la racine du répertoire où vous allez lancer les commandes ansible (celui ou il y a le fichier inventory)

Il y a déja un fichier exemple pour démarrer

```
[defaults]
timeout = 10
host_key_checking = False
ansible_python_interpreter=/usr/bin/python3
interpreter_python=/usr/bin/python3
remote_user=ubuntu
private_key_file=../terraform/tf-key
```

AQ04 : BONUS
En vous aidant de la documentation officielle ansible (ou en testant), quelle est la précédence entre `private_key_file` dans le ansible.cfg et `ansible_ssh_private_key_file` dans l'inventaire ?

Variables

Un élément très important de Ansible concerne les variables que vous allez pouvoir utiliser (pour celles qui sont définies par défaut) ainsi celles que vous allez définir et ce de différentes manières.

Dans le répertoire ansible, vous verrez un sous répertoire group_vars, comme son nom l'indique, ce répertoire permet d'affecter des variables à certains groupes ou à tous les groupes.

Vous pouvez utiliser le module debug pour afficher la valeur d'une variable
Testez avec quelques variables qui sont déjà définies par ansible

```
ansible group1 -i inventory -m debug -a "var=ansible_user"
ansible group1 -i inventory -m debug -a "var=groups"
```

Vous pouvez aussi afficher plusieurs variables avec msg

```
ansible all -i inventory -m debug -a "msg='{{ all_var }} {{ group1_var }}'"
```

AQ05 :
Qu'obtenez-vous comme résultat ?
Comment expliquez-vous qu'un des hosts ait deux fois le résultat all.yml ? (vous pouvez consulter le contenu des fichiers group_vars et avoir en tête que ansible va charger les fichiers suivant la présence des hosts dans certains groupes)

Vous pouvez également définir des variables directement dans l'inventaire ansible
Ajouter dans l'inventaire ces lignes (par exemple tout à la fin du fichier)

```
[group1:vars]
my_var=inventory
```

Jouer maintenant la commande suivante :

```
ansible all -i inventory -m debug -a "msg='{{ my_var }} {{ all_var }} {{ group1_var }}'" -e all_var=myvalue
```

AQ06 :
Qu'obtenez-vous comme résultat ?
Avez-vous une idée de différentes façons de régler l'erreur de variable non définie pour un des hosts ?

Avez-vous une idée pour expliquer le comportement de la valeur de all_var ?

Pour votre information, sachez que vous pouvez définir des variables durant l'exécution d'ansible avec set_fact pour par exemple assigner une valeur que vous récupérez dynamiquement lors de l'exécution. Ceci n'est pas possible en mode ad-hoc et uniquement au sein d'un playbook ou d'un rôle (que vous allez aborder plus tard). En effet, le mode ad-hoc (ligne de commande) ne permet de lancer qu'un seul module à la fois.

Enfin, ansible définit pour vous toute une série de variables que vous pouvez afficher avec le module setup

```
ansible group1 -i inventory -m setup
```

Attention, ces variables ne sont pas toutes disponibles quand on utilise le module debug car il faudrait lancer aussi le module setup. Nous pourrons surtout faire cela quand nous ferons des playbooks et des roles. C'est la limite de ansible en ligne de commande adhoc.

AQ07 :
Arrivez-vous à trouver une manière en utilisant le module setup pour afficher l'adresse ipv4 interne des hosts ansible de l'inventaire (vous pouvez utiliser grep). Vous ne pourrez pas afficher l'ExternalIP car elle n'est pas connue de l'OS, seulement au niveau du cloud provider.

Voilà, nous avons vu plusieurs manières de définir des variables et vous avez sans doute remarqué que si une variable est définie plusieurs fois, seule une valeur sera prise en compte.
On parle de précédence. Il faut consulter la documentation pour voir tous les endroits où vous pouvez définir une variable et l'ordre dans lequel est gérée la précédence :
https://docs.ansible.com/ansible/latest/playbook_guide/playbooks_variables.html#variable-precedence-where-should-i-put-a-variable

Documentation et aide sur les modules

Nous avons vu rapidement l'utilisation des modules en ligne de commande. Ces modules sont des fonctions ansible qui vont nous permettre d'automatiser les tâches.

Il existe un très grand nombre de modules intégrés dans ansible (et on peut étendre en développant ses propres modules ou en ajoutant des collections externes qui ne sont pas intégrées dans le noyau ansible)

Pour obtenir la liste de tous les modules :
```
ansible-doc -l
```

Pour consulter la documentation sur un module :
```
ansible-doc setup
ansible-doc debug
```

On trouve également très facilement la documentation en mode web d'un module en recherchant ansible nom du module dans google, exemple :
https://docs.ansible.com/ansible/latest/collections/ansible/builtin/debug_module.html

AQ08 :
En consultant la documentation du module setup, pouvez-vous trouver une nouvelle façon d'afficher l'adress ipv4 des hosts de l'inventaire ?

Playbooks

Nous avons beaucoup utilisé ansible en mode adhoc / inline ou encore appelé en ligne de commande.

La plupart du temps, nous allons utiliser ansible pour réaliser une multitude de tâches d'installation et de configuration sur plusieurs hosts appartenant à des groupes différents car ils auront eux mêmes des fonctions différentes et nous voudront configurer certains logiciels sur un groupe (par exemple des webservers) et d'autres ailleurs (bases de données).

Les playbooks sont l'élément de plus haut niveau qui vont nous permettre de lancer des tâches sur des hosts.

Toujours dans le répertoire ansible, il y a un fichier playbook_standalone.yml
Regarder le contenu du fichier. Nous pouvons remarquer que le format est le yaml (le principe important concerne l'indentation des éléments qui doit se faire avec des séries d'espaces, les tabulations sont interdites)

Vous voyez qu'on va cibler l'ensemble des hosts (on pourrait cibler un groupe particulier ou plusieurs ou un seul host)

On va ensuite lancer une série de tâches (tasks) auxquelles on a donné un nom (c'est optionnel mais c'est une bonne pratique)
Vous voyez qu'on appelle le module en y faisant référence (debug , shell, ansible.builtin.user) et on passe ensuite les options

On a parfois des options globale à la tâche comme **failed_when** qui va ici indiquer que l'on ne veut jamais que la tâche soit considérée en erreur quel que soit son résultat
Un autre exemple est **become** qui va indiquer que l'on veut prendre l'identité d'un autre utilisateur, par défaut il s'agit du user root et cela correspond à faire des commandes sudo (c'est nécessaire pour créer un utilisateur)

Vous pouvez ensuite exécuter le playbook (on utilise le -v pour avoir une sortie verbeuse)
```
ansible-playbook -i inventory playbook_standalone.yml -v
```

Vous pouvez relancer plusieurs fois le playbook pour voir si vous constatez des différences.
Vous pouvez également faire une série de tests en commentant les lignes **failed_when** ou **become** ainsi que le **state: absent** pour le module user

AQ09 :
Quand vous lancez plusieurs fois le playbook, est-ce que la sortie pour le module ansible.builtin.user change ? Pouvez-vous expliquer ce qui se passe ?

