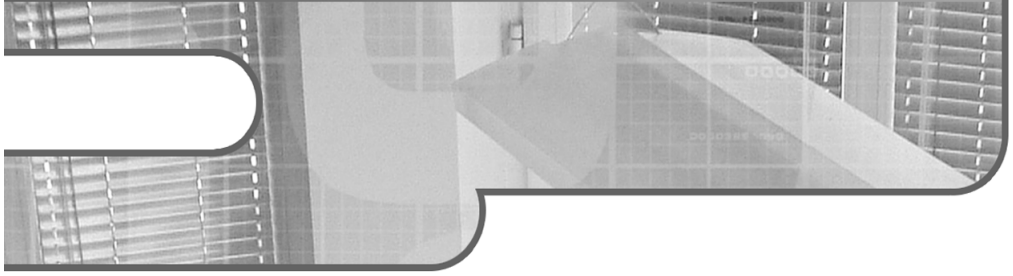

Chapitre 1-3
Écrire du code Terraform

1. Le langage déclaratif HCL

HCL, pour *HashiCorp Configuration Language*, est le langage dédié pour décrire les ressources manipulées par Terraform.

Ce langage se veut déclaratif. Plutôt que d'utiliser des commandes de manière séquentielle, comme « Créer une base de données » ou « Attacher une adresse IP », Terraform propose aux utilisateurs de décrire l'état de l'infrastructure souhaité.

Sa structure est proche de JSON ou YAML. Il est d'ailleurs possible d'écrire du code Terraform en JSON. Cependant, HCL est moins verbeux, et plus simple d'écriture que JSON.

Pour permettre aux utilisateurs de décrire leur infrastructure, Terraform propose deux types de blocs principaux. Les blocs `resource` permettent de décrire un objet de cloud ou d'API qui devra être créé et maintenu par Terraform. Une ressource sera par exemple une machine virtuelle ou une base de données sur un cloud public, un groupe Active Directory ou LDAP, un enregistrement DNS, etc.

Voici un exemple d'un bloc `resource` déclarant une base de données PostgreSQL sur le cloud public Scaleway :

```
resource "scaleway_rdb_instance" "main" {
  name = "sw-db"
  node_type = "DB-DEV-S"
  engine = "PostgreSQL-15"
  is_ha_cluster = true
  user_name = "yoda"
  password = "do_or_do_not"
}
```

Les blocs `data` permettent de référencer un objet déjà existant dans une infrastructure, et d'utiliser ses informations. Cet objet peut être géré par Terraform par ailleurs (dans un autre code, géré par une autre équipe), ou avoir été créé manuellement sur le cloud ou l'API. Une data sera par exemple une IP publique déjà existante, un groupe LDAP dans lequel nous souhaitons ajouter un utilisateur, une zone DNS existante, etc. En fonction des providers, nous accéderons à une data à partir de son identifiant, son nom, ou d'éléments permettant de l'identifier de manière unique.

Voici un exemple de bloc `data` qui requête une base de données existante, toujours sur le cloud Scaleway :

```
data "scaleway_rdb_instance" "instance" {
  name = "sw-rdb"
}
```

Remarque

Pour résumer, les blocs `data` sont les objets qui sont déjà existants et que l'on souhaite requêter, les blocs `resource` sont les objets que l'on souhaite créer.

La plupart des providers fournissent des blocs `data` et des blocs `resource` pour le même type d'objet, afin de pouvoir les créer, les modifier et les détruire, mais aussi simplement les requêter.

Au-delà de la définition des objets en blocs, le langage permet également de déclarer des variables d'entrée, de sortie et locales, et propose une gestion de types complexes, tuples, listes, dictionnaires, ainsi que des fonctions de manipulation de ces types.

2. Les fichiers .tf

Le code Terraform doit être écrit dans des fichiers dont l'extension est `.tf`. Voici un exemple d'arborescence de fichiers contenant du code Terraform :

```
.
├── .gitignore
├── main.tf
├── outputs.tf
├── provider.tf
├── README.md
└── variables.tf
```

Ce code contient plusieurs fichiers en plus des fichiers `.tf`. Le fichier `README.md` est un fichier de documentation à destination des utilisateurs du code et des développeurs. Le fichier `.gitignore` est un fichier de configuration interprété par Git. Un exemple d'un tel fichier est décrit dans le chapitre Outils externes - .gitignore.

