



# FORMATION TERRAFORM

From intermediate to  
advanced



# Checklist formateur

Avant la formation (☒ )

- Slide formateur personnalisée
- Envoyer l'événement aux participants avec les informations pratiques (heure de début, de fin, [remote] lien google meet, [sur site] quelle salle et étage)
- Créer le bucket de backend common
- Télécharger les TPs sur son poste et avoir consulté le contenu & solutions pour être à l'aise
- Télécharger la bonne version de terraform sur son poste



# SLIDE INTRO SPEAKERS

**#lorem ipsum**

**#lorem ipsum** #lorem ipsum

**#lorem** #lorem ipsum

**#lorem ipsum** #lorem



@jdoe



Jane Doe



<https://github.com/jdoe>



# Alexis Renard

# **Cloud & DevOps** - depuis 4 ans

# **AWS Architect & Security focus**

# **Mentor Blackbelt**

# **Manager Technique**



[Alexis Renard](#)



<https://github.com/alexis-renard>



# Who is around the table today ?

And **what** are you looking for doing this terraform training ?



## SOMMAIRE

### Journée 01

---

- A. Basics
- B. Variabilisation
- C. Manipulation
- D. Modules

### Journée 02

---

- A. Intégration pipeline
- B. Paramètres et secrets
- C. Tests & tools
- D. Au quotidien