Jouer le playbook avec le state absent pour le user afin de supprimer ce user. Ensuite, commentez la ligne failed_when et relancez le playbook, est-ce qu'une erreur s'affiche ? Que pensez-vous du module shell, est-ce une bonne idée de l'utiliser ?

Pouvez-vous expliquer votre compréhension de la signification des différents états fournis dans la sortie d'ansible (ok, changed, failed) ?

Choix du chemin d’apprentissage

A partir d’ici deux choix s’offrent à vous pour poursuivre le TP. Ce choix n’implique que l’ordre dans lequel vous allez réaliser les étapes suivantes :

- première possibilité : vous poursuivez avec l’ordre proposé. Cela signifie que vous allez maintenant voir les différents concepts d’Ansible avec des exemples et des manipulations.
- seconde possibilité, vous allez directement à la partie demoboard (Page 41) et vous suivez les 4 premières pages (jusqu’à Travaux d’optimisation - exercices) où vous allez déployer demoboard avec du code terraform et Ansible complet. Et vous reviendrez ensuite ici pour finaliser le déroulé des concepts Ansible avant de terminer ensuite par les exercices demoboard

Ces deux possibilités permettent aux étudiants préférant d’abord bien comprendre tous les aspects de choisir le premier chemin et à ceux qui préfèrent déjà voir du code en action et revenir ensuite sur chaque concept de choisir le second chemin

Rôles

Le playbook que nous avons vu ne faisait que lancer des tâches définies localement dans le playbook.
Souvent, on va souhaiter réutiliser des séries de tâches un peu comme des fonctions. On va pouvoir utiliser les rôles pour cela.
Combiner avec les notions de groupes, cela va nous permettre une grande flexibilité pour exécuter des tâches sur certains groupes de hosts.

Pour illustrer ces points nous allons regarder le fichier **playbook_with_role.yml**
Ce role contient beaucoup d’éléments et va nous permettre d’illustrer de nombreuses fonctionnalités et caractéristiques d’ansible

Il est important de consigner dans un fichier les résultats des différentes exécutions du playbook que vous réaliserez afin de pouvoir vous y reporter pour répondre à certaines questions posées ultérieurement (ne comptes pas uniquement sur l’historique de votre shell pour cela)
Utilisez bien systématiquement le -v dans vos commandes pour passer en mode verbose et afficher le résultat de l’exécution des tasks de type shell qui sont là pour contrôler des résultats et afficher des informations.

Organisation du role

Vous pouvez commencer par observer le contenu du fichier de playbook, on a des tasks mais aussi un appel aux roles (qui seront cherchés par défaut dans le sous-répertoire courant nommé roles, on peut modifier ce comportement via le ansible.cfg par exemple)

Vous voyez aussi lors de l’appel du role example_role que l’on en profite pour définir une nouvelle variable (example_var_list) qui est une liste. Elle sera de fait disponible dans le role et nous verrons comment l’utiliser.

Vous pouvez maintenant regarder le répertoire roles qui ne contient qu'un sous-répertoire example_role (on pourrait bien sûr avoir plusieurs roles ici).
Vous observez des sous-répertoires pour example_role :

- defaults : qui va pouvoir contenir des fichiers avec des variables définies pour ce rôle (et qui peuvent bien sûr être surchargées via les règles de précédence)
- files : qui peut contenir tout type de fichier statique qui seront à copier sur les machines distantes (cela peut être des fichiers de configuration voire même des applications). On peut bien sûr utiliser d'autres sources que ce répertoire mais ansible ira par défaut chercher ici à l'exécution de certains modules
- tasks : qui peut contenir un ou plusieurs fichiers contenant les tâches à exécuter (c'est toujours main.yml qui sera exécuté si rien d'autre n'est précisé à l'import du module)
- templates : un peu comme files (ansible vient chercher par défaut ici) mais cette fois on y trouvera des fichiers qui peuvent être construits dynamiquement en utilisant le langage de templating jinja

Nous allons donc maintenant pouvoir lancer ce playbook et le commenter tâches par tâches dans le code
Conseil, n'oubliez pas le -v et copier le résultat (sortie standard ansible) dans un fichier (au moins à la première exécution) pour vous y reporter et vous aider à répondre à certaines questions

```
ansible-playbook -i inventory playbook_with_role.yml -v
```

Analysons maintenant différentes fonctionnalités majeures de ansible en observant le code et le résultat de différentes exécutions.
Par défaut, nous consultons le fichier main.yml dans le répertoire example_role/tasks puisque c'est là que réside les directives du role

AQ10 :
La première tâche (du role, pas du playbook) affiche la valeur de deux variables. D'où provient la valeur de role_var ?
Est-ce que vous arrivez à expliquer la valeur qui s'affiche pour group1_var suivant le host ?

Conditions when

Ansible ne propose pas à proprement parler de structure if/else, c'est une de ses grosses limitations lorsque l'on est habitué à des langages de développement.

On observe donc dans le code une tâche qui va créer un utilisateur foo (nous avons déjà vu cela). Il y a une subtilité avec **register: user_creation**
ici, on va en fait déclarer une nouvelle variable dans laquelle on va affecter le résultat de la taĉhe (en mode simplifié, vous pouvez imaginer qu'après l'exécution de la taĉhe, vous aurez dans la variable user_creation le même contenu que ce que vous voyez en mode verbose pour la sortie d'une taĉhe)

La tâche suivante est un simple affichage de texte, mais justement, on ne souhaite l'afficher que si la création du user a réellement eu lieu. SI on relance le playbook et que le user existe déjà, on ne veut pas afficher ce texte.
On utilise donc une clause **when: user_creation.changed** pour indiquer que l'on ne veut exécuter cette tâche que si la condition est vraie (par défaut si on ne précise rien, il faut que la condition soit vraie).
Ici on va utiliser un élément de la variable qui indique toujours si ansible a réalisé ou non un changement.

AQ11 :
Si vous lancez plusieurs fois le playbook, est-ce que vous pouvez illustrer le fait que la tâche conditionnelle n'est pas exécutée à chaque fois ?

En vous aidant de la documentation et aussi en regardant dans le reste du code, pouvez-vous indiquer comment conditionner l'exécution de la tâche au fait que la variable role_var contiennent la valeur "execute"
https://docs.ansible.com/ansible/latest/playbook_guide/playbooks_conditionals.html

Copie de fichiers, templating jinja et manipulation

Souvent nous avons besoin de copier et manipuler des fichiers sur les machines distantes (notamment pour gérer les fichiers de configurations des logiciels, par exemple un serveur web apache)

Nous allons illustrer tout cela dans les tâches suivantes, celle qui sont situées entre :
- name: copy a local file to remote hosts
- name: print content of file to verify lineinfile behavior

La première utilise donc le module copy qui va chercher par défaut dans les répertoires du rôle ansible s'il trouve le fichier src
On utilise ensuite une simple commande module shell pour afficher (n'oubliez pas le mode verbose -v) si le fichier a bien été créé et avec les bons droits. Vous pouvez bien sûr vous connecter vous-mêmes en SSH sur la machine pour aller vérifier par vous mêmes que les fichiers sont bien là.

