UTILISATION DES MODULES SUR TERRAFORM

Introduction

Problèmes rencontrés

En gérant votre infrastructure avec Terraform, vous créerez des configurations de plus en plus complexes. Même s'il n'y a pas de limite intrinsèque à la complexité d'un seul fichier ou répertoire de configuration Terraform, il est possible de continuer à écrire et à mettre à jour vos fichiers de configuration dans un seul répertoire. Cependant, si vous le faite, vous pouvez rencontrer un ou plusieurs problèmes :

- La compréhension et la navigation de votre code Terraform deviendront de plus en plus difficiles.
- La mise à jour de la configuration deviendra plus risquée, car une mise à jour d'une section peut entraîner des conséquences inattendues sur d'autres parties de votre configuration.
- Il y aura une quantité croissante de duplication de blocs de code Terraform, ce qui entraînera une charge supplémentaire lors de la mise à jour de ces parties de votre configuration.
- Si vous partagez des parties de votre configuration entre les projets et les équipes, alors vous constaterez rapidement que copier et coller des blocs de configuration entre les projets est sujet aux erreurs et difficile à maintenir.

Pour résoudre ces problèmes, nous apprendrons donc dans ce guide **comment utiliser les modules sur Terraform** afin de simplifier notre flux de travail.

Qu'est-ce qu'un module Terraform?

Un module est tout simplement une partie de votre code Terraform que vous placez à l'intérieur d'un répertoire et que vous réutilisez à plusieurs endroits dans votre code. Au lieu donc d'avoir le même code copié et collé dans les différents projets, vous pourrez demander à vos différentes équipes de réutiliser le code du même module.

En effet, les modules sont l'ingrédient clé pour écrire du **code Terraform réutilisable, maintenable et testable**. Une fois que vous commencez à les utiliser, vous ne pouvez plus vous en passer. C'est d'ailleurs une bonne pratique de commencer à écrire votre configuration en pensant aux modules, pour les intégrer dans votre bibliothèque de modules qui seront ensuite partagées au sein de votre entreprise.

Le module racine

Comme vu ensemble de fichiers de configuration Terraform dans un dossier est un module. Toutes les configurations que vous avez vues jusqu'à présent dans cette série d'articles de blog sont techniquement des modules, bien que pour le moment elles ne soient pas particulièrement intéressantes en matière de réutilisation dans un autre code Terraform, vous les avez déployés directement. Ce type de module est appelé le "root volume" ou "module racine" en français.

Utilisation des modules Terraform

Structure d'un module

Une **structure de fichiers typique pour un nouveau module Terraform** est la suivante :

```
|___ LICENSE
|___ README.md
|___ main.tf
|___ variables.tf
|___ outputs.tf
```

Aucun de ces fichiers n'est requis ou n'a une signification particulière pour Terraform lorsqu'il utilise votre module. Vous pouvez créer un module avec seulement un seul fichier **.tf** ou utiliser toute autre structure de fichiers que vous souhaitez. Cependant, chacun de ces fichiers sert un objectif :

- **LICENSE** : licence sous laquelle votre module sera distribué. Il informera les personnes qui l'utilisent des conditions dans lesquelles il a été mis à disposition.
- **README.md** : contiendra de la documentation au format markdown décrivant comment utiliser votre module.
- **main.tf** : contiendra le code principal de votre configuration Terraform de votre module.
- **variables.tf** : contiendra les variables de votre module. Lorsque votre module est utilisé par d'autres, les variables seront configurées comme arguments dans le bloc `module` que nous verrons plus tard dans cet article.
- **outputs.tf** : comme son nom l'indique, il contiendra les variables de sortie de votre module. Elles sont souvent utilisées pour transmettre des informations sur les parties de votre infrastructure définies par le module à d'autres parties de votre configuration.

