LES EXPRESSIONS SUR TERRAFORM (BOUCLES, CONDITIONS, ARITHMÉTIQUE)

Introduction

Comme discuté dans les parties précédentes de cette série. Terraform reste en grande partie un **langage déclaratif**. L'avantage de cette caractéristique surtout concernant l'Infrastructure As Code, est qu'elle a tendance à nous fournir une vue plus précise de ce qui est réellement déployé, plus que sur un langage procédural. Cependant, sans accès à un langage de programmation complet, certains types de tâches deviennent plus difficiles dans un langage déclaratif. Imaginez par exemple, utilisez un langage déclaratif sans système de boucles, comment allez-vous faire pour créer plusieurs fois identiquement vos ressources ? Allez-vous copier-coller le code plusieurs fois ? Et si le langage déclaratif en question ne prend pas en charge les conditions, comment pouvez-vous configurer conditionnellement vos ressources ?

Même si Terraform reste avant tout un langage déclaratif, il fournit tout de même une boîte à outils nommée "**les Expressions**" qui sont utilisées pour faire référence ou à calculer des valeurs dans une configuration telles que **les conditions**, **les boucles** , **l'arithmétique** et un certain nombre de fonctions intégrées. Dans ce chapitre nous étudierons les plus importantes d'entre elles.

Les expressions de Boucles

Terraform propose plusieurs constructions de boucles différentes, chacune étant destinée à être utilisée dans un scénario légèrement différent. Vous avez

notamment :

- `count` : utilisée pour boucler des ressources.
- `for_each` : utilisée pour boucler des ressources et des lignes de bloc au sein d'une ressource.
- `for` : utilisée pour boucler sur des listes et des maps.

Les boucles avec le paramètre count

Utilisation du paramètre count

Le paramètre `count` est donc spécialement conçu pour nous simplifier les configurations et nous permettre de **scaler (mettre à l'échelle) les ressources** en incrémentant simplement son nombre. Dans l'exemple suivant je vais indiquer à Terraform de créer deux fois la même ressource :

```
provider "aws" {
    region = "us-east-2"
}

resource "aws_instance" "my_ec2_instance" {
    count = 2
    ami = "ami-07c1207a9d40bc3bd"
    instance_type = "t2.micro"
    tags = {
        Name = "terraform test"
    }
}
```

Des variables peuvent également être utilisées pour développer une liste de ressources à utiliser :

```
variable "NBR_INSTANCE" {
    type = "number"
    default = 2
}
```

```
resource "aws_instance" "my_ec2_instance" {
    count = var.NBR_INSTANCE
    ami = "ami-07c1207a9d40bc3bd"
    instance_type = "t2.micro"
    tags = {
        Name = "terraform test"
    }
}
```

Vous avez également la possibilité de **récupérer l'index du paramètre `count`** :

```
provider "aws" {
    region = "us-east-2"
}

resource "aws_iam_user" "team_iam_user" {
    count = 3
    name = "user-${count.index}"
}
```

Exécutons ce code afin d'analyser plus tard son résultat :

```
terraform init && terraform plan
```

Résultat :