Ensuite, nous définissons une variable avec une condition pour qu'elle n'existe que si le host appartiennent au group1

Maintenant, nous allons utiliser le module template qui prend en source un fichier de template que vous devez aussi analyser : **myfile.txt.j2**
Le principe du templating est que le moteur jinja va pouvoir remplacer des variables avec leur valeurs au moment de l'exécution
Vous pouvez voir que jinja est un langage complet intégrant des conditions et des boucles (cela dépasse le cadre du TP mais si besoin la documentation en ligne est très complète : https://jinja.palletsprojects.com/en/3.1.x/templates/)

Enfin la dernière tâche va également afficher le résultat et le contenu du fichier templatisé. Vous pouvez bien sûr vous connecter directement sur la machine pour voir le résultat.

AQ12 :
Pouvez-vous confirmer (capture écran ou autre) que le résultat pour les hosts du group1 contient la ligne supplémentaire dans le résultat du fichier templatisé ? Pouvez-vous expliquer pourquoi ?

Est-ce que vous pouvez indiquer le résultat du fichier templatisé si l'on changeait la boucle for avec groups['group2'] ?

La tâche template contient la directive become: true que nous avions déjà vue. Pourquoi est-elle indispensable ici ?

Le système des templates est parfait lorsque l'on maîtrise l'ensemble du fichier mais comment faire pour éditer des fichiers qui sont déjà existants et par défaut sur un serveur (fichier /etc/hosts ou fichier d'un serveur web dont on voudrait conserver le contenu existant même s'il a été modifié par un opérateur manuellement)

Nous allons donc utiliser le module lineinfile qui permet de faire cela.
Nous allons ici faire un exemple très simple où vous voyez la recherche d'un pattern et la modification. Si vous n'êtes pas familier des regex (expressions régulières) vous pouvez tout de même facilement voir le résultat lors de l'exécution

AQ13 :
Si vous lancez plusieurs fois le playbook, pourquoi est-ce que le lineinfile est toujours à changed alors que si le fichier a bien été modifié il ne devrait pas avoir besoin de le remodifier ? Pouvez-vous indiquer comment éviter ce changement à chaque fois (vous avez le droit de modifier les autres tâches pour cela si besoin)

A quoi sert l'option backup pour le lineinfile ? Pouvez-vous illustrer le contenu d'un backup en vous connectant sur une des machines ?

En consultant la documentation de lineinfile (https://docs.ansible.com/ansible/latest/collections/ansible/builtin/lineinfile_module.html), pouvez-vous expliquer comment ajouter une ligne après une autre ligne particulière déjà existante ?

Gestion des boucles dans les tâches

La task suivante illustre la possibilité de réaliser des boucles pour éviter de devoir dupliquer du code. C'est relativement limité par rapport à des langages de développement standard (notamment les conditions de sorties d'une boucle) mais cela est tout de même très utile.

On voit que la boucle va utiliser les éléments d'une variable de type list pour itérer dessus. Chaque élément de la liste sera disponible dans la variable item durant l'itération.
Sachez que l'on peut donner un nom de variable différent de item si on le souhaite en utilisant loop_control

Pour information pour faire des boucles, il existe aussi with_items qui est déprécié : https://docs.ansible.com/ansible/latest/playbook_guide/playbooks_loops.html#migrating-from-with-x-to-loop mais vous pourrez le voir dans du code existant sur des projets.

AQ14 :
Il y a une directive **run_once: true** sur cette tâche, quel est son effet ? (vous pouvez la commenter et lancer le playbook pour voir la différence)
A votre avis dans quels cas cela peut être utile ?

BONUS : Pouvez-vous donner un exemple de code en utilisant loop et loop_control pour faire la même chose mais utiliser element au lieu de item ?

Déléguer l'exécution des tâches à un host particulier

La task suivante va illustrer la délégation pour l'exécution d'une tâche sur un autre host.

Nous allons lancer une simple commande hostname via le module shell et vous notez la syntaxe delegate_to un peu particulière qui va en fait prendre le premier élément de la liste des hosts du group1.

Une utilisation commune du delegate est delegate_to localhost pour demander à exécuter la tâche sur le controller ansible lui-même. Il y a plusieurs cas d'usage mais l'un peut être de réaliser un test de connectivité depuis un certain host vers tous ceux que l'on vient de configurer.

AQ15 :
Que constatez-vous à l'exécution de la task ? Est-ce qu'elle est exécutée sur le host que nous avons spécifié ? Est-ce qu'elle est exécutée une ou plusieurs fois ?

Au passage, on a donc vu très rapidement que l'on peut utiliser un filtre sur une valeur ou une variable. Ansible propose une quantité non négligeable de filtres pour réaliser des transformations, tris ou sélection de valeurs, les curieux peuvent consulter la documentation : https://docs.ansible.com/ansible/latest/playbook_guide/playbooks_filters.html

Utilisation des tag pour conditionner l'exécution

La task suivante nous présente le concept de tags qui permet de ne jouer qu'une partie d'un playbook ou d'un rôle ou de sauter certaines étapes.

Le principe est que l'on peut ajouter un ou plusieurs tags sur des tâches (voire des rôles complets au niveau du playbook) et on va pouvoir choisir ensuite les tags que l'on souhaite jouer ou sauter.

Vous pouvez utiliser/tester les commandes suivantes :

```
ansible-playbook -i inventory playbook_with_role.yml --tags test_tag
ansible-playbook -i inventory playbook_with_role.yml --skip-tags
test_tag
```

AQ16 :
Est-ce que la sélection des tags fonctionne bien ? Quelles sont les limitations que vous pouvez envisager sur l'utilisation des tags ?

BONUS : Regarder la documentation --start-at-task et tentez de jouer le playbook

uniquement à partir de cette tâche mais sans utiliser les tags. Quelle est la différence avec les tags ?

Retry sur les tasks

La task suivante est là pour illustrer comment demander à Ansible de réessayer (retry) de jouer une task si cette dernière ne s'exécute pas correctement.

C'est un élément très intéressant car dans des systèmes distribués avec des appels API, il peut arriver qu'une API soit indisponible pour quelques secondes et il est dommage de devoir stopper toute une exécution et qu'un opérateur se demande s'il faut tout relancer (surtout qu'un long playbook peut prendre du temps)

C'est en revanche une gymnastique que d'implémenter tous les retry et de définir les conditions de succès.

AQ17 :
Que se passe-t-il à l'exécution de la tâche ? Est-ce qu'elle se termine en FAILED et est-ce que l'exécution continue ? Pouvez-vous expliquer pourquoi ?

Pouvez-vous proposer une condition until: qui va permettre que la task soit considérée comme réussie ?

BONUS : Pouvez-vous expliquer la différence entre failed_when: et ignore_errors: ?

Import et inclusion de tasks

La task suivante montre un include de tasks pour justement pouvoir définir des tasks dans d'autres fichiers.

La différence entre import tasks et include tasks réside dans le fait que le include permet des imports dynamique. Cela peut permettre d'inclure une task qui porte le nom de la valeur d'une variable.

https://docs.ansible.com/ansible/devel/playbook_guide/playbooks_reuse.html#includes-vs-imports

Exercices divers

AQ18 :
Pouvez-vous ajouter une tâche qui permet de supprimer en fin de rôle et à coup sûr le fichier mytestfile.txt (aidez-vous de la documentation du module file) ?