Il existe aussi d'autres fichiers à connaître et **assurez-vous de ne pas les distribuer dans le cadre de votre module** :

- **terraform.tfstate** et **terraform.tfstate.backup** : ces fichiers contiennent votre état Terraform et sont la façon dont Terraform garde une trace de la relation entre votre configuration et l'infrastructure réelle.
- **.terraform** Ce répertoire contient les plugins utilisés pour provisionner votre infrastructure.
- ***.tfvars** Étant donné que les variables d'entrée de module sont définies via des arguments sur le bloc `module` de votre configuration, vous n'avez pas besoin de distribuer votre propre fichier ***.tfvars** avec votre module, sauf si vous l'utilisez également en tant que configuration Terraform autonome.

Donc si jamais vous suivez les modifications apportées à votre module dans un système de contrôle de versions, tel que git, alors n'oubliez pas de configurer votre fichier **.gitignore** en rajoutant les fichiers et dossiers mentionnés ci-dessus. Sans oublier bien sûr de ne pas inclure des informations secrètes telles que des mots de passe ou des clés d'accès .

Les objectifs de notre module

Pour cet article nous allons **concevoir un module** qui permet de **créer un site web statique dans un bucket S3**. En effet, vous pouvez utiliser Amazon S3 pour héberger un site Web statique avec du contenu statique.

Pour héberger un site Web statique sur Amazon S3, il faut :

- Créer un notre bucket S3.

- Activer l'hébergement de sites Web.
- Définir des autorisations.
- Créer et ajouter un document d'index.
- En bonus : créer un fichier d'erreur personnalisé.

Notre module doit donc être capable d'effectuer les tâches listées ci-dessus et doit être réutilisable dans différents code Terraform.

Création de notre module

Création du dossier projet

Commencez par créer un répertoire pour y héberger vos différents modules. Dans mon cas j'ai décidé de le nommer **modules** , et créer dedans un sous-répertoire appelé **aws-s3-static-website-bucket** qui sera le nom de notre module. Pour automatiser cela, vous pouvez lancer la commande suivante :

```
mkdir -p modules/aws-s3-static-website-bucket
```

Création du fichier de licence et documentation

Dans ce dossier **aws-s3-static-website-bucket** , créez un fichier nommé **LICENSE** et un autre fichier **README.md**. À ce moment, vous devriez avoir l'arborescence suivante :

```
modules

|___ aws-s3-static-website-bucket
    |___ LICENSE

|___ README.md
```

Me concernant, j'ai décidé de choisir la licence Apache pour mon fichier **LICENSE** et la documentation suivante en anglais pour mon fichier **README.md** :

```
# AWS S3 static website bucket

This module provisions AWS S3 buckets configured for static website hosting.

## Usage

```hcl
module "<module name>" {
    source = "path of your module"
    bucket_name = "<UNIQ BUCKET NAME>"
    tags = {
        key   = "<value>"
    }
}
```

When your bucket is created, upload an `index.html` file and an `error.html` file in
```

Ajouter une configuration de module

Pour ce module j'ai choisi de travailler avec trois autres fichiers de configuration Terraform à l'intérieur de notre dossier **aws-s3-static-website-bucket** nommés respectivement: **main.tf**, **variables.tf** et **outputs.tf**.

Dans notre fichier **main.tf** nous allons coder le principal code Terraform réalisant les actions inventoriées plus haut, pour cela nous utiliserons la ressource aws_s3_bucket avec les arguments suivants :

- **`bucket`** : nom unique de votre bucket S3.
- **`acl`** : Access control lists (ACL) qui vous permettent de gérer l'accès aux buckets et aux objets (Nous utiliserons les ACLs prédéfinies).
- **`policy`** : contenu json de notre Bucket Policy afin de spécifier les conditions d'actions autorisées ou refusées de votre bucket S3.
- **`website`** : activation de l'hébergement statique avec les arguments suivants :

- `index_document` : fichier d'index html à utiliser lorsque des demandes sont adressées à votre endpoint S3.
- `error_document` : fichier d'erreur html à retourner en cas d'erreur 4XX.