Les autres fichiers ont l'extension `.tf` et contiennent donc du code Terraform. Par convention, le fichier `main.tf` est considéré comme le point d'entrée du code. Le fichier `provider.tf` contient la configuration des providers (cf. section Le bloc provider de ce chapitre). Les fichiers `variables.tf` et `outputs.tf` sont consacrés aux blocs portant le même nom.

Voici un autre exemple d'arborescence de fichiers de code Terraform :

```
.
├── .gitignore
├── databases.tf
├── instances.tf
├── main.tf
├── networks.tf
├── outputs.tf
├── provider.tf
├── README.md
└── variables.tf
```

Dans cet exemple de fichiers, le code a été découpé dans plusieurs fichiers `.tf`. Le fichier `databases.tf` contient les blocs `resource` et `data` permettant de créer une base de données. Le fichier `instance.tf` contient le code permettant de créer des instances de machines virtuelles. Le fichier `networks.tf` déclare le code de la partie réseau.

Remarque

Organisez le code comme il vous semble le plus pertinent. En général, si votre code ne contient que quelques ressources, un seul fichier `main.tf` est suffisant. Si votre code contient de nombreuses ressources, découpez-le en plusieurs fichiers, un par élément important de votre infrastructure.

3. Les blocs resource et data

3.1 Le bloc resource

Les blocs `resource`, qui permettent de créer et manipuler des objets avec Terraform, ont la structure suivante :

```
resource "<TYPE>" "<NOM>" {
  <ARGUMENT> = <EXPRESSION>
}
```

Le mot-clé `resource` est suivi du `<TYPE>` qui correspond au type d'objet que l'on souhaite instancier. Ces types sont définis dans les providers Terraform de chaque service cloud. Par exemple, le provider *Scaleway* définit des ressources comme `"scaleway_rdb_instance"` ou `"scaleway_container"`. Le provider AWS définit des ressources comme `"aws_instance"` ou `"aws_vpc"`. Le provider OVH définit des ressources comme `"ovh_domain_zone"` ou `"ovh_cloud_project_database"`. Le `"<NOM>"` est celui que l'on souhaite donner à sa ressource dans le code Terraform. Ce nom sert à référencer la ressource par ailleurs. Attention à bien distinguer ce nom associé au code, au nom réel de l'objet qui sera créé sur le cloud, qui lui sera défini le plus souvent par un argument.

Les "<TYPE>" et "<NOM>" doivent bien être déclarés sous la forme de chaîne de caractères en utilisant les guillemets doubles : ". Le contenu du bloc `resource` est défini entre accolades `{}`, et contient un ensemble de couples argument/valeur. Les arguments sont en quelque sorte les paramètres que l'on va pouvoir définir quand on souhaite créer une ressource. Pour une machine virtuelle, les arguments que l'on retrouve souvent sont sa localisation, le type d'instance, et un nom par exemple. Pour une base de données, les arguments courants sont similaires à une machine virtuelle, en y ajoutant le moteur de base de données souhaité. En fonction des ressources, les arguments acceptés diffèrent et certains peuvent être optionnels. Il faut se reporter à la documentation des providers pour chaque type de ressource pour connaître le nom des arguments attendus.

L'exemple suivant définit une ressource de type "scaleway_rdb_instance", qui est une base de données. Cette ressource est nommée "main" dans le code. Elle définit également plusieurs arguments, par exemple le nom de l'instance de base de données qui sera créée sur le cloud est "sw-db". Les autres arguments définissent le type de machine, ainsi que sa configuration :

```
resource "scaleway_rdb_instance" "main" {
  name = "sw-db"
  node_type = "DB-DEV-S"
  engine = "PostgreSQL-15"
  is_ha_cluster = true

}
```

Une ressource Terraform définit également des attributs. Les attributs d'une ressource sont des valeurs qui sont accessibles dans le code en lecture seule, et ne peuvent pas être passés en paramètre. Les attributs sont souvent issus de la construction des objets sur les clouds ou API appelées par les providers. Les exemples les plus parlants sont l'adresse IP d'une machine virtuelle ou d'une base de données, ou encore l'identifiant interne d'une machine virtuelle sur un cloud.

Dans la documentation des ressources Terraform, les arguments et les attributs sont toujours bien séparés dans des sections dédiées.