AQ19 :
En consultant la documentation du module copy, pouvez-vous faire un renommage/déplacement d'un fichier sur une machine distante ? Pourquoi est-il préférable d'utiliser copy plutôt qu'un simple mv via le module shell ?

AQ20 :
Depuis la console AWS, vous allez éteindre une des deux instances, relancez ensuite le playbook, que constatez-vous ? Quels sont les avantages et inconvénients de ce comportement ?

AQ21 :
Ajoutez une task au début du rôle (ou à la fin mais ce sera plus long) de type shell selon le code suivant :

```
- shell:
    cmd: mkdir /var/tmp/test
```

Lancer votre playbook une première fois puis une seconde. Vous devriez avoir un problème, votre task n'est pas idempotente. Comment pouvez-vous la modifier pour qu'elle soit idempotente (et idéalement gère correctement les code retour Ansible ok et changed) ? (vous pouvez utiliser un autre module que shell)

AQ22 (BONUS) :
BONUS : Ajoutez une nouvelle instance (ou 2) en augmentant le count dans le code terraform, relancez un apply pour les ajouter. Récupérez ensuite l'adresse IP correspondante et ajoutez les dans l'inventaire (group1 et/ou group2 à votre choix).
Vous pouvez relancer le playbook complet, que se passe-t-il ? Est-ce que tout fonctionne bien ? Notez-vous des éléments intéressants (notamment les tâches avec des conditions pour une première exécution uniquement ?)

Autres fonctionnalités intéressantes

Si vous n'êtes pas en avance, vous pouvez sauter cette section qui est uniquement informative et à laquelle vous pourrez revenir un autre jour pour creuser.

Ansible est très riche (et continue à s'étoffer), il est donc difficile ne serait-ce que de survoler l'ensemble. Les paragraphes suivants décrivent certaines fonctionnalités intéressantes pour la gestion d'un parc complet, nous ne les détaillerons pas (pas d'exercice) mais vous pourrez les voir ultérieurement implémentés et en action dans le cas concret en fin de TP.

Stratégies de déploiement

Sachez que l'on peut définir le comportement d'ansible sur la manière dont il va gérer les groupes de hosts et les tasks.
Par défaut, il utilise une stratégie "linear" qui exécute les tasks en séquence et attend d'avoir traité tous les hosts concernés avant de passer à la task suivante. Mais il existe d'autres stratégie comme "free" qui propose de passer un host à la task suivante dès qu'il a terminé même si ses petits camarades sont encore en cours.

Autre axe, vous pouvez spécifier le nombre de hosts par batch avec "serial". Par défaut, ansible va exécuter les tasks en parallèle sur l'ensemble des hosts. Vous pouvez avec serial lui indiquer de ne le faire que par groupe de 3, cela signifie qu'il va jouer l'ensemble du playbook pour les 3 premiers puis recommencer avec les 3 suivants, etc....

Intéressant, vous pouvez utiliser des pourcentage dans serial et spécifier une liste. Vous pouvez ainsi spécifier votre plan de déploiement progressif pour tester sur une machine puis si tout va bien continuer :

```
- name: test play
  hosts: webservers
  serial:
    - 1
    - "50%"
    - "100%"
```

“throttle” permet de limiter le nombre de hosts que l’on va traiter en parallèle dans une task (rappel serial concerne le nombre de hosts par lot pour l’ensemble du playbook)

Vous pouvez aussi spécifier avec “order” comment traiter vos serveurs (dans l’ordre de l’inventaire, ou dans l’ordre alphabétique de l’inventaire, ou en reverse order, etc…)

Certains mot-clés dans les tâches ou le playbook gère le comportement de ansible en cas d’erreur (comme ignore_errors, any_error_fatal) si on veut arrêter ou non une exécution suivant ce qui se passe.

https://docs.ansible.com/ansible/latest/playbook_guide/playbooks_strategies.html

Enfin, les patterns permettent aussi de spécifier seulement un certain nombre de hosts dans la liste de l’inventaire plutôt que de tous les prendre.
https://docs.ansible.com/ansible/latest/inventory_guide/intro_patterns.html#using-group-position-in-patterns

Inventaire dynamique

Nous avons construit notre fichier d’inventaire manuellement, c’est bien mais imaginez si vous aviez des centaines de machines ! Et surtout, nous utilisons un cloud provider (AWS) qui expose une API nous permettant de lister nos instances.

Il existe donc un concept d’inventaire dynamique (on peut écrire son propre inventaire dynamique en python assez simplement).
Il existe des inventaires pour les cloud providers, celui pour AWS est :
https://docs.ansible.com/ansible/latest/collections/amazon/aws/aws_ec2_inventory.html

Il faut qu’il puisse accéder à vos credentials et vous devez ensuite le configurer. Il va vous permettre de construire automatiquement l’inventaire avec plein de fonctionnalités intéressantes, notamment construire des groupes dynamiquement avec les tag présent sur vos VMs (que vous avez mis en place via terraform). On peut également inclure ou exclure des hosts de l’inventaire car la liste contiendra toutes les instances de votre compte (par exemple plusieurs environnements)

Durant le cas concret de fin de TP, nous utiliserons l’inventaire dynamique et vous pourrez le voir en action (vous disposerez ainsi d’un exemple sans toutefois rentrer dans le détail de sa configuration)

Handler

Les handlers permettent d'exécuter des tasks seulement lorsqu'une autre task a été réalisée. Bien sûr, on peut utiliser des conditions pour réaliser cela mais cela peut vite être lourd à gérer alors que les handlers le font automatiquement.

Un cas d'usage standard est de redémarrer un serveur (webserver apache par exemple) uniquement lorsque sa configuration a changée.
Ainsi, vous avez une première tâche qui génère le fichier de config avec un template, la première exécution va modifier ce fichier et il faut le prendre en compte au niveau serveur. Mais les fois suivantes, le fichier ne sera pas changé et il est inutile de redémarrer/reloader le serveur.

La particularité des handlers est qu'ils ne sont exécutés qu'à la fin de l'ensemble des tâches même si la notification d'appel est au milieu du rôle. Cela peut poser problème parfois et il est donc possible de forcer l'exécution des handlers à l'aide de la directive flush_handlers que l'on appelle dans une task.

https://docs.ansible.com/ansible/latest/playbook_guide/playbooks_handlers.html

Dans le cas concret en fin de TP, vous verrez également l'utilisation de handler en action.

ansible-galaxy

Vous avez vu que le nombre de modules "built-in" est assez impressionnant. Mais ce n'est rien, vous avez sans doute envie de configurer d'autres logiciels ou services (un peu comme avec les providers terraform).

Ansible peut être étendu car tout le monde peut écrire ses propres modules et les publier sous forme de collections. ansible-galaxy est un ensemble de règles pour développer et publier ses travaux ainsi qu'un registry de référence pour la publication.

