UTOPIOS
TRANSFORMATION DIGITALE INTELLIGENTE

Terraform avec Azure

Programme

1. Introduction
   1.1 Objectifs de la formation
   1.2 Présentation de Terraform
   1.3 Pourquoi utiliser Terraform avec Azure
2. Mise en place de l'environnement
   2.1 Installation de Terraform
   2.2 Configuration de Terraform dans l'écosystème Azure
3. HCL
   3.1 Présentation de Terraform HCL
   3.2 Les différentes variables et commandes
   3.3 Le flux de travail de Terraform
   3.4 Les dépendances explicites et implicites entre les ressources
   3.5 Les cycles de vie des ressources
   3.6 Les expressions conditionnelles et itératives
   3.7 Les templates et fonctions intégrées
   3.8 Comprendre et gérer l'état de Terraform

Programme

4. Azure et terraform
   - 4.1 Provider
   - 4.2 Gestion du réseau Azure
   - 4.3 Gestion de l'Infrastructure
5. Azure et terraform avancée
   - 5.1 Terraform Provisioner
   - 5.2 Automatiser son workflow avec Terraform
   - 5.3 Les modules Terraform
   - 5.4 Gestion des outputs Terraform
   - 5.5 Terraform et les environnements
6. Troubleshooting

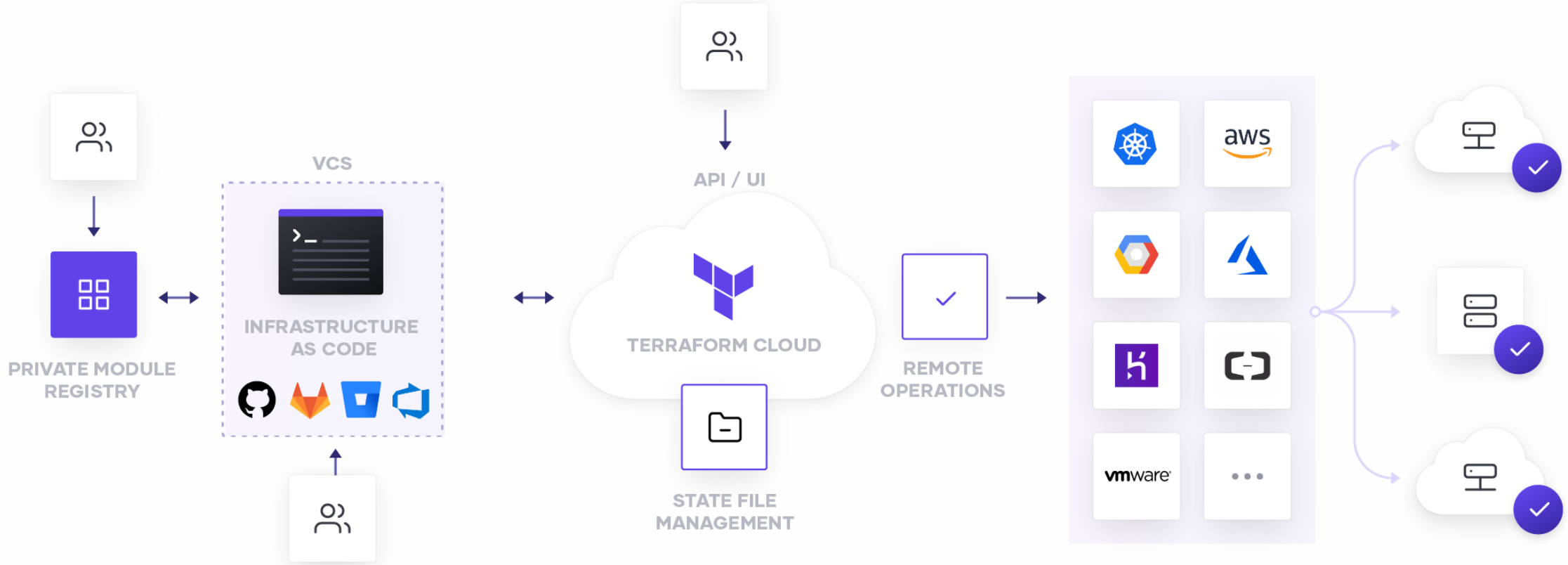
Objectifs de la formation

- Comprendre les concepts clés de Terraform
- Apprendre à créer et gérer une infrastructure cloud avec Terraform et Azure
- Maîtriser les meilleures pratiques pour utiliser Terraform avec Azure

Présentation de Terraform

- **Définition de Terraform** :
  - Terraform est un outil open-source développé par HashiCorp qui permet de définir, provisionner et gérer des ressources d'infrastructure en tant que code (IaC) dans divers fournisseurs de cloud, tels qu'OpenStack.
  - Terraform utilise son propre langage de configuration déclaratif appelé HCL (HashiCorp Configuration Language) pour décrire les ressources souhaitées, puis génère un plan d'exécution pour atteindre l'état souhaité.
- **Avantages de l'Infrastructure as Code (IaC)** :
  - Versionnage : Le code de l'infrastructure peut être versionné et contrôlé dans des systèmes de gestion de version (comme Git), permettant un suivi clair des modifications.
  - Reproductibilité : Les infrastructures peuvent être déployées de manière cohérente et reproductible, réduisant les erreurs humaines et les différences entre les environnements.
  - Automatisation : Le processus de déploiement de l'infrastructure peut être automatisé, réduisant les coûts et les efforts manuels.

Présentation de Terraform



Pourquoi utiliser Terraform avec Azure ?

Gestion simplifiée de l'infrastructure: Avec Terraform, vous pouvez définir toute votre infrastructure Azure dans un fichier de configuration. Cela rend la gestion de l'infrastructure plus facile et plus transparente car tout est codé dans un fichier ou un ensemble de fichiers.
Modularité et réutilisabilité: Terraform vous permet de créer des modules pour les différents composants de votre infrastructure. Cela permet de réutiliser les configurations pour différents environnements ou projets.
Gestion des dépendances: Terraform gère les dépendances entre les différents composants de l'infrastructure. Par exemple, si une machine virtuelle dépend d'un réseau virtuel, Terraform s'assurera que le réseau virtuel est créé avant la machine virtuelle.
Gestion des changements: Terraform compare l'état actuel de l'infrastructure avec la configuration définie et ne fait que les modifications nécessaires pour aligner les deux. Cela permet une gestion des changements plus efficace et réduit le risque d'erreurs.
Interopérabilité: Terraform peut être utilisé avec de nombreux autres fournisseurs de services cloud et d'outils de gestion de l'infrastructure. Cela le rend idéal pour les entreprises qui ont une infrastructure hybride ou multi-cloud.
Outil open-source: Terraform est un outil open-source, ce qui signifie qu'il est gratuit à utiliser et a une grande communauté de développeurs qui contribuent à son développement et à son support.

Installation de Terraform

1. **Téléchargez Terraform**: Allez sur le site officiel de Terraform et téléchargez la dernière version de Terraform pour votre système d'exploitation. Voici le lien pour télécharger Terraform : [Terraform Downloads].
2. **Décompressez le fichier**: Une fois le fichier téléchargé, décompressez le fichier dans un répertoire de votre choix.
3. **Ajoutez Terraform à votre PATH**:

- Sur Windows:
  - Copiez le fichier terraform.exe dans un répertoire qui est dans votre PATH, ou ajoutez le répertoire où se trouve terraform.exe à votre PATH.
  - Pour ajouter un répertoire à votre PATH, vous pouvez le faire via les paramètres système ou en utilisant la ligne de commande. Par exemple, si vous avez décompressé terraform.exe dans le répertoire C:\terraform, vous pouvez ajouter ce répertoire à votre PATH en exécutant la commande suivante dans l'invite de commande : setx PATH "%PATH%;C:\terraform".

Installation de Terraform

- Sur macOS et Linux:
  - Déplacez le fichier terraform dans un répertoire qui est dans votre PATH. Par exemple, vous pouvez déplacer terraform dans le répertoire /usr/local/bin en exécutant la commande suivante : mv terraform /usr/local/bin/.

4. **Vérifiez l'installation**: Ouvrez une nouvelle invite de commande ou un terminal et exécutez la commande terraform -v. Si Terraform est correctement installé, cette commande affichera la version de Terraform que vous avez installée.

Configuration de Terraform dans l'écosystème Azure

1. **Installer Azure CLI**: Azure CLI est un outil en ligne de commande pour gérer les ressources Azure. Vous pouvez installer Azure CLI en suivant les instructions disponibles ici.
2. **Se connecter à Azure**: Ouvrez un terminal et exécutez la commande az login. Cette commande ouvrira une nouvelle fenêtre de navigateur où vous pourrez vous connecter à votre compte Azure.
3. **Créer un Service Principal**: Un Service Principal est une identité d'application dans Azure Active Directory (AAD) qui est utilisée par les applications, les services et les outils d'automatisation pour accéder à des ressources spécifiques dans Azure.

```
az ad sp create-for-rbac --role="Contributor" --scopes="/subscriptions/YOUR_SUBSCRIPTION_ID"
```

Remplacez YOUR_SUBSCRIPTION_ID par l'ID de votre abonnement Azure. Cette commande vous retournera un appId, displayName, name, password, et tenant. Conservez ces valeurs car vous en aurez besoin pour configurer le fournisseur Azure dans Terraform

- Vous devrez également configurer les variables d'environnement ARM_SUBSCRIPTION_ID, ARM_CLIENT_ID, ARM_CLIENT_SECRET, et ARM_TENANT_ID avec les valeurs que vous avez obtenues lors de la création du Service Principal.
- Vous pouvez configurer ces variables d'environnement dans votre terminal ou dans un fichier .env

Présentation de Terraform HCL

- **HCL (HashiCorp Configuration Language)** est un langage de configuration développé par HashiCorp, la société derrière Terraform. HCL est conçu pour être à la fois lisible par l'homme et lisible par la machine, ce qui le rend idéal pour la configuration de l'infrastructure.
- HCL est utilisé pour écrire les fichiers de configuration Terraform. Les fichiers de configuration Terraform décrivent les ressources et les infrastructures à provisionner.
- **Structure de base**: Les fichiers de configuration Terraform, écrits en HCL, sont structurés en trois blocs principaux: provider, resource et output.
    1. Le bloc provider configure le fournisseur de cloud à utiliser (par exemple, AWS, Azure, GCP, etc.).
    2. Le bloc resource définie une ressource à provisionner (par exemple, une machine virtuelle, un réseau, une base de données, etc.).
    3. Le bloc output définie les valeurs à afficher à l'utilisateur après le provisionnement de l'infrastructure.

