1-00 A propos de Terraform

Uptime Formation

21/09/2023

Objectifs

- Avoir une perspective historique sur Terraform

2014 : Naissance (Terraform v0.1)

Le projet est né de l'absence d'alternative libre au projet cloudFormation de AWS

Voici un extrait de l'article de Mitchell Hashimoto en 2011 sur le sujet.

> However, we need an open source, cloud-agnostic solution to this problem. Libraries such as Fog, Boto, libcloud, etc. are not the answer. These libraries are just that… libraries to cloud APIs. Ideally, we would have a declarative language to define infrastructure (incrementally, as well), and tools to consume this language and use the various libraries noted previously to spin up cloud infrastructure, perhaps even across different cloud providers.
> CloudFormation is a brilliant move by AWS and is a technology worth watching, but at this stage its uses are limited, and it leaves space open for an open source alternative, which I hope will come about.

- La cible d'origine est Amazon AWS : le premier provider, fourni par défaut au départ.
- La licence opensource fait partie du projet de base.

2017 : La phase de croissance (Terraform v0.10)

Le "year of Terraform" : décollage des usages et premier partenariat officiel avec un fournisseur de cloud : Microsoft Azure.

- Séparation du core et des providers
- Création du programme pour les providers
- Mise à disposition de la plateforme Registry pour télécharger des providers
- Les partenaires développent le code nécessaire pour utiliser leurs API selon la norme Terraform.

2021 : la phase de stabilisation (Terraform v1.0.0)

Le projet est un succès, devenu un standard de l'IAC provisioning avec un millier de providers

- Le business se développe avec de grandes entreprises qui sont clientes de Hashicorp
- Terraform Enterprise : solution pour gérer l'état de l'infrastructure
- Terraform Cloud : version SAAS de Enterprise fournie par Terraform

L'entreprise se prépare à entrer en bourse, avec une estimation à 13 milliards de dollars en novembre 2021.

2023 : la phase de rentabilisation (Terraform v1.5.6)

- juin : L'action décroche en bourse, Hashicorp promet des réductions de poste et une rentabilité à 2025
- août : Annonce d'un changement de licence avec un passage de la Mozilla Public Licence à la Business Source Licence.
- septembre : annonce officielle de mise à disposition de OpenTF, le fork de Terraform.

Une situation de *Walled Garden* dans laquelle on enferme des utilisateurs après les avoir fait venir.

Par exemple Facebook qui a affiché longtemps "*It's free and always will be*" jusqu'à ce que la mention disparaisse.

La clôture se fait notamment face à des projets concurrents : * Des projets qui utilisent Terraform pour fournir un langage d'IAC : Pulumi * Des projets qui pilotent Terraform pour faire de l'IAC en équipe : Spacelift

---

Quel avenir ? dans quel contexte ?

Le projet de OpenTF

Fournir une alternative à Terraform * conservant la licence Mozilla Public Licence * basée sur la communauté (évolutions selon des RDF) * fondée légalement et institutionnellement (rattachement à la Linux Foundation ou CNCF) * fournissant les outils associés à Terraform et touchés par le changement de licence (Registry)

Une tendance à la fermeture des licences des projets libres qui marchent

Un mouvement historique qui touche notamment les projets libres de bases de données :

> https://linuxfr.org/news/virevoltantes-valses-de-licences-libres-et-non-libres-dans-les-bases-de-donnees

Des exemples : * IBM / RedHat * Oracle / MySQL * Microsoft / Github

Utilisations de licences (Business Source Licence / Server Side Public License) qui visent à empêcher des concurrents d'utilisar le logiciel.

```
> The BSL is not an Open Source license and we do not claim it to be one.
```

Un précédent intéressant : Docker, dont la normalisation permet de dépasser une éventuelle faillite de l'entreprise ou une clôture du projet.

Les forks de logiciels libres sont souvent des projets solides

Le projet OpenTF est soutenu par de nombreuses entreprises comme Spacelift.

L'inertie des projets logiciels est importante

La quantité de projets utilisant Terraform lui donne une garantie de stabilité.

La plupart des entreprises ne vont pas changer immédiatement de solution : former les équipes, produire le code, etc.

En revanche une solution "drop-in replacement" est rassurante a priori.

Un rappel important de l'importance des choix de dépendances logicielles

Ce virage dont les effets sont encore à venir nous rappelle l'importance des choix d'outils et de plateformes pour les métiers de l'ingénierie informatique.

De la même manière qu'il faut être vigilant quand on produit du code (dette technique), il faut s'assurer qu'il existe des portes de sortie pour les fournisseurs de solutions qu'on utilise.

En l'occurence, le code sous licence libre offre une forme de garantie avec le fork.

Les années à venir nous diront ce qu'il en est pour Terraform / OpenTF

Objectifs

- Avoir une perspective historique sur Terraform

1-00 Introduction, IAC et Devops

Uptime Formation

21/09/2023

Objectifs

- Avoir une vision globale de la formation
- Comprendre Terraform dans une logique IAC et Devops

La formation

Jour 1 : Language de déploiement, dans le cloud et au-delà
Jour 2 : Architecture Terraform et bonnes pratiques

Objectifs pédagogiques

- J1/ Comprendre les avantages des approches IAC et Devops
- J1/ Savoir faire des déploiements en utilisant des providers existants
- J2/ Savoir créer ses propres modules
- J2/ Savoir déployer en équipe sur le long terme

Devops et Terraform

Le mouvement DevOps

- Dépasser l'opposition culturelle et de métier entre les développeurs et les administrateurs système.
- Intégrer tout le monde dans une seule équipe et …
- Calquer les rythmes de travail sur l'organisation agile du développement logiciel
- Rapprocher techniquement la gestion de l'infrastructure du développement avec l'infrastructure as code.
  - Concrètement on écrit des fichiers de code pour gérer les éléments d'infra
  - l'état de l'infrastructure est plus claire et documentée par le code
  - la complexité est plus gérable car tout est déclaré et modifiable au fur et à mesure de façon centralisée
  - l'usage de git et des branches/tags pour la gestion de l'évolution d'infrastructure

---

Objectifs du DevOps

- Rapidité (celerité) de déploiement logiciel (organisation agile du développement et livraison jusqu'à plusieurs fois par jour)
  - Implique l'automatisation du déploiement et ce qu'on appelle la CI/CD c'est à dire une infrastructure de déploiement continu à partir de code.
- Passage à l'échelle (horizontal scaling) des logiciels et des équipes de développement (nécessaire pour les entreprises du cloud qui doivent servir pleins d'utilisateurs)
- Meilleure organisation des équipes
  - meilleure compréhension globale du logiciel et de son installation de production car le savoir est mieux partagé
  - organisation des équipes par thématique métier plutôt que par spécialité technique (l'équipe scale mieux)

Apports de Terraform pour le DevOps

- **Automatisation de l'infrastructure**
  Terraform permet de définir l'infrastructure comme du code, ce qui facilite l'automatisation et l'orchestration de la configuration, du déploiement et de la gestion de l'infrastructure.
- **Gestion de l'état**
  Terraform stocke l'état de l'infrastructure dans un fichier, ce qui permet de suivre l'évolution de l'infrastructure et de la gérer efficacement.
- **Gestion multi-cloud**
  Terraform prend en charge de nombreux fournisseurs de cloud (AWS, GCP, Azure, etc.), ce qui facilite la gestion d'infrastructures multi-cloud.
- **Collaboration**
  Terraform permet à plusieurs membres de l'équipe de travailler sur le même code d'infrastructure, ce qui facilite la collaboration et la gestion des modifications.
- **Versioning**
  Terraform permet de versionner le code d'infrastructure, ce qui facilite la gestion des modifications et des versions de l'infrastructure.

IAC et Terraform

Infrastructure as Code L'infrastructure as code (IaC) est une pratique consistant à décrire l'infrastructure informatique comme du code source, à l'aide d'un langage de programmation.

L'objectif est de définir l'infrastructure nécessaire à l'exécution du logiciel d'une manière * documentée * historisée * mutualisée

Cela signifie que l'infrastructure (serveurs, réseaux, stockage, etc.) n'est plus gérée et provisionnée plutôt par des processus manuels.

Ce qui permet d'automatiser les changements d'infrastructure, comme par exemple lancer une infrastructure pour des tests.

L'infrastructure as code est une pratique qui permet de gérer l'infrastructure informatique de manière plus automatisée, reproductible, collaborative, agile et rentable.

Les avantages de l'infrastructure as code sont nombreux :

Automatisation
L'infrastructure as code permet d'automatiser la configuration, le déploiement et la gestion de l'infrastructure, ce qui permet de réduire les erreurs humaines et de gagner du temps.

Reproductibilité
Les scripts d'infrastructure as code peuvent être versionnés, ce qui permet de reproduire facilement des environnements de développement, de test ou de production.

Collaboration
Les scripts d'infrastructure as code peuvent être partagés, modifiés et testés par plusieurs membres de l'équipe, ce qui facilite la collaboration et la gestion des changements.

Agilité
L'infrastructure as code permet de provisionner et de déprovisionner rapidement des ressources, ce qui facilite l'adaptation aux changements et aux besoins de l'entreprise.

Coût
L'infrastructure as code permet de réduire les coûts de maintenance et de gestion de l'infrastructure, car elle permet de minimiser le temps et les ressources nécessaires pour gérer l'infrastructure.

Avantages de Terraform comme solution d'IAC Terraform offre des avantages qui en fait une solution d'IaC populaire pour les équipes DevOps.

- **Base d'utilisateur**
  Terraform a acquis une maturité qui le place comme l'un des acteurs essentiels dans son domaine : le pilotage des API de fournisseurs de services cloud au sens large.
- **Planification**
  La fonction clef de Terraform est sa capacité à reconnaître les parties qui doivent être déployées avant les autres et également ne redéployer que ce qui est nécessaire.
- **Modularité**
  Terraform permet de définir des modules d'infrastructure réutilisables, ce qui facilite la création et la maintenance d'infrastructures complexes.
- **Adaptabilité**
  Terraform permet de définir une infrastructure en apportant la "glue" nécessaire pour interconnecter de manière originale des solutions existantes en branchant ensemble différents modules.

Comparaison entre Terraform et d'autres solutions IAC

Il existe 5 grandes familles d'outils d'IAC.

Scripts ad hoc

C'est la façon la plus basique de faire, en mettant dans des scripts les opérations répétables.
Ex: Scripts Bash

```
## Update the apt-get cache
sudo apt-get update

## Install PHP and Apache
sudo apt-get install -y php apache2

## Copy the code from the repository
sudo git clone https://github.com/brikis98/php-app.git /var/www/html/app

## Start Apache
sudo service apache2 start
```

Outils de gestion de configuration

On utilise des outils de gestion de configuration, ce qui signifie qu'ils sont conçus pour installer et gérer des logiciels sur des serveurs existants. Avantages : conventions de code, idempotence, nombreuses cibles ex: Chef, Puppet et Ansible

```
- name: Update the apt-get cache
  apt:
    update_cache: yes

- name: Install PHP
  apt:
    name: php
```

```
- name: Install Apache
  apt:
    name: apache2

- name: Copy the code from the repository
  git: repo=https://github.com/brikis98/php-app.git dest=/var/www/html/app

- name: Start Apache
  service: name=apache2 state=started enabled=yes
```

Outils de modèles de serveur

Les outils de création de modèles de serveur sont devenus populaire.
Au lieu de lancer un tas de serveurs et de les configurer en exécutant le même code sur chacun d'eux, on crée une image autonome avec le logiciel, les fichiers et tous les autres détails pertinents. Un autre outil IaC déploie cette image sur les serveurs, qu'il s'agisse de VMs ou de conteneurs. Ex: Docker, Packer

```
{
  "builders": [{
    "ami_name": "packer-example-",
    "instance_type": "t2.micro",
    "region": "us-east-2",
    "type": "amazon-ebs",
    "source_ami": "ami-0fb653ca2d3203ac1",
    "ssh_username": "ubuntu"
  }],
  "provisioners": [{
    "type": "shell",
    "inline": [
      "sudo apt-get update",
      "sudo apt-get install -y php apache2",
      "sudo git clone https://github.com/brikis98/php-app.git
       ↪  /var/www/html/app"
    ],
    "environment_vars": [
      "DEBIAN_FRONTEND=noninteractive"
    ],
    "pause_before": "60s"
```

```
  }]
}
```

Outils d'orchestration

Les outils d'orchestration se basent sur des modèles de serveur pour assurer la gestion du cycle de vie des services pilotés par les équipe Devops. L'orchestrateur va piloter ces instances en : démarrage / arrêt, configuration, démultiplication à la demande, et autres opérations nécessaires à la bonne marche du service.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: example-app
spec:
 selector:
    matchLabels:
      app: example-app
  replicas: 3
  strategy:
    rollingUpdate:
      maxSurge: 3
      maxUnavailable: 0
    type: RollingUpdate
  template:
    metadata:
      labels:
        app: example-app
    spec:
      containers:
        - name: example-app
          image: httpd:2.4.39
          ports:
            - containerPort: 80
```

Outils de provisionnement

Provisioning tools don't define the code that runs on each server, but create server, databases, caches, load balancers, queues, monitoring, subnet configurations, firewall settings, routing rules, Secure Sockets Layer (SSL) certificates, and almost every other aspect of your infrastructure.
Ex: Terraform, CloudFormation, OpenStack Heat, and Pulumi

```
resource "aws_instance" "app" {
  instance_type     = "t2.micro"
  availability_zone = "us-east-2a"
  ami               = "ami-0fb653ca2d3203ac1"

  user_data = <<-EOF
              #!/bin/bash
              sudo service apache2 start
              EOF
}
```

Rappel des objectifs

- Avoir une vision globale de la formation
- Comprendre Terraform dans une logique IAC et Devops

1-01 Installation et versions de Terraform

Uptime Formation

21/09/2023

Objectifs

- Appréhender les outils qu'on utilisera au court de la formation

---

Les outils de la formation

Les outils classiques

Nous utiliserons beaucoup durant la formations les outils suivants.

- Desktop via guacamole / VNC
- IDE : vscode / codium
- terminal
- git

Les outils utiles pour gérer terraform

- terraform
- tfenv

Avec quelques utilitaires supplémentaires * jq * dot / graphviz * kubectl

---

AWS

- l'utilitaire aws en ligne de commande

Et quelques concepts importants * IAM * User ID / Access Key / Access Key Secret * Regions * Virtual Private Cloud * AMI * instance * load balancer

---

Rappel des objectifs

- Appréhender les outils qu'on utilisera au court de la formation

1-02 Lancer une première recette Terraform

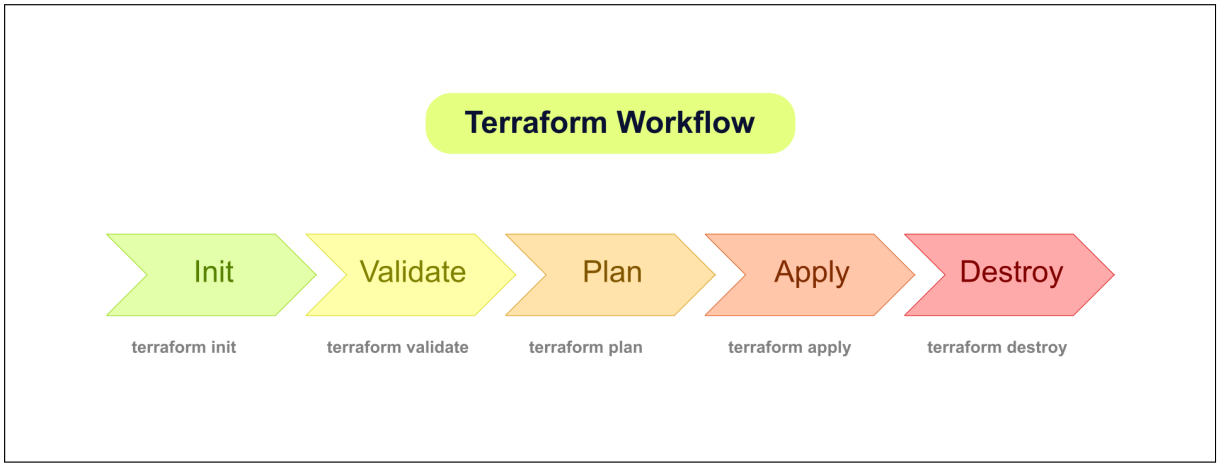
Uptime Formation

21/09/2023

Objectifs

-

Avoir une vue globale du fonctionnement de Terraform



10-ssh-new.tf 186 B

```
resource "openstack_compute_keypair_v2" "certification_key" {
  name       = var.ovh_ssh_key_name
  public_key = file(var.ovh_ssh_key_path) # Path to your previously generated SSH key
}
```

50-instances-basic.tf 482 B

```
data "openstack_images_image_v2" "some_distro" {
  name = "Ubuntu 23.04"
}

resource "openstack_compute_instance_v2" "certification_instance" {
  name        = "terraform_instance"
  image_id    = data.openstack_images_image_v2.some_distro.id
  flavor_name = "d2-2"
  key_pair    = openstack_compute_keypair_v2.certification_key.name
  network {
    name = "Ext-Net"
  }
}

output "instance_addresses" {
  value = openstack_compute_instance_v2.certification_instance.access_ip_v4
}
```

Le dépôt git “fil rouge”

Un dépôt va vous fournir des exemples tout au long de la formation.

> https://github.com/Uptime-Formation/terraform-code-projects

---

Accès au compte AMAZON AWS

Votre poste a été pré-configuré pour que l’utilitaire AWS trouve le fichier de configuration ad hoc.