# Journée 1

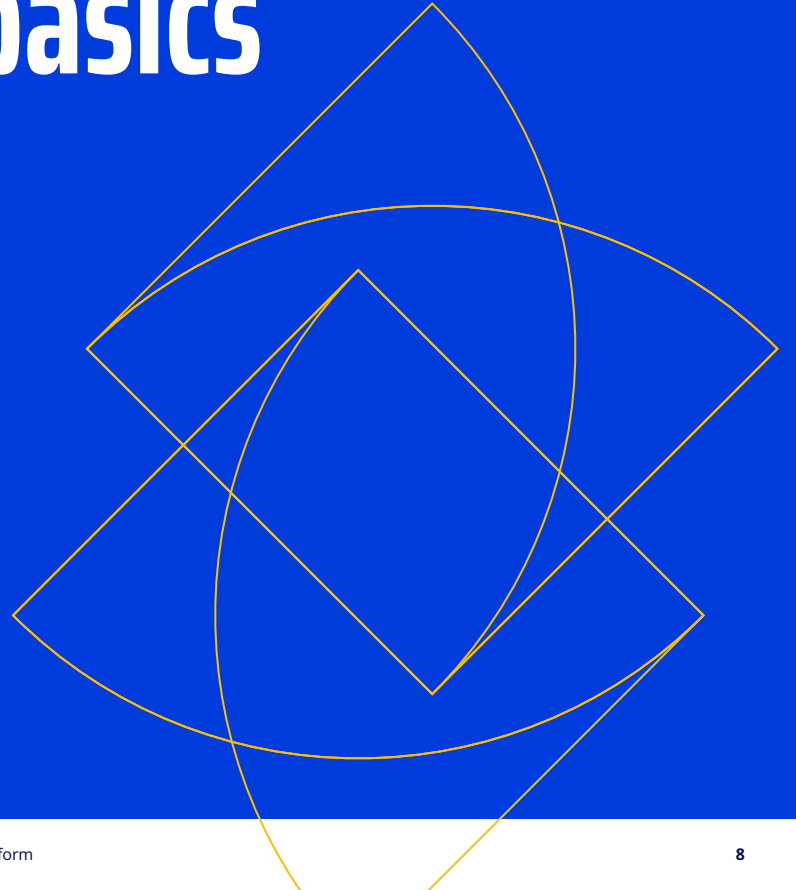


- A. Basics
- B. Workspace
- C. Manipulation
- D. Modules



# — A. Terraform basics

## Rappels





# 01 — Qu'est ce que Terraform ?





# Qu'est ce que Terraform ?

- Développé par HashiCorp en 2014 (première version)
- Outil d'Infrastructure As Code (IaC) pour tout type de provider
- Versionner, partager et réutiliser son infrastructure
- Intégration très facile avec dans un contexte de CI/CD

```
resource "aws_s3_bucket" "my_bucket" {  
    bucket = "my_unique_bucket_name"  
}  
  
<BLOCK TYPE> "<BLOCK LABEL>" "<CUSTOM NAME>" {  
    # Block body  
    <IDENTIFIER> = <EXPRESSION> # Argument  
}
```



# Pourquoi terraform ?

- Un software qui a fait ses preuves pour gérer de “vrais” projets en production
- Permet de construire son infrastructure as code (IaC) et embarque donc tous les bénéfices de la méthode
- C’est un software *open source* qui est facilement extensible (par les providers)
- Une communauté très active et une documentation très bien faite
- Une prise en main assez facile (même s’il faut apprendre les logiques de l’HCL!)

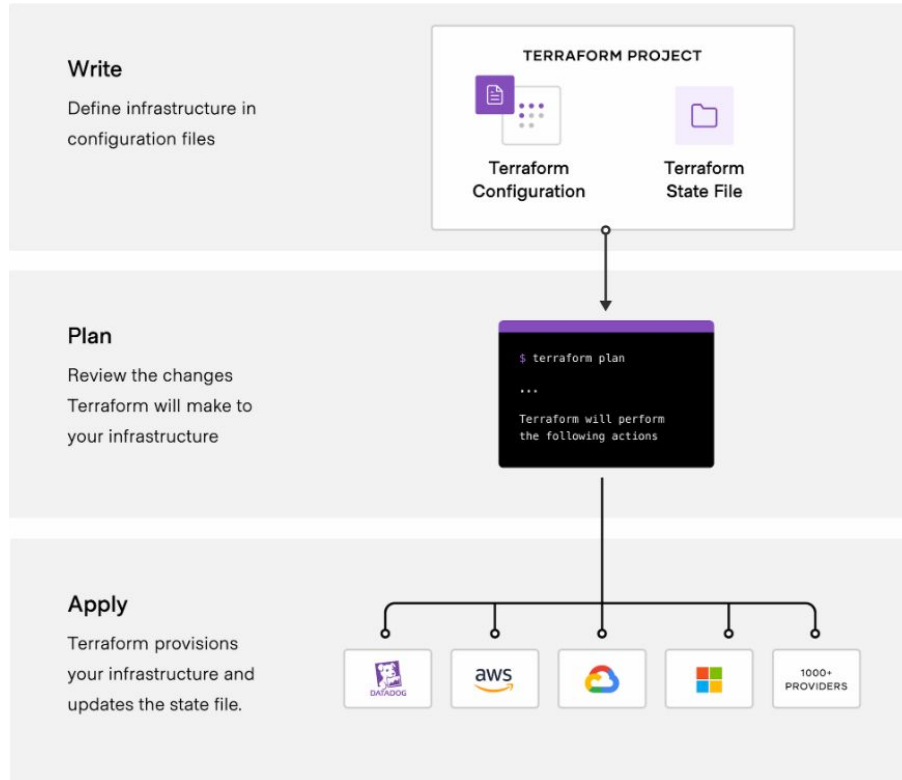


# Comment ça marche ?

- Permet la création de ressources sur les différentes cibles (Cloud, On-Prem, SaaS,...) au travers de leurs API
- Communauté très active, plus de 1700 providers existent
- Provider disponible sur le registre Terraform (<https://registry.terraform.io/>)