Les différentes variables et commandes

Commandes terraform

- **terraform init**: Initialise le répertoire de travail de Terraform. Cette commande doit être exécutée avant d'autres commandes Terraform.
- **terraform plan**: Crée un plan d'exécution. Cette commande montre quelles actions Terraform effectuera pour aligner l'infrastructure sur la configuration.
- **terraform apply**: Applique les changements nécessaires pour aligner l'infrastructure sur la configuration.
- **terraform destroy**: Détruit les ressources créées par Terraform.

Le flux de travail de Terraform

1. **Write** : La première étape du flux de travail de Terraform consiste à écrire un fichier de configuration. Ce fichier, souvent appelé main.tf, définira toutes les ressources que vous souhaitez créer ou gérer. Dans le contexte d'Azure, cela signifierait définir des ressources comme des machines virtuelles, des groupes de ressources, des réseaux virtuels, etc., en utilisant le fournisseur Azure et le langage de configuration de Terraform (HCL).
2. **Plan** : Après avoir écrit votre configuration, l'étape suivante est de créer un plan d'exécution. Vous pouvez faire cela en exécutant la commande terraform plan. Cette commande permet à Terraform de simuler les actions qu'il effectuera en fonction de la configuration actuelle par rapport à l'état actuel de votre infrastructure. Cela permet de voir quelles actions Terraform effectuera sans effectuer réellement de modifications.
3. **Apply** : Une fois que vous avez vérifié que le plan d'exécution est correct, la dernière étape est d'appliquer les modifications en exécutant la commande terraform apply. Cela demandera à Terraform de créer, mettre à jour ou détruire les ressources nécessaires pour correspondre à la configuration spécifiée dans votre fichier main.tf.

Les différentes variables et commandes

variables terraform

- Les variables dans Terraform sont un moyen de définir des valeurs qui peuvent être réutilisées dans toute votre configuration. Vous pouvez définir des valeurs par défaut pour vos variables et les substituer au moment de l'exécution si nécessaire
- *__Déclaration__ : Vous devez d'abord déclarer vos variables avant de les utiliser. Habituellement, les variables sont déclarées dans un fichier séparé appelé variables.tf, mais elles peuvent être déclarées dans n'importe quel fichier .tf dans votre configuration.

Les différentes variables et commandes

variables terraform

- variables.tf

```
variable "region" {
  description = "The region to deploy to"
  type        = string
  default     = "us-west-1"
}

variable "instance_types" {
  description = "The types of instances to deploy"
  type        = list(string)
  default     = ["t2.micro", "t2.small"]
}

variable "tags" {
  description = "The tags to apply to resources"
  type        = map(string)
  default     = { Owner = "DevOps", Environment = "Production" }
}
```

Les différentes variables et commandes

variables terraform

- Utilisation dans main.tf

```
output "selected_region" {
  value = var.region
}

output "selected_instance_types" {
  value = var.instance_types
}

output "common_tags" {
  value = var.tags
}
```

Les différentes variables et commandes

variables terraform

- Dans Terraform, les variables peuvent être de plusieurs types, dont voici quelques-uns des plus couramment utilisés

1. **String**: Il s'agit du type le plus basique. Une variable de type string est simplement une chaîne de caractères.

```
variable "example_string" {
  type    = string
  default = "Hello, World"
}
```

2. **Number**: Ce type est utilisé pour les variables qui sont des nombres. Ils peuvent être des entiers ou des nombres à virgule flottante.

```
variable "example_number" {
  type    = number
  default = 42
}
```

Les différentes variables et commandes

variables terraform

3. **Bool**: Ce type est utilisé pour les variables qui sont soit true, soit false.

```
variable "example_bool" {
  type    = bool
  default = true
}
```

4. **List (ou tuple)**: Une liste est une collection ordonnée d'éléments du même type.

```
variable "example_list" {
  type    = list(string)
  default = ["one", "two", "three"]
}
```

Les différentes variables et commandes

variables terraform

5. **Map**: Une carte est une collection non ordonnée d'éléments de type clé-valeur

```
variable "example_map" {
  type = map(string)
  default = {
    key1 = "value1"
    key2 = "value2"
  }
}
```

6. **Set**: Un ensemble est une collection non ordonnée d'éléments uniques du même type.

```
variable "example_set" {
  type    = set(string)
  default = ["one", "two", "three"]
}
```

Les différentes variables et commandes

variables terraform

7. **Object**: Un objet est une collection d'attributs avec des types spécifiés

```
variable "example_object" {
  type = object({
    attribute1 = string
    attribute2 = list(number)
  })
  default = {
    attribute1 = "value"
    attribute2 = [1, 2, 3]
  }
}
```

Les différentes variables et commandes

Exercice

- Configurer une configuration de base Terraform qui utilise différentes variables pour personnaliser les valeurs des sorties, y compris des variables qui dépendent d'autres variables.
- Dans le fichier variables.tf, déclarez les variables suivantes :
  - Une variable first_name de type string sans valeur par défaut.
  - Une variable last_name de type string sans valeur par défaut.
  - Une variable age de type number sans valeur par défaut.
  - Une variable is_student de type bool avec une valeur par défaut de false.
  - Une variable courses de type list(string) avec une valeur par défaut de ["Math", "Science"].
  - Une variable grades de type map(number) avec une valeur par défaut de { Math = 90, Science = 85 }.
  - Une variable student de type object qui a les attributs suivants : first_name de type string. last_name de type string. age de type number. is_student de type bool. courses de type list(string). grades de type map(number).

Les dépendances explicites et implicites entre les ressources

- Les dépendances entre les ressources sont essentielles pour déterminer l'ordre dans lequel les ressources doivent être créées, mises à jour ou détruites

1. **Dépendances implicites** :

- Les dépendances implicites sont créées automatiquement par Terraform lorsque vous utilisez une référence à une autre ressource dans la configuration d'une ressource.
- Par exemple, dans le code ci-dessous, une machine virtuelle Azure dépend implicitement d'un réseau virtuel et d'une interface réseau car leurs identifiants sont utilisés pour configurer la'instance

```
resource "azurerm_virtual_network" "example" {
  # ...
}
resource "azurerm_network_interface" "example" {
  # ...
  virtual_network_name = azurerm_virtual_network.example.name
}
resource "azurerm_virtual_machine" "example" {
  # ...
  network_interface_ids = [azurerm_network_interface.example.id]
}
```

Les dépendances explicites et implicites entre les ressources

2. **Dépendances explicites** :

- Vous pouvez également spécifier des dépendances explicites à l'aide de l'attribut depends_on. Cela peut être nécessaire dans certaines situations où Terraform ne peut pas déterminer les dépendances automatiquement.
- Par exemple, si une ressource doit attendre qu'une configuration externe soit complétée avant d'être créée, vous pouvez utiliser depends_on pour spécifier cette dépendance

```
resource "azurerm_virtual_machine" "example" {
  # ...
  depends_on = [azurerm_virtual_network.example]
}
```

Les cycles de vie des ressources

- Les cycles de vie des ressources dans Terraform définissent la'ordre et le comportement des actions qui seront effectuées sur une ressource. Par exemple, lorsqu'une ressource doit être créée, mise à jour ou détruite. Le bloc lifecycle dans la configuration d'une ressource permet de contrôler ce comportement.
- **create_before_destroy**: Un booléen qui, lorsqu'il est défini sur true, indique à Terraform de créer la nouvelle ressource avant de détruire l'ancienne lors d'une mise à jour.
- **prevent_destroy**: Un booléen qui, lorsqu'il est défini sur true, empêchera la ressource d'être détruite. Cela peut être utile pour éviter la suppression accidentelle de ressources critiques.
- **ignore_changes**: Une liste d'attributs qui seront ignorés lors de la mise à jour d'une ressource. Cela peut être utile pour ignorer les modifications apportées par des sources externes à Terrafor
- **replace_triggered_by**: Il permet de spécifier une liste d'attributs de la ressource qui, lorsqu'ils sont modifiés, déclenchent la destruction et la recréation de la ressource au lieu de simplement la mettre à jour

Les cycles de vie des ressources

- **Exemple**

```
resource "example_resource" "example" {
  # ...

  lifecycle {
    create_before_destroy = true
    prevent_destroy       = true
    ignore_changes        = [example_attribute]
    replace_triggered_by = [example_attribute]
  }
}
```

Les cycles de vie des ressources Exercice

1. Déclarez deux variables, example_attribute1 et example_attribute2, toutes deux de type string avec des valeurs par défaut.
2. Configurez quatre ressources fictives : example_resource_1, example_resource_2, example_resource_3, et example_resource_4.
    - Pour example_resource_1, utilisez la variable example_attribute1 et configurez son bloc lifecycle pour que la nouvelle ressource soit créée avant la destruction de l'ancienne, et pour ignorer les modifications de example_attribute1.
    - Pour example_resource_2, utilisez la variable example_attribute2, configurez une dépendance explicite sur example_resource_1 en utilisant depends_on, et configurez son bloc lifecycle pour empêcher la destruction de la ressource.
    - Pour example_resource_3, configurez une dépendance explicite sur example_resource_2 et example_resource_1 en utilisant depends_on.
    - Pour example_resource_4, configurez une dépendance explicite sur example_resource_3 en utilisant depends_on, et configurez son bloc lifecycle pour ignorer les modifications de toutes les propriétés, à l'exception de example_attribute2.