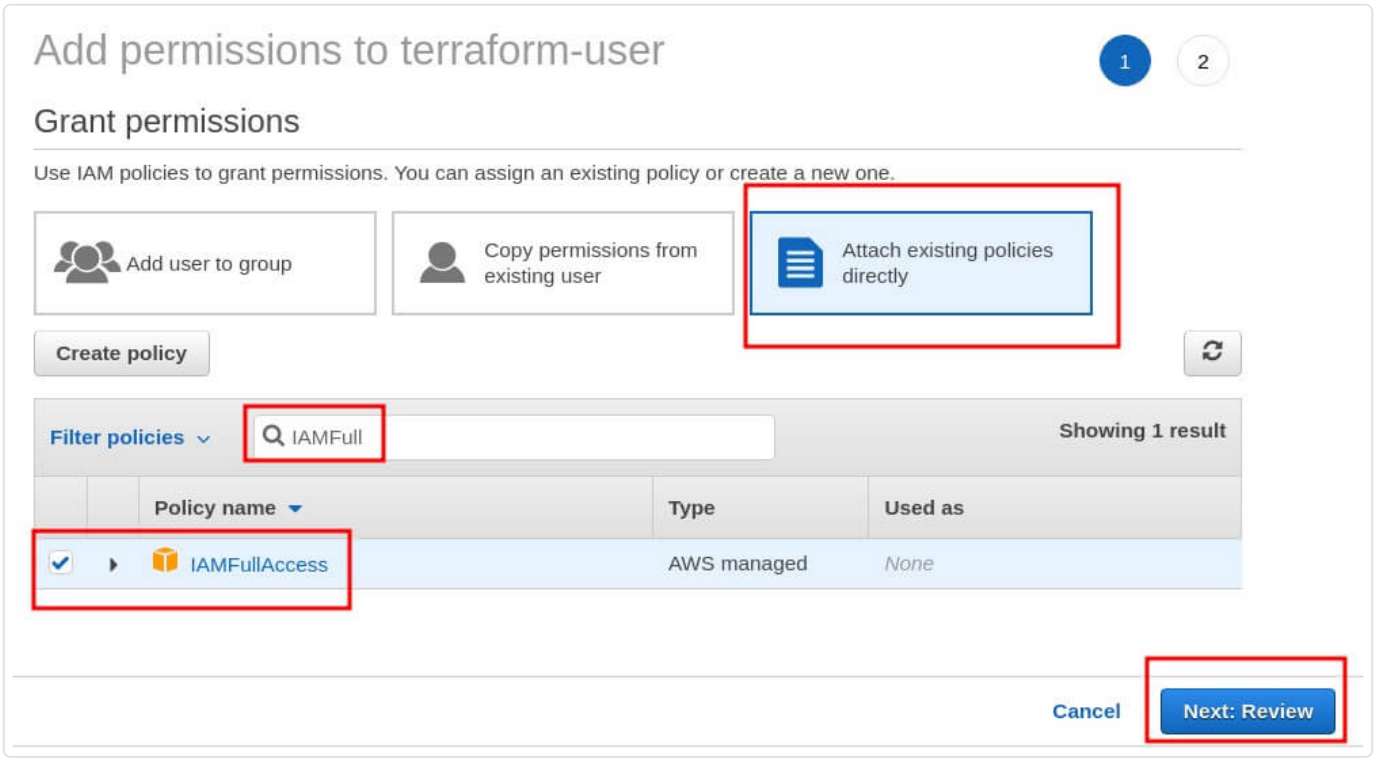
```
# aws_iam_user.team_iam_user[0] will be created
+ resource "aws_iam_user" "team_iam_user" {
    ...
    + name          = "user-0"
}

# aws_iam_user.team_iam_user[1] will be created
+ resource "aws_iam_user" "team_iam_user" {
    ...
    + name          = "user-1"
}

# aws_iam_user.team_iam_user[2] will be created
+ resource "aws_iam_user" "team_iam_user" {
    ...
    + name          = "user-2"
}

Plan: 3 to add, 0 to change, 0 to destroy.
```

On remarque que Terraform souhaite créer trois utilisateurs IAM, chacun avec un nom légèrement différent ("user-0", "user-1", "user-2"). Cependant, vous souhaitez peut-être personnaliser le nom de vos utilisateurs pour chaque itération de la boucle ? Dans ce cas je vous suggère de créer une liste contenant des noms personnalisés de vos utilisateurs et d'**utiliser le paramètre `count` pour accéder aux éléments de la liste**, ce qui nous donnera le code suivant :