```
$ cat ~/.aws/credentials
[default]
aws_access_key_id = <AWS_ACCESS_KEY_ID>
aws_secret_access_key = <AWS_SECRET_ACCESS_KEY>
```

On peut également utiliser des variables d’environnement pour s’identifier.

```
$ export AWS_ACCESS_KEY_ID=(your access key id)
$ export AWS_SECRET_ACCESS_KEY=(your secret access key)
```

Lancement du projet

Pour démontrer la capacité de Terraform à fonctionner sans connaissance, on va exécuter le projet.

```
$ git clone https://github.com/Uptime-Formation/terraform-code-projects
$ cd terraform-code-projects
$ cd <PROJET>
$ terraform init
$ terraform apply
```

Observons ce qui se passe !

Terraform en résumé

Terraform est un outil open source créé par HashiCorp et écrit dans le langage de programmation Go.

Vous pouvez utiliser ce binaire pour déployer une infrastructure à partir de votre ordinateur portable ou d'un serveur de build ou à peu près n'importe quel autre ordinateur, et vous n'avez pas besoin d'exécuter une infrastructure supplémentaire pour y arriver.

En effet, sous le capot, le binaire terraform effectue des appels d'API en votre nom à un ou plusieurs fournisseurs, tels qu'AWS, Azure, Google Cloud, DigitalOcean, OpenStack, etc.

Cela signifie que Terraform peut tirer parti de l'infrastructure que ces fournisseurs utilisent déjà pour leurs serveurs d'API, ainsi que des mécanismes d'authentification que vous utilisez déjà avec ces fournisseurs (par exemple, les clés API que vous possédez déjà pour AWS).

Comment Terraform sait-il quels appels d'API à effectuer ?

La réponse est que vous créez des configurations Terraform, qui sont des fichiers texte qui spécifient l'infrastructure que vous souhaitez créer.

Ces configurations sont le « code » dans « l'infrastructure en tant que code ». Voici un exemple de configuration Terraform :

```
resource "aws_instance" "example" {
  ami           = "ami-0fb653ca2d3203ac1"
  instance_type = "t2.micro"
}

resource "google_dns_record_set" "a" {
  name         = "demo.google-example.com"
  managed_zone = "example-zone"
  type         = "A"
  ttl          = 300
  rrdatas      = [aws_instance.example.public_ip]
}
```

Cet extrait demande à Terraform d'effectuer des appels d'API vers AWS pour déployer un serveur, puis d'effectuer des appels d'API vers Google Cloud pour créer une entrée DNS (Domain Name System) pointant vers l'adresse IP du serveur AWS.

Terraform vous permet de déployer des ressources interconnectées sur plusieurs fournisseurs de cloud.

Vous pouvez définir l'ensemble de votre infrastructure (serveurs, bases de données, équilibreurs de charge, topologie du réseau, etc.) dans les fichiers de configuration Terraform et valider ces fichiers dans le contrôle de version.

Vous exécutez ensuite certaines commandes Terraform, telles que terraform apply, pour déployer cette infrastructure.

Le binaire terraform analyse votre code, le traduit en une série d'appels d'API aux fournisseurs de cloud spécifiés dans le code et effectue ces appels d'API aussi efficacement que possible en votre nom.

Rappel des objectifs

- Avoir une vue globale du fonctionnement de Terraform

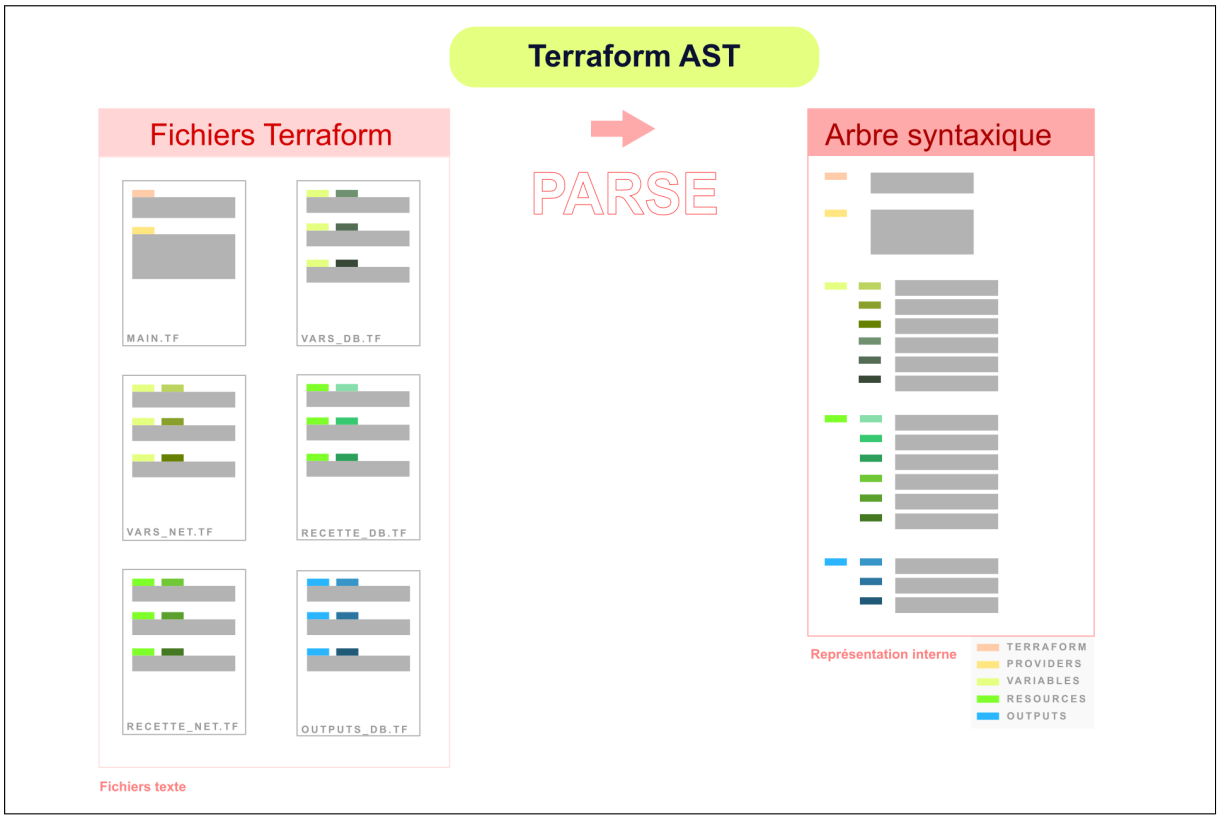
1-03 Concepts de base de Terraform : le langage HCL

Uptime Formation

21/09/2023

Objectifs

- Savoir utiliser le langage utilisé par Terraform



Chlorure d'hydrogène (HCl)

Le chlorure d'hydrogène, de symbole chimique HCl, est un corps composé de chlore et d'hydrogène, incolore, toxique et hautement corrosif. Dans les conditions ambiantes de température et de pression, c'est un gaz qui forme des fumées blanches au contact de l'humidité. Ces fumées sont constituées d'acide chlorhydrique, solution ionique de chlorure d'hydrogène dans l'eau.

HashiCorp Configuration Language (HCL)

Documentation :

- Générale : https://github.com/hashicorp/hcl
- Specs : https://github.com/hashicorp/hcl/blob/main/hclsyntax/spec.md
- Specs agnostiques : https://github.com/hashicorp/hcl/blob/main/spec.md
-

Terraform : https://developer.hashicorp.com/terraform/language/

Un langage intermédiaire entre représentation et exécution

- Langage inventé par HashiCorp pour remplacer les langages de configuration plus verbeux comme JSON et XML.
- Équilibre entre la lisibilité humaine et machine
- Influencé par des tentatives antérieures sur le terrain, telles que la configuration libucl et Nginx.
- Compatible avec JSON, ce qui signifie que HCL peut être converti 1:1 en JSON et vice versa.
- Pensé pour être utilisé par des langages typés, en l'occurence Go

Exercice : Transcrire un json => HCL

```
{
    "name": "Jason Ray",
    "profession": "Software Engineer",
    "age": 31,
    "address": {
        "city": "New York",
        "postalCode": 64780,
        "Country": "USA"
    },
    "languages": ["Java", "Node.js", "JavaScript", "JSON"],
    "socialProfiles": [
        {
            "name": "Twitter",
            "link": "https://twitter.com"
        },
        {
            "name": "Facebook",
            "link": "https://www.facebook.com"
        }
```

```
    ]
}

## We can use comments !

"name" = "Jason Ray"

"profession" = "Software Engineer"

"age" = 31

"address" = {
  "city" = "New York"

  "postalCode" = 64780

  "Country" = "USA"
}

"languages" = ["Java", "Node.js", "JavaScript", "JSON"]

"socialProfiles" = {
  "name" = "Twitter"

  "link" = "https://twitter.com"
}

"socialProfiles" = {
  "name" = "Facebook"

  "link" = "https://www.facebook.com"
}
```

https://www.convertsimple.com/convert-json-to-hcl/

Les composants du langage Terraform Les blocs

Structure imbriquée qui a un nom de type, zéro ou plusieurs étiquettes de chaîne (par exemple, des identifiants) et un corps imbriqué.

En général leur contenu est défini par un schéma préalable qui contient :

- un schéma des attributs
- un schéma du bloc d'en-tête (header)

Ensemble, les éléments structurels créent une structure de données hiérarchique, avec des attributs destinés à représenter les propriétés directes d'un objet particulier dans l'application appelante, et des blocs destinés à représenter des objets enfants d'un objet particulier.

```
$ cat main.tf

variable "base_cidr_block" {
  description = "A /16 CIDR range definition, such as 10.1.0.0/16, that the
 ↪  VPC will use"
  type        = string
  default = "10.1.0.0/16"
}

resource "aws_vpc" "main" {
  cidr_block = var.base_cidr_block
}
```

Les structures résultantes sont multi dimensionnelles

```
dictionary "level_1" "level_2" {
  var = "value"
}

En JSON :

{
  "dictionary":
    {
      "level_1":
        {
          "level_2":
            {
              "var": "value"
            }
        }
    }
}
```

Le fichier de configuration

Généralement produit en lisant un fichier sur le disque et en l'analysant comme une syntaxe particulière.

Dans terraform, le fichier de configuration est souvent le résultat d'une concaténation de plusieurs fichiers.

C'est là une des forces du langage qui autorise * une composition des fichiers * une surcharge potentielle

---

Les expressions

Les valeurs d'attribut sont représentées par des expressions.

Selon la syntaxe concrète utilisée, une expression peut n'être qu'une valeur littérale ou elle peut décrire un calcul en termes de valeurs littérales, de variables et de fonctions.

```
## Basic types
my_absent = null
my_bool = true
my_int = 1024
my_number = 3.14159265358979323846264338327950288419716939937510 58

## Strings and templates
my_str = "Something blue"
my_heredoc =  <<-EOT
  A multiline
    string
      with indents
  EOT
some_string = "My int is ${my_int}"

## Lists and maps
my_list = ["blue", "green"]
some_color = my_list[0]
my_map = {type = "apple", color = "red"}
some_type = my_map["type"]
```

On peut effectuer des opérations de transtypages sur les variables (ex 0 => "0").

Le langage étant typé, il existe toute une complexité de situations qui sont rares dans l'usage courant de Terraform.

Les fonctions

L'usage de fonctions rapprche HCL des langages de templating comme jinja2.

Il existe des dizaines de fonctions dans Terraform

```
$ terraform console
> keys({a=1, c=2, d=3})
[
  "a",
  "c",
  "d",
]
```

Elles permettent de filtrer, transformer, augmenter, qualifier, et toutes autres manipulations utiles des données gérées dans le fichier de configuration.

Les opérateurs et les conditions

HCL utilise des les opérateurs pour les

- opérations arithmétiques `{ a = (1+2) }`
- opérations de comparaisons `1 >= 2`
- opérations logiques `true == false || 1 == 1`

HCL utilise aussi la structure ternaire conditionnelle

```
> { a = true ? 1 : 2 }
```

Un exemple réaliste

```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 1.0.4"
    }
  }
}

variable "aws_region" {}

variable "base_cidr_block" {
  description = "A /16 CIDR range definition, such as 10.1.0.0/16, that the
 ↪  VPC will use"
  type        = string
  default     = "10.1.0.0/16"
```

```
}

variable "availability_zones" {
  description = "A list of availability zones in which to create subnets"
  type = list(string)
  default = ["us-east-2","eu-west-3"]
}

provider "aws" {
  region = var.aws_region
}

resource "aws_vpc" "main" {
  # Referencing the base_cidr_block variable allows the network address
  # to be changed without modifying the configuration.
  cidr_block = var.base_cidr_block
}

resource "aws_subnet" "az" {
  # Create one subnet for each given availability zone.
  count = length(var.availability_zones)

  # For each subnet, use one of the specified availability zones.
  availability_zone = var.availability_zones[count.index]

  # By referencing the aws_vpc.main object, Terraform knows that the subnet
  # must be created only after the VPC is created.
  vpc_id = aws_vpc.main.id

  # Built-in functions and operators can be used for simple transformations
   ↪  of
  # values, such as computing a subnet address. Here we create a /20 prefix
   ↪  for
  # each subnet, using consecutive addresses for each availability zone,
  # such as 10.1.16.0/20 .
  cidr_block = cidrsubnet(aws_vpc.main.cidr_block, 4, count.index+1)
}
```

On peut agir dans le terminal avec cette recette