Les expressions conditionnelles et itératives

- Les expressions conditionnelles dans Terraform permettent d'évaluer une condition et de retourner une valeur en fonction du résultat de cette condition. La syntaxe de base pour une expression conditionnelle est la suivante

```
condition ? true_value : false_value
```

- Exemple :
  - *variables.tf*

```
variable "environment" {
  description = "The environment"
  type        = string
}
```

- *main.tf*

```
resource "example_resource" "example" {
example_attribute = var.environment == "production" ? "production_value" : "non_production_value"
}
```

Les expressions conditionnelles et itératives

- les expressions itératives, comme for et for_each, permettent de créer plusieurs ressources ou d'effectuer des opérations sur chaque élément d'une collection

1. **Expression for** :

- L'expression for peut être utilisée pour transformer une liste ou un ensemble de valeurs. Elle peut également être utilisée pour filtrer ou transformer les éléments d'une collection.

```
variable "list_of_strings" {
  description = "A list of strings"
  type        = list(string)
  default     = ["apple", "banana", "cherry"]
}

output "lengths_of_strings" {
  value = [for s in var.list_of_strings : length(s)]
}
```

Les expressions conditionnelles et itératives

2. **L'attribut for_each** :

- L'attribut for_each peut être utilisé pour créer une instance de ressource pour chaque élément dans une collection.

```
variable "set_of_names" {
  description = "A set of names"
  type        = set(string)
  default     = ["Alice", "Bob", "Charlie"]
}

resource "example_resource" "example" {
  for_each = var.set_of_names

  name = each.value
}
```

Les expressions conditionnelles et itératives

Exercice

Structures conditionnelles :

- Créez un fichier main.tf et déclarez une variable environment qui prendra deux valeurs possibles : "production" ou "development".
- Déclarez une autre variable instance_type qui prendra deux valeurs possibles : "small" ou "large".
- Utilisez une expression conditionnelle pour définir une troisième variable instance_size basée sur la valeur de instance_type:
  - Si instance_type est "small", instance_size devrait être "t2.micro".
  - Si instance_type est "large", instance_size devrait être "t2.large".

Structures itératives :

- Déclarez une variable names qui est une liste de noms.
- Utilisez une expression for pour créer une liste de messages de bienvenue pour chaque nom dans la liste names. Chaque message de bienvenue doit être formaté comme suit : "Welcome, [name]!"
- Affichez la valeur de instance_size et la liste des messages de bienvenue.

Les templates et fonctions intégrées

1. **Templates** :

Les templates dans Terraform sont utilisés pour insérer des valeurs dans des chaînes de caractères. Vous pouvez utiliser les expressions et les fonctions de Terraform dans un template. La syntaxe des templates est ${...}.

```
variable "name" {
  default = "world"
}

output "greeting" {
  value = "Hello, ${var.name}!"
}
```

Les templates et fonctions intégrées

2. **Fonctions intégrées** :

Terraform a un grand nombre de fonctions intégrées qui peuvent être utilisées pour transformer et combiner des valeurs. Ces fonctions peuvent être utilisées dans n'importe quelle expression dans une configuration Terraform

```
variable "list" {
  default = [1, 2, 3]
}
output "sum" {
  value = sum(var.list)
}
```

- Quelques autres fonctions intégrées communes sont :
  - length(list) : Retourne le nombre d'éléments dans une liste.
  - join(separator, list) : Combine les éléments d'une liste en une seule chaîne, en insérant un séparateur entre chaque élément.
  - split(separator, string) : Divise une chaîne en une liste d'éléments, en utilisant un séparateur pour déterminer les limites de chaque élément.

Les templates et fonctions intégrées

Exercice

- Créez un fichier main.tf et déclarez une variable names qui est une liste de noms.
- Combinez les noms de la liste names en une seule chaîne séparée par des virgules.
- Comptez le nombre de noms dans la liste names.
- Créez un fichier example.tpl avec le contenu suivant :

```
Hello, ${name}!
```

- Utilisez le fichier example.tpl et remplacer la variable ${name} par chaque nom de la liste names.

Comprendre et gérer l'état de Terraform

- L'état de Terraform est un aspect crucial de Terraform.
- Il garde une trace de l'infrastructure que Terraform a provisionnée et est utilisé pour planifier et appliquer des modifications à cette infrastructure
- L'état de Terraform est un fichier JSON qui contient les propriétés actuelles de l'infrastructure provisionnée. Il est utilisé par Terraform pour mapper les ressources réelles aux ressources dans votre configuration, pour stocker les attributs des ressources managées, et pour optimiser les performances pour les grandes infrastructures.
- Par défaut, Terraform stocke l'état localement dans un fichier appelé terraform.tfstate. Cependant, il est recommandé de stocker l'état dans un emplacement distant, tel qu'un bucket S3 ou Azure Blob Storage, pour des raisons de sécurité, de performance et de collaboration

Comprendre et gérer l'état de Terraform

Gestion de l'état de Terraform

- **Stockage distant** : Vous pouvez configurer Terraform pour stocker l'état dans un emplacement distant. Cela est particulièrement important lorsque vous travaillez en équipe, car il permet de partager l'état de Terraform entre les membres de l'équipe.
- **Verrouillage d'état** : Le verrouillage d'état empêche d'autres utilisateurs de modifier l'état de Terraform en même temps. Certains backends d'état, comme S3 avec DynamoDB, prennent en charge le verrouillage d'état.
- **Séparation de l'état** : Pour les grandes infrastructures, il peut être utile de séparer l'état de Terraform en plusieurs fichiers d'état. Cela peut être accompli en utilisant les workspaces de Terraform ou en définissant des configurations de Terraform distinctes pour différentes parties de votre infrastructure.
- Vous pouvez inspecter l'état actuel de Terraform en utilisant la commande ```terraform show``````.
- `terraform state list` : Liste toutes les ressources dans l'état de Terraform.
- `terraform state show` : Affiche les détails d'une ressource spécifique dans l'état de Terraform.
- `terraform state mv` : Déplace une ressource d'un état à un autre ou à un autre emplacement dans le même état.
- `terraform state rm` : Supprime une ressource de l'état de Terraform

Comprendre et gérer l'état de Terraform

Sécurisation l'état de Terraform

- L'état de Terraform peut contenir des informations sensibles, comme des mots de passe ou des clés d'accès. Il est donc important de sécuriser l'état de Terraform.
- Stockez l'état dans un emplacement sécurisé et contrôlez l'accès à cet emplacement.
- Utilisez le chiffrement pour stocker l'état.
- Ne stockez jamais l'état de Terraform dans un dépôt de code public ou non sécurisé

Comprendre et gérer l'état de Terraform

Exercice

- Créez un fichier main.tf avec une ressource fictive

```
resource "null_resource" "example" {
  triggers = {
    always_run = "${timestamp()}"
  }
}
```

- Listez toutes les ressources dans votre état de Terraform.
- Affichez les détails de la ressource null_resource.example.
- Supprimez la ressource null_resource.

Gestion des ressources Azure avec Terraform - Provider

- Un **provider** est un service qui interagit avec les ressources d'un cloud provider spécifique (comme AWS, Azure, GCP, etc.). Un provider définit les ressources (comme les instances de machines virtuelles, les groupes de sécurité, les réseaux, etc.) et les propriétés de ces ressources que Terraform peut gérer.
- Pour interagir avec les ressources d'Azure, vous utiliserez le provider **azurerm** de Terraform.

```
provider "azurerm" {
  features {}
}
```

- La section features est obligatoire pour le provider azurerm, mais vous n'avez pas besoin de spécifier des valeurs dedans

Gestion du réseau Azure

- Gérer le réseau Azure avec Terraform implique la création et la configuration de diverses ressources telles que les réseaux virtuels, les sous-réseaux, les interfaces réseau, les groupes de sécurité réseau, etc

Gestion du réseau Azure (Groupe de ressources)

- Les réseaux virtuels dans Azure doivent être créés dans un groupe de ressources
- Vous pouvez créer un groupe de ressources en ajoutant la ressource azurerm_resource_group à votre fichier de configuration

```
resource "azurerm_resource_group" "example" {
  name     = "example-resources"
  location = "East US"
}
```

Gestion du réseau Azure (Azure VNET)

- Avec Terraform, vous pouvez créer et gérer des réseaux virtuels Azure en utilisant la ressource azurerm_virtual_network

```
resource "azurerm_virtual_network" "example" {
  name                = "example-vnet"
  address_space       = ["10.0.0.0/16"]
  location            = azurerm_resource_group.example.location
  resource_group_name = azurerm_resource_group.example.name
}
```

- **Options possible**:
  - name (Requis) : Le nom du réseau virtuel.
  - address_space (Requis) : La ou les plages d'adresses IP pour le réseau virtuel. C'est une liste de plages d'adresses.
  - location (Requis) : La région Azure où créer le réseau virtuel.
  - resource_group_name (Requis) : Le nom du groupe de ressources dans lequel créer le réseau virtuel.
  - dns_servers (Optionnel) : Liste des adresses IP des serveurs DNS.
  - subnet (Optionnel) : Configuration des sous-réseaux du réseau virtuel. C'est un bloc de configuration (voir exemple ci-dessous).