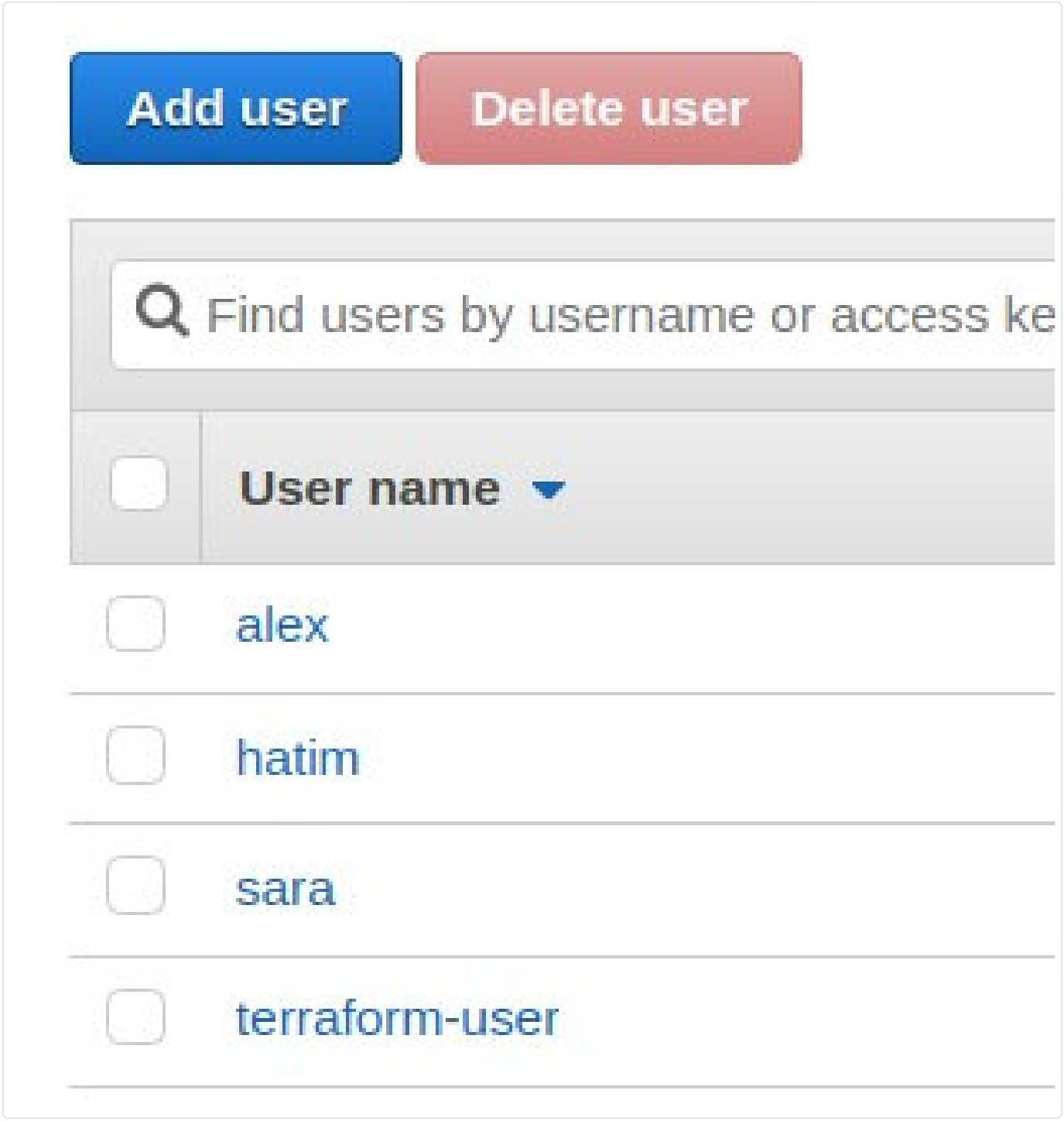
```
variable "IAM_USERS" {
    type = "list"
    default = ["hatim", "sarah", "alex"]
}

provider "aws" {
    region = "us-east-2"
}

resource "aws_iam_user" "team_iam_user" {
    count = length(var.IAM_USERS)
    name  = var.IAM_USERS[count.index]
}
```

Comme vous l'aurez probablement le deviner, la fonction **`length()` retourne le nombre d'éléments dans une liste**. Cette fonction fonctionne également avec des variables de type **`string`** et **`map`**.

