Aurélie Vache
Développeuse chez Continental Intelligent Transportation Systems
Duchess France & DevFest Toulouse Leader
Toulouse Data Science core-team & Marraine Elles bougent
@aurelievache

Infrastructure as Code avec Terraform

Vous ne voulez plus créer manuellement vos machines physiques, vos VMs, vos VPC, vos conteneurs, vos lambda … ? Nous allons voir dans cet article qu'il existe un outil permettant de faire de l'Infrastructure as Code qui vous permettra de passer du ClickOps au DevOps.

Nous commencerons par voir ce qu'est Terraform, les concepts, sa CLI. Nous ferons une mise en pratique de l'outil, puis nous verrons quelques tips et outils sympa. Nous ferons aussi une comparaison avec CloudFormation.

Qu'est ce que Terraform ?

Terraform est un outil créé en 2014 par HashiCorp, la société qui a créé d'autres outils que vous connaissez sûrement : Consul, Vagrant, Vault, Atlas, Packer et Nomad.

C'est un outil Open Source, écrit en Go, avec une communauté active de plus de 1000 contributeurs qui repose sur une architecture basée sur les plugins.

Terraform (TF) est un outil qui permet de **construire**, **modifier** et **versionner** une infrastructure. Contrairement à ce que l'on peut lire sur internet, la technologie n'est pas plateforme agnostic MAIS elle permet d'utiliser plusieurs providers dans un même template de configuration. Il existe en effet des plugins pour des providers de Cloud, des services d'hébergement, des SCM … Nous le verrons un peu plus tard dans cet article.

Que fait l'outil ?

- Il assure la création et la cohérence d'infrastructure ;
- Il permet d'appliquer des modifications incrémentales ;
- On peut détruire des ressources si besoin ;
- On peut prévisualiser les modifications avant de les appliquer.

HCL

Les fichiers de configurations s'écrivent en HCL (HashiCorp Configuration Language). Le principe est d'écrire des ressources.

Les ressources peuvent être écrites en JSON également mais il est recommandé de les écrire en HCL.

Lire un fichier de configuration HCL est plutôt simple et intuitif. 1

C'est un langage dit "human readable". Ce qu'il faut savoir c'est que Terraform scanne tous les fichiers se terminant par .tf dans le répertoire courant; en revanche, il ne va pas scanner les répertoires enfants.

CONCEPTS

On va voir ensemble quelques concepts :

Provider

Un provider est responsable du cycle de vie/du CRUD d'une ressource : sa création, sa lecture, sa mise à jour et sa suppression.

Plusieurs providers sont actuellement supportés :

- AWS, GCP, Azure, OpenStack …
- Heroku, OVH, 1&1 …
- Consul, Chef, Vault …
- Docker, Kubernetes …
- GitLab, BitBucket, GitHub …
- MySQL, PostgreSQL …
- mais encore RabbitMQ, DNSimple, CloudFlare …

La liste complète des providers est disponible sur le site de TF : https://www.terraform.io/docs/providers/index.html

Vous ne trouvez pas le provider de votre rêve ? Pas de souci vous pouvez écrire votre propre plugin et participer à l'élaboration de nouvelles fonctionnalités de Terraform. Si vous ne codez pas encore en Go, ce sera l'occasion de vous y mettre ;-).

Pour configurer un provider, nous voulons par exemple déployer une infra AWS, il suffit de créer une ressource "aws" dans un fichier se terminant par .tf :

root.tf :

```
provider "aws" {
  region = "eu-central-1"
}
```

Bonne pratique : ne pas mettre les credentials directement dans la ressource provider aws mais configurez vos variables d'environnement :

```
$ export AWS_ACCESS_KEY=YOUR_ACCESS_KEY
$ export AWS_SECRET_ACCESS_KEY=YOUR_SECRET_KEY
$ export AWS_DEFAULT_REGION=eu-central-1
```

Ou bien vous pouvez les mettre dans un fichier .aws/credentials :

```
[default]
aws_access_key_id = YOUR_ACCESS_KEY
aws_secret_access_key = YOUR_SECRET_KEY
```

Attention, il va de soi que vous devrez prendre les tokens d'un utilisateur qui aura les droits/la bonne policy pour créer les services que vous voulez.