```
$ cd TH-01-HCL
```

```
$ terraform init
$ terraform console
> var.base_cidr_block
"10.1.0.0/16"
```

Rappel des objectifs

- Savoir utiliser le langage utilisé par Terraform

1-04 La CLI de Terraform

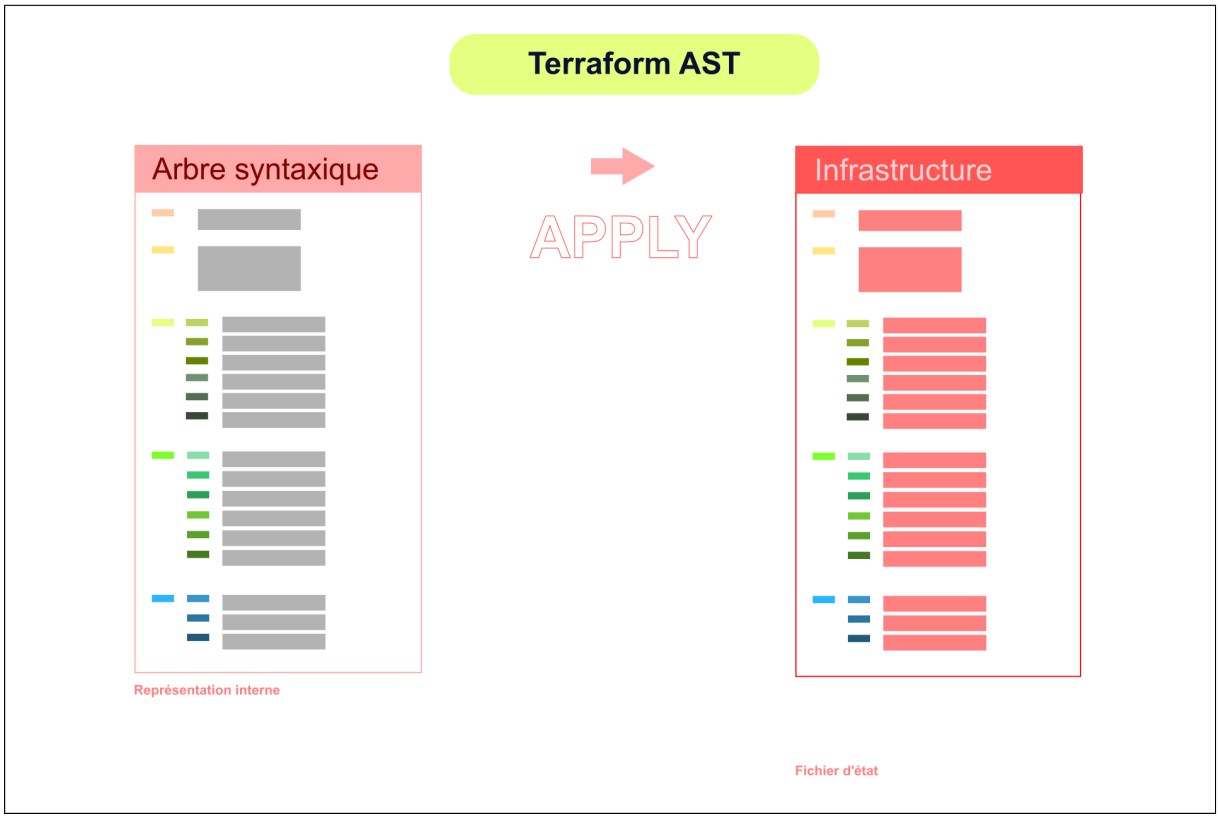
Uptime Formation

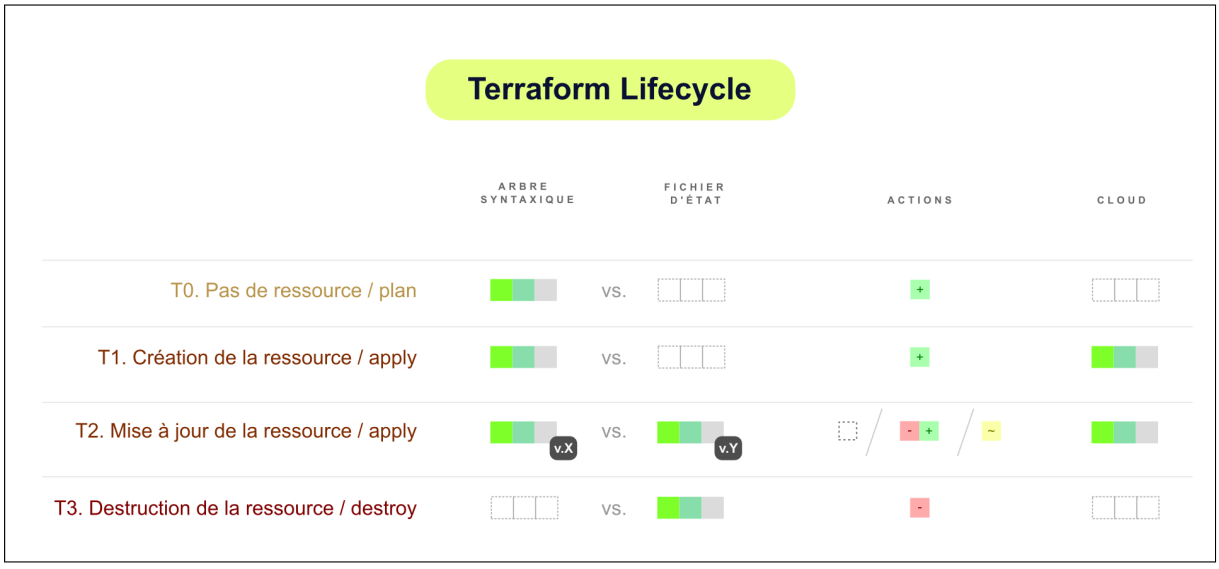
21/09/2023

Objectifs

-

Comprendre et utiliser les principales commandes de la CLI terraform





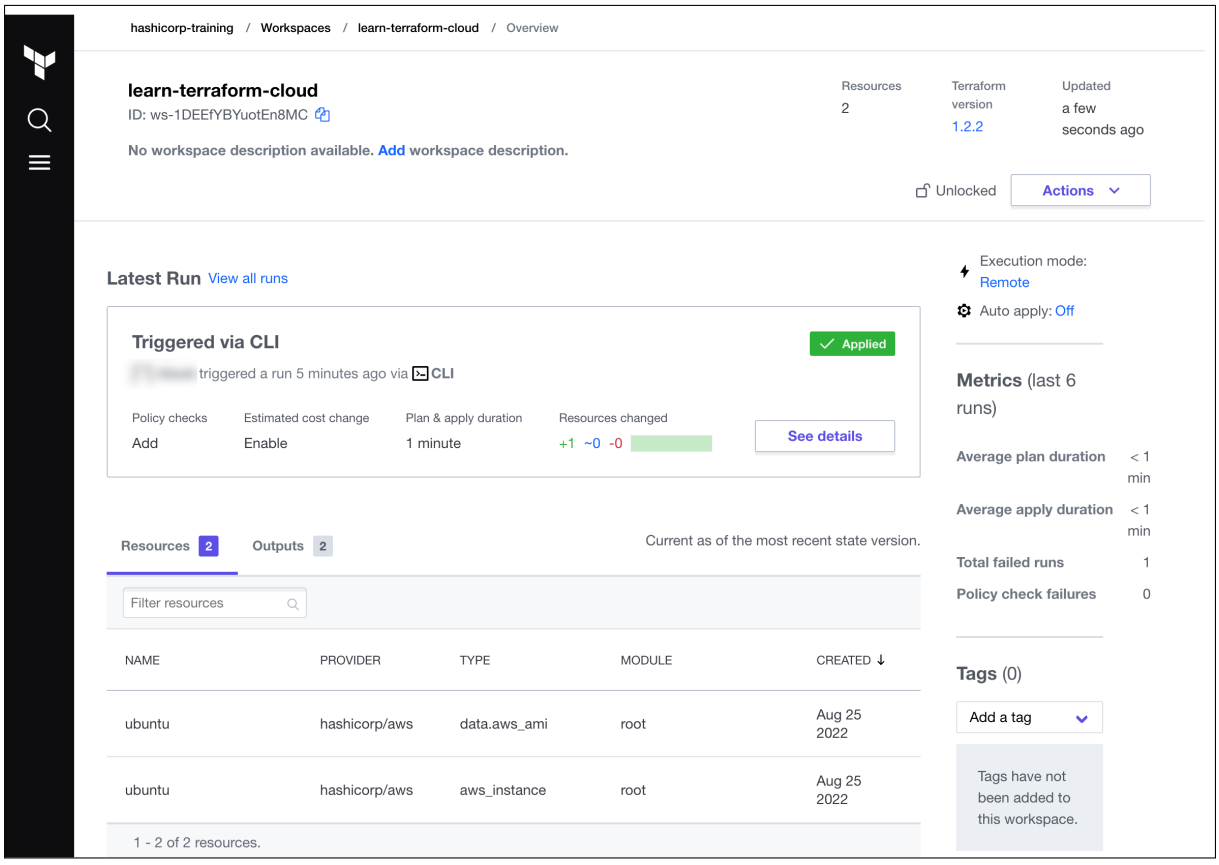
Les IHM de Terraform

CLI L'outil en ligne de commande est le moyen d'accès le plus simple et essentiel pour apprendre et maîtriser Terraform.

- Exposition progressive aux concepts et aux pratiques
- Approfondissement progressif de chaque commande selon le besoin
-

Intégralité des opérations disponibles

Une autre solution est Terraform Enterprise qui présente une interface HTML.



Terraform Cloud est le SASS proposé par la société Hashicorp pour cet outil.

Il offre de nombreux avantages pour le travail en équipe, comme par exemple l'intégration de GIT, des états d'infrastructure.

CDK for Terraform On peut également piloter Terraform avec un langage comme TypeScript, Python, Java, C#, ou Go.

> https://developer.hashicorp.com/terraform/cdktf

```
// Copyright (c) HashiCorp, Inc
// SPDX-License-Identifier: MPL-2.0
import { Construct } from "constructs";
import { App, TerraformStack, TerraformOutput } from "cdktf";
import {
  DataAwsRegion,
  AwsProvider,
  DynamodbTable,
  SnsTopic,
} from "@cdktf/provider-aws";

export class HelloTerra extends TerraformStack {
  constructor(scope: Construct, id: string) {
    super(scope, id);

    new AwsProvider(this, "aws", {
      region: "eu-central-1",
    });

    const region = new DataAwsRegion(this, "region");

    const table = new DynamodbTable(this, "Hello", {
      name: `my-first-table-${region.name}`,
      hashKey: "temp",
      attribute: [{ name: "id", type: "S" }],
      billingMode: "PAY_PER_REQUEST",
    });
```

```
    table.addOverride("hash_key", "id");
    // table.addOverride('hash_key', 'foo')
    table.addOverride("lifecycle", { create_before_destroy: true });

    const topicCount = 1;
    const topics = [...Array(topicCount).keys()].map((i) => {
      return new SnsTopic(this, `Topic${i}`, {
        displayName: `my-first-sns-topic${i}`,
      });
    });

    new TerraformOutput(this, "table_name", {
      value: table.name,
    });

    topics.forEach((topic, i) => {
      new TerraformOutput(this, `sns_topic${i}`, {
        value: topic.name,
      });
    });
  }
}

const app = new App();
new HelloTerra(app, "hello-terra");
app.synth();
```

Les commandes principales de `terraform`

Documentation: * https://developer.hashicorp.com/terraform/cli

```
## Obtenir de l'aide sur la CLI
$ terraform --help
Usage: terraform [global options] <subcommand> [args]
The available commands for execution are listed below.
The primary workflow commands are given first, followed by
less common or more advanced commands.

Main commands:
  init          Prepare your working directory for other commands
  validate      Check whether the configuration is valid
```

```
  plan           Show changes required by the current configuration
  apply          Create or update infrastructure
  destroy        Destroy previously-created infrastructure

All other commands:
  console        Try Terraform expressions at an interactive command prompt
  fmt            Reformat your configuration in the standard style
  force-unlock   Release a stuck lock on the current workspace
  get            Install or upgrade remote Terraform modules
  graph          Generate a Graphviz graph of the steps in an operation
  import         Associate existing infrastructure with a Terraform resource
  login          Obtain and save credentials for a remote host
  logout         Remove locally-stored credentials for a remote host
  output         Show output values from your root module
  providers      Show the providers required for this configuration
  refresh        Update the state to match remote systems
  show           Show the current state or a saved plan
  state          Advanced state management
  taint          Mark a resource instance as not fully functional
  test           Experimental support for module integration testing
  untaint        Remove the 'tainted' state from a resource instance
  version        Show the current Terraform version
  workspace      Workspace management

Global options (use these before the subcommand, if any):
  -chdir=DIR    Switch to a different working directory before executing the
                given subcommand.
  -help        Show this help output, or the help for a specified subcommand.
  -version      An alias for the "version" subcommand.
```

```
## Obtenir de l'aide sur une commande
$ terraform apply -h
...
## Utiliser une commmande avec une option globale
$ terraform -chdir /opt/workplace apply
```

Les domaines de commandes

Essentiels

- Gérer l'espace de travail
  - get

  - init
- Gestion du cycle de vie des ressources
  - plan
  - apply
  - destroy
- Inspection / visualisation de l'infrastructure
  - show
  - providers
  - graph
  - output
- Gestion de l'état de l'infrastructure
  - state
- Test et réécriture
  - console
  - validate
  - fmt

Avancés

- Gestion des modules
  - get
- Authentification (Terraform Enterprise)
  - login
  - logout
- Gestion d'environnements
  - wordspace
- Import de ressources
  - import
- Tests
  - test

Commandes essentielles

---

init La commande `init` encapsule tout ce qui est nécessaire pour rendre utilisable une recette d'infrastructure Terraform.

- Initialisation du stockage de l'état afin de permettre de le stocker dans un backend (on y reviendra)
- Téléchargement des dépendances (modules et providers: idem on y reviendra)

```
$ ls -a1
.
..
main.tf

$ terraform init
Initializing the backend...

Initializing provider plugins...
- Finding latest version of hashicorp/aws...
- Installing hashicorp/aws v4.57.0...
- Installed hashicorp/aws v4.57.0 (signed by HashiCorp)

Terraform has created a lock file .terraform.lock.hcl to record the provider
selections it made above. Include this file in your version control
 ↪  repository
so that Terraform can guarantee to make the same selections by default when
you run "terraform init" in the future.

Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan" to see
any changes that are required for your infrastructure. All Terraform commands
should now work.

If you ever set or change modules or backend configuration for Terraform,
rerun this command to reinitialize your working directory. If you forget,
 ↪  other
commands will detect it and remind you to do so if necessary.

$ ls -a1
```

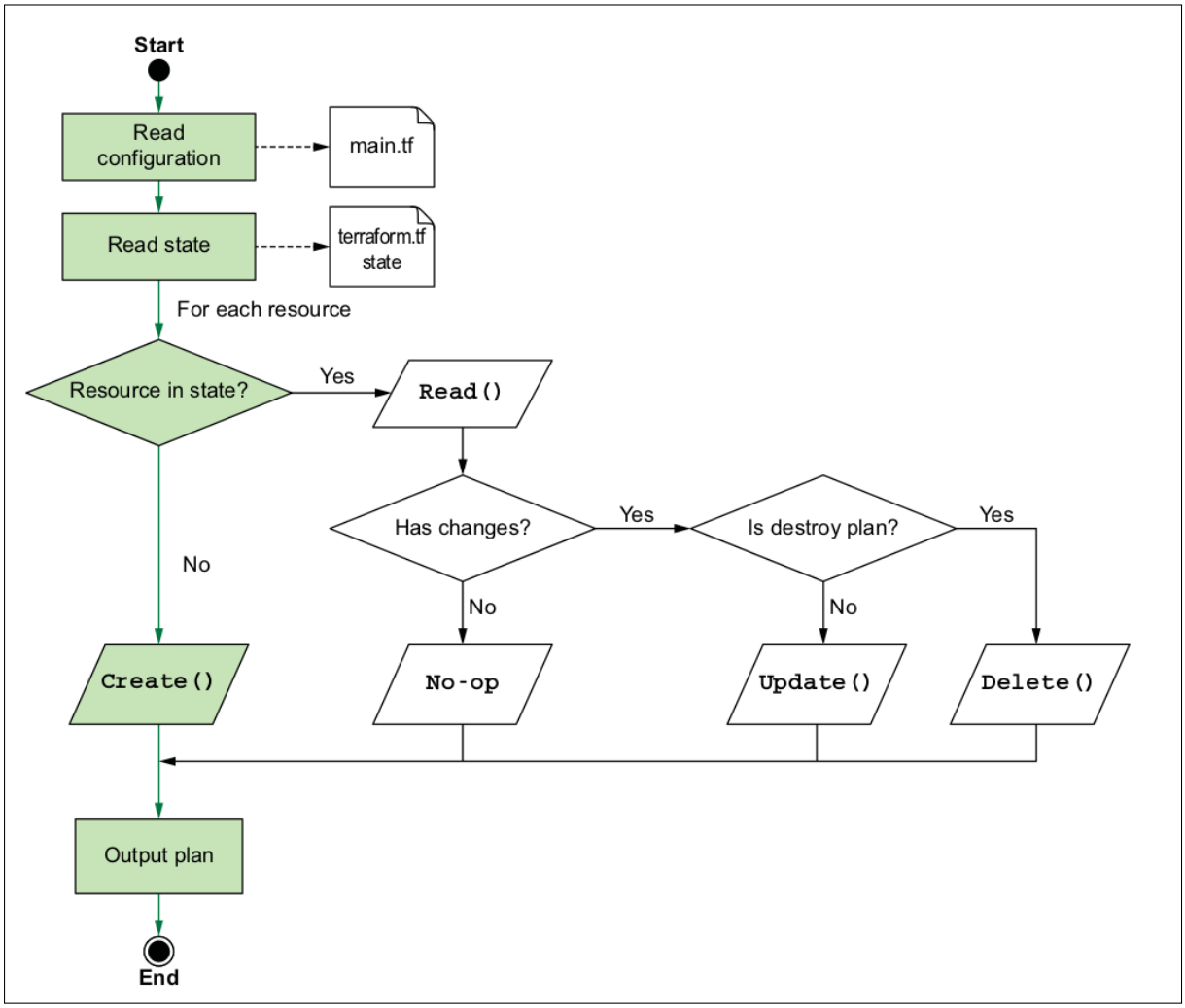
```
.
..
main.tf
.terraform
.terraform.lock.hcl
```

On voit que la commande `init` crée deux éléments cachés

- Le dossier `.terraform` contient les plugins / providers requis par la recette
- Le fichier `.terraform.lock.hcl` stocke la version des dépendances utilisées.
  Ce fichier peut être ajouté au contrôle de version afin d'éviter que les versions déployées sont incompatibles.

---

plan La commande `plan` encapsule la logique de gestion du cycle de vie pour planifier les actions nécessaires.



Elle est fondamentale dans le fonctionnement de Terraform.

Elle prend en compte l'état et les dépendances entre ressources à générer pour orchestrer les actions à entreprendre.

---

apply La commande `apply` exécute un plan, encapsulant par défaut un appel à `plan`.

On peut aussi obtenir un plan sous forme de fichier et le passer à `apply`.

---

destroy La commande `destroy` génère un plan de destruction, encapsulant par défaut un appel à `plan`.

C'est simplement un alias vers `terraform apply -destroy`

---

show La commande show affiche le contenu d'un plan ou de l'état.

```
$ terraform show -json terraform.tfstate
$ terraform plan -out plan.out && terraform show plan.out
```

---

providers La commande `providers` affiche les plugins / providers de la recette.

```
$ terraform providers schema -json  | jq | less
```

---

graph La commande `graph` produit un graph au format DOT(.dot) de l'infrastructure.

Elle permet de mieux comprendre / documenter des recettes.

```
$ terraform graph | dot -Tsvg > graph.svg
```

---

output La commande `output` extraie les valeurs qui ont été produites par la recette.

Ces valeurs ne sont pas connues avant l'exécution et sont utiles pour communiquer entre différents modules.

---

state La commande `state` permet d'afficher et d'interagir avec l'état de l'infrastructure.

```
$ terraform state list

$ terraform state show <object>
```

---

console La commande `console` permet d'exécuter des expressions en HCL au sein de l'état actuel de l'infrastructure.

Cela permet de tester des appels de fonction avant de les tester dans le code.

---

validate La commande `validate` permet de tester la validité programmatique de la recette, sans faire appel aux ressources.

Elle est pratique pour s'assurer de la qualité du code.

---

fmt La commande `fmt` sert à rendre le style du code conforme aux normes.

Elle fixe notamment les problèmes d'espaces et d'indentation pour uniformiser le style du code.