Si vous souhaitez lancer le code précédent, n'oubliez de rajouter la policy "IAMFullAccess" à votre utilisateur IAM Terraform :



Lancez ensuite la commande `terraform init && terraform apply` et rendez-vous dans votre console IAM pour vérifier la création de vos nouveaux utilisateurs IAM :



Limite du paramètre count

Malheureusement, **le paramètre `count` possède une limitation** qui réduit considérablement son utilité. Je m'explique, le décompte utilisé précédemment est sensible pour tout changement dans l'ordre d'une liste, cela signifie que si pour une

raison quelconque l'ordre de la liste est modifié, terraform forcera le remplacement de toutes les ressources dont l'index dans la liste a changé. Une exemple sera beaucoup plus parlant, considérez la liste des utilisateurs IAM créés auparavant `["hatim", "sarah", "alex"]`. Imaginez maintenant que vous souhaitez supprimer l'utilisateur "sarah" de cette liste. À votre avis que se passera t-il lorsque vous lancerez à nouveau la commande `terraform plan` ? Essayons cela :

Voici déjà le contenu de notre nouveau code avec l'utilisateur "sarah" de supprimé :

```
variable "IAM_USERS" {
    type = "list"
    default = ["hatim", "alex"]
}

provider "aws" {
    region = "us-east-2"
}

resource "aws_iam_user" "team_iam_user" {
    count = length(var.IAM_USERS)
    name  = var.IAM_USERS[count.index]
}
```

Exécutons notre code :

```
terraform init && terraform plan
```

Résultat :

```
Terraform will perform the following actions:

  # aws_iam_user.team_iam_user[1] will be updated in-place
  ~ resource "aws_iam_user" "team_iam_user" {
      ...
    ~ name          = "sarah" -> "alex"
    }

  # aws_iam_user.team_iam_user[2] will be destroyed
  - resource "aws_iam_user" "team_iam_user" {
      ...
    - name          = "alex" -> null
    }
```

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

Attendez une seconde! Ce n'est probablement pas ce que vous attendiez ? Au lieu de simplement supprimer l'utilisateur IAM "sarah", la sortie de la commande plan indique que Terraform souhaite renommer l'utilisateur IAM "sarah" par "alex" et supprimer ensuite l'utilisateur IAM "alex". Que se passe-t-il vraiment ?

Lorsque vous utilisez le paramètre `count` sur une ressource, cette ressource devient un élement d'une liste de ressources. Malheureusement, la façon dont Terraform identifie chaque ressource dans cette liste est par sa position et donc son index. Autrement dit, après avoir exécuté la commande plan la première fois avec trois noms d'utilisateurs, la représentation interne de Terraform de ces utilisateurs IAM ressemble à ceci:

```
aws_iam_user.team_iam_user [0]: hatim
aws_iam_user.team_iam_user [1]: sarah
aws_iam_user.team_iam_user [2]: alex
```

Lorsque vous supprimez un élément du milieu d'un tableau, tous les éléments après celui-ci reculent de 1, donc après avoir exécuté la commande plan avec seulement deux noms dans votre tableau, la représentation interne de Terraform ressemblera à ceci :

```
aws_iam_user.team_iam_user [0]: hatim
aws_iam_user.team_iam_user [1]: alex
```

Vous remarquez alors que l'utilisateur IAM "alex" est passé de l'index 2 à l'index 1. Étant donné que Terraform voit l'index comme l'identifiant d'une ressource, ce changement est traduit par "renommer la ressource de l'index 1 en 'alex' et supprimer la ressource à l'index 2».

Cette manipulation est très dangereuse ! Car au final nous supprimons l'utilisateur IAM 'alex' avec toute sa configuration AWS (ce compte aurait pu être un compte admin par exemple) au lieu de réellement supprimer l'utilisateur IAM 'sara' qui est pour le moment juste renommé par 'alex'. Pour résoudre cette limitation, Terraform 0.12.6 a introduit les expressions `for_each`.