Les collections vont de petits projets d'une personne ou d'une petite communauté à des produits commerciaux ou open-source de grande ampleur (f5bigip, mysql, …)
https://galaxy.ansible.com/ui/repo/published/f5networks/f5_bigip/
https://galaxy.ansible.com/ui/repo/published/community/mysql/

Les commandes ansible-galaxy vous permettent d'installer des collections mais aussi de générer un squelette de répertoire pour votre propre module ou role en suivant les bonnes pratiques d'organisation ansible.

Retenez surtout que ansible-galaxy est un catalogue permettant d'étendre les capacités d'ansible à tout type de services à manager.
Allez faire un tour rapide sur le site et regardez si vous trouvez les infrastructures ou logiciels que vous aimeriez manager avec ansible.

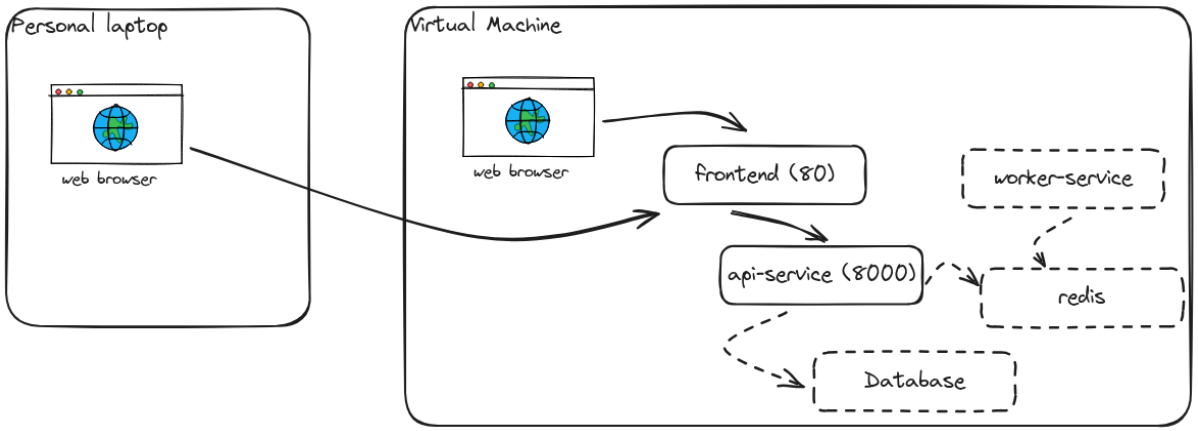
Un exemple concret : l'application demoboard

Afin de rendre cela plus concret, nous allons déployer une application de démonstration.

à des fins pédagogiques, nous allons donc utiliser l'application demoboard conçue pour illustrer les TP.

demoboard permet d'ajouter (et supprimer) des tâches fictives dans une liste et éventuellement de lancer un traitement associé à ces tâches en mode asynchrone et afficher le statut de ces traitements.

Architecture globale de l'application



Les éléments en pointillés correspondent au mode complet (permettant de lancer les traitements).

L'API de demoboard et le worker service sont écrits en Python. On peut utiliser des moteurs SGBD standards (mysql, postgreSQL) pour le backend.

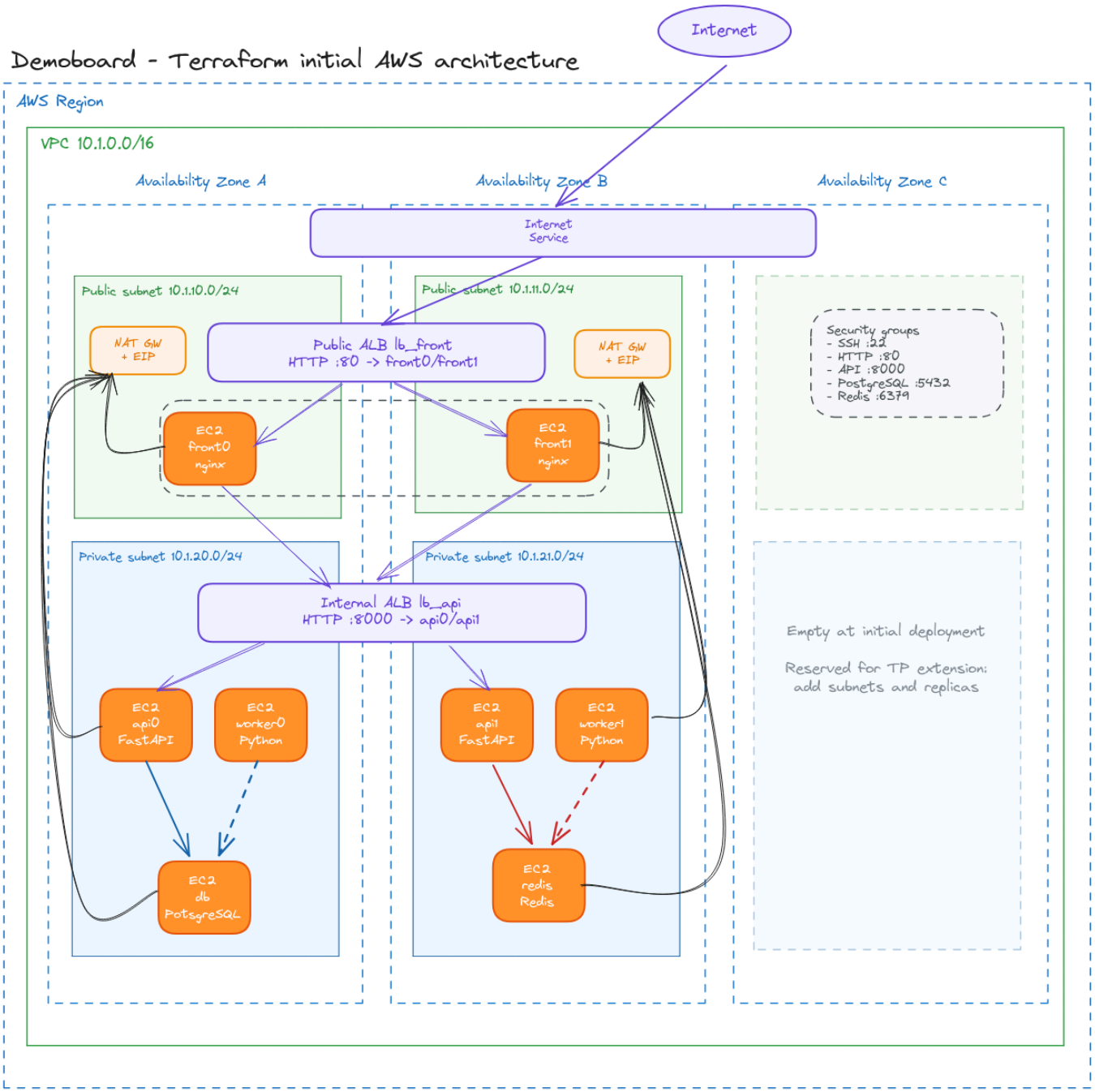
Vous n'allez pas devoir créer tout le code terraform et ansible car ce serait beaucoup trop long dans le cadre de ce TP.
Vous allez disposer d'un code fonctionnel que vous exécuterez et sur lequel vous pourrez apporter des améliorations.