```
$ cat <<EOF | terraform fmt -
> foo =    "bar"
> l = [1,2,  3]
> EOF
foo = "bar"
l   = [1, 2, 3]
```

Rappel des objectifs

- Comprendre et utiliser les principales commandes de la CLI terraform

1-05 Concepts de base de Terraform : provider, resource et data

Uptime Formation

21/09/2023

Objectifs

- Comprendre et savoir utiliser les objets fondamentaux de Terraform

Les providers

Terraform s'appuie sur des plugins appelés providers (fournisseurs) pour interagir avec les fournisseurs de cloud, les fournisseurs SaaS et d'autres API.

Les configurations Terraform doivent déclarer les providers dont elles ont besoin pour que Terraform puisse les installer et les utiliser.

De plus, certains fournisseurs nécessitent une configuration (comme les URL de point de terminaison ou les régions cloud) avant de pouvoir être utilisés.

La liste des providers est disponible sur le Resistry Terraform.

> https://registry.terraform.io/browse/providers

Terraform effectue des validations cryptographiques du contenu des providers téléchargés pour s'assurer qu'ils sont conformes.

Il existe des fournisseurs : * Officiels : développés par hashicorp, namespace: hashicorp * Partners : développés par des entreprises en relation étroite avec Hashicorp, namespace : "company" * Community : développés par la communauté, namespace : "user" * Archived : versions dépréciées

---

La documentation pour utiliser un provider est dans son code et sa page Terraform.

> https://registry.terraform.io/providers/hashicorp/boundary/latest/docs
> https://registry.terraform.io/providers/hashicorp/vault/latest/docs

---

Chaque fournisseur ajoute un ensemble de `resources types` (types de ressources) et/ou de `data sources` (sources de données) que Terraform peut gérer.

Chaque type de ressource est implémenté par un fournisseur ; sans fournisseurs, Terraform ne peut gérer aucun type d'infrastructure.

La plupart des fournisseurs configurent une plate-forme d'infrastructure spécifique (cloud ou auto-hébergée).

Les fournisseurs peuvent également proposer des utilitaires locaux pour des tâches telles que la génération de nombres aléatoires pour des noms de ressources uniques.

Les providers sont codés en Go.

> Ex: https://github.com/hashicorp/terraform-provider-boundary

Pour gérer le cycle de vie des ressources, ils prennent en compte les quatres méthodes CRUD

- Create
- Read
- Update
- Delete

> ex: https://github.com/hashicorp/terraform-provider-boundary/blob/main/internal/provider/resource_group.go

```
> resourceGroup(...)
> resourceGroupCreate(...)
> resourceGroupRead(...)
> resourceGroupUpdate(...)
> resourceGroupDelete(...)
```

Le lock file pour les providers permet de les versionner sous contrôle.

```
## This file is maintained automatically by "terraform init".
## Manual edits may be lost in future updates.

provider "registry.terraform.io/hashicorp/aws" {
  version = "4.57.0"
  hashes = [
    "h1:0bd5IKkEF1TGE4tgm0VuVMFQg2s6GOXJBU+/b/siYKw=",
    "zh:07d89ad94267b7d6285fd65fbd67f8680e111abf9bbcbcac2e30154262fbbe46",
    "zh:0eeee044e6fc285c20241d3de7f9b79450cab2df1452a9c18c0bed1090085a25",
    "zh:306ba8ac99a0d9f9eba0386cb11459323696e69dcb28bc5e55b6fb2de28640cd",
    "zh:40afc24b94e7cae387f22dd3045b09311a120e429aa4f06168d7498995a98f67",
    "zh:5a2c846a2cc463841ca2353fb734ba6f9502e662196c85fd3332a4e18acec72e",
    "zh:854fbf7d058e4e31ce4ed882e2085bd94c53be4b38b15f3b5d3d897a2c5102df",
```

```
    "zh:89a7a5e7de6400662804d5dc43251172e3f0522853dcab304d637a7bbb266654",
    "zh:89ef96a1b36396f555e80505f55fd29432be3dc518bd75b72a1aae29e8171b4a",
    "zh:9b12af85486a96aedd8d7984b0ff811a4b42e3d88dad1a3fb4c0b580d04fa425",
    "zh:b28516cc8e614fad40738ec73ce70528e2a817dae3118895333c1d63f1e22a89",
    "zh:c3f1c6a7d56b0838da2f880a74e19df65ca9006cb3ebdc34403cb9d6e4ee046d",
    "zh:d036a7355494792e2347b92d766431ba91cb399a4bd2bb719db3025542c0e674",
    "zh:d3299b9507085238aaf24f38faffb5d6226a31f916e47320b31d728c7062be16",
    "zh:d9f5c04f4648d593d91be2a66c6c61f6c52512c78ac6fb3077f0911a6b95fa2f",
    "zh:f84143ee0cff2ad0af8ad40074fb5dcd83bfb8a514c7f43ca23c7422caa42330",
  ]
}
```

Les hashs sont tirés des zips qui constituent l'archive

- zh : "Zip Hash" (legacy) sha256 des fichiers zip
- h1 : "Hash Scheme 1" (prefered) sha256 des contenus

On peut mettre à jour les dépendances de providers avec la commande suivante.

```
$ terraform init -upgrade
```

L'ajout à git du fichier de lock permet de gérer collaborativement le versionnage des providers.

Les data sources

Un fournisseur peut exposer des datas sources, soit des informations sur des types d'objets.

La syntaxe HCL pour déclarer une data contient le nom du provider qui la fournit :

```
data "<PROVIDER>_<DATA_SOURCE>" "<NAME>" {
    [CONFIG ...]
}
```

La plupart des éléments du corps d'un bloc de données sont définis par et spécifiques à la source de données sélectionnée, et ces arguments peuvent tirer pleinement parti des expressions et d'autres fonctionnalités dynamiques du langage Terraform.

```
data "aws_ami" "ubuntu" {
  most_recent = true
```

```
  filter {
    name   = "name"
    values = ["ubuntu/images/hvm-ssd/ubuntu-focal-20.04-amd64-server-*"]
  }

  owners = ["099720109477"]
}
```

Attention, l'argument `filter` dans ce bloc n'est pas générique : il dépend directement de l'implémentation fournie par le provider.

Une fois l'appel effectué, la valeur demandée est disponible pour être utilisée dans des expressions HCL.

```
  id = data.aws_ami.ubuntu.id
```

Les resources

Chaque fournisseur peut exposer des ressources, soit des consommations types d'objets.

Ces objets peuvent être des instances, des bases de données, des noms de domaines, ou toute autre structure utile au montage d'une infrastructure.

La syntaxe HCL pour déclarer une ressource contient le nom du provider qui la fournit :

```
resource "<PROVIDER>_<TYPE>" "<NAME>" {
  [CONFIG ...]
}
```

Rappel des objectifs

- Comprendre et savoir utiliser les objets fondamentaux de Terraform

1-06 Les entrées et sorties de Terraform

Uptime Formation

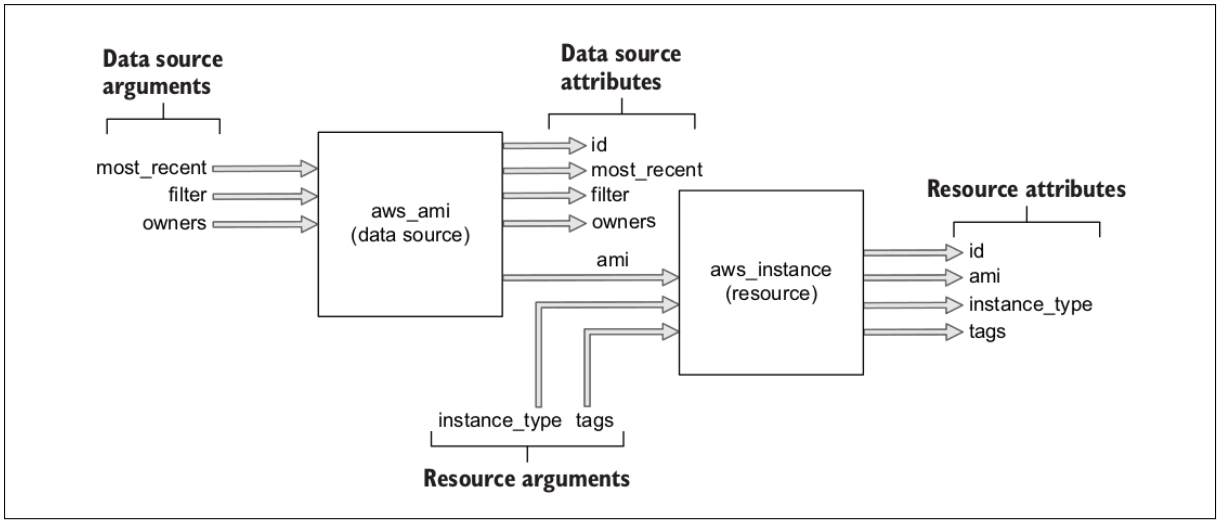
21/09/2023

Objectifs

- Comprendre les variables dans Terraform
- Comprendre les outputs dans Terraform

Terraform et le cycle des données

Terraform suit un cycle de données en plusieurs étapes pour gérer l'infrastructure.



Tout d'abord, Terraform doit connaître le contexte de l'utilisateur, tel que l'environnement, les mots de passe, les variables, et autres contraintes d'exécution.

Les données en entrée sont utilisées sous formes d'`arguments` pour les ressources.

Ensuite, le déploiement fournit des données telles que les data sources, les ID de machine, les adresses IP et d'autres informations d'infrastructure.

Ce sont les `attributs` qui sont un mélange d'argument et d'attributs générés à l'instanciation.

Lorsque la configuration est appliquée, Terraform utilise un mélange de variables fournies et produites via des sorties pour créer la recette de déploiement.

La force de Terraform est de planifier l'exécution en fonction des informations requises, c'est un ordonnancement par les variables.

Les objets Terraform sont tous des fournisseurs de données potentielles via leurs attributs.

Terraform sait aussi exporter certains attributs pour leur réutilisation : ce sont les `outputs`.

Ces variables, outputs, attributs de ressources sont parfois sensibles (sensitive) car il s'agit de secrets.

Les variables en entrées

Dans Terraform, les variables et les locals sont deux concepts importants pour définir des valeurs qui peuvent être réutilisées dans la configuration.

Les variables **Les variables sont des valeurs qui peuvent être utilisées dans une configuration Terraform.**

Elles peuvent être définies dans le fichier principal de configuration, ou dans un fichier séparé.

Les variables peuvent être de différents types, tels que des chaînes de caractères, des nombres, des booléens ou des listes.

Les variables peuvent également être utilisées pour des valeurs sensibles telles que les clés d'accès à des API.

Les variables peuvent être définies à l'aide de la syntaxe suivante :

```
variable "nom_de_variable" {
  type = type_de_variable
  default = valeur_par_defaut
}
```

Lorsque la configuration est appliquée, Terraform demande à l'utilisateur de fournir les valeurs pour les variables non définies.

Attribuer des valeurs de variable avec l'argument par défaut n'est pas une bonne idée car cela ne facilite pas la réutilisation du code.

Une meilleure façon de définir des valeurs de variable consiste à utiliser un fichier de définition de variables, qui est tout fichier `terraform.tfvars` ou `terraform.tfvars.json`.

Un fichier de définition de variables utilise la même syntaxe que le code de configuration Terraform, mais se compose exclusivement d'affectations de variables.

```
## file: terraform.tfvars
default_cidr = 10.4.0.0/24
```

On peut spécifier à Terraform des variables ou fichiers de variable spécifiques en ligne de commande.

```
$ terraform apply -var="image_id=ami-abc123"
$ terraform apply -var-file="testing.tfvars"
```

Terraform charge par défaut tous les fichiers nommés : `* terraform.tfvars` or `terraform.tfvars.json` `*.auto.tfvars` or `*.auto.tfvars.json`

On peut également affecter les variables via des variables d'environnement.

On utilise la syntaxe `TF_VAR_{ma variable}`.

```
$ export TF_VAR_image_id=ami-abc123
$ export TF_VAR_availability_zone_names='["us-west-1b","us-west-1d"]'
```

Terraform charge les variables dans l'ordre suivant, les sources ultérieures ayant priorité sur les précédentes :

- Variables d'environnement
- Le fichier `terraform.tfvars`, s'il existe.
- Le fichier `terraform.tfvars.json`, s'il est présent.
- Tous les fichiers `*.auto.tfvars` ou `*.auto.tfvars.json`, traités dans l'ordre lexical de leurs noms de fichiers.
-

  Toutes les options `-var` et `-var-file` sur la ligne de commande, dans l'ordre dans lequel elles sont fournies. (Cela inclut les variables définies par un espace de travail Terraform Cloud.)

Les variables sont référencées dans le code en utilisant la syntaxe `var.nom_de_variable`.

Il s'agit d'une expression comme une autre, on peut accéder à des items d'un objet complexe ou lui appliquer une fonction.

Les locals

Les locals sont des valeurs calculées dans une configuration Terraform.

Ils sont similaires aux variables, mais sont utilisés pour éviter de répéter certaines variables dans la recette.

Attention, les locals définis dans le fichier sont utilisés dans le même fichier et ne sont pas accessibles en dehors.

Les locals sont définis à l'aide de la syntaxe suivante

```
locals {
  nom_de_local = expression,
  [CONFIG ...]
}
```

L'expression peut utiliser des variables, d'autres locals et des fonctions Terraform.

```
locals {
  # Ids for multiple sets of EC2 instances, merged together
  instance_ids = concat(aws_instance.blue.*.id, aws_instance.green.*.id)
}
```

Les locals peuvent être référencés dans le code en utilisant la syntaxe `local.nom_de_local`.

Comme d'habitude ce sont des expressions donc on peut appliquer des fonctions ou accéder à des éléments de leur structure.

Les outputs

Les `outputs` (valeurs de sortie) rendent les informations sur votre infrastructure disponibles sur la ligne de commande et peuvent exposer des informations que d'autres configurations Terraform peuvent utiliser.

Les valeurs de sortie ont plusieurs utilisations :

- Afficher certains résultats de la recette après `terraform apply`
- Exporter certaines variables pour les modules

Chaque valeur de sortie doit être déclarée à l'aide d'un bloc de sortie.

```
output "instance_ip_addr" {
   value = aws_instance.server.private_ip
}
```

Les sorties ne sont disponibles que lorsque Terraform applique votre plan.

Les données sensibles

Les `variables` et les `outputs` peuvent être définis comme sensibles afin de masquer leur contenu.

```
variable "user_password" {
  type = string
  sensitive = true
}

output "db_password" {
  value       = aws_db_instance.db.password
  description = "The password for logging in to the database."
  sensitive   = true
}
```

Terraform masquera les valeurs marquées comme sensibles dans les messages de `terraform plan` et `terraform apply`.

Les données sensibles sont accessibles en clair dans l'état de Terraform.

Rappel des objectifs

- Comprendre les variables dans Terraform
- Comprendre les outputs dans Terraform

1-07 Fonctions intégrées à Terraform

Uptime Formation

21/09/2023

Objectifs

- Connaître et savoir utiliser les fonctions intégrées dans Terraform

Les fonctions dans Terraform

Documentation: * https://developer.hashicorp.com/terraform/language/functions

Terraform intègre de nombreuses fonctions mais ne permet pas l'ajout de fonctions utilisateurs.

Les fonctions Terraform sont des blocs de code intégrés et réutilisables qui exécutent des tâches spécifiques dans les configurations Terraform.

Collection : fonctions pour manipuler des structures - tableaux, objets, listes et maps

> Ces fonctions sont courantes pour faire des créations, extractions et conversions de structures complexes.

Codage : fonctions d'encodage et de décodage qui traitent de divers formats - base64, texte, JSON, YAML, etc.

> Utiles pour transformer un contenu d'un format vers un autre.

Système de fichiers : fonctions pour effectuer des opérations essentielles sur les fichiers

> Peu de fonctions, car contrairement à ansible Terraform opère sur des API réseaux et les accès au système de fichiers sont limités.

Numérique : fonctions pour effectuer des opérations mathématiques sur des valeurs numériques

> Une dizaine de fonctions pour faire des opérations mathématiques de base.

Date et heure : fonctions de manipulation de la date et de l'heure

> Quelques fonctions pour manipuler des timestamps.

Hachage et crypto : fonctions qui fonctionnent avec des mécanismes de hachage et cryptographiques

> En cas de besoin, on peut vérifier une checksum ou obtenir un UUID

Réseau IP : fonctions pour travailler avec les plages CIDR

> Utile pour calculer des IPs ou obtenir des CIDR

Conversion de types : fonctions pour convertir les types de données

> Des fonctions utiles pour valider le contenu d'une variable ou la transformer à chaud

---

Une sélection de fonctions couramment utilisées

format(string_format, unformatted_string) La fonction format est similaire à la fonction printf en C et fonctionne en formatant un certain nombre de valeurs selon une chaîne de spécification.

Il peut être utilisé pour créer différentes chaînes pouvant être utilisées conjointement avec d'autres variables. Voici un exemple d'utilisation de cette fonction :

```
> format("%s_%s_%d", "foo", "bar", 2)
"foo_bar_2"
```

---

formatlist(string_format, unformatted_list) La fonction formatlist utilise la même syntaxe que la fonction format mais modifie les éléments d'une liste.

Voici un exemple d'utilisation de cette fonction :

```
> formatlist("Bonjour, %s!", ["A", "B", "C"])
tolist([
  "Bonjour, A!",
  "Bonjour, B!",
  "Bonjour, C!",
])
```

---

length (liste / chaîne / carte) Renvoie la longueur d'une chaîne, d'une liste ou d'une carte.