Les Boucles avec des expressions for_each

Résolution du problème

L'expression `for_each` nous permet essentiellement de faire la même chose que le paramètre `count`, donc de créer plusieurs instances de la même ressource, avec une petite mais très importante différence! Elle prend en entrée une variable de type `map` et utilise donc la clé de la map comme index des instances de la ressource créée et non son index.

Pour mieux comprendre l'utilisation de cette expression réutilisons le même exemple qu'avant, avec cette fois-ci l'expression `for_each`. Afin de repartir de 0, je vous invite également à supprimer vos anciens utilisateurs IAM créés antérieurement dans cet article avec la commande suivante :

```
terraform destroy
```

Ensuite, comme cette expression s'utilise avec des variables de type `map`, vous devez **convertir votre variable de type `list` en type `map`** à l'aide de la fonction `toset()`. Dans l'exemple ci-dessous, notre variable initiale est toujours définie comme une liste. Nous commencerons par remplacer le paramètre `count` par `for_each` et convertir notre liste en type map avec la fonction `toset()` et ensuite je planifierai notre état. Voici déjà notre nouveau code :

```
variable "IAM_USERS" {
    type = "list"
    default = ["hatim", "sarah", "alex"]
}

provider "aws" {
    region = "us-east-1"
}

resource "aws_iam_user" "team_iam_user" {
    for_each = toset(var.IAM_USERS)
    name  = each.value

    tags = {
        key = each.key
        value = each.value
    }
}
```

Vous remarquerez, que j'utilise `each.key` et `each.value` pour accéder à la clé et à la valeur de l'élément actuel notamment dans l'argument `tags` afin de les visualiser par la suite. Commençons donc par créer nos nouveaux utilisateurs avec la commande suivante :

```
terraform init && terraform apply
```

Résultat :

```
aws_iam_user.team_iam_user["sarah"]: Creation complete after 2s [id=sarah]
aws_iam_user.team_iam_user["alex"]: Creation complete after 2s [id=alex]
aws_iam_user.team_iam_user["hatim"]: Creation complete after 2s [id=hatim]
```

Vous pouvez voir dans la sortie du résultat que Terraform a bien créé trois utilisateurs IAM tel que demandé , en utilisant les clés de l'expression `for_each` (dans ce cas, par défaut il prend les noms d'utilisateurs). Maintenant supprimons à nouveau l'utilisateur IAM "sarah" de notre liste et observons le résultat obtenu :

```
Terraform will perform the following actions:

  # aws_iam_user.team_iam_user["sarah"] will be destroyed
  - resource "aws_iam_user" "team_iam_user" {
```

```
      - id            = "sarah" -> null
      - name          = "sarah" -> null
      - tags          = {
          - "key"   = "sarah"
          - "value" = "sarah"
        } -> null
}
```

Dans ce résultat, vous remarquerez que Terraform supprime maintenant uniquement la ressource exacte souhaitée, sans déplacer toutes les autres. C'est pourquoi vous devriez presque toujours préférer utiliser `for_each` plutôt que de créer plusieurs copies d'une ressource avec le paramètre `count` car cela permet d'**éviter la recréation accidentelle/indésirable de ressources** lorsqu'une liste d'entrée (ou simplement son ordre) a été modifiée.