L'ensemble du code se trouve dans le répertoire $HOME/tpcs-iac/demoboard
Nous allons travailler à partir de là

Avant cela, vérifiez bien que vous avez supprimé (`terraform destroy`) l'ensemble de vos précédents travaux. Il vous faut pour cela être situé dans le répertoire où vous aviez vos précédents exercices terraform.

Création de l’infrastructure avec terraform

Nous allons maintenant déployer une infrastructure complète avec son propre réseau (VPC) proche d’un design qui pourrait être déployé en production.
Voici un schéma non exhaustif (les security groups sont simplifiés) de ce qui va être déployé par le code terraform



Les éléments à savoir :

- Les public subnet le sont du fait d’une route vers l’internet service (ils sont donc accessibles depuis INTERNET)

- Les private subnets ont une route vers la NAT gateway du subnet public correspondant pour sortir sur INTERNET (téléchargement des paquets d'installation par exemple)
- Le public et private subnet sur la 3ème AZ n'existe pas dans le code. Il est représenté car c'est un des travaux possibles ultérieurement dans le TP
- Seul le frontend (nginx) doit être accessible depuis l'extérieur pour les appels des clients, donc le LB pour l'API est situé dans le private subnet.
- Des security groups regroupent les différentes ressources (tout n'est pas représenté sur le schéma) pour pouvoir ensuite autoriser par exemple les frontend à appeler le LB api.

Vous pouvez lancer la création de l'infrastructure sur votre compte AWS en allant dans le sous répertoire terraform du répertoire demoboard

```
terraform init
terraform apply
```

Durant l'exécution, vous pouvez regarder le code terraform pour voir la correspondance avec le schéma. Vous noterez le découpage dans différents fichiers :

- network : création du vpc, des subnets et des services internet et nat gateway + tables de routage
- loadbalancers : la création des lb avec target groups
- servers : les différents serveurs ainsi que la clé SSH qui sera ensuite déposée partout
- firewall : qui contient les security groups pour autoriser les communications (de façon un peu trop large mais nous en reparlerons)
- outputs : pour récupérer les adresses et record DNS des instances
- variables : pour la déclaration des variables et un auto pour définir des valeurs (il y a peu de variables définies mais nous en reparlerons)

Au bout de quelques minutes, félicitations, vous avez une infrastructure complexe et complète entièrement créée !

TP01 :
Prenez une capture d'écran de la console AWS ou l'output d'exécution (synthèse) de terraform pour montrer que vous avez réussi le provisioning de l'infrastructure

Configuration des serveurs avec Ansible

Nous allons maintenant déployer demoboard sur cette nouvelle infrastructure
Vous allez pour cela travailler dans le répertoire demoboard/ansible

Nous allons cette fois utiliser l'inventaire dynamique ainsi qu'une collection ansible-galaxy que nous allons devoir installer et configurer.
L'inventaire dynamique nécessite certaines librairies python. Nous allons utiliser un environnement virtuel python pour cela (**depuis le sous répertoire demoboard/ansible**)

```
python3 -m venv ansiblevenv
source ansiblevenv/bin/activate
pip install ansible boto3 botocore
ansible-galaxy collection install community.mysql
```

Pour information, si vous voulez sortir du venv python tapez simplement ***`deactivate`*** *et si vous souhaitez ensuite y revenir* ***`source ansiblevenv/bin/activate`***
Attention : pour toutes les commandes ansible qui suivent, vous avez besoin d'être dans le venv python pour bénéficier de l'inventaire dynamique

Vérifiez que l'inventaire dynamique fonctionne :

```
ansible-inventory -i aws_ec2.yml --graph
ansible-inventory -i aws_ec2.yml --list
ansible-inventory -i aws_ec2.yml --host api1
ansible-inventory -i aws_ec2.yml --host front1 | jq 'keys'
```

TP02 :
Pouvez-vous expliquer le fonctionnement des directives suivantes issues du fichier d'inventaire dynamique aws_ec2.yml ?

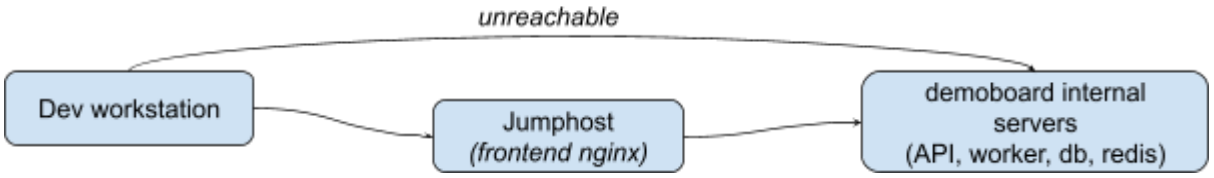
```
keyed_groups:
  - key: aws_tags
```

Pour information, nous avons ajouté ***`hostvars_prefix: aws_`*** *afin que tous les attributs renvoyés par l'inventaire dynamique soient préfixés par* ***`aws_`***
Ceci permet d'éviter des warnings ansible lorsque des noms d'attributs sont identiques à des mots clés réservés Ansible (ce n'est pas bloquant mais plus propre)
Exemple des WARNING wue l'on peut avoir avec ansible sans ce prefix : ***`[WARNING]: Found variable using reserved name: tags`***

Comme vous l'avez peut-être remarqué, une partie de nos hosts est située dans des subnets private. Cela signifie qu'ils ne sont pas accessibles depuis INTERNET directement. Quand votre controller ansible va vouloir s'y connecter en SSH, il y aura une erreur "unreachable".

Nous allons donc utiliser un concept de jumphost qui permet un rebond SSH. C'est directement intégré au client SSH et cela permet de se connecter à une machine d'un réseau privé en passant par un jumphost qui lui est accessible sur INTERNET et a accès au réseau privé également. (https://man.openbsd.org/ssh#J)

Afin de ne pas trop surcharger notre architecture terraform, nous utilisons les frontend (nginx) comme jumphost. Ce n'est pas idéal dans une véritable architecture de production car cela nécessite de laisser l'accès SSH sur les serveurs frontend.



Nous allons utiliser un fichier ssh-config spécifique qui est référencé dans le ansible.cfg, ainsi il sera pris en compte à chaque exécution de ansible.

Nous devons dans ce fichier ssh-config, indiquer à ansible qu'à chaque fois qu'il veut se connecter à un host, il doit utiliser un ProxyJump (un jumphost). Il faut toutefois indiquer que pour le host ou adress IP du ProxyJump lui-même, on ne doit pas passer par le jumphost, sinon cela va faire une boucle de connexion SSH infinie.