# Comment ça marche vraiment ?

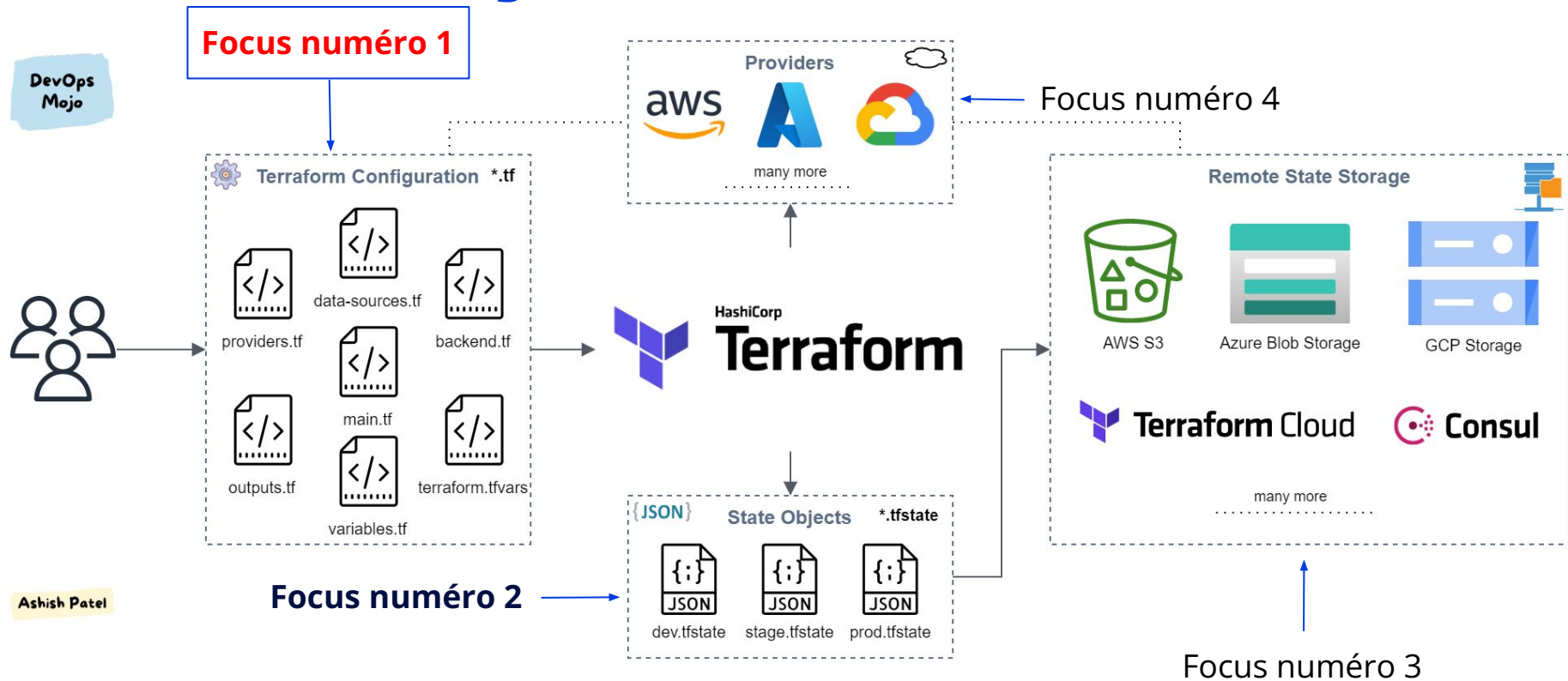


# 02 — Terraform

## Ecosystème



# Eléments de configuration



Ashish Patel

# Le state Terraform : éléments de configuration

## Les blocs de ressources

Le type de bloc dit ressource sera votre principal allié. Grâce à lui, vous définirez n'importe quel objet de votre infrastructure. Par exemple :

- Création d'un bucket S3

```
resource "aws_s3_bucket" "my_bucket" {  
  bucket = "my_unique_bucket_name"  
}
```

- Création d'une instance EC2

```
resource "aws_instance" "my_instance" {  
  ami = "my_ami"  
  instance_type = "t3.micro"  
}
```

```
resource "aws_s3_bucket" "my_already_created_bucket" {  
  bucket = aws_s3_bucket.my_bucket.bucket  
}
```





# Le state Terraform : éléments de configuration

## Les datasources

Le type de bloc datasource permet de récupérer un élément de configuration qui a été créé soit en dehors de Terraform soit dans un autre projet Terraform.