```
> length([10, 20, 30])
3
```

---

join (séparateur, liste) Cette fonction crée une chaîne en concaténant tous les éléments d'une liste et un séparateur. Par exemple, considérez le code suivant :

```
> join(",", ["a", "b", "c"])
"a,b,c"
```

---

flatten (liste) Dans Terraform, vous pouvez travailler avec des types de données complexes pour gérer votre infrastructure. Dans ces cas, vous souhaiterez peut-être aplatir une liste de listes en une seule liste.

```
> flatten([[1, 2, 3], [4, 5], [6]])
[
  1,
  2,
  3,
  4,
  5,
  6,
]
```

---

keys(map) & values(map)

```
> keys({ "a": 1, "b": 2 })
[
  "a",
  "b",
]
> values({ "a": 1, "b": 2 })
[
```

```
  1,
  2,
]
```

---

slice(list, startindex, endindex)

```
> slice([1, 2, 3, 4], 2, 4)
[
  3,
  4,
]
```

---

range(...) Crée une plage de nombres :

- un argument (limite)
- deux arguments (valeur_initiale, limite)
- trois arguments (initial_value, limit, step)

```
> range(14)
> range(10,26,2)
```

lookup(map, key, fallback_value) Récupère une valeur d'une map à l'aide de sa clé.

Si la valeur n'est pas trouvée, il renverra la valeur par défaut à la place.

```
> lookup({ "a": 1, "b": 2 }, "c", 3)
3
```

---

concat(listes) Prend deux ou plusieurs listes et les combine en une seule.

```
> concat([1, 2, 3], [4, 5, 6])
```

---

merge(maps) Fusionne des maps.

```
> merge( {"a": 1, "b": 2 },{"c": 3, "z": 26 })
```

---

file(path_to_file)

Stocke le contenu d'un fichier dans une variable.

```
> file("/etc/resolv.conf")
```

---

templatefile(path, vars) Renvoie le contenu d'un fichier avec des interpolations de variables.

```
## $cat /tmp/template
## Hello ${name}

> templatefile("/tmp/template", {"name": "world"})
```

(json|yaml)(decode|encode)(string|value) Effectue des conversions depuis et vers les formats YAML / JSON.

```
> jsonencode({"a":1,"file":2}
```

Rappel des objectifs

- Connaître et savoir utiliser les fonctions intégrées dans Terraform

1-08 Boucles et expressions conditionnelles dans un langage déclaratif comme Terraform

Uptime Formation

21/09/2023

Objectifs

- Savoir utiliser les boucles et conditions logiques dans Terraform

Le meta argument `count` Ce méta-argument count peut provisionner dynamiquement plusieurs instances d'un Ressource.

Il prend pour paramètre un nombre entier qui détermine le nombre d'objets désirés.

```
variable "user_names" {
  description = "Create IAM users with these names"
  type        = list(string)
  default     = ["neo", "trinity", "morpheus"]
}

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

La syntaxe `count.index` permet d'accéder à l'entier correspondant à l'itération courante, de `{0...N}`.

Pour accéder à une instance d'une ressource créée avec count, utilisez la notation crochet [], car il produit logiquement une liste de ressources.

```
aws_iam_user.example[0]
```

Il existe plusieurs limitations qui limitent l'usage de `count`.

1. Il n'est utilisable que sur des blocks de ressources, et non sur des blocs intérieurs.
2. Il peut introduire des erreurs en raison de l'index d'itération.

Q: Que se passe-t-il si j'enlève l'utilisateur ayant l'index 1 de la liste ?
R: La valeur de `aws_iam_user.example[1]` n'est plus la même et Terraform va replanifier en fonction.

L'expression `for_each` Cette expression comparable à `count` itére sur des listes et des maps pour créer plusieurs copies d'une ressource.

On peut l'utiliser comme `count` en tant que paramètre de la ressource, en lui fournissant une structure complexe à parcourir.

```
variable "user_names" {
  description = "Create IAM users with these names"
  type        = list(string)
  default     = ["neo", "trinity", "morpheus"]
}

resource "aws_iam_user" "example" {
  for_each = toset(var.user_names)
  name     = each.value
}
```

Quelques remarques importantes * On utilise la fonction `toset` pour convertir le type `list` en type `set` * On utilise la syntaxe `each.value` pour accéder à la valeur courante du pointeur dans la structure. * On peut aussi utiliser `each.key` pour utiliser la clef d'une `map` * Terraform utilise la chaîne `value` d'un set ou la `key` d'une map comme index du conteneur des ressources produites

```
aws_iam_user.example["neo"]
aws_iam_user.example["trinity"]
aws_iam_user.example["morpheus"]
```

L'intérêt de `for_each` est qu'il s'agit d'une expression qui peut être utilisée plus généralement.

Au contraire de `count` on peut l'utiliser dans toutes sortes de blocs à condition d'utiliser la directive `dynamic`.

Ici on l'utilise pour générer plusieurs tags définis dans une variable.

```
variable "
```

```
resource "aws_autoscaling_group" "example" {
  launch_configuration = aws_launch_configuration.example.name
  vpc_zone_identifier  = data.aws_subnets.default.ids
  target_group_arns    = [aws_lb_target_group.asg.arn]
  health_check_type    = "ELB"

  min_size = var.min_size
  max_size = var.max_size

  tag {
    key                 = "Name"
    value               = var.cluster_name
    propagate_at_launch = true
  }

  dynamic "tag" {
    for_each = var.custom_tags

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

La structure complexe parcourue par une expression `for_each` doit être connue à l'avance.

On ne peut donc pas utiliser une structure construire sur la base d'informations d'une ressource connues après son obtention.

L'expression `for` L'expression `for` est utilisée pour parcourir et traiter une structure complexe.

Son usage est assez proche de celui des compréhensions de liste en python.

Elle produit des listes et des maps

```
> [for k,v in [1,3,5] : "${k}:${v}"]
[
  "0:1",
  "1:3",
  "2:5",
]

> {for k,v in ["apple","kiwi","bananas"] : v => k}
{
  "apple" = 0
  "bananas" = 2
  "kiwi" = 1
}
```

On peut également filtrer les résultats avec un `if` et utilliser des fonctions.

```
> [for k,v in ["aba","aca","ada","afa","aga"] : upper("${k}:${v}") if k % 2
 ↪  == 0]
[
  "0:ABA",
  "2:ADA",
  "4:AGA",
]
```

La directive de chaînes de caractère `for` Elle est utile pour créer des contenus de chaînes à partir de structures complexes.

```
> "%{ for k,v in ["apple","kiwi","bananas"] }${k}:${v}, %{ endfor }"
"0:apple, 1:kiwi, 2:bananas, "
```

Utilisation des conditions

Avec `count` En combinant les conditionals et `count` on peut désactiver certaines ressources.

```
resource "aws_autoscaling_schedule" "scale_out_during_business_hours" {
  count = var.enable_autoscaling ? 1 : 0

  scheduled_action_name  =
  ↪  "${var.cluster_name}-scale-out-during-business-hours"
  min_size               = 2
  max_size               = 10
  desired_capacity       = 10
  recurrence             = "0 9 * * *"
  autoscaling_group_name = aws_autoscaling_group.example.name
}
```

On peut aussi faire des conditions inverses

```
resource "aws_iam_user_policy_attachment" "neo_cloudwatch_full_access" {
  count = var.give_user_cloudwatch_full_access ? 1 : 0

  user       = aws_iam_user.the_user.name
  policy_arn = aws_iam_policy.cloudwatch_full_access.arn
}

resource "aws_iam_user_policy_attachment" "neo_cloudwatch_read_only" {
  count = var.give_user_cloudwatch_full_access ? 0 : 1

  user       = aws_iam_user.the_user.name
  policy_arn = aws_iam_policy.cloudwatch_read_only.arn
}
```

Utiliser des conditions pour `for_each` On peut transformer la structure complexe à parcourir avant de la passer à `for_each`.

```
  dynamic "tag" {
    for_each = {
```

```
      for key, value in var.custom_tags:
      key => upper(value)
      if key != "Name"
    }

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

Utiliser des conditions pour la directive de chaîne `for` Dans l'exemple précédent, on avait le problème classique de la chaîne terminant par une virgule.

```
## console
> "%{ for k,v in ["apple","kiwi","bananas"] }${k}:${v}%{ if k < length(
 ↪  ["apple","kiwi","bananas"]) - 1}, %{ endif}%{ endfor }"
"0:apple, 1:kiwi, 2:bananas"
```

Rappel des objectifs

- Savoir utiliser les boucles et conditions logiques dans Terraform

1.09 Les ressources au delà des fournisseurs de cloud: fichiers, modèles (templates) et null_resource

Uptime Formation

21/09/2023

Objectifs

- Utiliser Terraform avec des fournisseurs non API
- Utiliser les fonctions null resource et template de Terraform

---

Utilisation du TP madlibs

Cet exemple utilise de nombreux aspects du langage pour produire des textes en local.

On va discuter de son fonctionnement en parcourant le code et en l'exécutant.

- Comment fait le code pour lire les fichiers source et interpoler les variables ?
- Comment fait le code pour boucler le nombre voulu de fois ?

Un autre cas intéressant : Null resource

Documentation: * https://registry.terraform.io/providers/hashicorp/null/latest/docs/resources/resource

Ce provider particulier permet d'exécuter des actions sans réellement faire appel à une API ou générer de contenu.

L'idée est de pouvoir déclencher des actions lorsqu'une partie de la recette change.

Dans l'exemple de la documentation, on détecte un changement d'adresses IP avec le ***trigger***.

Quand le contenu de l'attribut évolue, le ***provisioner*** est exécuté. En l'occurence il s'agit d'un ***remote-exec***

Les provisioners sont un moyen dans Terraform de lancer une action custom, cf. https://developer.hashicorp.com/terra

```
resource "null_resource" "configmap" {

  triggers = {
    value = aws_instance.[*].private_ip # A list of strings
  }

  provisioner "local-exec" {
```

```
    command = <<EOT
    /opt/scripts/alert.sh "New servers ${aws_instance.[*].private_ip}"
    EOT
  }
}
```

Rappel des objectifs

- Utiliser Terraform avec des fournisseurs non API

1-10 Évaluation et point sur la journée

Uptime Formation

21/09/2023

Objectifs

- Discuter du contenu de la formation
- Évaluer les progrès

Rappel

Jour 1 : Language de déploiement, dans le cloud et au-delà

Objectifs pédagogiques

- J1/ Comprendre les avantages des approches IAC et Devops
- J1/ Savoir faire des déploiements en utilisant des providers existants
- J2/ Savoir créer ses propres modules
- J2/ Savoir déployer en équipe sur le long terme

Compréhension directe Terraform

- Existence d'un projet commun : J2
- Structuration du projet : J2
- **Définition de la recette** : J1
- Passage de secrets : J2
- **Initialisation du projet** : J1
- **Planification** : J1
- Utilisation de données entre modules : J2
- **Stockage d'un état** : J1
- **Cycle de vie** : J1

Les objectifs de la journée

- Avoir une vision globale de la formation
- Comprendre Terraform dans une logique IAC et Devops
- Appréhender les outils qu'on utilisera au court de la formation
- Avoir une vue globale du fonctionnement de Terraform

- Savoir utiliser le langage utilisé par Terraform
- Comprendre et utiliser les principales commandes de la CLI terraform
- Comprendre et savoir utiliser les objets fondamentaux de Terraform
- Comprendre les variables dans Terraform
- Comprendre les outputs dans Terraform
- Connaître et savoir utiliser les fonctions intégrées dans Terraform
- Savoir utiliser les boucles et conditions logiques dans Terraform
- Utiliser Terraform avec des fournisseurs non API

Les TPs

L'évaluation

Comprendre les avantages des approches IAC et Devops

Savoir faire des déploiements en utilisant des providers existants

---

Objectifs

- Discuter du contenu de la formation
- Évaluer les progrès

2-00 Objectifs du jour

Uptime Formation

21/09/2023

Objectifs

- Comprendre le contenu pédagogique de la journée
- Faire un point sur le questionnement personnel IAC/Devops envers Terraform

---

Rappel

On est sur une formation à la fois théorique et pratique.

L'important est de comprendre les concepts et les enjeux mais aussi leur articulation dans une démarche professionnelle.

Objectifs pédagogiques

- J1/ Comprendre les avantages des approches IAC et Devops
- J1/ Savoir faire des déploiements en utilisant des providers existants
- J2/ Savoir créer ses propres modules
- J2/ Savoir déployer en équipe sur le long terme

Compréhension directe Terraform

- **Existence d'un projet commun** : J2
- **Structuration du projet** : J2
- Définition de la recette : J1
- **Passage de secrets** : J2
- Initialisation du projet : J1
- Planification : J1
- **Utilisation de données entre modules** : J2
- Stockage d'un état : J1
- Cycle de vie : J1

Questionnement personnel

- Quelles questions avez-vous ?
- Qu'est-ce que vous avez appris / compris ? Ou pas appris / compris ?

Rappel des objectifs

- Comprendre le contenu pédagogique de la journée
- Faire un point sur le questionnement personnel IAC/Devops envers Terraform

2-01 Qu’est-ce que l’état Terraform ?

Uptime Formation

21/09/2023

Objectifs

- Comprendre le fonctionnement et les limitations de l'état Terraform

---

Qu'est ce que l'état de Terraform (state file) ?

Voici ce que contient le fichier `.terraform/terraform.tfstate` après une exécution d'exemple simple.


```
{
  "version": 4,
  "terraform_version": "1.2.3",
  "serial": 1,
  "lineage": "86545604-4aa5-7463-e9e8-a2a221de98d2",
  "outputs": {},
  "resources": [
    {
      "mode": "managed",
      "type": "aws_instance",
      "name": "example",
      "provider": "provider[\"registry.terraform.io/hashicorp/aws\"]",
      "instances": [
        {
          "schema_version": 1,
          "attributes": {
            "ami": "ami-0fb653ca2d3203ac1",
            "availability_zone": "us-east-2b",
            "id": "i-843875430bc4bbe5b",
            "instance_state": "running",
            "instance_type": "t2.micro",
            "(...)": "(truncated)"
          }
        }
      ]
    }
  ]
}
```

En utilisant ce format JSON, Terraform sait qu'une ressource avec le type aws_instance et le nom example correspond à une instance EC2 dans votre compte AWS avec l'ID i-843875430bc4bbe5b.

Chaque fois que vous exécutez Terraform, il peut récupérer le dernier statut de cette instance EC2 auprès d'AWS et le comparer à ce qui se trouve dans vos configurations Terraform pour déterminer les modifications à appliquer.

En d'autres termes, la sortie de la commande plan est un diff entre le code sur votre ordinateur et l'infrastructure déployée dans le monde réel, comme découvert via les ID dans le fichier d'état.

Attention, seul Terraform doit manipuler ce fichier !

Le format de fichier d'état est une API privée destinée uniquement à un usage interne dans Terraform. Vous ne devez jamais modifier les fichiers d'état Terraform à la main ou écrire du code qui les lit directement.

Seules les commandes `terraform import` ou `terraform state` sont à utiliser pour manipuler ce fichier depuis l'extérieur de Terraform.

Rappel des objectifs

- Comprendre le fonctionnement et les limitations de l'état Terraform

2.02 Gestion de l’état dans Terraform pour le DevOps

Uptime Formation

21/09/2023

Objectifs

- Savoir gérer l'état de Terraform pour les pratiques DevOps
- Stocker et partager l'état dans une équipe

Quand les problèmes commencent

Dans la pratique DevOps, la capacité à agir librement avec des responsabilités au sein d'une équipe est la clé.

Mais l'état Terraform étant un fichier statique introduit plusieurs problème.

Travail en commun Si j'ai plusieurs partenaires dans le projet, il faut qu'on puisse travailler tous sur la même infrastructure.

On a donc besoin de partager le fichier d'état.

Je peux certes ajouter le fichier tfstate au dépôt git, mais c'est une mauvaise formule, car il est quasi certain qu'à un moment ça ne marchera pas. * deux personnes vont lancer une commande en simultané (lock) * quelqu'un va oublier de commit * quelqu'un va utiliser un état ancien * des secrets vont se retrouver accessibles

---

Drift Le dérapage (drift) de l'infrastructure survient lorsqu'il existe une différence entre ce qui se trouve dans le tfstate et ce qui est déployé. * dans le fichier d'état * et dans les clouds des fournisseurs

Le drift peut survenir * à cause d'opérations manuelles * en utilisant un tfstate périmé * en raison d'incidents techniques

---

Environnements Le DevOps recommande évidemment d'avoir plusieurs environnements aussi similaires les uns des autres que possible.

Chaque environnement représente une infrastructure spécifique, et donc un fichier d'état différent.

Il y a deux façons courantes d'isoler le fichier d'état dans Terraform.

Isolement via les espaces de travail

Les espaces de travail (workspaces en anglais) sont des environnements isolés dans Terraform.

Par défaut Terraform crée un workspace nommé `default`.

```
$ terraform workspace show
```

On peut générer autant de workspaces que requis par le nombre d'environnements.

```
$ terraform workspace new test1
$ terraform workspace select test1
```

Chaque espace de travail possède son propre fichier d'état géré séparément.

Créer un nouveau workspace va déployer une nouvelle architecture d'une même configuration Terraform pour ce nouvel environnement.

Cependant les workspaces ne sont pas des solutions idéales

Les fichiers d'état de tous vos espaces de travail sont stockés dans le même backend (par exemple, le même bucket S3).

Cela signifie que vous utilisez les mêmes contrôles d'authentification et d'accès pour tous les espaces de travail, ce qui est l'une des principales raisons pour lesquelles les espaces de travail ne sont pas un mécanisme approprié pour isoler les environnements (par exemple, isoler la mise en scène de la production).