Voici à quoi ressemblera noter fichier **main.tf** :

```
resource "aws_s3_bucket" "s3_bucket" {
    bucket = var.bucket_name
    acl    = "public-read"
    policy = <<EOF
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "PublicReadGetObject",
            "Effect": "Allow",
            "Principal": "*",
            "Action": [
                "s3:GetObject"
            ],
            "Resource": [
                "arn:aws:s3:::${var.bucket_name}/*"
            ]
        }
    ]
}
EOF

    website {
        index_document = "index.html"
        error_document = "error.html"
    }
}
```

Vous remarquerez qu'il n'y a pas de bloc `provider` dans cette configuration. Car lorsque Terraform traite un bloc de module, il hérite automatiquement du provider de la configuration mère qui fait appel à votre module. Pour cette raison, il est recommandé de ne pas l'inclure dans vos modules.

Tout comme le module racine de votre configuration, les modules utilisent aussi des variables, dans notre cas c'est la variable `bucket_name` qui sera définie dans notre

fichier **variables.tf** situé dans le répertoire **aws-s3-static-website-bucket** comme suit :

```
variable "bucket_name" {
    description = "Name of the s3 bucket. Must be unique."
    type = string
}
```

Les variables dans les modules fonctionnent presque exactement de la même manière que pour le module racine. Lorsque vous exécutez une commande d'exécution Terraform sur votre configuration racine, on avait vu qu'il existe différentes façons de définir des valeurs de variable Terraform , telles que leur transmission par la ligne de commande ou depuis un fichier **.tfvars**. Cependant, lorsque vous utilisez un bloc `module`, les variables sont définies en passant des arguments au module dans votre configuration. Nous verrons cette partie un peu plus loins dans notre article.

Nous aurons également besoin d'ajouter quelques valeurs en tant que sortie afin de renvoyer des informations à l'utilisateur final sur les ressources configurées par notre module. Pour ce faire, créez un fichier **outputs.tf** à l'intérieur du dossier **aws-s3-static-website-bucket**. Dans ce fichier nous récupérerons l'attribut `id` qui est le nom de notre bucket S3 et `website_endpoint` qui est le nom DNS utilisé pour accéder directement à notre site Web statique . Ce fichier ressemblera donc à ceci :

```
output "website_endpoint" {
    description = "Domain name of the bucket"
    value       = aws_s3_bucket.s3_bucket.website_endpoint
}

output "name" {
    description = "Name (id) of the bucket"
    value       = aws_s3_bucket.s3_bucket.id
}
```

Nous avons à présent finaliser la création de notre module, il est tant maintenant de passer à la création de notre module racine afin de faire appel à notre module `aws-s3-static-website-bucket`.

Référencer un module

La **syntaxe d'utilisation d'un module** est la suivante :

```
module "<NAME>" {
    source = "<SOURCE>"
    [ARGUMENTS ...]
  }
```

- `NAME` : identifiant que vous pouvez utiliser tout au long du code Terraform pour faire référence à votre module.
- `source` : chemin relatif de votre module (d'autres méthodes seront étudiées à la fin de cet article).
- `ARGUMENTS` : variables d'entrées spécifiques à votre module.

Maintenant que nous avions créé votre module, recréons un fichier **main.tf** dans notre module racine et ajoutons une référence à notre nouveau module :

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

module "website_s3_bucket" {
    source = "./modules/aws-s3-static-website-bucket"
    bucket_name = "devopssec-terraform"
}
```

Ici j'ai décidé de mettre le chemin relatif de notre module dans l'argument `source`. Et pour info les buckets S3 doivent être uniques au monde, c'est donc pour cette raison, que vous devriez choisir un nom unique pour votre bucket S3 pour la valeur

de l'argument `bucket_name`.

Une des dernières étapes à réaliser est d'utiliser les variables de sortie `website_endpoint` et `name` de notre module `aws-s3-static-website-bucket` que l'on ajoutera un fichier nommé lui aussi **outputs.tf** dans le répertoire de notre module racine :

```
output "website_bucket_name" {
    description = "Name (id) of the bucket"
    value       = module.website_s3_bucket.name
}

output "website_endpoint" {
    description = "Domain name of the bucket"
    value       = module.website_s3_bucket.website_endpoint
}
```

Jusqu'à présent, nous sommes dorénavant capables de créer notre site web statique sur notre bucket S3. Et Histoire de ne pas se perdre, notre architecture finale ressemblera à ceci :