- Récupération d'un bucket S3 :

```
data "aws_s3_bucket" "my_already_created_bucket" {  
  bucket = "my_unique_bucket_name"  
}
```

- Récupération d'une instance EC2 :

```
data "aws_instance" "my_already_created_instance" {  
  instance_id = "instance_id_already_created"  
}
```

```
resource "aws_s3_bucket" "test" {  
  bucket = data.aws_s3_bucket.my_already_created_bucket.bucket  
}
```



# Le state Terraform : éléments de configuration

## Les modules

Un module correspond à un ensemble de ressources utilisées ensemble dans un "conteneur" -> un dossier

Le module est la fonctionnalité principale de Terraform permettant le packaging et la réutilisabilité de code.

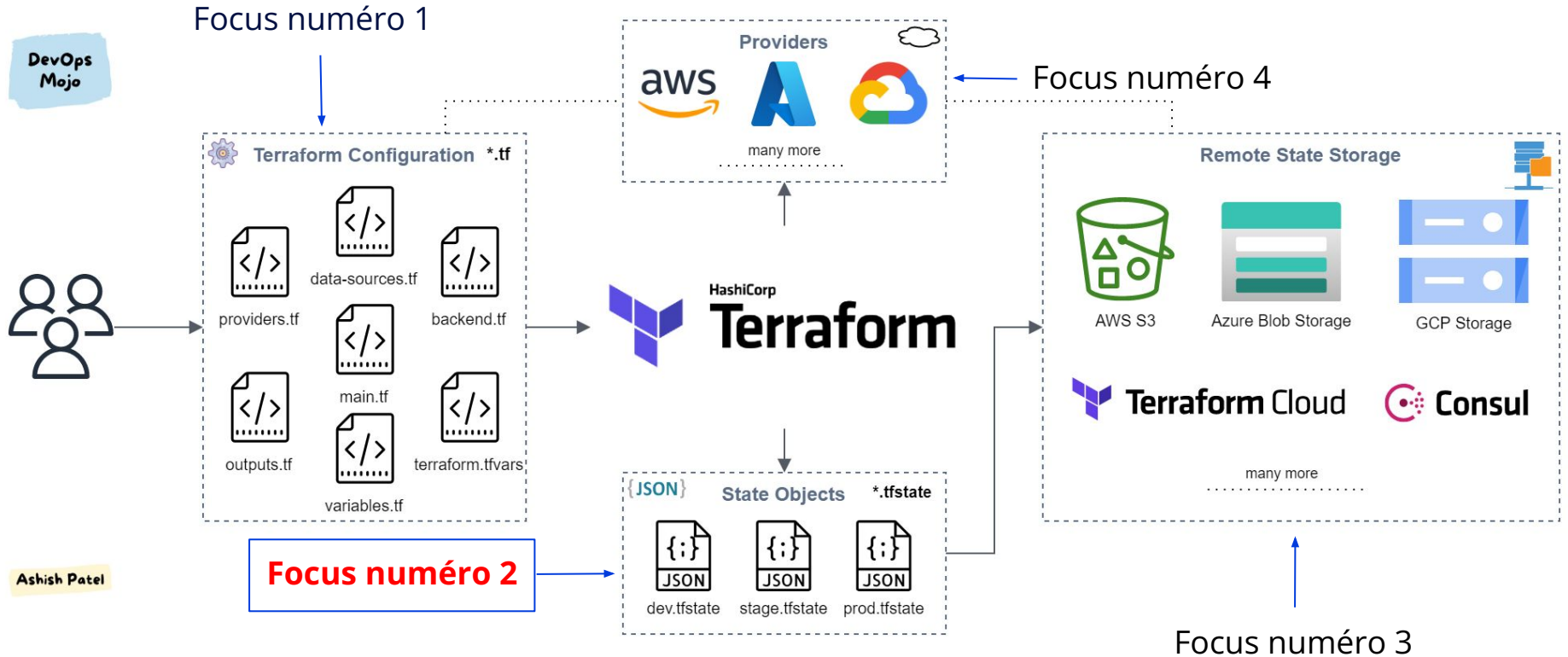
On peut trouver des modules built-in fait par des providers :