Bloc dynamique

L'expression `for_each` offre également la possibilité de **créer des blocs de code dynamique au sein d'une ressource**. Pour vous expliquer son intérêt, laissez-moi vous comparer la syntaxe utilisée sur les versions antérieures et celle utilisée actuellement pour la création de multiples tags d'une ressource autoscaling_group. Sur l'exemple ci-dessous, je vous dévoile la syntaxe utilisée sur Terraform en version 0.11 et ses antérieures versions pour ce type de besoin :

```
resource "aws_autoscaling_group" "instance_sg" {
    # ...

    tag {
      key                 = "Env"
      value               = "prod"
      propagate_at_launch = true
    }

    tag {
      key                 = "App"
      value               = "finance"
      propagate_at_launch = true
    }
```

```
    tag {
        key                 = "Name"
        value               = "devopssec"
        propagate_at_launch = true
    }
}
```

Vous remarquerez deux choses : premièrement que le code reste assez répétitif, et que deuxièmement les blocs imbriqués du paramètre `tag` sont validés statiquement. En effet, il était auparavant difficile d'implémenter des comportements plus dynamiques. Certains utilisateurs ont trouvé des moyens d'exploiter certains détails d'implémentation pour inciter Terraform à prendre en charge partiellement la génération dynamique de blocs, mais ces solutions de contournement n'étaient pas fiables.

Heureusement que Terraform 0.12 a introduit une nouvelle construction de bloc dynamique spécialement conçue pour prendre en charge ces cas d'utilisation de configuration dynamique. Voici donc le même exemple converti sous la syntaxe de la version 0.12 de Terraform. Je vais même d'ailleurs en profiter pour fournir la possibilité à l'utilisateur de définir ses propres tags dynamiquement :

```
variable "AWS_CUSTOM_TAGS" {
    type = "map"
    default = {
        "Env" = "prod"
        "App" = "finance"
        "Name" = "devopssec"
    }
}

resource "aws_autoscaling_group" "AG" {
    # ...

    dynamic "tag" {
        for_each = var.AWS_CUSTOM_TAGS

        content {
            key                 = tag.key
            value               = tag.value
            propagate_at_launch = true
```

```
        }
    }
}
```

> **Information**
>
> Notez que lorsque vous utilisez `for_each` sur une liste, `key` sera l'index de l'élément.

Les Boucles avec des expressions for

Vous avez maintenant vu comment boucler sur les ressources et les blocs en ligne, nous utiliserons cette fois-ci l'**expression `for` pour générer une valeur unique**. La syntaxe de base de cette expression est la suivante :

```
[for <KEY>, <VALUE> in <MAP> : <OUTPUT>]
```

Où **`MAP`** est la variable à parcourir, **`KEY`** et **`VALUE`** sont respectivement la clé et la valeur de chaque élément de votre map et **`OUTPUT`** est le résultat final retourné par chaque itération de la boucle. Voici un exemple ci-dessous qui génère des outputs personnalisés :

```
provider "aws" {
    region = "us-east-1"
}

variable "IAM_USERS" {
    description = "utilisateurs iam avec leur description"
    type        = map(string)
        default     = {
        hatim     = "notre créateur du site devopssec"
        sarah      = "notre cobaye Terraform"
        alex      = "notre voleur des indexs Terraform"
    }
}
```

```
output "users" {
    value = [for name, role in var.IAM_USERS : "${name} est ${role}"]
}
```

En lançant notre code nous obtenons le résultat suivant :

```
terraform init && terraform apply
```

Résultat :

```
users = [
  "alex est notre voleur des indexs Terraform",
  "hatim est notre créateur du site devopssec",
  "sarah est notre cobaye Terraform",
]
```

Le résultat est retourné sous forme de liste, pour le convertir en type map, il suffit d'envelopper l'expression entre des accolades plutôt que des crochets. vous utiliserez ainsi la syntaxe suivante :

```
output "users" {
    value = {for name, role in var.IAM_USERS : "Qui est ${name} ?" => "C'est ${role}"
}
```

```
terraform init && terraform apply
```

Résultat :

```
users = {
    "Qui est alex ?" = "C'est notre voleur des indexs Terraform"
    "Qui est hatim ?" = "C'est notre créateur du site devopssec"
    "Qui est sarah ?" = "C'est notre cobaye Terraform"
}
```

Vous pouvez également **utiliser l'expression `for` sur une liste**, voici la syntaxe à respecter :

```
[for <ITEM> in <LIST> : <OUTPUT>]
```