Dans le code, on accède aux attributs ou aux arguments d'une ressource à travers une expression `<TYPE>.<NOM>.<ATTRIBUT>` (cf. section Les expressions).

3.2 Le bloc data

Les blocs `data`, qui permettent de requêter des objets existants avec Terraform, ont la structure suivante :

```
data "<TYPE>" "<NOM>" {
  <ARGUMENT> = <EXPRESSION>
}
```

La structure est similaire en tout point aux blocs `resource`. Le mot-clé `data` est suivi du `<TYPE>` d'objet requêté, ainsi que du nom associé au bloc. Ce nom servira également pour référencer la data par ailleurs dans le code. Comme pour un bloc `resource`, un bloc `data` définit son contenu entre accolades `{}`, et contient un ensemble de couples argument/valeur.

L'exemple suivant définit un bloc `data`, nommé `my_instance`, de type `scaleway_rdb_instance`, avec pour argument un nom valant `test-rdb` :

```
data "scaleway_rdb_instance" "instance" {
  name = "sw-rdb"
}
```

Ce bloc `data` permet de requêter la base de données qui peut être créée avec le bloc `resource` vu dans la section précédente.

Comme pour un bloc `resource`, un bloc `data` expose également des attributs. Nous accéderons aux attributs d'un bloc `data` à travers une expression `data.<TYPE>.<NOM>.<ATTRIBUT>`.

Remarque

En général, il est préférable d'éviter d'isoler les blocs `data` dans un fichier `data.tf`. Essayez de conserver les blocs `data` au plus proche de l'endroit où ils sont utilisés. Les blocs `data` sont souvent juste au-dessus des blocs `resource` qui les utilisent.

4. Les blocs variable, locals et output

Terraform permet aussi de définir des entrées et sorties dans le code, pour rendre le code réutilisable. Les entrées sont définies par des blocs `variable`, et les sorties par des blocs `output`.

Pour rendre le code plus lisible, il est également possible de définir des variables locales, qui pourront être réutilisées dans l'ensemble du code et qui sont définies par un bloc `locals`.

4.1 Le bloc variable

Les blocs `variable` permettent d'introduire de la réutilisabilité dans le code Terraform. Le code pourra déclarer un ensemble de variables, avec une valeur par défaut si besoin. À l'exécution, l'utilisateur devra fournir la valeur pour chacune des variables n'ayant pas de valeur par défaut (cf. chapitre Architecture et CLI Terraform - Le workflow et les commandes principales).

Voici un exemple de code déclarant une variable. La variable est utilisée dans la déclaration d'une ressource :

```
variable "instance_type" {
  description = "type de machine"
  type = string
  default = "DB-DEV-S"
}

resource "scaleway_rdb_instance" "main" {
  node_type = var.instance_type
  engine = "PostgreSQL-11"
}
```

Remarque

Ces deux blocs devraient être dans des fichiers différents ! Les variables sont toujours déclarées dans un fichier nommé `variables.tf`. Mais certains exemples de cet ouvrage listeront des blocs `variable` et des blocs `resource` ensemble pour rendre la lecture des exemples plus cohérente.

Les définitions de variable ont la structure suivante :

```
variable "<NOM>" {
  description = "<STRING>"
  type = <TYPE>
  default = <EXPRESSION>
}
```

Le `"<NOM>"` permet de nommer la variable, et doit être unique pour toutes les variables dans l'ensemble du code Terraform. Le `<TYPE>` peut être `string`, `number` ou `bool` pour les types simples, mais aussi `list(<TYPE>)` ou `map(<TYPE>)` pour des variables plus complexes.

Les arguments `description` et `default` sont optionnels. Cependant, il est recommandé de toujours inclure l'argument description. Cet argument peut également être utilisé par d'autres outils, pour générer de la documentation par exemple.

Remarque

Les variables sont les premiers points d'entrée de l'utilisation de votre code Terraform. Soignez leur description en indiquant quel est leur but et quelles sont les valeurs attendues.

Comme illustré dans l'exemple, on accède à une variable d'entrée avec la syntaxe `var.<NOM>`.

Par convention, l'ensemble des variables est souvent rassemblé dans un unique fichier que l'on nomme `variables.tf`.

Remarque

Retenez bien cette convention, elle sera réutilisée dans le chapitre Utiliser et développer des modules.