Gestion du réseau Azure (Subnet)

- Pour créer un subnet Azure avec Terraform, vous utilisez la ressource azurerm_subnet. Les sous-réseaux dans Azure doivent être créés dans un réseau virtuel

```
resource "azurerm_subnet" "example" {
  name                 = "example-subnet"
  resource_group_name  = azurerm_resource_group.example.name
  virtual_network_name = azurerm_virtual_network.example.name
  address_prefixes     = ["10.0.1.0/24"]
  service_endpoints    = ["Microsoft.Storage"]
}
```

- **Options possible**:
  - name (Requis) : Le nom du sous-réseau.
  - resource_group_name (Requis) : Le nom du groupe de ressources dans lequel créer le sous-réseau.
  - virtual_network_name (Requis) : Le nom du réseau virtuel dans lequel créer le sous-réseau.
  - address_prefixes (Requis) : Liste des plages d'adresses IP pour le sous-réseau.
  - network_security_group_id (Optionnel) : ID du groupe de sécurité réseau à associer au sous-réseau.
  - route_table_id (Optionnel) : ID de la table de routage à associer au sous-réseau.

Gestion du réseau Azure - Structures conditionnelles et itératives

1. **Structures itératives** :

- Avec les boucles for de Terraform, vous pouvez créer plusieurs ressources du même type sans avoir à définir chacune d'elles individuellement dans votre configuration.
- Par exemple, pour créer plusieurs sous-réseaux dans un réseau virtuel, vous pouvez utiliser la construction count de Terraform

```
resource "azurerm_subnet" "example" {
  count                = 3
  name                 = "subnet-${count.index}"
  resource_group_name  = azurerm_resource_group.example.name
  virtual_network_name = azurerm_virtual_network.example.name
  address_prefixes     = ["10.0.${count.index + 1}.0/24"]
}
```

Gestion du réseau Azure - Structures conditionnelles et itératives

2. **Structures conditionnelles** :

- Vous pouvez également utiliser des expressions conditionnelles pour contrôler la création de ressources.
- Par exemple, vous pouvez utiliser une expression conditionnelle pour créer un sous-réseau seulement si une certaine variable est définie

```
variable "create_subnet" {
  description = "Create subnet"
  type        = bool
  default     = false
}

resource "azurerm_subnet" "example" {
  count                = var.create_subnet ? 1 : 0
  name                 = "example-subnet"
  resource_group_name  = azurerm_resource_group.example.name
  virtual_network_name = azurerm_virtual_network.example.name
  address_prefixes     = ["10.0.1.0/24"]
}
```

Gestion du réseau Azure - NSG

- Un groupe de sécurité réseau (NSG) sur Azure est utilisé pour filtrer le trafic réseau vers et depuis les ressources Azure dans un réseau virtuel Azure. Un NSG peut être associé à des interfaces réseau (NIC) attachées à une machine virtuelle ou à un sous-réseau. Lorsqu'un NSG est associé à un sous-réseau, les règles s'appliquent à toutes les ressources connectées à ce sous-réseau.

```
resource "azurerm_network_security_group" "example" {
  name                = "example-nsg"
  location            = azurerm_resource_group.example.location
  resource_group_name = azurerm_resource_group.example.name

  security_rule {
    name                       = "SSH"
    priority                   = 1001
    direction                  = "Inbound"
    access                     = "Allow"
    protocol                   = "Tcp"
    source_port_range          = "*"
    destination_port_range     = "22"
    source_address_prefix      = "*"
    destination_address_prefix = "*"
  }
}
```

Gestion du réseau Azure - Lab

- **Objectif** : Le but de ce TP est de vous familiariser avec la'automatisation de la création des ressources Azure en utilisant Terraform. Vous apprendrez à créer des réseaux virtuels (VNET), des sous-réseaux, des groupes de ressources et à utiliser des expressions conditionnelles pour contrôler la création des ressources.
- **Contexte** : Vous êtes un ingénieur DevOps et on vous a demandé d'automatiser la création des infrastructures réseau dans Azure pour un nouveau projet. Vous avez besoin de créer un réseau virtuel, plusieurs sous-réseaux et des groupes de ressources. De plus, l'équipe de développement a demandé la possibilité de créer un sous-réseau supplémentaire pour des tests, mais seulement si une variable spécifique est définie.
- **Exigences** :

1. Créer un groupe de ressources Azure.
2. Créer un réseau virtuel (VNET) Azure.
3. Créer trois sous-réseaux dans le réseau virtuel.
4. Utiliser une expression conditionnelle pour créer un quatrième sous-réseau seulement si une variable spécifique est définie à true.
5. Créer un groupe de sécurité réseau (NSG) qui autorise uniquement le trafic HTTP (port 80) et HTTPS (port 443) entrant . Associer ce NSG à l'un des sous-réseaux créés.

Gestion de l'Infrastructure - machines virtuelles

- Pour créer une **machine virtuelle**, vous devez définir le groupe de ressources, le réseau virtuel, le sous-réseau, l'interface réseau, et la machine virtuelle elle-même
- Création d'une interface réseau :

```
resource "azurerm_network_interface" "example" {
  name                = "example-nic"
  location            = azurerm_resource_group.example.location
  resource_group_name = azurerm_resource_group.example.name

  ip_configuration {
    name                          = "internal"
    subnet_id                     = azurerm_subnet.example.id
    private_ip_address_allocation = "Dynamic"
  }
}
```

Gestion de l'Infrastructure - machines virtuelles

- Création d'une machine virtuelle:

```
resource "azurerm_virtual_machine" "example" {
  name                = "example-vm"
  location            = azurerm_resource_group.example.location
  resource_group_name = azurerm_resource_group.example.name
  network_interface_ids = [azurerm_network_interface.example.id]
  vm_size             = "Standard_DS1_v2"
  storage_image_reference {
    publisher = "Canonical"
    offer     = "UbuntuServer"
    sku       = "16.04-LTS"
    version   = "latest"
  }
  storage_os_disk {
    name              = "example-osdisk"
    caching           = "ReadWrite"
    create_option     = "FromImage"
    managed_disk_type = "Standard_LRS"
  }
  os_profile {
    computer_name  = "hostname"
    admin_username = "adminuser"
    admin_password = "P@ssword1234!"
  }
  os_profile_linux_config {
    disable_password_authentication = false
  }
}
```

Gestion de l’Infrastructure - machines virtuelles

- Options possible :
- Resource Group: La VM doit être créée dans un groupe de ressources existant.
- Location: Emplacement de la VM. Par exemple, "East US".
- Virtual Network: La VM doit être connectée à un réseau virtuel existant.
- Subnet: Le sous-réseau dans lequel la VM sera placée.
- Network Interface: L'interface réseau qui sera attachée à la VM.
- VM Size: La taille de la VM. Par exemple, "Standard_DS1_v2".
- Storage Image: L'image du système d'exploitation à utiliser pour la VM. Vous pouvez choisir parmi une variété d'images fournies par Azure ou utiliser votre propre image personnalisée.
- OS Disk: Les paramètres du disque du système d'exploitation, y compris le nom, le type de cache, l'option de création, et le type de disque managé.
- Data Disks: Les disques de données à attacher à la VM.
- Boot Diagnostics: Si vous souhaitez activer ou désactiver les diagnostics de démarrage

Gestion de l'Infrastructure - machines virtuelles

- Utilisation et création de clé ssh avec machine virtuelle.

```
provider "azurerm" {
  features {}
}
resource "random_pet" "ssh_key_name" {}
resource "azapi_resource" "ssh_public_key" {
  type      = "Microsoft.Compute/sshPublicKeys@2022-11-01"
  name      = random_pet.ssh_key_name.id
  location  = azurerm_resource_group.rg.location
  parent_id = azurerm_resource_group.rg.id
}

resource "azurerm_virtual_machine" "vm" {
  name                  = "example-vm"
  location              = azurerm_resource_group.rg.location
  resource_group_name   = azurerm_resource_group.rg.name
  network_interface_ids = [azurerm_network_interface.nic.id]
  vm_size               = "Standard_DS1_v2"

  ...

  os_profile {
    computer_name  = "hostname"
    admin_username = "adminuser"
  }

  os_profile_linux_config {
    disable_password_authentication = true
    ssh_keys {
      path     = "/home/adminuser/.ssh/authorized_keys"
      key_data = azapi_resource.ssh_public_key.properties.sshPublicKey
    }
  }
}
```

Gestion de l’Infrastructure - machines virtuelles - Exercice

- Votre mission est de créer une machine virtuelle (VM) sur Azure à l'aide de Terraform. La VM doit être accessible via SSH en utilisant une paire de clés SSH générée par Terraform.