Les espaces de travail ne sont pas visibles dans le code ou sur le terminal, sauf si vous exécutez les commandes de `terraform workspace`.

Lorsque vous parcourez le code, un module qui a été déployé dans un espace de travail ressemble exactement à un module déployé dans 10 espaces de travail. Cela rend la maintenance plus difficile, car vous n'avez pas une bonne image de votre infrastructure.

En réunissant les deux éléments précédents, le résultat est que les espaces de travail peuvent être assez sujets aux erreurs.

Le manque de visibilité permet d'oublier facilement dans quel espace de travail vous vous trouvez et de déployer accidentellement des modifications dans le mauvais (par exemple, exécuter accidentellement `terraform destroy` dans un espace de travail de "production" plutôt qu'un espace de travail de "dev").

Parce que vous devez utiliser le même mécanisme d'authentification pour tous les espaces de travail, vous n'avez pas d'autres couches de défense pour vous protéger contre de telles erreurs.

On va voir plus loin avec Terragrunt comment trouver une solution à ce problème.

Les backends Documentation : - https://developer.hashicorp.com/terraform/language/settings/backends/configu

La meilleure façon de partager les fichiers d'état consiste à utiliser la prise en charge intégrée de Terraform pour les backends distants.

Un backend Terraform détermine la façon dont Terraform charge et stocke l'état. Le backend par défaut, que vous avez utilisé tout ce temps, est le backend local, qui stocke le fichier d'état sur votre disque local.

Les backends distants vous permettent de stocker le fichier d'état chez Amazon S3, Azure Storage, Google Cloud Storage et Terraform Cloud / Enterprise de HashiCorp.

La mise en oeuvre est simple (en apparence).

```
terraform {
  backend "<BACKEND_NAME>" {
    [CONFIG...]
  }
}
```

Les avantages des backends Après avoir configuré un backend distant, Terraform chargera automatiquement le fichier d'état à partir de ce backend chaque fois que vous exécuterez un plan ou appliquerez.

Il stockera automatiquement le fichier d'état dans ce backend après chaque application, il n'y a donc aucun risque d'erreur manuelle.

Verrouillage

Un bon backend distant prend en charge nativement le verrouillage.

Pour exécuter terraform apply, Terraform acquiert automatiquement un verrou ; si quelqu'un d'autre est déjà en cours d'exécution, il aura déjà le verrou et vous devrez attendre.

Vous pouvez exécuter apply avec le paramètre -lock-timeout=(TIME) pour demander à Terraform d'attendre jusqu'à TIME pour qu'un verrou soit libéré (par exemple, -lock-timeout=10m attendra 10 minutes).

Secrets

Un bon backend distant prend en charge nativement le chiffrement en transit et le chiffrement au repos du fichier d'état.

De plus, ces backends exposent généralement des moyens de configurer les autorisations d'accès (par exemple, en utilisant des politiques IAM avec un compartiment Amazon S3), afin que vous puissiez contrôler qui a accès à vos fichiers d'état et aux secrets qu'ils peuvent contenir.

Il serait encore préférable que Terraform prenne en charge nativement le chiffrement des secrets dans le fichier d'état, mais ces backends distants réduisent la plupart des problèmes de sécurité, étant donné qu'au moins le fichier d'état n'est stocké en texte brut sur le disque nulle part.

Prise en compte des workspaces

Un bon backend distant prend en charge nativement la gestion des workspaces.

Documentation : - https://developer.hashicorp.com/terraform/language/state/workspaces

Ces backends prennent en charge plusieurs espaces de travail nommés, ce qui permet d'associer plusieurs états à une seule configuration.

La configuration n'a toujours qu'un seul backend, mais vous pouvez déployer plusieurs instances distinctes de cette configuration sans configurer un nouveau backend ni modifier les informations d'authentification.

Sans workspace, Terraform sauverait le fichier d'état dans

```
s3://bucket/key
```

Avec un workspace, cela devient automatiquement

```
s3://bucket/env:/$workspace/key
```

Rappel des objectifs

- Savoir gérer l'état de Terraform pour les pratiques DevOps
- Stocker et partager l'état dans une équipe

2-03 Comment gérer les secrets avec Terraform

Uptime Formation

21/09/2023

Objectifs

- Comprendre les enjeux de sécurité des données sensibles dans Terraform
- Savoir passer et protéger des informations sensibles dans Terraform

---

Sécurité et données en *clear text*

Est-il utile de rappeler combien de fois des identifiants AWS en clair ont été trouvés dans des repos sur Github?

Il s'agit de cas assez extrême, mais dont les conséquences sont révélatrices du danger à laisser apparaître des données sensible en clair dans le code.

Une fois dépassés ces cas simples, on va voir qu'il n'est pour autant pas forcément simple de sécuriser ces informations.

Il existe une règle importante et simple

> NE JAMAIS STOCKER DE DONNÉES SENSIBLES EN CLAIR.

Note: C'est valable pour les données stockées dans git, mais pas seulement.

D'autres informations cruciales à retenir Les fichiers d'état et les fichiers de planificationde Terraform contiennent forcément** des données sensibles en clair.**

C'est pourquoi il faut être vigilant sur la façon dont ils sont stockés (cf. backends)

C'est aussi pourquoi on doit absolument éviter de les ajouter dans Git.

Le stockage des secrets doit être un équilibre entre solution pour développeurs et solution pour automatisation.

À terme l'objectif du DevOps est d'automatiser au maximum.

Si on choisit une solution qui ne marche que sur les postes individuels, ça devient un blocage.

On ne peut vraiment faire confiance à personne.

Le disque dur des membres de l'équipe est-il chiffré en cas de vol ? (cf. Attaque LastPass sur un poste admin)

Que se passe-t-il si un membre de l'équipe devient "hostile" ?

Est-ce que les stagiaires doivent avoir accès aux secret de la production en agissant sur l'environnement de dev ?

La sécurité est un processus qui doit tenir compte de nombreux facteurs.

Quel est votre modèle de menace ? Tout le monde n'est pas une banque ou la CNAM.

Quels sont les moyens financiers et techniques de votre structure ?

Quelle est la taille de l'équipe et la fréquence de modifications des mots de passe ?

La sécurité est un processus qui résulte d'une entente DevOps.

La bonne solution sera probablement toujours dépendante d'une certaine manière de fonctionner dans l'application.

Par exemple en utilisant des variables d'environnement (cf. 12-Factor App) pour récupérer des secrets déployés au dernier moment dans les instances.

Différentes solutions pour passer des secrets dans Terraform

Les deux méthodes les plus courantes pour stocker des secrets sont

- Stockage dans un fichier chiffré
- Stockage dans un service centralisé

Stockage dans un fichier chiffré **Les stockages de secrets basés sur des fichiers stockent les secrets dans des fichiers chiffrés, qui sont généralement ajoutés dans git.**

Pour chiffrer les fichiers, vous avez besoin d'une clé de chiffrement. Cette clé est elle-même un secret !

Cela crée une petit souci : comment stocker cette clé en toute sécurité ?

L'anti pattern dans cette situation devient d'avoir le même mot de passe pour toutes les clefs de chiffrement.

Assez rapidement tout le monde dans l'équipe connaît le même mot de passe (devops101 par exemple?), qui n'est plus sécure et modifiable.

La solution la plus courante à cette énigme consiste à stocker la clé dans un kKey Management Service (KMS) fourni par votre fournisseur de cloud, tel qu'AWS KMS, GCP KMS ou Azure Key Vault.

Une autre solution consiste à utiliser des clés PGP, et de chiffrer le fichier avec.

Mais les clefs PGP ont un mode de fonctionnement un peu complexe, et il faut que les utilisateurs stockent le mot de passe de leur clef PGP ailleurs… on ne fait que repousser le problème.

Il faut que l'utilisateur ait un système de gestion de mots de passe personnel (ex: Keepass) fiable, redondé, etc..

Stockage dans un service centralisé Les stockages de secrets centralisés sont généralement des services Web auxquels vous parlez sur le réseau qui chiffrent vos secrets et les stockent dans une base de données.

Pour chiffrer ces secrets, ces magasins de secrets centralisés ont besoin d'une clé de chiffrement. Généralement, la clé de chiffrement est gérée par le service lui-même, ou le service s'appuie sur le KMS d'un fournisseur de cloud.

Les services centralisés

À terme la bonne solution est d'utiliser un service centralisé pensé pour les secrets d'infrastructure.

Voici une liste des solutions, on voit que les fournisseurs de cloud proposent directement cette solution.

- HashiCorp Vault
- AWS Secrets Manager
- Google Secrets Manager
- Azure Key Vault
- Confidant

Le choix d'un service centralisé dépend de facteurs contextuels, comme évoqué plus haut.

Utiliser le service d'un fournisseur de cloud est rapide et simple, mais il implique une dépendance lourde à ce fournisseur.

Utiliser une solution tierce est plus complexe et demande du support, mais elle offre une certaine liberté.

Vault + AWS dans un lab Pour illustrer le stockage de secrets dans un service centralisé, on va opter pour Vault dans cet exemple.

Il se trouve que c'est la même société Hashicorp qui produit Terraform et Vault, et que son intégration dans AWS est assez simple.

Lancement de Vault via Docker

```
$ docker run --rm -d --name vault -p 8200:8200 hashicorp/vault
$ docker logs vault
```

Lancement des opérations

On peut suivre la doc officielle de Terraform ici :

> https://developer.hashicorp.com/terraform/tutorials/secrets/secrets-vault

En utilisant le code fourni dans le projet Git de la formation : `TP2.03`

Un cas simple… voir simpliste

On voit que cet exemple est un peu limité, car il utilise des méthodes pas très sécures.

- fait appel à `terraform_remote_state` qui permet d'aller chercher des informations dans un fichier d'état tiers.
- utilise un compte d'administration pour générer des secrets

Cependant, il est intéressant dans la mesure où dans le fond il est souhaitable d'éviter d'utililser des tokens de durée longue pour piloter les infrastructures.

Sans cette logique, on peut stocker des authentifiants AWS dans Vault avec

```
provider "vault" {
  address = var.vault_address
}

data "vault_aws_access_credentials" "creds" {
  backend = "aws"
  role    = "prod-role"
}

provider "aws" {
  access_key = data.vault_aws_access_credentials.creds.access_key
  secret_key = data.vault_aws_access_credentials.creds.secret_key
  region     = "us-west-2"
}
```

Et ensuite ?

Voici comment on peut utiliser plus raisonnablement Vault comme source de secrets.

On part du principe qu'on a mis en place :

- une méthode d'authentification sécure ;
- un utilisateur avec un groupe et des droits définiis ;
- un espace de secrets auquel il a accès avec une

Documentation : * Vault https://developer.hashicorp.com/vault/docs/auth * Vault provider for Terraform https://registry.terraform.io/providers/hashicorp/vault/latest/docs#

```
## Set by CI/CD or on Dev machine via ENV VAR
```

```
variable vault_url {}
variable  vault_token {}
variable  vault_aws_role {}

provider "vault" {
    address = var.vault_url
    # Token conf
    token = var.vault_token
    # OR Aws conf
    auth_login_aws = true
    role = var.vault_aws_role
}

data "vault_generic_secret" "app_secrets" {
  path = "tf/my_app/${environment}/"
}

provider "aws" {
  # [CONFIG ...]
}

## Use the secret in configuration
resource "aws_instance" "my_server" {
  ami           = "ami-059af0b76ba105e7e"
  instance_type = "t2.nano"
  provisioner "file" {
    content     = "PASSWD = ${PASSWD}"
    destination = "/etc/environ"
  }
}
```

Informations sur les provisioners : > https://developer.hashicorp.com/terraform/language/resources/provisioners/fil

Et ensuite c'est aux personnes en charge du développement d'utiliser la variable d'environnement comme source d'information.

Un mot sur Kubernetes

La gestion native des secrets est une des raisons pour lesquelles Kubernetes a un succès grandissant.

Dans Terraform, quoi qu'on fasse il restera toujours des traces de données sensibles en clair.

Rappel des objectifs

- Comprendre les enjeux de sécurité des données sensibles dans Terraform
- Savoir passer et protéger des informations sensibles dans Terraform

2-04 Qu’est ce que larchitecture en modules de Terraform

Uptime Formation

21/09/2023

Objectifs

- Comprendre les modules dans Terraform
- Savoir créer ses propres modules

Les différentes formes de structuration du code

Avant d'aborder les modules, voyons quels sont les structures utilisées pour organiser le code.

La structure "monolithique" Tout le code est dans un seul fichier, sans séparation entre responsabilités.

```
prod
├─ outputs.tf
├─ main.tf
└─ variables.tf
```

Rapidement elle peut devenir insuffisante.

La structure "flat" Tout le code est dans un seul dossier, avec chaque partie "logique" de l'infrastructure dans un fichier distinct.

```
prod
├─ database.tf
├─ load-balancer.tf
├─ outputs.tf
├─ variables.tf
└─ webserver.tf
```

Elle résulte d'une analyse des composants logiques de la recette.

La structure en recettes séparées **Les différents composants sont séparés dans des dossiers différents.**

```
└── prod
    ├── webserver
    │   ├── main.tf
    │   ├── outputs.tf
    │   └── variables.tf
    └── database
        ├── main.tf
        ├── outputs.tf
        └── variables.tf
```

Elle permet de séparer les problèmes, mais elle implique une duplication des ressources avec par exemple des dossiers `.terraform` distincts.

La structure "liens symboliques" **On déploie le code dans plusieurs dossiers, avec les fichiers communs définis comme des liens symboliques.**

```
├── dev
│   ├── infra_override.tf
│   ├── infra.tf -> ../common/infra.tf
│   ├── outputs.tf -> ../common/outputs.tf
│   ├── state.tf
│   ├── terraform.tfvars
│   └── variables.tf -> ../common/variables.tf
└── common
    ├── infra.tf
    ├── outputs.tf
    └── variables.tf
```

Cette structure fonctionne en chargeant les variables locales et en surchargeant des parties de la recette avec des fichiers via `_override.tf`

À noter également, le système de fichier de Windows ne supporte pas historiquement les liens symboliques, et il faut ajouter le mode developpeur pour y accéder.

Les modules dans Terraform

Documentation: - https://developer.hashicorp.com/terraform/language/modules

Les modules sont des packages de code autonomes qui vous permettent de créer des composants réutilisables en regroupant des ressources associées.

C'est un compossant essentielle de toute architecture d'infrastructure logicielle Terraform mature, car les modules offrent de nombreux avantages

- Lisibilité
- Réusabilité
- Testabilité

Vous n'avez pas besoin de savoir comment fonctionne un module pour pouvoir l'utiliser ; il suffit de savoir paramétrer les entrées et les sorties. Les modules sont des outils utiles pour promouvoir l'abstraction logicielle et la réutilisation du code.

On va utiliser un exemple standard d'infrastructure modulaire.

> TP2.05-modules

Dans le premier `main.tf`, notez l'usage de

```
module "<NAME>" {
  source = "<SOURCE>"

  [CONFIG ...]
}
```

On observe que ce module est principalement composé d'autres modules.

Ce modèle est connu sous le nom de composition logicielle : la pratique consistant à décomposer un code volumineux et complexe en sous-systèmes plus petits.

HashiCorp recommande fortement que chaque module suive certaines conventions de code connues sous le nom de structure de module standard.

Au minimum, cela signifie avoir trois configurations Terraform fichiers de figuration par module.

- main.tf—le point d'entrée principal
- outputs.tf—déclarations pour toutes les valeurs de sortie
- variables.tf—déclarations pour toutes les variables d'entrée

---

Le module racine est le module de niveau supérieur.

C'est là que les variables d'entrée fournies par l'utilisateur sont configurées et que les commandes Terraform telles que `terraform init` et `terraform apply` sont exécutées.

Le module racine ne fait pas grand-chose à part déclarer des modules de composants et leur permettre de transmettre des données entre eux.

Du point de vue du module racine, les modules sont des fonctions avec des effets secondaires (c'est-à-dire des fonctions non pures), qui sont les ressources provisionnées à la suite de l'application de terraform.

Nous pouvons voir que les sorties de chaque module "s'écoulent" via le module racine qui fait appel à leur outputs.

```
module.<MODULE_NAME>.<OUTPUT_NAME>
```

Le module avec le moins de dépendances est appelé en premier, et ses sorties sont transmises au module avec le plus de dépendances (l'équilibreur de charge) par le module racine.

Bonnes pratiques pour le module racine Il est conseillé d'intégrer certains fichiers sont dans le module racine : * versions.tf

```
terraform {
  required_version = ">= 0.14"
  required_providers {
    aws = "= 3.28"
  }
}
```

- providers.tf

```
provider "google" {
  project = var.project_id
  region = var.region
}
```

- README.md

Ces fichiers définissent la documentation et configuration de base de votre recette.

Cependant rien n'interdit un module d'avoir sa propre documentation ou de surcharger un provider (par exemple pour utiliser une version différente).

Sources des modules

On peut utiliser les modules disponibles sur le registre Terraform, créer les siennes et utiliser un mélange des deux.

Créer ses propres modules La plupart du temps on crée ses modules au sein de son projet, la syntaxe dans ce cas fait appel à un dossier relatif.

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

On peut ensuite ajouter ce module dans le Registry Terraform, et en réalité on peut aussi charger des modules depuis Github, des buckets S3 et encore d'autres.