```
|___ aws-s3-static-website-bucket
|   |___ modules
|   |
|___ aws-s3-static-website-bucket
|   |       |___ LICENSE
|   |       |___ main.tf
|   |       |___ outputs.tf
|   |       |___ README.md
|   |
|___ variables.tf
|   |___ main.tf

|___
|___ outputs.tf
```

Execution de notre module

Vous pouvez télécharger le projet complet en cliquant ici, j'en ai également profité pour vous ajouter un fichier **index.html** et **error.html** dans le dossier `www` afin de

simuler et tester votre page web statique.

Maintenant que tout est prêt, exécutons notre code terraform en vous plaçant sur votre module racine et en lançant la commande suivante :

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

Résultat :

```
Enter a value: yes

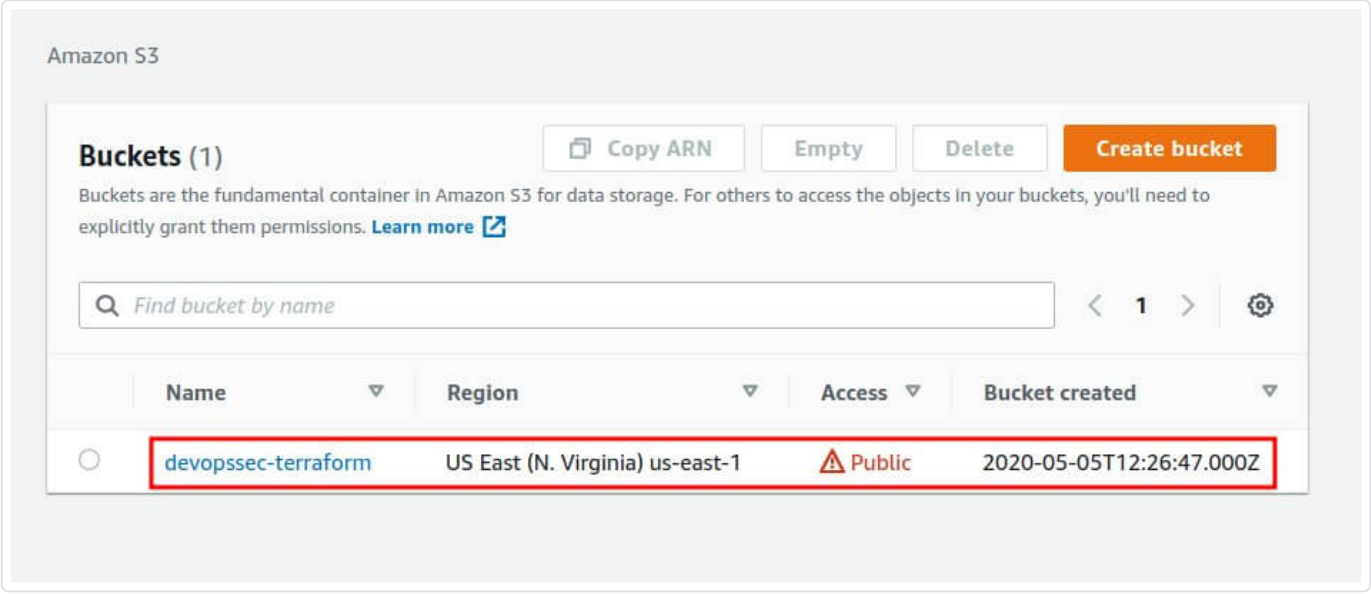
module.website_s3_bucket.aws_s3_bucket.s3_bucket: Creating...
module.website_s3_bucket.aws_s3_bucket.s3_bucket: Still creating... [10s elapsed]
module.website_s3_bucket.aws_s3_bucket.s3_bucket: Creation complete after 12s [id=dev

Apply complete! Resources: 1 added, 0 changed, 0 destroyed.