Gestion de l'Infrastructure - compte de stockage

- Un compte de stockage Azure est un compte qui donne accès à tous les services de stockage Azure tels que les BLOBs, les Files, les Queues, et les Tables. Les données dans un compte de stockage Azure sont toujours répliquées pour garantir leur durabilité et leur haute disponibilité.

```
resource "azurerm_storage_account" "example" {
  name                     = "examplestorageaccount"
  resource_group_name      = azurerm_resource_group.example.name
  location                 = azurerm_resource_group.example.location
  account_tier             = "Standard"
  account_replication_type = "GRS"

}
```

Options possibles:

- name : (Obligatoire) Le nom du compte de stockage. Il doit être unique dans l'ensemble d'Azure.
- resource_group_name : (Obligatoire) Le nom du groupe de ressources dans lequel créer le compte de stockage.
- location : (Obligatoire) La localisation géographique du groupe de ressources.
- account_kind : (Facultatif) Définit le type de compte de stockage à créer. Les valeurs possibles sont Storage, StorageV2, et BlobStorage. Par défaut, il est réglé sur StorageV2.

Gestion de l'Infrastructure - compte de stockage

- account_tier : (Facultatif) Définit le niveau de performance du compte de stockage. Les valeurs possibles sont Standard et Premium.
- account_replication_type : (Obligatoire) Le type de réplication à utiliser. Les valeurs possibles sont :
  - LRS : Locally-redundant storage.
  - GRS : Geo-redundant storage.
  - RAGRS : Read-access geo-redundant storage.
  - ZRS : Zone-redundant storage.
  - GZRS : Geo-zone-redundant storage.
  - RAGZRS : Read-access geo-zone-redundant storage.
- enable_https_traffic_only : (Facultatif) Force seulement le trafic HTTPS pour le compte de stockage. Les valeurs possibles sont true et false. Par défaut, il est réglé sur true.
- access_tier : (Facultatif) Définit le niveau d'accès en dehors des heures pour le compte de stockage. Les valeurs possibles sont Hot et Cool. Par défaut, il est réglé sur Hot.
- allow_blob_public_access : (Facultatif) Permet l'accès public aux blobs ou aux conteneurs. Par défaut, il est réglé sur true.

Gestion de l’Infrastructure - compte de stockage - Exercice

Vous êtes un ingénieur DevOps et on vous a demandé d'automatiser la création des comptes de stockage Azure pour un nouveau projet. Vous avez besoin de créer deux comptes de stockage : un pour stocker les fichiers de l'application et un autre pour stocker les logs. De plus, vous devez restreindre l'accès à ces comptes de stockage aux adresses IP spécifiées.

- Créez un groupe de ressources Azure.
- Créez un compte de stockage de usage général v2 pour stocker les fichiers de l'application.
- Créez un compte de stockage de blobs pour stocker les logs.
- Configurez les règles d'accès au réseau pour restreindre l'accès aux comptes de stockage aux adresses IP spécifiées.
- Utilisez des variables et des outputs dans votre configuration Terraform.

Gestion de l'Infrastructure - déploiement d'applications Web

- Déployer des applications web sur Azure avec Terraform implique plusieurs composants
- **App Service** : C'est un service entièrement géré pour créer, déployer et échelonner rapidement des applications web et des API.
- **Azure Container Instances** : Ce service permet de déployer des conteneurs sur Azure sans avoir à gérer les serveurs.
- **Azure Kubernetes Service (AKS)** : C'est un service entièrement géré pour déployer, gérer et échelonner des applications conteneurisées en utilisant Kubernetes, l'orchestrateur de conteneurs open-source.
- **Azure Functions** : C'est un service qui permet de créer, déployer et gérer des applications sans serveur.

Gestion de l'Infrastructure - déploiement d'applications Web

App Service

```
resource "azurerm_app_service_plan" "example" {
  name                = "example-appserviceplan"
  location            = azurerm_resource_group.example.location
  resource_group_name = azurerm_resource_group.example.name
  kind                = "Linux"
  reserved            = true
  sku {
    tier = "Basic"
    size = "B1"
  }
}
resource "azurerm_app_service" "example" {
  name                = "example-appservice"
  location            = azurerm_resource_group.example.location
  resource_group_name = azurerm_resource_group.example.name
  app_service_plan_id = azurerm_app_service_plan.example.id
  site_config {
    linux_fx_version = "DOCKER|nginx"
  }
  app_settings = {
    "WEBSITES_ENABLE_APP_SERVICE_STORAGE" = "false"
  }
}
```

Gestion de l'Infrastructure - déploiement d'applications Web

App Service

Options possible:

- name (Requis): Le nom de l'App Service.
- location (Requis): La localisation géographique de l'App Service.
- resource_group_name (Requis): Le nom du groupe de ressources dans lequel créer l'App Service.
- app_service_plan_id (Requis): L'ID du Plan de Service d'App à associer à cet App Service.
- site_config (Optionnel): Un bloc site_config qui contient des configurations pour l'App Service (voir ci-dessous pour plus de détails).
- app_settings (Optionnel): Un mappage de valeurs de paramètres d'application.
- connection_string (Optionnel): Un bloc connection_string qui contient des chaînes de connexion pour l'App Service.
- client_affinity_enabled (Optionnel): Un booléen indiquant si l'affinité client est activée ou non. Par défaut, c'est false.
- https_only (Optionnel): Un booléen qui force l'usage du HTTPS pour les requêtes. Par défaut, c'est false.
- backup (Optionnel): Un bloc backup qui contient des configurations pour les sauvegardes automatisées.

Gestion de l'Infrastructure - déploiement d'applications Web

Azure Container Instances

- Avec Terraform, vous pouvez gérer les instances de conteneur Azure en utilisant la ressource azurerm_container_group.

```
resource "azurerm_container_group" "example" {
  name                = "example-containergroup"
  location            = azurerm_resource_group.example.location
  resource_group_name = azurerm_resource_group.example.name
  os_type             = "Linux"
  container {
    name   = "example-container"
    image  = "nginx:latest"
    cpu    = "0.5"
    memory = "1.5"
    ports {
      port     = 80
      protocol = "TCP"
    }
  }
  ip_address {
    type          = "Public"
    dns_name_label = "example"
  }
  tags = {
    environment = "testing"
  }
}
```

Gestion de l'Infrastructure - déploiement d'applications Web

Azure Container Instances

Options possible:

- name (Requis): Le nom du groupe de conteneurs.
- location (Requis): La localisation géographique du groupe de conteneurs.
- resource_group_name (Requis): Le nom du groupe de ressources dans lequel créer le groupe de conteneurs.
- os_type (Requis): Le système d'exploitation du groupe de conteneurs. Peut être Windows ou Linux.
- container (Requis): Un ou plusieurs blocs de container qui définissent les conteneurs à déployer dans le groupe de conteneurs.
- ip_address (Optionnel): Un bloc ip_address qui défini les paramètres d'accès au réseau pour le groupe de conteneurs.
- volume (Optionnel): Un bloc volume qui défini les volumes à monter dans les conteneurs.

Gestion de l'Infrastructure - déploiement d'applications Web

Azure Kubernetes Service (AKS)

- Pour créer un cluster AKS avec Terraform, vous devez utiliser la ressource azurerm_kubernetes_cluster.

```
resource "azurerm_kubernetes_cluster" "example" {
  name                = "example-aks"
  location            = azurerm_resource_group.example.location
  resource_group_name = azurerm_resource_group.example.name
  dns_prefix          = "exampleaks"

  default_node_pool {
    name       = "default"
    node_count = 1
    vm_size    = "Standard_D2_v2"
  }

  identity {
    type = "SystemAssigned"
  }

  tags = {
    Environment = "Production"
  }
}
```

Gestion de l'Infrastructure - déploiement d'applications Web

Azure Kubernetes Service (AKS)

Options possible:

- default_node_pool: C'est un bloc qui définit le pool de nœuds par défaut du cluster. Il contient plusieurs sous-options:
- name: (Requis) Le nom du pool de nœuds par défaut.
- node_count: (Requis) Le nombre de nœuds dans le pool de nœuds par défaut.
- vm_size: (Requis) La taille des machines virtuelles dans le pool de nœuds par défaut.
- enable_auto_scaling: (Optionnel) Définit si l'auto-scaling doit être activé pour le pool de nœuds par défaut.
- min_count: (Optionnel) Le nombre minimum de nœuds dans le pool de nœuds par défaut lors de l'utilisation de l'auto-scaling.
- max_count: (Optionnel) Le nombre maximum de nœuds dans le pool de nœuds par défaut lors de l'utilisation de l'auto-scaling.

Gestion de l'Infrastructure - déploiement d'applications Web

Azure Kubernetes Service (AKS)

Options possible:

- identity: C'est un bloc qui définit l'identité du cluster. Il contient plusieurs sous-options:
- type: (Requis) Le type d'identité à utiliser pour le cluster. Les valeurs possibles sont SystemAssigned (identité managée) et UserAssigned (identité managée assignée par l'utilisateur).
- dns_prefix: (Requis) Le préfixe DNS pour les adresses IP assignées au cluster.
- linux_profile: C'est un bloc qui définit le profil Linux pour les nœuds du cluster. Il contient plusieurs sous-options:
- admin_username: (Requis) Le nom d'utilisateur administrateur pour les nœuds du cluster.
- ssh_key: (Requis) C'est un bloc qui contient les clés SSH pour l'accès aux nœuds du cluster.
- network_profile: C'est un bloc qui définit le profil réseau du cluster. Il contient plusieurs sous-options:
- network_plugin: (Requis) Le plugin réseau à utiliser pour le cluster. Les valeurs possibles sont azure et kubenet.
- service_cidr: (Optionnel) Le CIDR pour les services Kubernetes dans le cluster.
- dns_service_ip: (Optionnel) L'adresse IP du service DNS dans le cluster.