Créer un module Terraform est très simple : tout ensemble de fichiers de configuration Terraform dans un dossier est un module.

Toutes les configurations que vous avez écrites jusqu'à présent étaient techniquement des modules, bien que pas particulièrement intéressants, puisque vous les avez déployés directement : si vous exécutez apply directement sur un module, il est appelé module racine.

Pour voir de quoi les modules sont réellement capables, vous devez créer un module réutilisable, c'est-à-dire un module destiné à être utilisé dans d'autres modules.

Structurer le module **Comme on l'a vu la norme est de mettre**

- La génération de ressources dans le fichier `main.tf`
- Les inputs dans le fichier `variables.tf`
- Les outputs dans le fichier `outputs.tf`

Rendre son module customisable **La bonne gestion des variables du module est essentielle, car la capacité à pouvoir être réutilisable dépend d'elle.**

Il faut donc essayer de multiplier intelligemment les variables, avec des défauts intelligents.

Éviter les pièges **Le premier piège apparaît quand on fait appel à des fonctions utilisant les fichiers locaux, comme `template`.**

Terraform utilise le dossier courant comme référence par défaut des chemins relatifs, mais ça ne fonctionnera plus quand vous serez dans un module.

On utilise donc des variables spéciales `path.*` qui permettent à Terraform de charger les fichiers correctement.

```
user_data = templatefile("${path.module}/user-data.sh", {
  server_port = var.server_port
  db_address  = data.terraform_remote_state.db.outputs.address
  db_port     = data.terraform_remote_state.db.outputs.port
})
```

Le deuxième piège survient avec les déclarations de bloc dans les ressources.

Avec la déclaration suivante dans un module, il est impossible de surcharger le `ingress` déclaré dans le module, car le bloc est déterminé.

```
resource "aws_security_group" "alb" {
  name = "${var.cluster_name}-alb"

  ingress {
    from_port   = local.http_port
    to_port     = local.http_port
    protocol    = local.tcp_protocol
    cidr_blocks = local.all_ips
  }
  ...
}
```

On utillise donc des ressources séparées, qui seront aggrégées au moment de l'exécution.

C'est pourquoi on trouve dans les listes d'objets mis à disposition par les providers une myriade de sous-éléments.

```
resource "aws_security_group" "alb" {
  name = "${var.cluster_name}-alb"
}

resource "aws_security_group_rule" "allow_http_inbound" {
  type              = "ingress"
  security_group_id = aws_security_group.alb.id

  from_port   = local.http_port
  to_port     = local.http_port
  protocol    = local.tcp_protocol
  cidr_blocks = local.all_ips
}
```

En exportant le Security Group…

```
output "alb_security_group_id" {
  value       = aws_security_group.alb.id
  description = "The ID of the Security Group attached to the load balancer"
}
```

On peut le réutiliser pour ajouter de nouvelles règles hors module

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

```
resource "aws_security_group_rule" "allow_testing_inbound" {
  type              = "ingress"
  security_group_id = module.webservers.alb_security_group_id

  from_port   = 12345
  to_port     = 12345
  protocol    = "tcp"
  cidr_blocks = ["0.0.0.0/0"]
}
```

Rappel des objectifs

- Comprendre les modules dans Terraform
- Savoir créer ses propres modules

2-05 Bonnes pratiques d’organisation des fichiers et dossiers d’un projet

Uptime Formation

21/09/2023

Objectifs

- Comprendre comment bien structurer ses projets dans Terraform avec Terragrunt

À l'inverse, le code d'une application peut évoluer plusieurs fois par jour, et dans une infrastructure élastique les noeuds de calculs sont par nature transients.

On va appeler "dynamiques" ces parties de l'infrastructure.

Il y a a plusieurs avantages à séparer dans l'IAC les parties statiques et dynamiques.

- Assurance : éviter le risque de détruire par erreur une base de données
- Rapidité : éviter de perdre du temps dans de la planification et du déploiement
- Sobriété : éviter les surconsommations d'énergie / réseau / disque

---

Conséquence 1 : séparer le code des différentes parties

On va segmenter le projet en modules différents.

Dans notre exemple, le déploiement de la base de données et la gestion des serveurs web sont dans deux modules séparés.

Conséquence 2 : utiliser des méthodes de déploiement légères pour les parties dynamiques

On va utiliser des outils adaptés à de la CI/CD pour les déploiements réguliers.

La bonne solution dépend des compétences techniques de l'équipe.

Aujourd'hui on conseillerait d'utiliser Kubernetes et des images de conteneurs comme méthode de déploiement du code.

Mais K8s n'est pas simple, et dans des cas plus raisonnables on peut utiliser ansible en coordination avec des utilitaires de déploiement continu.

Ansible va utiliser un inventaire dynamique, basé sur les API du cloud provider voire même en utilisant l'état Terraform.

Les utilitaires de déploiement vont récupérer à la demande les nouvelles versions du code, qui peut être déployé via

- des conteneurs : docker-compose
- des `git pull` : capistrano et ses alternatives

Point sur l'architecture modulaire, les workspaces et les fichiers d'états Notre vision de Terraform a évolué : il y a des limitations d'usage dans ces techniques un peu complexes, et on voit qu'il faut bien structurer son code à terme.

Nous allons voir que sur cette base le logiciel `terragrunt` va nous permettre de résoudre plusieurs problèmes avec quelques principes de base.

- Don't Repeat Yourself (DRY) : éviter la répétition du code
- Usage de conventions structurelles : travailler uniquement avec des modules
- Standardisation des environnements : gestions des états d'infrastructure

La ligne de commande terragrunt

À la base, Terragrunt est une "surcouche" qui va piloter terraform pour vous.

Les versions de Terragrunt sont à aligner avec celles de Terraform.

Pour lancer Terragrunt, on utilisera les mêmes actions que Terraform

```
$ terragrunt init
$ terragrunt plan
$ terragrunt apply
$ terragrunt destroy
```

Mais là où Terragrunt va présenter une différence, c'est qu'il peut opérer sur plusieurs recettes à la fois.

```
$ terragrunt run-all apply
$ terragrunt run-all destroy
```

Terragrunt : Terraform sous stéroïdes Avec Terragrunt, on passe d'une structure redondante :

```
└── live
    ├── prod
    │   ├── app
    │   │   └── main.tf + variables.tf + output.tf + ...
    │   ├── mysql
    │   │   └── main.tf + variables.tf + output.tf + ...
    │   └── vpc
    │       └── main.tf + variables.tf + output.tf + ...
    ├── qa
    │   ├── app
    │   │   └── main.tf + variables.tf + output.tf + ...
    │   ├── mysql
    │   │   └── main.tf + variables.tf + output.tf + ...
    │   └── vpc
    │       └── main.tf + variables.tf + output.tf + ...
    └── stage
        ├── app
        │   └── main.tf + variables.tf + output.tf + ...
        ├── mysql
        │   └── main.tf + variables.tf + output.tf + ...
        └── vpc
            └── main.tf + variables.tf + output.tf + ...
```

à ça

```
└── live
    ├── terragrunt.hcl
    ├── prod
    │   ├── app
    │   │   └── terragrunt.hcl (v 1.7.4)
    │   ├── mysql
    │   │   └── terragrunt.hcl (v 2.1.0)
    │   └── vpc
    │       └── terragrunt.hcl (v 1.8.2)
    ├── qa
    │   ├── app
    │   │   └── terragrunt.hcl (v 1.7.5)
    │   ├── mysql
    │   │   └── terragrunt.hcl (v 2.1.0)
    │   └── vpc
    │       └── terragrunt.hcl (v 1.8.3)
```

```
└── stage
    ├── app
    │   └── terragrunt.hcl (v 1.7.7)
    ├── mysql
    │   └── terragrunt.hcl (v 2.1.2)
    └── vpc
        └── terragrunt.hcl (v 1.8.3)
```

Gestion des dépendances Chaque fichier `terragrunt.hcl` contient les informations nécessaires pour gérer l'infrastructure via Terraform.

```
terraform {
  # Deploy version v0.0.3 in stage
  source = "git::git@github.com:foo/modules.git//app?ref=v0.0.3"
}

inputs = {
  instance_count = 3
  instance_type  = "t2.micro"
}
```

Gestion des backends de fichiers d'état On a vu une limitation des backends : impossible d'utiliser des variables quand on déclare un backend de remote state.

Et il fallait créer le bucket S3 pour stocker notre remote state séparément.

Avec Terragrunt, on a une syntaxe qui va prendre ça en charge.

```
remote_state {
  backend = "s3"
  generate = {
    path      = "backend.tf"
    if_exists = "overwrite"
  }
  config = {
    bucket         = "my-terraform-state"
```

```
    key            = "${path_relative_to_include()}/terraform.tfstate"
    region         = "us-east-1"
    encrypt        = true
    dynamodb_table = "my-lock-table"
  }
}
```

Et d'autres avantages... Parmi les aspects pratiques de Terraform, on peut utiliser des "hooks" qui déclenchent des actions à des moments clefs.

```
terraform {
 # Pull the terraform configuration at the github repo "acme/infrastructure-
modules", under the subdirectory
  # "networking/vpc", using the git tag "v0.0.1".
 source = "git::git@github.com:acme/infrastructure-modules.git//networking/vpc?re

 # For any terraform commands that use locking, make sure to configure a lock timeout
  extra_arguments "retry_lock" {
    commands  = get_terraform_commands_that_need_locking()
    arguments = ["-lock-timeout=20m"]
  }

 # You can also specify multiple extra arguments for each use case. Here we configure
  # `common.tfvars` var file located by the parent terragrunt config.
  extra_arguments "custom_vars" {
    commands = [
      "apply",
      "plan",
      "import",
      "push",
      "refresh"
    ]

   required_var_files = ["${get_parent_terragrunt_dir()}/common.tfvars"]
  }
```

```
  # The following are examples of how to specify hooks

  # Before apply or plan, run "echo Foo".
  before_hook "before_hook_1" {
    commands     = ["apply", "plan"]
    execute      = ["echo", "Foo"]
  }

 # Before apply, run "echo Bar". Note that blocks are ordered, so this hook will run
 # "echo Foo". In this case, always "echo Bar" even if the previous hook failed.
  before_hook "before_hook_2" {
    commands     = ["apply"]
    execute      = ["echo", "Bar"]
    run_on_error = true
  }


 # After running apply or plan, run "echo Baz". This hook is configured so that it wi
  # or plan failed.
  after_hook "after_hook_1" {
    commands     = ["apply", "plan"]
    execute      = ["echo", "Baz"]
  }

 # After an error occurs during apply or plan, run "echo Error Hook executed". This h
  # after any error, with the ".*" expression.
  error_hook "error_hook_1" {
    commands  = ["apply", "plan"]
    execute   = ["echo", "Error Hook executed"]
    on_errors = [
      ".*",
    ]
  }

}
```

Gestion de l'architecture DRY **Pour éviter de se répéter entre configurations d'environnement, Terragrunt utilise une directive `include`.**

```
└── live
    ├── terragrunt.hcl
    ├── _env
    │   ├── app.hcl
    │   ├── mysql.hcl
    │   └── vpc.hcl
    ├── prod
    │   ├── app
    │   │   └── terragrunt.hcl
    │   ├── mysql
    │   │   └── terragrunt.hcl
    │   └── vpc
    │       └── terragrunt.hcl
```

```
## File: _env:app.hcl
terraform {
  source = "github.com/<org>/modules.git//app?ref=v0.1.0"
}

dependency "vpc" {
  config_path = "../vpc"
}

dependency "mysql" {
  config_path = "../mysql"
}

inputs = {
  basename       = "example-app"
  vpc_id         = dependency.vpc.outputs.vpc_id
  subnet_ids     = dependency.vpc.outputs.subnet_ids
  mysql_endpoint = dependency.mysql.outputs.endpoint
}

## File: prod/app/terragrunt.hcl
include "root" {
  path = find_in_parent_folders()
}

include "env" {
  path = "${get_terragrunt_dir()}/../../_env/app.hcl"
```

```
}

inputs = {
  env = "prod"
}
```

Rappel des objectifs

- Comprendre comment bien structurer ses projets dans Terraform avec Terragrunt

2-06 Créer, refactorer et réutiliser du code avec des modules

Uptime Formation

21/09/2023

Objectifs

- Savoir gérer sur le long terme un projet modulaire Terraform

Le “style Terraform”

Il existe un guide de style Terraform, augmenté par Terragrunt

Documentation : * https://developer.hashicorp.com/terraform/language/syntax/style * https://docs.gruntwork.io/gu
style-guide/

On peut reformater automatiquement son code avec la commande fmt

```
$ terraform fmt -recursive
```

Utiliser un style commun est la première étape pour rendre son code gérable en équipe.

- Indentez deux espaces pour chaque niveau d’imbrication.
- Lorsque plusieurs arguments avec des valeurs sur une seule ligne apparaissent sur des lignes consécutives au même niveau d’imbrication, alignez leurs signes égal :

```
## BAD
ami = "abc123"
type_instance = "t2.micro"

## GOOD
ami             = "abc123"
type_instance   = "t2.micro"
```

- Dans un corps de bloc les arguments vont ensemble en haut, séparés par une ligne des blocs imbriqués.
- Utilisez des lignes vides pour séparer des groupes logiques d’arguments dans un bloc.
- Pour les blocs contenant à la fois des arguments et des “méta-arguments” (tels que définis par la sémantique du langage Terraform), répertoriez d’abord les méta-arguments et séparez-les des autres arguments par une ligne vide. Placez les blocs de méta-arguments en dernier et séparez-les des autres blocs par une ligne vide.

```
ressource "aws_instance" "exemple" {
   count = 2 # méta-argument en premier

   ami = "abc123"
   type_instance = "t2.micro"

   interface réseau {
     # ...
   }

  lifecycle { # bloc de méta-arguments en dernier
    create_before_destroy = true
  }
}
```

- Utilisez des lignes vides pour séparer les blocs de niveau supérieur et les blocs imbriqués sauf si ces derniers sont du même type .
- Limitez à 120 colonnes la longueur de ligne, sauf pour les chaînes de description dans les blocs de variable et de sortie, où les chaînes à une seule ligne sont préférées.
- Utilisez le *camel_case* pour les étiquettes de bloc, les variables et les sorties, ex: `example_instance` et non `ExampleInstance` ou `example-instance`.

Créer un code modulaire de qualité

Les fondamentaux pour passer avoir une architecture Terraform prête pour la production implique d'avoir :

- des modules atomiques
- des modules composables
- des modules testables
- des modules versionnés

Les fonctions / contenus d'un module Il est important avant d'écrire un module d'avoir en tête une liste de toutes les opérations potentielles dans le Devops.

Type	Description	Outils
Installer	Installer les fichiers binaires du logiciel et toutes les dépendances.	Bash, Ansible, Docker, Packer
Configurer	Configurer le logiciel lors de l'exécution. Inclut les paramètres de port, les certificats TLS, la découverte de services, les leaders, les suiveurs, la réplication, etc.	Chef, Ansible, Kubernetes
Mise à disposition	Provisionner l'infrastructure. Inclut les serveurs, les équilibreurs de charge, la configuration réseau, les paramètres de pare-feu, les autorisations IAM, etc.	Terraform, CloudFormation
Déployer	Déployer le service au-dessus de l'infrastructure. Déployer les mises à jour sans temps d'arrêt. Inclut les déploiements bleu-vert, roulant et canari.	ASG, Kubernetes, ECS
Évolutivité	Échelle vers le haut et vers le bas en réponse à la charge. Évoluer horizontalement (plus de serveurs) et/ou verticalement (plus gros serveurs).	Auto scaling, réplication

Type	Description	Outils
Performances	Optimiser l'utilisation du CPU, de la mémoire, du disque, du réseau et du GPU. Comprend le réglage des requêtes, l'analyse comparative, les tests de charge et le profilage.	Dynatrace, Valgrind, VisualVM
Réseau	Configurer les adresses IP statiques et dynamiques, les ports, la découverte de services, les pare-feu, le DNS, l'accès SSH et l'accès VPN.	VPC, pare-feu, Route 53
Sécurité	Chiffrement en transit (TLS) et sur disque, authentification, autorisation, gestion des secrets, durcissement des serveurs.	ACM, Let's Encrypt, KMS, Vault
Métriques	Métriques de disponibilité, métriques commerciales, métriques d'application, métriques de serveur, événements, observabilité, traçage et alerte.	CloudWatch, Datadog
Journaux	Rotation des journaux sur le disque. Agrégation des données du journal dans un emplacement central.	Pile élastique, Sumo Logic
Documents	Documentez du code, de votre architecture et vos pratiques. Playbooks pour répondre aux incidents.	README, wikis, Slack, IaC

Type	Description	Outils
Essais	Écrivez des tests automatisés pour votre code d'infrastructure. Exécutez des tests après chaque commit et tous les soirs.	Terratest, tflint, OPA, InSpec

---

Caractéristiques d'une bonne modularisation. Maintenant que vous avez vu tous les ingrédients de la création de code Terraform de qualité, il est temps de les assembler.