La syntaxe reste similaire à celle d'une map, sauf que nous ne spécifiant pas de clé cette fois-ci. Par exemple, voici le code Terraform pour convertir les éléments de notre liste en majuscule avec la fonction `upper()` :

```
variable "IAM_USERS" {
    description = "utilisateurs iam"
    type        = list(string)
    default     = ["hatim", "sarah", "alex"]
}

output "users" {
    value = [for name in var.IAM_USERS : upper(name)]
}
```

```
terraform init && terraform apply
```

Résultat :

```
users = [
    "HATIM",
    "sarah",
    "ALEX",
]
```

Les conditions

Création d'une condition

Dans les langages de programmation traditionnels, vous aurez tendance à utiliser l'instruction `if` mais malheureusement Terraform ne prend pas en charge ce type d'instruction. Cependant, vous pouvez accomplir la même chose en profitant des **conditions ternaires**. Pour cela, vous devez respecter la syntaxe suivante :

```
<CONDITION> ? <TRUE_RESULT> : <FALSE_RESULT>
```

Si votre condition est bonne, il exécutera `TRUE_RESULT`, et si la condition est fausse, il exécutera `FALSE_RESULT`. Voici un exemple d'utilisation pour nommer notre instance EC2 selon l'environnement choisi par l'utilisateur :

```
provider "aws" {
    region = "us-east-1"
}
variable "is_prod" {
    type        = list(bool)
    default     = [false, true]
}

resource "aws_instance" "my_ec2_instance" {
    ami = "ami-085925f297f89fce1"
    count = 2
    instance_type = "t2.micro"

    tags = {
        Name = "${var.is_prod[count.index] ? "prod-ec2" : "test-ec2"}"
    }
}

output "key_name" {
    value = aws_instance.my_ec2_instance.*.tags
}
```

Dans cet exemple, j'utilise le paramètre `count` sur une liste afin de fournir la possibilité à l'utilisateur de créer à la fois une instance EC2 pour l'environnement de production et une autre pour l'environnement de test. Enfin, j'utilise ensuite la condition ternaire afin de mieux traiter les conditions de nommage de mes instances EC2. Lançons maintenant la commande d'exécution :

```
terraform init && terraform apply
```

Résultat :

```
Outputs:

key_name = [
  {
    "Name" = "test-ec2"
  },
```

```
  {
    "Name" = "prod-ec2"
  },
]
```

Les opérateurs

Terraform vous offre la possibilité d'**utiliser une multitude d'opérateurs dans vos conditions** que nous étudierons ci-dessous :

Opérateurs d'égalité

Les opérateurs d'égalité vont vous permettre de **comparer deux valeurs** :

Opérateur	Description	Exemple	Résultat
`==`	Compare deux valeurs et vérifie leur égalité	x==4	La condition est bonne si x est égal à 4
`<`	Vérifie qu'une variable est strictement inférieure à une valeur	x<4	La condition est bonne si x est strictement inférieure à 4
`<=`	Vérifie qu'une variable est inférieure ou égale à une valeur	x<=4	La condition est bonne si x est inférieure ou égale à 4
`>`	Vérifie qu'une variable est strictement supérieure à une valeur	x>4	La condition est bonne si x est strictement supérieure à 4
`>=`	Vérifie qu'une variable est supérieure ou égale à une valeur	>=4	La condition est bonne si x est supérieure ou égale à 4
`!=`	Vérifie qu'une variable est différente à une valeur	>=4	La condition est bonne si x est différent à 4

Voici quelques exemples d'utilisation :

```
variable "x" {
    type        = number
    default = 4
}
```

```
output "result" {
    value = [
        var.x == 4,
        var.x  4,
        var.x >= 4,
        var.x != 4
    ]
}
```

```
terraform init && terraform apply
```

Résultat :

```
result = [
  true,
  false,
  true,
  false,
  true,
  false,
]
```

Les opérateurs logiques

Les opérateurs logiques vont vous permettre de **vérifier si plusieurs conditions sont bonnes**. Il existe trois types d'opérateurs logiques :