Vous allez donc devoir remplacer dans le fichier ssh-config, l'adresse IP par celle d'un des jumphosts que vous venez de provisionner via terraform (il s'agit des deux front qui font office de jumphost et de serveur web nginx).
Vous pouvez récupérer cette adresse directement depuis l'inventory (vous pouvez aussi utiliser le nom DNS public dynamique fourni par AWS si vous préférez)

```
ansible-inventory -i aws_ec2.yml --host front1 | jq -r
'.aws_public_ip_address'
```

```
ansible-inventory -i aws_ec2.yml --host front1 | jq -r
'.aws_public_dns_name'
```

Et faites le remplacement dans le fichier ssh-config à la racine du répertoire courant (attention à bien laisser le ! à la première ligne)

```
Host * !<ADRRES_IP>
      ProxyJump <ADRRES_IP>
```

Pour les amateurs de shell, vous pouvez utiliser sed (avec option -i pour changement in-place) afin de mettre à jour directement le fichier.

```
IP=$(ansible-inventory -i aws_ec2.yml --host front1 | jq -r
'.aws_public_ip_address')
sed -i -e
"s/[0-9]\{1,3\}\.[0-9]\{1,3\}\.[0-9]\{1,3\}\.[0-9]\{1,3\}/${IP}/g"
ssh-config
```

*Enfin, n'oubliez pas de **mettre à jour dans le fichier ssh-config le chemin vers la clé privée** que vous avez utilisée pour terraform et dont la clé publique est maintenant déployée sur tous vos host et va vous permettre de vous connecter. Il s'agit de la ligne `IdentityFile` (si vous avez utilisé le code terraform par défaut, une clé ssh avait normalement déjà été provisionnée et utilisée pour vous sur la partie demoboard et renseignée dans le ssh-config, donc si vous n'avez rien modifié, vous n'avez rien à faire)*

Vous pouvez maintenant tester les commandes ansible :

```
ansible -i aws_ec2.yml all -m shell -a hostname
```

Si tout fonctionne bien, il est temps de lancer l'installation et configuration de demoboard via le playbook setup.yml !

```
ansible-playbook -i aws_ec2.yml setup.yml -v
```

Si tout se déroule comme prévu (sinon vous devrez débugger), vous pouvez tester l'accès à l'application demoboard en utilisant l'adresse IP ou le record DNS fourni par AWS du LoadBalancer frontend. Pour faciliter sa récupération, nous l'avons intégré dans des tags sur les VMs via terraform, vous pouvez donc utiliser l'inventory
`ansible-inventory -i aws_ec2.yml --host front0 | jq -r '.aws_tags.front_lb_dns'`

Ouvrez ensuite l'URL du LB du frontend que vous venez de récupérer dans un navigateur. Bravo !
Vous pouvez aussi utiliser curl, votre URL LB devrait être plus ou moins similaire à l'exemple ci-dessous
`curl lb-front-vm00-220743085.eu-central-1.elb.amazonaws.com`

Vous devriez voir l'IHM de demoboard, vous pouvez ajouter des tâches, lancer des traitements et supprimer des tâches, tout doit fonctionner.

Si ça ne fonctionne pas, vous pouvez taper F12 pour afficher le mode développeur du navigateur et essayer de comprendre d'où vient le problème (appelez le formateur).

Petit test sympa au passage, vous pouvez appeler des URLs de monitor qui ont été déployées (cf. code ansible du role nginx) pour montrer que le loadBalancing fonctionne bien : `<DNS du LB front>/monitor.html` et rafraîchissez la page plusieurs fois, vous devriez voir l'autre front répondre. Vous devriez aussi pouvoir tester la page de santé de l'API : `<DNS du LB front>/api/healthz`
Si vous ne voyez pas de changement, testez aussi depuis une fenêtre de navigation privée avec un refresh complet (pour éviter le cache ou les cookies)

Quelques tips de DEBUG ou pistes en cas de problème :

- Se connecter sur un host (nécessite de passer par le jumphost et donc le ssh-config) - adaptez suivant le host souhaité (ici front0)
  - `ansible-inventory -i aws_ec2.yml --host front0 | jq -r '.aws_private_ip_address'`
  - Puis : `ssh -F ./ssh-config <private_ip_address_recuperee>`
- Vérifiez depuis une VM front la config du lb
  - `aws elbv2 describe-target-health --target-group-arn "$(aws elbv2 describe-target-groups --names lb-tg-api-$(hostname) --query 'TargetGroups[0].TargetGroupArn' --output text)"`
- Depuis un front (nginx) regardez les logs
  - `sudo tail -f /var/log/nginx/access.log /var/log/nginx/error.log`
- Se connecter sur un host API et vérifier l'état du service :
  - `sudo systemctl status demoboard-api --no-pager`
  - `sudo tail -n 100 /var/log/demoboard/api-service.log`
  - `curl -sv http://127.0.0.1:8000/`
  - `journalctl -u demoboard-api`
- On peut aussi se connecter à la base depuis un host API pour vérifier les flux :
  - `sudo systemctl status postgresql --no-pager`
  - `sudo ss -lntp | grep 5432`
  - `sudo -u postgres psql -l`
  - `sudo -u postgres psql -d tasks -c '\dt'`
- On peut aussi sur hosts redis
  - `sudo systemctl status redis-server --no-pager`
  - `sudo ss -lntp | grep 6379`
  - `redis-cli ping`

Travaux d'optimisation - exercices

Vous allez maintenant disposer d'une liste de différents travaux possibles autour de cette architecture applicative et des outils terraform et ansible.
L'objectif est de pouvoir mettre en pratique les premières connaissances que vous avez apprises sur un cas concret avec la base de code existante du projet demoboard.
Il y a volontairement beaucoup de sujets pour vous permettre de faire un choix (de 2 à 3 sujets maximum) suivant le niveau de difficulté, le composant (terraform, ansible ou parfois les deux combinés) et le type de travaux (ajout de fonctions, optimisation, …)

Prenez quelques minutes pour lire les énoncés et choisissez 2 sujets (si vous avez du temps, vous pourrez en faire d'autres) et lancez-vous !

En termes de rendu, reprenez bien l'intitulé complet de chaque sujet que vous avez choisi avec la description. Ensuite, libre à vous d'expliquer ce que vous avez réalisé en l'illustrant avec des extraits de code et de captures d'écrans ou de sortie de logs (voire de schémas). N'hésitez pas également à expliquer votre cheminement, les problématiques rencontrées, les succès et les échecs. L'important, c'est la démarche, pas tellement le résultat (peu importe si à la fin tout ne fonctionne pas)

FACILE

Terraform

TP03 (au choix) :
Trouvez au moins 3 exemples de paramètres ou éléments du code terraform qui pourraient être variabilisés et expliquer le gain.
Montrez du code modifié pour au moins un exemple.

TP04 (au choix) :
En observant le contenu du fichier variables.tf, vous pouvez voir
aws_private_subnet_cidr_prefix = "10.1.2"
et dans network.tf cidr_block =
format("%s%s.0/24",var.aws_private_subnet_cidr_prefix, count.index)

Quelle va être la limitation de cette astuce qui permet ainsi d'avoir un CIDR différent (obligatoire) pour chaque subnet ?

TP05 (au choix) :
Dans servers.tf, que permettrait aws_subnet.private_subnet[count.index % length(var.demoboard_aws_zones)] au lieu de aws_subnet.private_subnet[count.index] ?

Ansible

TP06 (au choix) :
Dans setup.yml, pourquoi les rôles postgres et redis sont-ils exécutés avant api, puis nginx après api ?