Gestion de l'Infrastructure - déploiement d'applications Web

Azure Functions

- Les fonctions Azure ont besoin d'un compte de stockage pour stocker les fichiers de fonction et les journaux. Vous pouvez créer un compte de stockage en utilisant la ressource azurerm_storage_account
- Les fonctions Azure ont besoin d'un plan de service azurerm_app_service_plan

options possible:

- name: Le nom de l'application de fonction Azure.
- resource_group_name: Le nom du groupe de ressources dans lequel créer l'application de fonction.
- location: L'emplacement géographique où la ressource sera créée.
- app_service_plan_id: L'ID du plan de service d'application dans lequel créer cette application de fonction.
- storage_account_name: Le nom du compte de stockage Azure.
- storage_account_access_key: La clé d'accès pour le compte de stockage.
- version: La version de la runtime Functions. Les valeurs possibles sont ~1, ~2, et ~3.
- app_settings: Un mappage de paramètres d'application.

Gestion de l'Infrastructure - déploiement d'applications Web

Azure Functions

```
resource "azurerm_storage_account" "example" {
  name                     = "examplestoracc"
  resource_group_name      = azurerm_resource_group.example.name
  location                 = azurerm_resource_group.example.location
  account_tier             = "Standard"
  account_replication_type = "LRS"
}
resource "azurerm_app_service_plan" "example" {
  name                = "example-asp"
  location            = azurerm_resource_group.example.location
  resource_group_name = azurerm_resource_group.example.name
  kind                = "FunctionApp"
  sku {
    tier = "Dynamic"
    size = "Y1"
  }
}
resource "azurerm_function_app" "example" {
  name                       = "example-functionapp"
  location                   = azurerm_resource_group.example.location
  resource_group_name        = azurerm_resource_group.example.name
  app_service_plan_id        = azurerm_app_service_plan.example.id
  storage_account_name       = azurerm_storage_account.example.name
  storage_account_access_key = azurerm_storage_account.example.primary_access_key
  version                    = "~3"

  app_settings = {
    "FUNCTIONS_WORKER_RUNTIME" = "node"
  }
}
```

Gestion de l'Infrastructure - déploiement d'applications Web - Lab

- **Objectif** : Le but de ce TP est de vous familiariser avec l'automatisation du déploiement d'applications web sur Azure en utilisant Terraform. Vous apprendrez à créer et à configurer une application web Azure, un Container Instance, un Azure Kubernetes Service (AKS), et une Azure Function.
- **Contexte** : Vous êtes un ingénieur DevOps et on vous a demandé d'automatiser le déploiement d'une application web sur Azure pour un nouveau projet. L'application se compose de plusieurs composants : une application web, un service en conteneur, un service Kubernetes et une fonction. Vous devrez créer et configurer chacun de ces composants.
- **Exigences** :
- Créer un groupe de ressources Azure.
- Créer une application web Azure (App Service).
- Créer une instance de conteneur Azure (Container Instance).
- Créer un service Azure Kubernetes (AKS).
- Créer une fonction Azure (Azure Functions).
- Configurer les paramètres d'application de l'application web.

Gestion de l'Infrastructure - serveur Base de données

- **Azure SQL Database** : C'est un service de base de données relationnelle entièrement géré, basé sur le moteur SQL Server de Microsoft. Avec Terraform, vous pouvez créer, configurer et gérer des instances de bases de données SQL, ainsi que configurer les pare-feux, les règles d'audit, les élasticités, etc.
- **Azure Cosmos DB** : C'est un service de base de données multi-modèle et distribuée globalement. Il permet de stocker et d'interroger des données NoSQL, et prend en charge les modèles de données clé-valeur, documents, colonnes et graphes. Avec Terraform, vous pouvez créer et gérer des comptes, des bases de données, des conteneurs, etc., ainsi que configurer les politiques d'index, les politiques de cohérence, les règles d'audit, etc.
- **Azure Database for PostgreSQL, MySQL, et MariaDB** : Ce sont des services de bases de données relationnelles entièrement gérés, basés sur les moteurs de bases de données open-source PostgreSQL, MySQL et MariaDB. Avec Terraform, vous pouvez créer et gérer des instances de serveur, configurer les paramètres de base de données, les règles de pare-feu, les paramètres d'audit, etc.

Gestion de l'Infrastructure - serveur Base de données

Azure SQL Database

```
provider "azurerm" {
  features {}
}

resource "azurerm_resource_group" "example" {
  name     = "example-resources"
  location = "East US"
}

resource "azurerm_sql_server" "example" {
  name                         = "example-sqlserver"
  resource_group_name          = azurerm_resource_group.example.name
  location                     = azurerm_resource_group.example.location
  version                      = "12.0"
  administrator_login          = "admin"
  administrator_login_password = "Password123!"
}

resource "azurerm_sql_database" "example" {
  name                = "example-database"
  resource_group_name = azurerm_resource_group.example.name
  server_name         = azurerm_sql_server.example.name
  location            = azurerm_resource_group.example.location
  requested_service_objective_name = "S0"
}

output "sql_server_fqdn" {
  value = azurerm_sql_server.example.fully_qualified_domain_name
}
```

Gestion de l'Infrastructure - serveur Base de données

Azure Cosmos DB

```
provider "azurerm" {
  features {}
}

resource "azurerm_resource_group" "example" {
  name     = "example-resources"
  location = "East US"
}

resource "azurerm_cosmosdb_account" "example" {
  name                = "example-cosmosdb"
  resource_group_name = azurerm_resource_group.example.name
  location            = azurerm_resource_group.example.location
  offer_type          = "Standard"
  kind                = "GlobalDocumentDB"

  consistency_policy {
    consistency_level       = "BoundedStaleness"
    max_interval_in_seconds = 10
    max_staleness_prefix    = 200
  }

  geo_location {
    location          = azurerm_resource_group.example.location
    failover_priority = 0
  }
}
resource "azurerm_cosmosdb_sql_database" "example" {
  name                = "example-db"
  resource_group_name = azurerm_resource_group.example.name
  account_name        = azurerm_cosmosdb_account.example.name
  throughput          = 400
}
```

Gestion de l'Infrastructure - serveur Base de données

Azure Cosmos DB

```
provider "azurerm" {
  features {}
}
resource "azurerm_resource_group" "example" {
  name     = "example-resources"
  location = "East US"
}
resource "azurerm_postgresql_server" "example" {
  name                = "example-psqlserver"
  location            = azurerm_resource_group.example.location
  resource_group_name = azurerm_resource_group.example.name

  sku_name = "B_Gen5_2"

  storage_mb                   = 5120
  backup_retention_days        = 7
  geo_redundant_backup_enabled = false
  auto_grow_enabled            = false

  administrator_login          = "psqladminun"
  administrator_login_password = "H@Sh1CoR3!"

  version = "9.5"

  ssl_enforcement_enabled = true
}
resource "azurerm_postgresql_database" "example" {
  name                = "example-database"
  resource_group_name = azurerm_resource_group.example.name
  server_name         = azurerm_postgresql_server.example.name
  charset             = "UTF8"
  collation           = "English_United States.1252"
}
```

Gestion de l'Infrastructure - Lab

- **Objectif**
  Le but de ce TP est de vous familiariser avec l'automatisation de la création des ressources Azure en utilisant Terraform. Vous apprendrez à créer des machines virtuelles, des comptes de stockage, à déployer des applications Web sur différents services, à configurer une base de données Azure SQL et à utiliser des structures conditionnelles et itératives.
- **Contexte**
  Vous travaillez pour une entreprise qui développe une application Web complexe qui nécessite une base de données Azure SQL, un stockage de fichiers, un serveur Web sur une machine virtuelle, une application sur Azure App Service, un conteneur sur Azure Container Instances, et un cluster Azure Kubernetes Service (AKS). Vous avez été chargé de créer et de configurer l'infrastructure nécessaire pour cette application sur Azure. Vous devez automatiser la création et la configuration de cette infrastructure en utilisant Terraform.

Gestion de l'Infrastructure - Lab

- **Exigences** :

1. Création de machines virtuelles : Créer une machine virtuelle pour le serveur Web.
2. Création d'un compte de stockage : Créer un compte de stockage Azure pour stocker les fichiers de l'application.
3. Déploiement d'un serveur Web : Déployer un serveur Web sur la machine virtuelle créée à l'étape 1.
4. Déploiement d'une base de données Azure SQL : Créer et configurer une base de données Azure SQL.
5. Déploiement d'une application sur Azure App Service : Déployer une application Web sur Azure App Service.
6. Déploiement d'un conteneur sur Azure Container Instances : Déployer un conteneur sur Azure Container Instances.
7. Déploiement d'un cluster Azure Kubernetes Service (AKS) : Créer un cluster AKS et déployer une application sur celui-ci.
8. Utilisation de structures conditionnelles et itératives : Utiliser des expressions conditionnelles pour déployer un cluster AKS seulement si une variable spécifique est définie à true. Utiliser des structures itératives pour créer plusieurs instances de conteneurs sur Azure Container Instances.

Terraform Provisioner

- Les provisionneurs Terraform sont utilisés pour exécuter des commandes spécifiques sur la machine locale ou sur une machine distante après le déploiement de la ressource. Cela peut être utile pour initialiser une ressource fraîchement créée, par exemple pour installer un logiciel sur une machine virtuelle nouvellement créée.
- Par exemple, pour provisionner une machine virtuelle Azure avec Terraform, vous pourriez utiliser le provisionneur remote-exec pour exécuter des scripts sur la machine virtuelle après sa création.

```
resource "azurerm_virtual_machine" "example" {
  name                  = "example-vm"
  location              = azurerm_resource_group.example.location
  resource_group_name   = azurerm_resource_group.example.name
  network_interface_ids = [azurerm_network_interface.example.id]
  vm_size               = "Standard_D2s_v3"

  ...

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

    connection {
      type     = "ssh"
      user     = "testadmin"
      password = "${var.admin_password}"
      host     = "${azurerm_public_ip.example.ip_address}"
    }
  }
}
```

Terraform Provisioner

- **local-exec** : Ce provisionneur exécute des commandes sur la machine locale, c'est-à-dire la machine qui exécute Terraform.
- **remote-exec** : Ce provisionneur exécute des commandes sur une machine distante. Dans l'exemple que je vous ai montré, nous avons utilisé le provisionneur remote-exec pour exécuter des commandes sur la machine virtuelle Azure nouvellement créée.
- **file** : Ce provisionneur permet de copier des fichiers ou des répertoires d'une machine locale vers une machine distante.