<https://registry.terraform.io/modules/terraform-aws-modules/lambda/aws/latest>

On peut aussi très bien créer nos propres modules (from scratch, voir même en prenant un built-in et rajouter des choses par dessus).



# Le state Terraform



# Le state Terraform

## Le fichier .tfstate

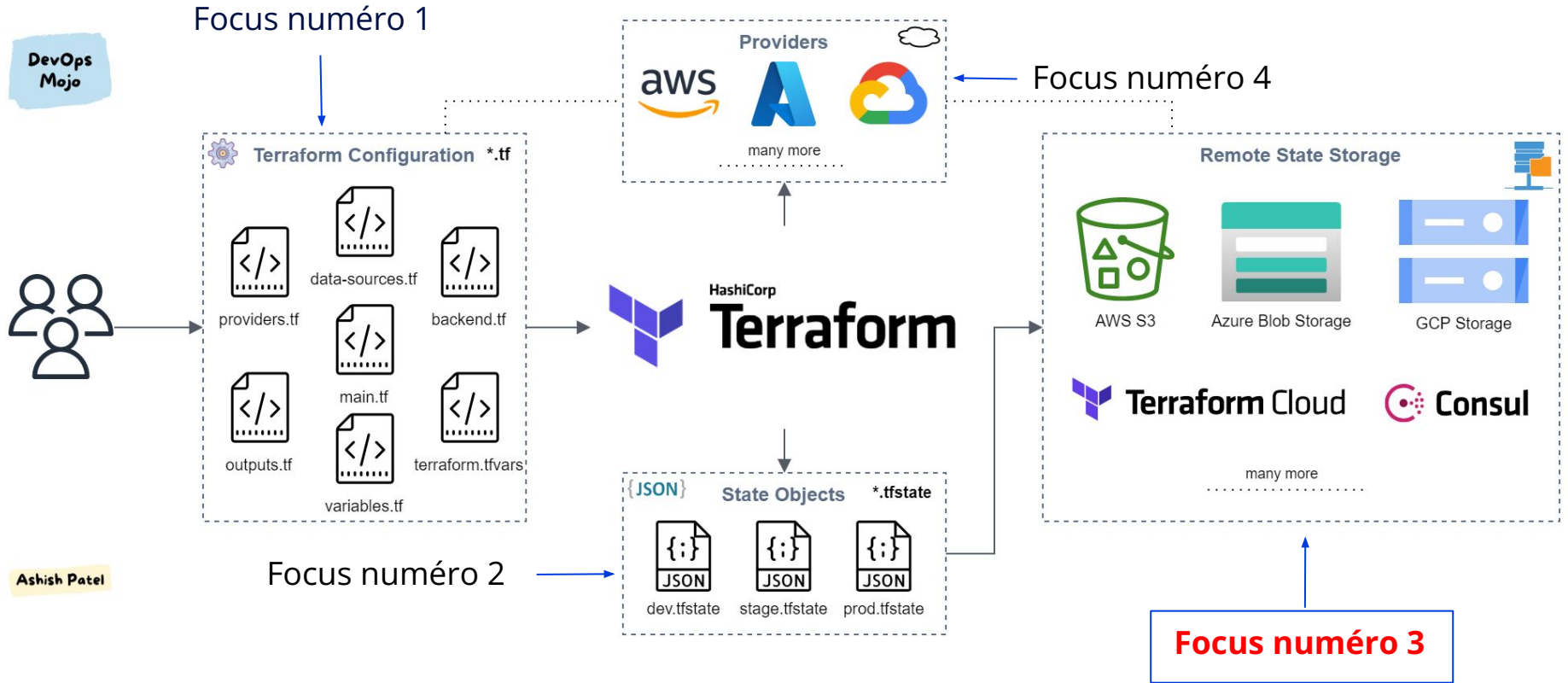
Le state terraform sert à :