Outputs:

website_bucket_name = devopssec-terraform
website_endpoint = devopssec-terraform.s3-website-us-east-1.amazonaws.com
```

Après avoir répondu avec un "yes", votre bucket sera provisionné. Vous pouvez vérifier cela en vous rendant sur votre console S3 :



Vous avez maintenant **configuré et utilisé votre propre module** pour créer un site Web statique. Vous voudrez peut-être visiter votre site Web statique ? À l'heure actuelle,

il n'y a rien dans votre bucket, donc il n'y aurait rien à voir si vous tentez de visiter le site Web de votre bucket. Pour voir tout contenu, vous devrez **télécharger des objets dans votre bucket S3**. Dans les sources je vous ai fourni un répertoire `www` contient les sources de notre site web statique, nous téléchargerons son contenu à l'aide de la CLI AWS comme suit :

```
aws s3 cp www/ s3://$(terraform output -raw website_bucket_name)/ --recursive
```

Résultat :

```
upload: www/error.html to s3://devopssec-terraform/error.html
upload: www/index.html to s3://devopssec-terraform/index.html
```

Information

L'option `-raw` permet de récupérer la valeur de sortie sans aucun formatage spécial (Il va par exemple éviter de rajouter des guillemets par-dessus votre sortie).

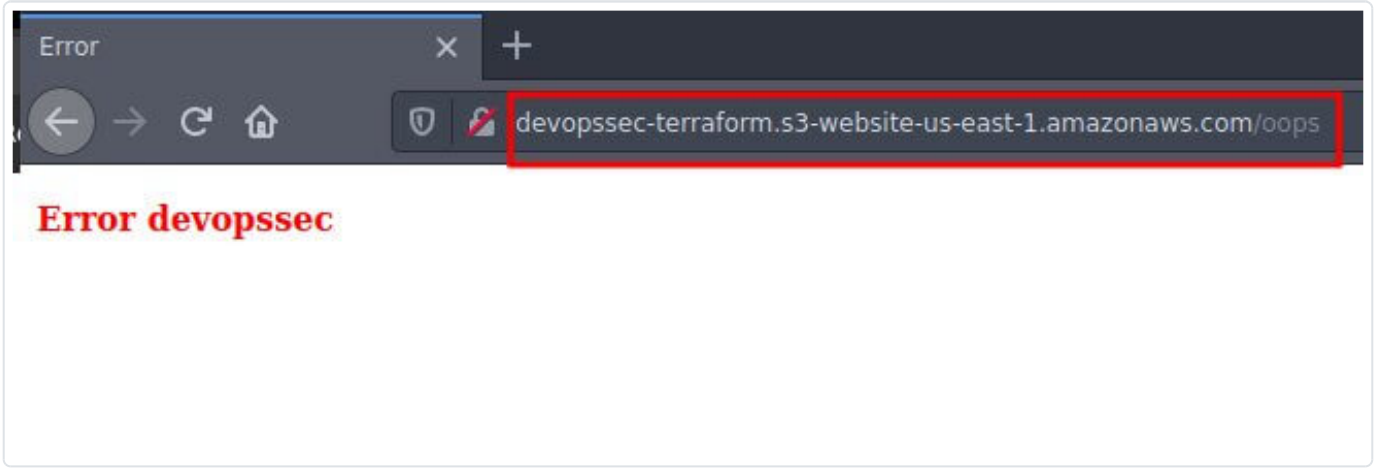
Enfin, récupérerons notre endpoint S3 pour l'utiliser en tant qu'url dans notre navigateur afin d'accèder enfin à notre site web statique :

```
terraform output -raw website_endpoint
```

Résultat :

```
devopssec-terraform.s3-website-us-east-1.amazonaws.com
```

Maintenant, ouvrez votre navigateur et utilisez ce endpoint en tant qu'url :



Nettoyer l'infrastructure

Pour **supprimer votre infrastructure créée par votre module** précédemment, vous devrez en amont détruire tous les objets disponibles dans votre bucket S3. Nous utiliserons une nouvelle fois la CLI AWS :

```
aws s3 rm s3://$(terraform output -raw website_bucket_name)/ --recursive
```

Résultat :

```
delete: s3://devopssec-terraform/error.html
delete: s3://devopssec-terraform/index.html
```

Une fois votre bucket vide, détruisez vos ressources Terraform à l'aide de la commande suivante :

```
terraform destroy
```

Résultat :

```
Do you really want to destroy all resources?
Terraform will destroy all your managed infrastructure, as shown above.
There is no undo. Only 'yes' will be accepted to confirm.