L'utilisation des provisionneurs est souvent déconseillée, car elle peut entraîner des configurations difficilement reproductibles. Il est généralement préférable d'utiliser des outils de configuration tels que Ansible, Chef ou Puppet pour configurer les machines virtuelles après leur création

Terraform Provisioner - Exercice

- Utiliser un provisionneur installer un serveur web (par exemple, Nginx) sur la machine virtuelle après sa création. Vous devrez générer une paire de clés SSH pour vous connecter à la machine virtuelle.
- Utiliser un provisionneur pour copier un fichier index.html de votre machine locale vers le répertoire racine du serveur web sur la machine virtuelle.

Automatiser son workflow avec Terraform

- Automatiser son workflow avec Terraform signifie utiliser Terraform pour codifier et gérer toutes les étapes de votre processus de déploiement d'infrastructure, afin de minimiser l'intervention manuelle, les erreurs et le temps nécessaire pour déployer et gérer votre infrastructure.

1. **Codifier l'Infrastructure**: Écrivez le code qui décrit l'infrastructure que vous souhaitez créer ou modifier. Cela inclut la définition de toutes les ressources dont vous avez besoin (comme les machines virtuelles, les réseaux, etc.) en utilisant le langage de configuration de Terraform (HCL).
2. **Gestion du Code Source**: Stockez votre code Terraform dans un système de gestion de version de code source (comme Git) pour suivre les modifications, collaborer avec d'autres membres de l'équipe et gérer les versions de votre code
3. **Automatisation des Tests**: Écrivez des tests pour votre code Terraform pour vous assurer qu'il fonctionne comme prévu. Cela peut inclure des tests unitaires pour des morceaux de code individuels, des tests d'intégration pour s'assurer que différentes parties de votre système fonctionnent ensemble, et des tests fonctionnels pour s'assurer que tout le système fonctionne comme prévu.

Automatiser son workflow avec Terraform

4. **Intégration Continue (CI)**: Utilisez un système d'intégration continue pour automatiser l'exécution de vos tests chaque fois que vous apportez des modifications à votre code. Cela vous aidera à identifier et à corriger rapidement les éventuels problèmes
5. **Déploiement Continu (CD)**: Utilisez un système de déploiement continu pour automatiser le déploiement de votre code en production. Cela peut inclure l'automatisation de l'exécution de terraform plan pour afficher les modifications qui seront apportées à votre infrastructure, et terraform apply pour appliquer ces modifications
6. **Gestion de l'État**: Utilisez un backend distant pour stocker l'état de votre infrastructure. Cela permet à plusieurs membres de l'équipe de travailler sur le même projet en même temps et de gérer l'état de manière sécurisée.
7. **Gestion des Variables**: Utilisez des fichiers de variables et des variables d'environnement pour gérer les configurations spécifiques à chaque environnement. Cela permet de réutiliser le même code pour différents environnements (comme le développement, le test et la production) en changeant simplement les valeurs des variables
8. **Modules**: Réutilisez le code en créant des modules pour des morceaux de votre infrastructure que vous utilisez fréquemment. Cela vous permet de ne pas réécrire le même code encore et encore.

Les modules Terraform

- Les modules Terraform sont utilisés pour encapsuler des groupes de ressources dans des blocs réutilisables. Cela permet de créer des morceaux d'infrastructure réutilisables et partageables, d'organiser l'infrastructure en petits morceaux maniables et de mieux gérer l'infrastructure complexe.

1. **Création de Modules**:

- Pour créer un module, vous créez un nouveau répertoire et y placez un fichier de configuration Terraform. Par exemple, si vous avez un ensemble de configurations que vous utilisez fréquemment pour créer des machines virtuelles dans Azure, vous pouvez créer un module pour cela.
- Supposons que vous créiez un module pour une machine virtuelle Azure. Vous aurez un fichier principal (par exemple, main.tf) dans lequel vous définirez les ressources nécessaires pour créer une machine virtuelle, par exemple azurerm_virtual_machine, azurerm_network_interface, etc.
- Vous définirez également les variables dans un fichier (par exemple, variables.tf) et les valeurs par défaut ou les sorties dans un autre fichier (par exemple, outputs.tf).

Les modules Terraform

2. **Partage de Modules**:

- Une fois que vous avez créé un module, vous pouvez le partager avec d'autres en le mettant dans un registre de modules comme le Terraform Registry ou un dépôt Git.
- Si vous travaillez au sein d'une organisation, vous pouvez également utiliser un registre privé, comme Azure Artifact ou le registre de modules privés de Terraform Enterprise.
- Assurez-vous que le module est bien documenté, de sorte que d'autres sachent comment l'utiliser, quelles sont les variables requises, etc.

3. **Utilisation de Modules**:

- Pour utiliser un module, vous l'invoquez dans votre configuration Terraform avec le bloc module.
- Vous spécifiez la source du module (c'est-à-dire l'emplacement où il est stocké, par exemple, le Terraform Registry, un dépôt Git, etc.) et les valeurs pour les variables que le module requiert.

Les modules Terraform

3. **Utilisation de Modules**:

- Tarraform registry

```
module "azure_virtual_machine" {
  source  = "Azure/virtual-machine/azure"
  version = "2.0.0"
  #...
}
```

- Github

```
provider "azurerm" {
  features {}
}

module "vm" {
  source = "git::https://github.com/your-org/azure-vm-module.git"

  resource_group_name = "my-rg"
  location            = "East US"
  vm_name             = "my-vm"
}
```

Les modules Terraform - Lab

- **Objectif** :L'objectif de ce TP est de vous apprendre à créer un module Terraform, à le partager sur GitHub et à l'utiliser dans une configuration Terraform pour déployer des ressources dans Azure.
- **Contexte** : Vous travaillez en tant qu'ingénieur DevOps pour une entreprise qui utilise intensivement Microsoft Azure. On vous a demandé de créer un module Terraform réutilisable pour déployer des machines virtuelles dans Azure. Vous devez ensuite utiliser ce module pour déployer une machine virtuelle dans Azure.
- **Exigences :**

1. Créez un nouveau répertoire pour votre module, par exemple azure-vm-module.
   Le module devra créer une machine virtuelle dans Azure.
2. Créez un nouveau dépôt sur GitHub et publiez-y votre module. Assurez-vous d'inclure un fichier README.md qui explique comment utiliser le module, les variables requises, etc.
3. Créez une nouvelle configuration Terraform dans un répertoire séparé.
   Dans cette configuration, utilisez le module que vous avez créé pour déployer une machine virtuelle dans Azure. Configurez les variables nécessaires pour le module.

Gestion des outputs Terraform

- La gestion des outputs dans Terraform est un moyen essentiel de récupérer des informations sur les ressources que vous créez, ce qui peut être particulièrement important dans un environnement cloud complexe comme Azure.

1. **Récupération des Outputs**:

- Les outputs dans Terraform sont les valeurs que Terraform renvoie après avoir exécuté votre configuration. Vous pouvez utiliser ces valeurs dans d'autres parties de votre configuration, ou en dehors de Terraform.
- Pour définir un output dans votre configuration Terraform, vous utilisez le bloc output

```
output "public_ip" {
  value = azurerm_virtual_machine.example.public_ip_address
  description = "The public IP address of the virtual machine."
}
```

Gestion des outputs Terraform

2. **Utilisation des Outputs dans d'autres modules**:

- Lorsque vous utilisez des modules dans Terraform, vous pouvez aussi récupérer et utiliser les outputs de ces modules. Par exemple, si vous avez un module qui crée une machine virtuelle dans Azure, et que vous voulez utiliser l'adresse IP publique de cette machine virtuelle dans une autre partie de votre configuration

```
module "virtual_machine" {
  source = "./modules/virtual_machine"
  ...
}

output "vm_public_ip" {
  value = module.virtual_machine.public_ip
}
```

Gestion des outputs Terraform

3. **Utilisation des Outputs en dehors de Terraform** :
   Vous pouvez aussi récupérer les outputs en dehors de Terraform en utilisant la commande terraform output

Les outputs peuvent être des valeurs sensibles, comme des mots de passe ou des clés d'accès. Par défaut, Terraform affichera ces valeurs dans la sortie lors de l'exécution de terraform apply ou terraform output. Vous pouvez empêcher cela en définissant l'argument sensitive à true dans la définition de l'output

```
output "admin_password" {
  value = azurerm_virtual_machine.example.admin_password
  description = "The administrator password of the virtual machine."
  sensitive = true
}
```

Gestion des outputs Terraform - Exercice

- Utilisez le module Terraform pour déployer une machine virtuelle dans Azure
- Ce module doit renvoyer l'adresse IP publique de la machine virtuelle créée en tant qu'output.
- Créez un second module Terraform pour déployer un serveur de base de données Azure SQL:
  - Le module doit accepter des variables pour le nom du serveur, l'emplacement, le nom de l'administrateur du serveur et le mot de passe de l'administrateur.
  - Le module doit créer un serveur de base de données Azure SQL et une base de données.
- Utilisez les outputs du premier module dans le second module:
  - Utilisez l'adresse IP publique de la machine virtuelle créée dans le premier module comme paramètre d'entrée pour le second module. Par exemple, vous pouvez utiliser cette adresse IP pour configurer les règles du pare-feu de la base de données Azure SQL.