- Stocker l'état actuel de notre application ⇒ ce fichier doit toujours décrire l'état actuel de notre application (sinon on a un "drift")
- Par défaut, il est stocké localement dans un fichier nommé "terraform.tfstate"
- Fichier utilisé pour créer le plan et faire les changements à l'infrastructure actuelle
- Avant chaque opération sur le state ⇒ terraform fait un refresh
- Fichier sous format JSON
- Depuis la version 1.0.0, rétrocompatibilité entre les states (attention, avant non)

**En bref ⇒ Ce fichier est vôtre précieux, ne le modifiez pas à la main, ce n'est pas prévu pour ! Prenez en soin**



# Sauvegarde du state



# Sauvegarde du state

## Où stocker son state terraform ?

- **Localement : pas recommandé**
- **Sur votre Cloud Provider préféré (AWS, GCP, Azure, Alibaba, OVHcloud, ...)**
- **On Prem si une API est disponible pour Terraform**

## Avantages de stocker dans AWS (sur S3) :

- **Service managé**
- **99,999999999% durabilité**
- **99,99% disponibilité**
- **Chiffrement à l'aide de KMS, CloudHSM et SSL**
- **Sécurisation via des outils du type IAM -> RBAC**
- **Locking nativement à l'aide de DynamoDB**
- **Versioning sur le bucket**
- **Coût de S3 insignifiant**

```
terraform {  
  backend "s3" {  
    bucket          = "s3-eu-west-1-ippon-formation-terraform"  
    dynamodb_table = "table-ippon-formation-terraform-lock"  
    encrypt         = true  
    key             = "/ippon-formation-terraform"  
    region          = "eu-west-1"  
  }  
}
```