1

Variables

Afin de faire du code propre et qui se réutilise, il est recommandé d'initialiser des variables et de les utiliser dans les autres fichiers .tf.

Définissez vos variables dans un fichier variables.tf, par exemple :

```
variable "default-aws-region" {
  default = "eu-central-1"
}

variable "aws_account_id" {
  default = "123456789123"
}

variable "tags-tool" {
  default = "Terraform"
}

variable "tags-contact" {
  default = "Aurelie Vache"
}

variable "aws_s3_folder" {
  default = "s3"
}

variable "aws_s3_bucket_terraform" {
  default = "com.programmez.terraform"
}
```

Appelez-les dans vos ressources :

```
resource "aws_s3_bucket" "com-programmez-terraform" {
  bucket = "${var.aws_s3_bucket_terraform}"
  acl    = "private"

  tags {
    Tool    = "${var.tags-tool}"
    Contact = "${var.tags-contact}"
  }
}
```

Modules

Les modules sont utilisés pour créer des composants réutilisables, améliorer l'organisation et traiter les éléments de l'infrastructure comme une boîte noire.

C'est un groupe de ressources qui prennent en entrée des paramètres et retournent en sortie des outputs.

Dans le même fichier, root.tf, dans lequel vous avez défini votre provider, vous pouvez ensuite définir vos modules :

```
module "lambda" {
  source         = "./lambda/"
  toto           = "${var.aws_s3_bucket_toto_lambda_key}"
  titi           = "${var.aws_s3_bucket_titi_key}"
  tutu           = "${var.aws_s3_bucket_tutu_key}"
}
```

Outputs

Les modules peuvent produire des outputs que l'on pourra utiliser dans d'autres ressources.

lambda/outputs.tf :

```
output "authorizer_uri" {
  value = "${aws_lambda_function.lambda_toto.invoke_arn}"
}
```

aws_api_gw.tf :

```
resource "aws_api_gateway_authorizer" "custom_authorizer" {
  name                          = "CustomAuthorizer"
  rest_api_id                   = "${aws_api_gateway_rest_api.toto_api.id}"
  authorizer_uri                = "${module.lambda.authorizer_uri}"
  identity_validation_expression = "Bearer .*"
  authorizer_result_ttl_in_seconds = "30"
}
```

State

Un state est un snapshot de votre infrastructure depuis la dernière fois que vous avez exécuté la commande **terraform apply.** 2

Terraform utilise un stockage local pour créer les plans et effectuer les changements sur votre infra. Mais il est possible de stocker ce state, dans le cloud.

Pour configurer le stockage de ce state en remote, il suffit de définir un backend.

backend.tf :

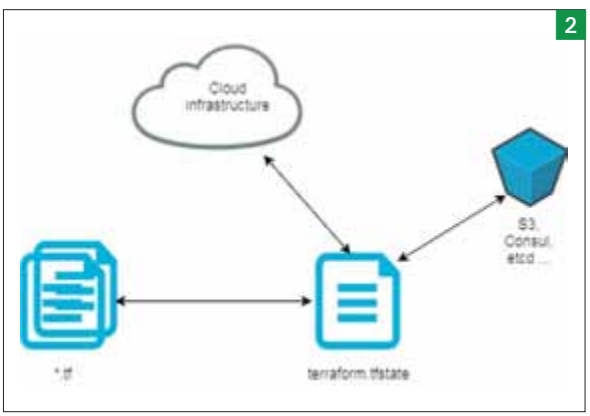
```
# Backend configuration is loaded early so we can't use variables

terraform {
  backend "s3" {
    region  = "eu-central-1"
    bucket  = "com.programmez.terraform"
    key     = "state.tfstate"
    encrypt = true
  }
}
```

CLI

Terraform a une CLI (Command-Line Interface) facile à utiliser composée de plusieurs commandes, nous allons en voir quelques-unes, pas toutes, uniquement celles que j'utilise le plus au quotidien.