Opérateur	Signification	Description
`||`	OU	Vrai si au moins une des comparaisons est vraie
`&&`	ET	Vrai si toutes les comparaisons sont vraies
`!`	NON	Retourne faux si la comparaison est vraie et vraie si la comparaison est fausse

Voici quelques exemples d'utilisation :

```
variable "x" {
    type        = number
    default = 1
}
```

```
output "result" {
    value = [
        var.x == 1 && 1 == 1,
        var.x == 1 && 1 == 0,
        var.x == 1 || 1 == 1,
        var.x == 1 || 1 == 0,
        !(var.x == 1),
        !(var.x == 0)
    ]
}
```

```
terraform init && terraform apply
```

Résultat :

```
result = [
    true,
    false,
    true,
    true,
    false,
    true,
  ]
```

Astuce Switch case

Maintenant que vous savez comment faire une instruction if, qu'en est-il de l'instruction switch ? Cette instruction permet de **tester l'égalité d'une variable par rapport à une liste de valeurs** (comme pour le choix d'un menu dans un restaurant). Vous souhaitez par exemple nommer votre instance différemment selon workspace utilisé ? Dans le cas courant, dans un langage de programmation traditionnel vous aurez tendance à utiliser l'instruction `switch case`, malheureusement ceci n'est pas vraiment possible sur Terraform, il faut donc bidouiller et trouver une autre solution. Voici une solution que je vous propose pour répondre à ce besoin, nous utiliserons pour cela les maps :

```
variable "EC2_NAME" {
    default = {
        default = "default-ec2"
        prod = "prod-ec2"
        test = "test-ec2"
    }
}

provider "aws" {
    region = "us-east-1"
}

resource "random_integer" "rand_int" {
    min     = 100000
    max     = 800000
}


resource "aws_instance" "my_ec2_instance" {
    ami = "ami-085925f297f89fce1"
    count = 2
    instance_type = "t2.micro"

    tags = {
        Name = "${random_integer.rand_int.result}-${var.EC2_NAME[terraform.workspace]
    }
}

output "key_name" {
    value = aws_instance.my_ec2_instance.*.tags
}
```

Ici nous utilisons comme condition le nom de notre workspace courant en tant que clé dans notre map `EC2_NAME`. Terraform choisira ensuite la valeur correspondant à cette clé. J'en ai profité également pour utiliser la fonction `random_integer()` afin de générer automatiquement un identifiant aléatoire de type entier. Pour tester notre code, j'ai créé un workspace de type prod :

```
terraform workspace new prod
terraform workspace select prod
terraform init && terraform  apply
```

Résultat :

```
key_name = [
    {
      "Name" = "568825-prod-ec2"
    },
    {
      "Name" = "568825-prod-ec2"
    },
]
```

Opérateurs arithmétiques

Vous avez également la possibilité d'utiliser des opérateurs arithmétiques afin d'**effectuer quelques calculs mathématiques simples**. Ces types d'opérateurs attendent des valeurs numériques et produisent des valeurs numériques comme résultats :

Opérateurs de calcul	Effet
`+`	Additionne deux valeurs
`-`	Soustrait deux valeurs
`*`	Multiplie deux valeurs
`/`	Divise deux valeurs
`%`	Calcul le reste de la division de deux valeurs

Voici un exemple d'utilisation :

```
variable "test" {
    type = number
    default = 4
}

output "result" {
    value = [
        var.test + 2,
        var.test - 2,
        var.test * 2,
        var.test / 2,
        var.test % 2
    ]
}
```

Résultat :

```
result = [
    6,
    2,
    8,
    2,
    0,
]
```

Conclusion

Bien que Terraform soit un langage déclaratif, il comprend un grand nombre d'outils de langage de programmation comme les variables, les conditions et les boucles. Ce qui fournit une quantité surprenante de **flexibilité**. Cependant comme ça reste avant tout un langage déclaratif, il faut certaines fois être créatif pour bidouiller le langage et arriver à ses fins.