Enter a value: yes

module.website_s3_bucket.aws_s3_bucket.s3_bucket: Destroying... [id=devopssec-terrafo
module.website_s3_bucket.aws_s3_bucket.s3_bucket: Destruction complete after 1s

Destroy complete! Resources: 1 destroyed.
```

Après avoir répondu à l'invite avec yes, Terraform détruira toutes les ressources créées en suivant ce guide.

Partager vos modules sur des registres Distant

L'argument `source` du bloc `module` indique à Terraform l'emplacement du code source du module souhaité. Terraform l'utilise lors de l'étape d'initialisation de la commande `terraform init` ou `terraform get` pour télécharger le code source dans un répertoire sur le disque local afin qu'il puisse être utilisé par d'autres commandes Terraform. Le programme d'installation du module prend en charge l'installation à partir d'un certain nombre de **types de sources différents**, Nous allons voir ci-dessous les plus populaires d'entre eux :

Source local

C'est la source que nous avons utilisée pour notre exemple S3. Le chemin local doit commencer par `./` ou `../` pour indiquer qu'un chemin relatif :

```
module "mon_module" {
    source = "./chemin"
}
```

Les chemins locaux sont spéciaux car ils ne sont pas installés au même titre que les autres sources, puisque les fichiers sont déjà présents sur votre disque local et

peuvent donc être utilisés directement. Leur code source est automatiquement mis à jour si le module parent est mis à niveau.

Source GitHub

Pour télécharger un module distant depuis votre repository Github, utilisez la source suivante :

```
module "mon_module" {
    source = "github.com/hajdaini/example"
}
```

Le schéma d'adresse ci-dessus sera cloné avec le protocole HTTPS. Pour le cloner avec le protocole SSH, utilisez la syntaxe suivante :

```
module "mon_module" {
    source = "git@github.com:hajdaini/example.git"
}
```

Source Bucket S3

Vous pouvez utiliser des archives zip stockées dans S3 comme sources de module à l'aide du préfixe spécial `s3::`, suivi d'une URL d'objet de votre bucket S3 :

```
module "mon_module" {
    source = "s3::https://s3-eu-west-1.amazonaws.com/devopssec-terraform/modules/aws-
}
```

L'objet résultant doit être une archive sous les extensions suivantes :

- `zip`
- `tar.bz2` et `tbz2`
- `tar.gz` et `tgz`

- `tar.xz` et `txz`

Terraform va ensuite extraire l'archive pour obtenir l'arborescence source du module.

Registre officiel Terraform

Vous pouvez également **utiliser le registre cloud officiel de Terraform** qui est un index des modules partagés publiquement par la communauté Terraform. Ce registre public est le moyen le plus simple de démarrer avec Terraform et de trouver des modules créés par d'autres membres de la communauté.

Les modules du registre Terraform public peuvent être référencés à l'aide d'une adresse source de registre que vous retrouverez sur la page d'information de chaque module sur le site du registre comprenant l'adresse exacte à utiliser, par exemple pour le registre aws consul ça sera la source suivante :

```
module "consul" {
    source  = "hashicorp/consul/aws"
    version = "0.7.4"
    # etc ...
}
```

Vous pouvez d'ailleurs télécharger et partager avec toute la communautés Terraform vos propres modules dans le registre public. Pour utiliser un registre privé , vous trouverez plus d'informations sur la documentation des registres privés.

Conclusion

En définissant des modules dans votre Infrastructure As Code, vous pouvez **appliquer diverses bonnes pratiques de codage à votre infrastructure**. Vous passez votre temps à développer des modules plutôt qu'à déployer manuellement du

code. Tout cela peut considérablement **augmenter votre capacité à construire une infrastructure rapidement** et de manière fiable. Vous pouvez encapsuler des éléments d'infrastructure compliqués derrière de simples modules qui peuvent être réutilisés dans toute votre pile technologique.