La prochaine fois que vous commencerez à travailler sur un nouveau module, utilisez le processus suivant :

- Parcourez la liste précédente et identifiez explicitement les éléments que vous implémenterez et les éléments que vous ignorerez.
- Créez un dossier d'exemples et écrivez d'abord l'exemple de code, en l'utilisant pour définir la meilleure expérience utilisateur et l'API la plus propre à laquelle vous pouvez penser pour vos modules.
- Créez un exemple pour chaque permutation importante de votre module et incluez suffisamment de documentation et des valeurs par défaut raisonnables pour rendre l'exemple aussi facile à déployer que possible.
- Créez un dossier de modules et implémentez l'API que vous avez créée sous la forme d'une collection de petits modules réutilisables et composables.
- Utilisez une combinaison de Terraform et d'autres outils tels que Docker, Packer et Bash pour implémenter ces modules.
- Assurez-vous de verrouiller les versions de toutes vos dépendances, y compris le noyau Terraform, vos fournisseurs Terraform et les modules Terraform dont vous dépendez.
- Créez un dossier de test et écrivez des tests automatisés pour chaque exemple.

Pièges courants de Terraform, difficultés de refactorisation

Avec une architecture modulaire, que se passe-t-il quand on renomme une ressource par exemple?

Le même probkème se pose quand on passe à une structure modulaire.

Par exemple l'objet

```
aws_iam_user.app
```

devient

```
module.iam.aws_iam_user["app"].user
```

La ressource n'est plus la même que celle définie dans l'état et Terraform va vouloir créer une nouvelle ressource.

Comment gérer ce problème ? La solution simple à ce changement est d'opérer une modification de l'état en déplaçant la ressource dans le fichier.

```
$ terraform state mv aws_iam_user.app module.iam[\"app\"].aws_iam_user.user
Move "aws_iam_user.app to "module.iam[\"app1\"].aws_iam_user.user
Successfully moved 1 object(s).
```

On peut également importer des ressources existantes, si le provider le permet.

```
$ terraform import module.iam[\"app\"].aws_iam_user.user app1-svc-account
```

Rappel des objectifs

- Savoir gérer sur le long terme un projet modulaire Terraform

2-07 Problématiques de production et “Zero-downtime”

Uptime Formation

21/09/2023

Objectifs

- Connaître les bonnes pratiques de production avec Terraform

---

Zero-Downtime

La logique Zero-Downtime consiste à s’assurer que l’application reste disponible durant un déploiement.

Si par exemple on redéploie une flotte de webservers avec des Load Balancers, il faut éviter que des clients voient leurs requêtes web échouer.

Comment faire ?

Utiliser lifecycle et create_before_destroy **Le comportement par défault lorsqu’une ressource dite “immutable” est modifiée consiste à supprimer puis recréer la ressource avec la même configuration.**

Une ressource immutable est une ressource dont on ne peut pas changer les attributs sans en créer une nouvelle.

On peut utiliser un meta-argument nommé `lifecycle` pour changer le comportement par défaut.

L’ordre par défaut d’une configuration Terraform est Create - Delete - Update

1. Créer des ressources qui existent dans la configuration mais qui ne sont pas associées à un objet d’infrastructure réel dans l’état.
2. Détruire les ressources qui existent dans l’état mais qui n’existent plus dans la configuration.
3. Mettre à jour les ressources sur place dont les arguments ont changé.
4. Détruire et recréer les ressources dont les arguments ont changé mais qui ne peuvent pas être mis à jour sur place en raison des limitations de l’API distante.

Dans le dernier cas par défaut Terraform détruira l’objet existant, puis créera un nouvel objet de remplacement avec les nouveaux arguments configurés.

Le méta-argument create_before_destroy modifie ce comportement afin que le nouvel objet de remplacement soit créé en premier et que l’objet précédent soit détruit après la création du remplacement.

Dans l’exemple suivant un Launch Template est un modèle d’instance démarrable par un Auto Scaler.

```
resource "aws_launch_template" "example" {
  image_id        = "var.image_id"
  instance_type   = "t2.micro"

  ...
  # Required when using a launch configuration with an auto scaling group.
  lifecycle {
    create_before_destroy = true
  }
}
```

Ainsi lorsqu’on modifie la variable `image_id`, les nouvelles instances seront créés *AVANT* la destruction des précédentes, sans moment où il n’y aurait pas de serveur disponible pour traiter les requêtes.

Stratégies de déploiements **Parmi les différentes stratégies de déploiement existantes, en voici une qui marche bien pour Terraform.**

Les autres possibilités impliquent des opérations manuelles, et notons que c’est ici un avantage de Kubernetes.

Green / Blue Deployments

Le déploiement bleu/vert est un modèle de publication d’application qui permet de transférer progressivement le trafic utilisateur depuis la version antérieure d’une application ou d’un microservice vers une nouvelle version pratiquement identique, ces deux versions s’exécutant en même temps dans l’environnement de production.

> Voir le TP-2.07-green-blue

Cette méthode utilise un module spécifique :

> https://github.com/terraform-in-action/terraform-bluegreen-aws/tree/v0.1.3

Cette méthode propose un remplacement complet des backends avant de basculer le load balancer.

Les autres méthodes de déploiement impliquent un remplacement progressif, voire un pilotage via deux load balancers (méthode canari).

Ces méthodes sont plus complexes et impliquent des lancements incrémentaux de l’infrastructure.

Déployer avec Ansible ou Kubernetes Pour déployer le code d’une application, il est toujours préférable de donner cette tâche à un outil adapté.

Le déploiement d’infrastructures complexes sur la base de quelques variables est la force de Terraform.

Kubernetes ou d’autres outils plus proches des développeurs sont plus adaptés pour les mises à jour d’application.

Rappel des objectifs

- Connaître les bonnes pratiques de production avec Terraform

2-08 Architecture et critères de vérification pour la production

Uptime Formation

21/09/2023

Objectifs

- Savoir utiliser à bon escient les mécanismes de validation et les tests dans Terraform

---

Les valideurs, Pre- et Post- Conditions Terraform

Les trois manières de tester la validité des objets au sein de la recette Terraform sont assez simples d'usage, et relativement similaires.

Documentation : * https://developer.hashicorp.com/terraform/language/expressions/custom-conditions

---

Les validateurs Ils émettent des messages d'erreur si les variables fournies ne sont pas conformes.

```
variable "instance_type" {
  description = "The type of EC2 Instances to run (e.g. t2.micro)"
  type        = string

  validation {
    condition     = contains(["t2.micro", "t3.micro"], var.instance_type)
    error_message = "Only free tier is allowed: t2.micro | t3.micro."
  }
}
```

La condition doit retourner un booléen pour être valide.

```
$ terraform apply -var instance_type="m4.large"
╷
│ Error: Invalid value for variable
│
│   on main.tf line 17:
│    1: variable "instance_type" {
│      ├────────────────
│      │ var.instance_type is "m4.large"
│
│ Only free tier is allowed: t2.micro | t3.micro.
```

```
│
│ This was checked by the validation rule at main.tf:21,3-13.
```

Pre- et Post conditions Ces conditions permettent de faire des vérifications plus générales sur les ressources et les outputs.

```
resource "aws_launch_configuration" "example" {
  image_id        = var.ami
  instance_type   = var.instance_type
  security_groups = [aws_security_group.instance.id]
  user_data       = var.user_data

  # Required when using a launch configuration with an auto scaling group.
  lifecycle {
    create_before_destroy = true
    precondition {
      condition     = data.aws_ec2_instance_type.instance.free_tier_eligible
      error_message = "${var.instance_type} is not part of the AWS Free
       ↪  Tier!"
    }
    postcondition {
      condition     = length(self.security_groups) > 1  # usage de
       ↪  self.attribute popssible
      error_message = "You must use more than one security groups!"
    }
  }
}
```

De la même manière une la condition intégrée dans un block de `lifeccyle` doit retourner un booléen.

```
$ terraform apply -var instance_type="m4.large"
│ Error: Resource precondition failed
│
│   on main.tf line 25, in resource "aws_launch_configuration" "example":
│   18:    condition =
 ↪  data.aws_ec2_instance_type.instance.free_tier_eligible
│      ├────────────────
│      │ data.aws_ec2_instance_type.instance.free_tier_eligible is false
│
│ m4.large is not part of the AWS Free Tier!
```

Comme pour les validations on peut avoir plusieurs conditions dans un même bloc

---

Les ressources et les Data acceptent `precondition` et `postcondition`

Les `precondition` sont vérifiées au moment où l'objet est construit, avec ses éventuels paramètres de `count` ou `for_each`.

Cela permet de s'assurer qu'on ne construit pas une donnée ou une ressource incorrecte.

Les `postcondition` sont vérifiées après la planification et l'obtention de l'objet.

Cela évite les dépendances envers un objet invalide pour d'autres objets qui auraient un effet de cascade.

Les outputs n'acceptent que `precondition`

Comme les validateurs vérifient que les entrées sont conformes, les conditions sur les outputs s'assurent que les sorties du module sont conformes afin d'éviter d'ajouter dans l'état Terraform des informations incorrectes.

Pour construire de bons modules, il est pratique de s'assurer qu'on a bien intégré des processus de `condition`.

Ce sont des garde-barrières efficaces pour les situations inattendues que peut rencontrer le code en raison de mauvaise configuration notamment.

Les tests Terraform

La pratique des tests, comme la sécurité, est un sujet complexe sur lequel on pourrait passer beaucoup de temps.

En effet, tester le code peut impliquer de nombreux types de tests utilsant de nombreuses approches différentes.

- Tests unitaires
- Tests d'intégration
- Tests End To End

Mais aussi

- Tests statiques
- Tests de performance
- Tests de sécurité

Le tester du code Terraform nécessite de lancer réellement des opérations de déploiement.

Les tests principaux consistent donc à créer des scénarios de test consistant à

- créer une infrastructure de test avec des variables spécifiques
- faire des tests sur l'infrastructure
- détruire l'infrastructure de test

Écrire des tests fonctionnels pour Terraform Il existe de nombreux outils pour écrire des tests qui lancent Terraform.

Les solutions standards sont écrites en Go :

- Terratest, par Gruntwork : https://github.com/gruntwork-io/terratest
- terraform-exec, par Hashicorp : https://github.com/hashicorp/terraform-exec

Mais il existe des librairies dans d'autres langages

- Python https://pypi.org/project/pytest-terraform/
- Java : https://github.com/deliveredtechnologies/terraform-maven

---

Si vous êtes assez à l'aise avec Go on peut visiter ce dépôt qui donne des exemples de test Terraform

> https://github.com/brikis98/terraform-up-and-running-code/tree/503ab1f5055917f2d0c715a6b1aa0b9dfb71635
> terraform-team/test

Quand on crée un module qu'on souhaite exporter, il est préférable d'avoir des tests.

Ces derniers sont placés dans un dossier `test` et on les lance automatiquement dans la CI / CD du dépôt du module.

Ceci dit, * il est particulièrement long et compliqué d'écrire des tests pour Terraform * il est remarquable que les modules AWS pour Terraforme n'intègrent pas de tests publics alors que des modules de moindre renommée le fassent * il est aussi remarquable que même Terraform se pose la question de la testabilité des modules Terraform avec la commande `terraform test` * il est au moins aussi important d'avoir une bonne page d'examples pour rendre son module compréhensible

---

Rappel des objectifs

- Savoir utiliser à bon escient les mécanismes de validation et les tests dans Terraform

2-09 Adopter Terraform dans une équipe

Uptime Formation

21/09/2023

Objectifs

- Connaître les bonnes pratiques du travaile en équipe dans Terraform

---

Adopter une nouvelle technique : un enjeu humain avant tout

Dans le mouvement DevOps, c'est la compréhension des processus humains qui doit primer.

On réduit trop souvent les choses à des cadres rigides qui sont les mêmes formes sclérosées contre lesquelles le mouvement luttait au départ.

Aucune équipe n'adoptera Terraform correctement sans un minimum de travail en commun.

Instaurer un process compréhensible et collectif Chaque étape doit être satisfaite, et dépend du capital financier, technique et humain de votre organisation.

Vous aurez besoin de faire progresser techniquement les autres membres de l'équipe, donc autant y aller progressivement.

1. Utiliser git

`La première étape évidente. Sans contrôle de version, le travail collaboratif est li`

2. Proposer une architecture commune

`Montrer une architecture de projet qui offre à la fois souplesse et sécurité, comme c`

3. Proposer des modules communs

`C'est le cataloque de service sur lequel vous pouvez proposer à l'équipe de s'habitue`

4. Faire des tests

`Instaurer la culture du test permet de garantir que les déploiements échouent moins s`

5. Faire des révisions du code

`Analyser le code avec les membres de votre équipe permet de contrôler que les projets`

6. Versionner le code et l'infrastructure

Utiliser des tags SEMVER est une garantie de qualité du code et de l'infrastructure.

7. Automatiser le déploiement

Dès que possible, suivre les bonnes recettes ou une formation avancée pour maîtriser

8. Fournir des environnements différents

Obtenir le soutien matériel pour payer ces environnements est une garantie de stabil

---

Travailler avec l'équipe pour faire la partie configuration de l'IAC

Terraform gère la mise à disposition des ressources, laissant la question de la configuration des instances à d'autres outils.

Les provisioners Certes on peut utiliser des provisioners, mais ce n'est pas considéré comme une bonne pratique.

Ansible Ansible est un outil de gestion de la configuration, et donc son usage peut sembler adapté.

Et en effet il est assez simple de faire parler Ansible et Terraform entre eux.

Mais à terme la complexité est nocive et engendre une dette technique importante.

Horror Stories : CI/CD => Script => Ansible + secrets => Terraform + Ansible => Helmfiles => Kubernetes

---

Packer Packer est un outil de templating de serveurs qui va se révéler adapté à Terraform.

Et de fait Packer est produit par Hashicorp comme Terraform.

Docker Docker est un autre outil de templating de serveurs, mais son usage dans Terraform directement n'est pas très indiqué.

Éventuellement on peut utiliser Terraform pour déployer un cluster Kubernetes qui va ensuite permettre de gérer une infrastructure à base de conteneurs.

Bonne gestion du dépôt git pour Terraform

La branche principale du dépôt git doit être une représentation "1:1" de ce qui est réellement déployé en production.

- Il ne faut pas faire de changements manuels qui causeraient un drift de l'état
- Il faut bien structurer le code pour avoir toutes les modules avec des variables : Terragrunt > workspaces
- Il ne faut pas avoir de branches anarchiques ex: *«Merci de checkout la branche fix-ok-db pour la db de prod»*

Rappel des objectifs

- Connaître les bonnes pratiques du travaile en équipe dans Terraform

2-10 Cadre de travail pour déployer du code applicatif et d’infrastructure

Uptime Formation

21/09/2023

Objectifs

- Savoir utiliser Terraform pour des pratiques CI/CD

Intégrer Terraform dans le CI/CD

Il n'existe pas de solution universelle pour le CI/CD et Terraform, tant le choix d'outils et de plateformes est vaste.

Les grandes lignes de plateformes actuelles sont : * **Classique** : les outils agnostiques ex: Jenkins * **Moderne** : les SASS ex: Github * **Futur** les outils GitOps ex: ArgoCD

L'aspect essentiel que vous aurez à gérer sur toutes ces plateformes est le passage de secrets.

Une fois encore, chaque solution a ses choix qui ont leurs avantages et leurs inconvénients. * **Indépendants** ex: Vault * **Services managés** Ex: Aws Secrets Manager * **Orchestrateur** Ex: Kubernetes

---

Les bonnes pratiques

Utiliser des labels ou des tags sur les ressources générées Chercher dans une masse de ressources est une perte de temps à terme.

Obligez les équipes à fournir des indicateurs :

- Environnement
- Équipe
- Application

1. Entreprise
2. Service / Équipe
3. Région / Zone
4. Environnment
5. Nom / App / Service

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

  # Tags to apply to all AWS resources by default
  default_tags {
    tags = {
      Owner     = "team-foo"
      ManagedBy = "Terraform"
    }
  }
}
```

Rappel des objectifs

- Savoir utiliser Terraform pour des pratiques CI/CD

2-12 Évaluation et point sur la formation

Uptime Formation

21/09/2023

Objectifs

- Discuter du contenu de la formation
- Évaluer les progrès

Rappel

Jour 1 : Language de déploiement, dans le cloud et au-delà

Objectifs pédagogiques

- J1/ Comprendre les avantages des approches IAC et Devops
- J1/ Savoir faire des déploiements en utilisant des providers existants
- **J2/ Savoir créer ses propres modules**
- **J2/ Savoir déployer en équipe sur le long terme**

Compréhension directe Terraform

- **Existence d'un projet commun** : J2
- **Structuration du projet** : J2
- Définition de la recette : J1
- **Passage de secrets** : J2
- Initialisation du projet : J1
- Planification : J1
- **Utilisation de données entre modules** : J2
- Stockage d'un état : J1
- Cycle de vie : J1

Les objectifs de la journée

- Savoir gérer l'états de Terraform pour les pratiques DevOps
- Comprendre le contenu pédagogique de la journée
- Faire un point sur le questionnement personnel IAC/Devops envers Terraform
- Savoir gérer l'état de Terraform pour les pratiques DevOps
- Stocker et partager l'état dans une équipe

- Savoir passer et protéger des informations sensibles dans Terraform
- Comprendre les modules dans Terraform
- Savoir créer ses propres modules
- Savoir gérer sur le long terme un projet modulaire Terraform
- Connaître les bonnes pratiques de production avec Terraform
- Savoir utiliser à bon escient les mécanismes de validation et les tests dans Terraform
- Connaître les bonnes pratiques du travaile en équipe dans Terraform

---

Les TPs

L'évaluation

Comprendre les avantages des approches IAC et Devops

Savoir faire des déploiements en utilisant des providers existants

---

Objectifs

- Discuter du contenu de la formation
- Évaluer les progrès