2



Init

```
$ terraform init
```

Cette commande va initialiser votre répertoire de travail qui contient vos fichiers de configuration en .tf.
C'est la première commande à exécuter pour une nouvelle configuration, ou après avoir fait un checkout d'une configuration existante depuis votre repo git par exemple.
La commande init va :

- télécharger et installer les providers ;
- initialiser le backend (si défini) ;
- télécharger et installer les modules (si définis).

Plan

```
$ terraform plan
```

La commande plan permet de créer un plan d'exécution. Terraform va déterminer quelles actions il doit faire afin d'avoir les ressources listées dans les fichiers de configuration par rapport à ce qui est actuellement en place sur l'environnement/le provider cible.
Cette commande n'effectue concrètement rien sur votre infra.
Bonne pratique : afin de sauver le plan, vous pouvez spécifier un fichier de sortie :

```
$ terraform plan -out programmez.out
```

Apply

```
$ terraform apply programmez.out
```

La commande apply, comme son nom l'indique, permet d'appliquer les changements à effectuer sur l'infra. C'est cette commande qui va créer nos ressources.
Attention, depuis la version 0.11 de Terraform, sortie le 16 novembre dernier, lorsque vous utilisez TF dans un environnement interactif, en local par exemple, mais pas en CI/CD, il est recommandé de ne plus passer par un plan mais de directement faire un apply et de répondre Yes si vous souhaitez appliquer ce plan.

Destroy

```
$ terraform destroy
```

La commande destroy permet de supprimer TOUTES les ressources. Un plan de suppression peut être généré au préalable :

```
$ terraform plan -destroy
```

Console

Grâce à la CLI console, vous pouvez connaître la valeur d'une ressource Terraform. Cette commande est pratique pour faire du debug, avant de créer un plan ou de l'appliquer.

```
$ echo "aws_iam_user.programmez.arn" | terraform console
arn:aws:iam::123456789123:user/programmez
```

Get

```
$ terraform get
```

Cette commande est utile si par le passé vous avez déjà fait un terraform init puis ajouté un module, il faut préciser maintenant à Terraform qu'il faut récupérer le module ou bien le mettre à jour. Si vous ne le faites pas, lors de d'un Terraform plan, TF vous demandera de le faire ;-).

Graph

```
$ terraform graph | dot -Tpng > graph.png
```

La CLI graph permet de dessiner un graphique de dépendances visuelles des ressources Terraform en fonction des fichiers de configuration. 3
Au bout d'un certain nombre de ressources dans un répertoire, Terraform n'arrive plus à générer ce graphique. J'espère que ce problème sera corrigé dans les futures version ;-).

TRÈVE DE BLABLA, PASSONS À LA PRATIQUE !

Commençons tout d'abord par installer Terraform :

```
$ curl -O https://releases.hashicorp.com/terraform/0.11.1/terraform_0.11.1_linux_amd64.zip
$ sudo unzip terraform_0.11.1_linux_amd64.zip -d /usr/local/bin/
$ rm terraform_0.11.1_linux_amd64.zip
```

0.11.1 étant la dernière version de Terraform, stable, à ce jour.
Ces commandes vont extraire un binaire dans /usr/local/bin/, qui est déjà dans votre PATH.
Afin de vérifier que Terraform est correctement installé, veuillez vérifier la version courante de l'outil :

```
$ terraform --version
Terraform v0.11.1
```

Info: terraform -version fonctionne également.

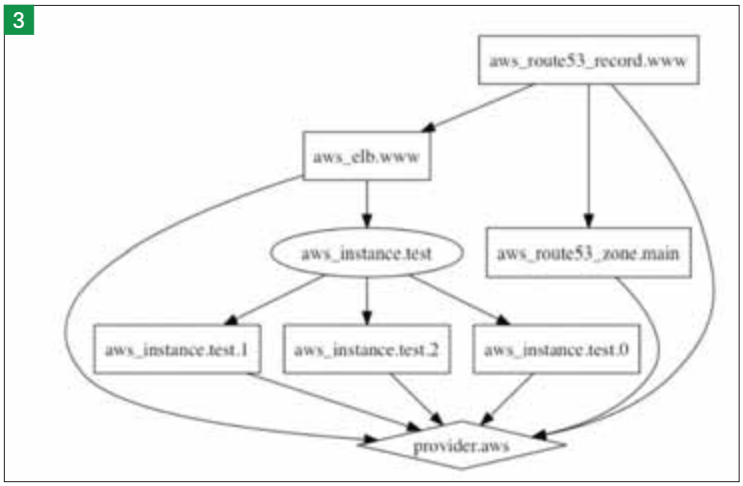
1/ Création d'un répertoire

```
$ mkdir programmez/
```