Terraform et les environnements

- Terraform et la gestion de plusieurs environnements dans le contexte d'Azure impliquent l'utilisation de configurations, de workspaces, de variables et, éventuellement, de modules pour gérer et déployer les ressources d'Azure dans différents environnements tels que le développement, la production, etc.

1. **Utiliser des Workspaces** :

- Terraform Workspaces vous permet de gérer plusieurs environnements en isolant leurs états. Par exemple, vous pouvez avoir un workspace pour 'dev' et un autre pour 'prod', chacun avec son propre état.

2. Utiliser des Variables :

- Vous pouvez utiliser des variables pour définir des valeurs spécifiques à l'environnement. Par exemple, la taille de la machine virtuelle dans l'environnement de production peut être différente de celle de l'environnement de développement.

3. **Utiliser des Modules** :

- Les modules permettent de regrouper et de réutiliser des configurations Terraform. Vous pouvez créer un module de base et l'utiliser dans différents environnements en passant des variables spécifiques à l'environnement.

Terraform et les environnements

4. Gestion des États :

- L'état de Terraform doit être stocké de manière sécurisée et accessible à partir des différents environnements. Azure Blob Storage peut être utilisé pour stocker l'état de Terraform.

```
terraform {
  backend "azurerm" {
    storage_account_name = "tfstatestorageaccount"
    container_name       = "tfstatecontainer"
    key                  = "terraform.tfstate"
  }
}
```

5. **Contrôle d'Accès** :

- Vous devez contrôler l'accès aux différents environnements. Azure Active Directory (Azure AD) peut être utilisé pour gérer l'accès aux ressources d'Azure. Vous pouvez également utiliser le contrôle d'accès basé sur les rôles (RBAC) d'Azure pour contrôler l'accès aux ressources d'Azure.

6. **Automatisation** :

- Vous pouvez automatiser le déploiement de vos configurations Terraform à l'aide des pipelines Azure DevOps ou d'autres outils d'intégration continue et de déploiement continu (CI/CD).

Terraform et les environnements - Workspaces

- Par défaut, chaque configuration Terraform a un seul "workspace", appelé "default". Cependant, vous pouvez créer plusieurs "workspaces" pour gérer différentes configurations ou environnements, comme le développement, le test et la production.

1. **Créer un Workspace** :

- Vous pouvez créer un nouveau "workspace" en utilisant la commande terraform workspace new <NOM_WORKSPACE>.

```
terraform workspace new development
```

2. **Sélectionner un Workspace** :

- Vous pouvez sélectionner un "workspace" existant en utilisant la commande terraform workspace select <NOM_WORKSPACE>.

```
terraform workspace select development
```

Terraform et les environnements - Workspaces

3. **Lister les Workspaces** :

- Vous pouvez lister tous les "workspaces" existants en utilisant la commande terraform workspace list.

```
terraform workspace list
```

4. **Supprimer un Workspace** :

Vous pouvez supprimer un "workspace" en utilisant la commande terraform workspace delete <NOM_WORKSPACE>.

```
terraform workspace delete development
```

Terraform et les environnements - Workspaces - Lab

- **Objectif** : Le but de ce TP est de vous apprendre à gérer différents environnements comme le développement, le test et la production dans Azure en utilisant les workspaces de Terraform.
- **Contexte** : Vous êtes un ingénieur DevOps et on vous a demandé de créer des ressources Azure pour différents environnements (développement, test, production) en utilisant Terraform. Vous devez créer une configuration Terraform qui peut être utilisée pour créer les mêmes ressources dans différents environnements en utilisant les workspaces de Terraform.
- **Exigences** :
- Configurer les crédentials Azure pour être utilisés par Terraform.
- Configurer le backend de Terraform pour stocker l'état de votre configuration dans Azure Blob Storage.
- Créer un fichier de variables pour définir toutes les variables que vous utiliserez dans votre configuration.
- Créer un fichier de configuration pour définir toutes les ressources que vous souhaitez créer dans Azure.
- Utiliser la commande terraform workspace pour créer et gérer les workspaces.

Troubleshooting

- Le dépannage dans Terraform peut impliquer plusieurs étapes et techniques différentes en fonction de la nature du problème.
- **Consultez les journaux et les messages d'erreur** : Les messages d'erreur affichés par Terraform peuvent souvent fournir des informations précieuses sur ce qui ne va pas. Assurez-vous de lire et de comprendre les messages d'erreur.
- **Utilisez terraform plan** : Avant d'appliquer les changements, utilisez toujours la commande terraform plan pour voir quels seront les changements apportés à votre infrastructure.
- **Utilisez la commande terraform show**: Cette commande vous permet de visualiser l'état actuel de votre infrastructure et peut être utile pour identifier les problèmes.
- **Vérifiez l'état de Terraform** : Utilisez la commande terraform state list pour voir toutes les ressources dans l'état actuel de Terraform. Cela peut être utile pour vérifier si une ressource a été créée, modifiée ou détruite.

Troubleshooting

- *__Vérifiez les versions de Terraform et des providers__: Assurez-vous d'utiliser des versions de Terraform et des providers compatibles. Vous pouvez spécifier les versions dans votre fichier de configuration Terraform.
- **Vérifiez les dépendances des ressources** : Assurez-vous que les ressources sont créées dans le bon ordre et que les dépendances sont correctement configurées. Terraform gère généralement les dépendances automatiquement, mais dans certains cas, vous devrez peut-être les spécifier explicitement à l'aide de l'argument depends_on.
- **Vérifiez les permissions et les politiques** : Assurez-vous que le compte ou le service principal utilisé par Terraform a les permissions nécessaires pour créer, modifier ou détruire les ressources.
- **Vérifiez les limites de votre compte** : Certains services cloud ont des limites sur le nombre de ressources que vous pouvez créer. Assurez-vous que vous n'avez pas atteint ces limites.
- **Consultez la documentation** : La documentation de Terraform et des providers peut souvent fournir des informations utiles pour résoudre les problèmes.

Troubleshooting

- **Vérifiez les valeurs des variables** : Assurez-vous que les valeurs des variables que vous avez spécifiées sont correctes. Vous pouvez utiliser la commande terraform console pour vérifier les valeurs des variables et tester les expressions Terraform.
- **Utilisez terraform validate** : Cette commande vérifie que votre fichier de configuration est syntaxiquement correct et valide.
- **Vérifiez les ressources créées manuellement** : Assurez-vous que les ressources que vous gérez avec Terraform n'ont pas été modifiées manuellement ou par un autre outil. Terraform peut avoir du mal à gérer les ressources qui ne sont pas entièrement gérées par lui-même.
- **Utilisez des sorties de débogage** : Vous pouvez obtenir des informations supplémentaires sur l'exécution de Terraform en définissant la variable d'environnement TF_LOG à DEBUG ou TRACE.
- **Utilisez le terraform graph** : Cette commande génère un graphique visuel de vos ressources Terraform et de leurs dépendances. Vous pouvez utiliser un outil comme Graphviz pour visualiser ce graphique.

Etude de cas

Déploiement d'une Application Web Multi-Tiers sur Azure avec Terraform

Contexte: Une entreprise souhaite déployer une application web multi-tiers sur Microsoft Azure. L'application est composée de trois tiers: une interface utilisateur frontend, une API backend, et une base de données. L'entreprise souhaite utiliser Terraform pour définir et gérer son infrastructure sur Azure. De plus, l'entreprise souhaite suivre les meilleures pratiques pour gérer les différents environnements (développement, test, production) et automatiser le workflow de déploiement.
Exigences:

1. **Infrastructure**:

- Créer un réseau virtuel (VNet) et des sous-réseaux pour les différentes couches de l'application.
- Créer des groupes de ressources pour organiser les ressources.
- Déployer des machines virtuelles (VMs) pour l'application frontend et l'API backend.
- Déployer un serveur de base de données Azure SQL.
- Configurer les règles de groupe de sécurité réseau (NSG) pour contrôler l'accès aux différentes ressources.
- Créer un compte de stockage pour stocker les fichiers statiques de l'application.

Etude de cas

Déploiement d'une Application Web Multi-Tiers sur Azure avec Terraform

2. **Provisionnement**:

- Utiliser les provisionneurs Terraform pour installer les dépendances et démarrer les applications sur les VMs.

3. **Gestion de l'État**:

- Configurer un backend distant pour stocker l'état de Terraform.
  Gérer les permissions pour accéder à l'état de Terraform.

4. **Modules**:

- Créer des modules Terraform réutilisables pour les différentes composantes de l'infrastructure (par exemple, un module pour les VMs, un autre pour la base de données, etc.).

Etude de cas

Déploiement d'une Application Web Multi-Tiers sur Azure avec Terraform

5. **Environnements**:

- Utiliser les workspaces Terraform pour gérer les différents environnements.

6. **Outputs**:

- Utiliser les outputs de Terraform pour récupérer des informations sur l'infrastructure déployée, comme les adresses IP des VMs ou l'URL de la base de données.

7. **Automatisation**:

- Automatiser le workflow de déploiement en utilisant les commandes terraform init, terraform plan, et terraform apply.
  Utiliser un système de contrôle de version (comme Git) pour gérer les versions de la configuration Terraform.