TP07 (au choix) :
Dans le role api, on réalise l'installation, on crée un service systemd pour lancer le binaire, on s'assure que c'est le cas avec un restart mais on ne vérifie pas vraiment si l'API fonctionne.
Pouvez-vous ajouter des tâches en fin de rôle pour tester si l'API fonctionne bien localement sur chaque host et également quand on l'appelle en passant par le loadbalancer ? (Pensez à utiliser aussi delegate_to pour pouvoir réaliser des tests depuis différentes machines afin d'avoir quelque chose de représentatif)

TP08 (au choix) :
Le rôle API utilise déjà un handler de restart. Améliorez le rôle pour garantir qu'un changement de code, de dépendances, d'environnement ou d'unité systemd redémarre l'API avant les tests.

Terraform + Ansible

TP09 (au choix) :
Simulez une perte matérielle : créez quelques tâches dans Demoboard, supprimez une instance front et une instance api depuis AWS. Pendant que cela supprime, continuez à faire des requêtes pour voir si le loadBalancer fonctionne et prend en compte l'incident (vous pourriez avoir quelques requêtes ratées le temps de la bascule, prenez des captures d'écrans).

Ensuite, gérez la crise et reconstruisez les VMs et le déploiement des serveurs en utilisant terraform et ansible, n'est-ce pas un peu plus rapide et serein avec ces outils ?

Un peu plus long, mais si vous le souhaitez, vous pouvez aussi utiliser Jmeter (présent sur la VM de travail) pour écrire un petit scénario de test qui par exemple appelle uniquement l'URL de monitor du LB webserver. Vous pouvez laisser tourner votre scénario durant toute l'opération pour montrer qu'il n'y a pas ou peu d'interruption de service.

Terraform

TP10 (au choix) :
Vous allez devoir améliorer la sécurité réseau du déploiement. En effet, si vous avez regardé le fichier firewall.tf, vous avez dû voir que l'on autorise souvent l'ensemble des hosts 0.0.0.0/0 à se connecter à beaucoup de choses.
Commencez par définir des règles théoriques de ce type :
- Les jumphosts/RP doivent être joignables en SSH depuis l'extérieur (pour ansible)
- Les API, front et DB ne doivent être joignables en SSH que depuis les jumphosts
- La DB ne doit être joignable que depuis les hosts API sur le port mysql
- Sur les ports HTTP/HTTPS seuls les LB doivent pouvoir joindre les hosts front ou api

Vous pouvez ajouter des règles et ensuite les implémenter via les security groups. Vous pouvez aussi ajouter des network ACL (regardez aws_network_acl et aws_network_acl_rule dans la doc du provider AWS) qui permettent de définir des règles valables pour tout un subnet (les security group concernent les instances qui appartiennent à ce security group uniquement)

Une partie difficile concerne la déclaration dynamique des subnets que l'on a définie avec des préfixes. Dans un premier temps, vous pouvez simplement ajouter manuellement dans chaque security groups les subnets comme c'est fait actuellement (ce n'est pas dynamique mais plus simple).
Si vous avez le temps (et attention, c'est plus difficile), vous pouvez en plus tenter de rendre cela dynamique en ajoutant une variable subnets et en utilisant la fonction cidr de terraform
Regardez la documentation et surtout les exemples :
https://registry.terraform.io/modules/hashicorp/subnets/cidr/1.0.0

Ansible

TP11 (au choix) :
Utilisez les collections/modules ansible community.crypto.x509_certificate et community.crypto.openssl_csr pour générer une autorité de certification locale (auto-signée) et ensuite délivrer à partir de celle-ci des certificats pour les serveurs frontend et API contenant le record des LoadBalancer respectifs dans les SAN (Subject Alternate Name) des certificats.

Vous devrez pouvoir montrer que votre configuration fonctionne en HTTPS et surtout que le certificat renvoyé est celui que vous avez généré. (Vous devriez pouvoir forcer vos serveurs à faire du HTTPS sur les ports existants, si nécessaire ou plus simple pour vous, vous pouvez aussi intervenir sur le code terraform pour autoriser d'autres ports dans les security groups et le LoadBalancer)

~~**TP12 (au choix) :**~~ **CANCELED (OLD VIKUNJA)**

~~**TP13 (au choix) :**~~ **CANCELED (OLD VIKUNJA)**

Terraform + Ansible

TP14 (au choix) :
Ajoutez ou améliorez une page monitor.html servie par nginx pour tester explicitement le load balancer frontend. Ajoutez ensuite un équivalent côté API si vous voulez tester le LB API sans dépendre de l'application complète.
Vous aurez à adapter le code terraform pour les security groups et le loadBalancer API.

TP15 (au choix) :
Modifiez le déploiement pour permettre de choisir le backend de données Demoboard entre PostgreSQL et SQLite. Prenez en compte l'option côté API et côté Ansible.

DIFFICILE

Ansible

TP17 (au choix) :
Automatisez une mise à jour rolling update de l'API Demoboard.
Le code source de l'API contient un fichier api-service/VERSION, initialement à 1.0.0. Ce numéro est exposé par l'endpoint /healthz.

Modifiez api-service/VERSION pour simuler une nouvelle version, par exemple 1.1.0.
Adaptez le playbook ou le rôle Ansible API pour déployer cette nouvelle version une instance API à la fois.
Après chaque déploiement, redémarrez (dans le rôle ansible) uniquement l'API concernée si le code a changé.

Vérifiez localement sur l'instance que /healthz retourne la version attendue.
Vérifiez aussi via le load balancer API que la nouvelle version apparaît progressivement.
Utilisez une stratégie de type rolling update avec serial: 1.
Vous pouvez ajouter des variables comme :
demoboard_api_target_version: "1.1.0"
demoboard_api_health_url: "http://127.0.0.1:8000/healthz"

Terraform + Ansible

TP17 (au choix) :
Ajoutez un 3ème subnet private ainsi que 3 serveurs API supplémentaires. Vous devrez répartir la charge sur ces 5 serveurs API qui sont eux mêmes déployés dans les 3 subnets private.
Utilisez bien sûr ansible pour déployer demoboard API sur les nouveaux serveurs et finaliser la configuration.

TP18 (au choix) :
Ajoutez une variable de type environnement afin de déclarer un second environnement (le premier serait dev et le second prod).
Créer ce second environnement en parallèle du premier avec une topologie un peu différente (par exemple plus de serveurs API sur la PROD)
Installez également une instance de demoboard sur cet environnement de production avec ansible.

TP19 (au choix) :
Ajoutez un volume externe sur la VM database avec terraform (regarder les ressources ebs_volume et aws_volume_attachment dansl a documentation du provider AWS)
Vous devrez ensuite utiliser du code ansible pour prendre en compte ce nouveau disque, le formater et le monter là où postgreSQL stocke ses fichiers de data (ou modifier la conf du postgreSQL pour lui indiquer d'utiliser votre nouveau point de montage)
Vous pouvez regarder des modules comme parted ou filesystem

!! Suppression de l'ensemble des ressources CLOUD

Avant de partir, il est très important que vous réalisiez les `terraform destroy` partout !
En effet, il est très fastidieux et risqué (oubli) de devoir le faire manuellement via la console.