2/ Création d'un fichier .tf

```
$ vi aws_s3.tf
```

3



```
############## AWS S3 ##############
resource "aws_s3_bucket" "com-programmez-terraform" {
bucket = "${var.aws_s3_bucket_terraform}"
acl = "private"
  tags {
    Tool = "${var.tags-tool}"
    Contact = "${var.tags-contact}"
  }
}
```

3/ Pré-requis

Pour indiquer à Terraform sur quel compte AWS vous souhaitez déployer l'infrastructure souhaitée, vous devez définir des variables d'environnement AWS au préalable, par exemple dans un fichier .aws/credentials ou avec des variables d'environnement :

```
$ export AWS_ACCESS_KEY_ID="an_aws_access_key"
$ export AWS_SECRET_ACCESS_KEY="a_aws_secret_key"
$ export AWS_DEFAULT_REGION="a-region"
```

4/ Initialiser Terraform

```
$ terraform init
[0m[1mDownloading modules...[0m
[0m[1mInitializing the backend...[0m
[0m[1mInitializing provider plugins...[0m
The following providers do not have any version constraints in configuration,
so the latest version was installed.
To prevent automatic upgrades to new major versions that may contain breaking
changes, it is recommended to add version = "..." constraints to the
corresponding provider blocks in configuration, with the constraint strings
suggested below.
* provider.aws: version = "~> 1.6"
```

5/ Plan

```
$ terraform plan -out crf.out
Refreshing Terraform state in-memory prior to plan...
The refreshed state will be used to calculate this plan, but will not be
persisted to local or remote state storage.
aws_s3_bucket.com-programmez-terraform: Refreshing state... (ID: com.
programmez.terraform)
...
Plan: 1 to add, 0 to change, 0 to destroy.
```

6/ On applique le plan

```
$ terraform apply plan.out
...
Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
```

OUTILS & TIPS

Terraforming

Afin de nous faciliter la vie, il y a quelques petits outils intéressants à connaître et à utiliser. Si vous avez une infra AWS existante et que vous devez la dupliquer dans plusieurs autres comptes, **Terraforming** (https://github.com/dtan4/terraforming) est fait pour vous. C'est un outil écrit en Ruby qui permet d'extraire des ressources AWS existantes et de générer un fichier Terraform correspondant.

Pour l'installer, voici la marche à suivre :

```
$ sudo apt install ruby
$ gem install terraforming
```

Pour extraire un type de ressources c'est très simple. Par exemple si l'on veut extraire des buckets s3 :

```
$ terraforming s3 > aws_s3.tf
```

Comme vous pouvez le constater si vous testez cette commande sur un compte AWS qui contient des buckets s3, cette dernière n'extrait que les buckets, elle n'extrait pas les objets : les zip, jar et divers fichiers contenus dans vos buckets. Il vous faudra du coup écrire les ressources `aws_s3_object` comme celle-ci :

```
resource "aws_s3_bucket_object" "authorizer_keystore" {
 bucket = "${var.aws_s3_bucket_authorizer}"
 key    = "${var.aws_s3_bucket_authorizer_keystore_key}"
 source = "${var.aws_s3_folder}/${var.aws_s3_bucket_authorizer_keystore_key}"
 etag   = "${md5(file("${var.aws_s3_folder}/${var.aws_s3_bucket_authorizer
_keystore_key}"))}"

 tags {
  Tool    = "${var.tags-tool}"
  Contact = "${var.tags-contact}"
 }
}
```

Attention, terraforming ne prend pas en compte toutes les ressources AWS, notamment les API Gateway. De ce fait, même si vous souhaitez dupliquer une API GW existante, il vous faudra vous les écrire à la mimine.

Gitignore

.gitignore :

```
#  Local .terraform directories
**/.terraform/*

# .tfstate files
*.tfstate
*.tfstate.*

# .tfvars files
*.tfvars

# .out files
*.out
```

Comment convertir un local state en remote state ?

Si vous avez déjà, par le passé, créé un local state et que vous avez maintenant défini un state en remote dans un fichier nommé backend.tf, par exemple, voici la marche à suivre pour uploader votre local state terraform.tfstate dans le cloud :

```
$ terraform init
$ terraform state push
```

Et CloudFormation, on l'oublie ?

Comme vous l'avez vu, nous avons opté pour Terraform comme

outil pour faire de l'IaC (Infrastructure as Code) pour nos services managés sur AWS mais nous aurions pu partir sur du CloudFormation. On ne s'outille pas pour suivre une mode, ou ce que fait le voisin mais pour répondre aux besoins d'un projet/d'un contexte donné.

Terraform

- Support de presque tous les services AWS et d'autres (cloud) providers ;
- Open Source ;
- Pas de rolling update pour les ASG (Auto Scaling Group) ;
- HCL/JSON ;
- State management ;
- Support de Vault.

CloudFormation

- Support de presque tous les services AWS (uniquement ce cloud provider!) ;
- Service géré et "offert" par AWS ;
- Le rolling update d'EC2 par un ASG est supporté ;
- JSON ;
- Pas de State management.

A noter qu'il est possible de charger du CloudFormation (CF) dans du Terraform, si par exemple vous avez déjà votre stack écrite en CF :

```
# Setup of an CloudformationStack with terraform
resource "aws_cloudformation_stack" "my_stack" {
  depends_on = [
    "aws_s3_bucket_object.lambda_code",
    "aws_s3_bucket_object.authorizer"
  ]
  name = "${var.cf_stack_name}"
  template_url = "https://${aws_s3_bucket.deployment_bucket.bucket
_domain_name}/cf_programmez.json"
  provider = "aws.region"

  parameters {
    LambdaCodeAuthorizer = "${basename(var.authorizer_lambda_jar)}",
    S3CodeBucket = "${aws_s3_bucket.deployment_bucket.id}",
    ...
  }

  tags {
    Tool   = "${var.tags-tool}"
    Contact = "${var.tags-contact}"
  }
}
```

AVANTAGES / INCONVÉNIENTS

Terraform n'est pas une baguette magique, cette techno a des inconvénients, comme pour toutes les technos. Il s'agit notamment d'un outil assez jeune, et en constante évolution donc il y a des bugs et toutes les ressources de tous les providers ne sont pas encore prises en compte.
On peut dresser une petite liste de "Pros" et de "Cons :

Avantages

- Permet de définir de l'Infrastructure as Code ;
- Syntaxe concise et lisible ;
- Réutilisation de : variables, inputs, outputs, modules ;
- Commande **plan ;**
- Multi cloud support ;
- Développement très actif.

Inconvénients

- Outil jeune (comporte des bugs) ;
- Pas de rollback possible ;
- Verbeux.

J'ajouterai que les messages d'erreurs ne sont pas super explicites :

```
* module.lambda.aws_lambda_function.lambda_toto: 1 error(s) occurred:

* module.lambda.aws_lambda_function.lambda_toto: aws_lambda_function
.lambda_toto: InvalidSignatureException: Signature expired: 20171218T075729Z
 is now earlier than 20171218T080647Z (20171218T081147Z - 5 min.)
  status code: 403, request id: 17c82bc7-e3cb-11e7-8a7e-6b18fbf66fae
```

CONCLUSION

Ma conclusion personnelle est que TF est un des outils qui permet de faire de l'Infrastructure as Code et de passer du ClickOps au DevOps. Mais faire du Terraform en local ce n'est pas la vraie vie, donc nous pourrons voir dans un prochain épisode comment automatiser la création de ressources écrites en Terraform dans une chaîne de Continuous Integration/Continuous Deployment avec notre fidèle compagnon : Jenkins. •