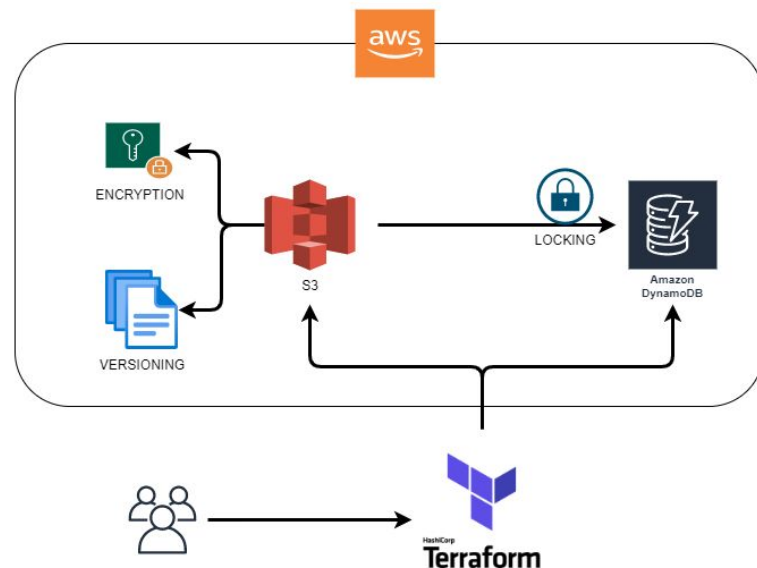


# Sauvegarde du state

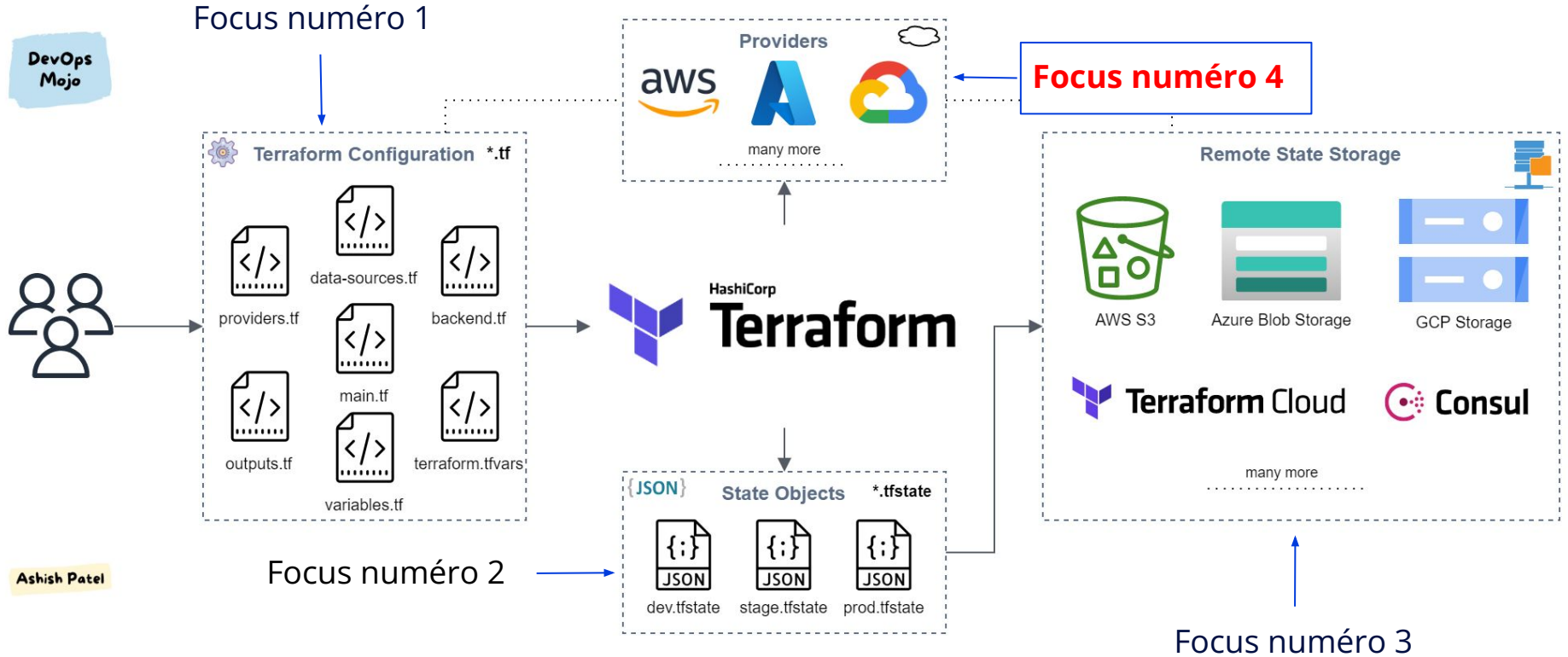
## Le state locking

Le state locking, c'est quoi ?

- Permet la gestion de la concurrence
- Deux utilisateurs en simultanés qui souhaitent modifier l'infrastructure = **impossible**
- Le state locking peut-être fait via API (AWS, Consul,...) ou par l'OS en lui même (pour du local)
- On peut désactiver la feature à l'aide du flag **-lock** : pas recommandé
- Possibilité de forcer le release du lock à l'aide de la commande **force-unlock**



# Providers

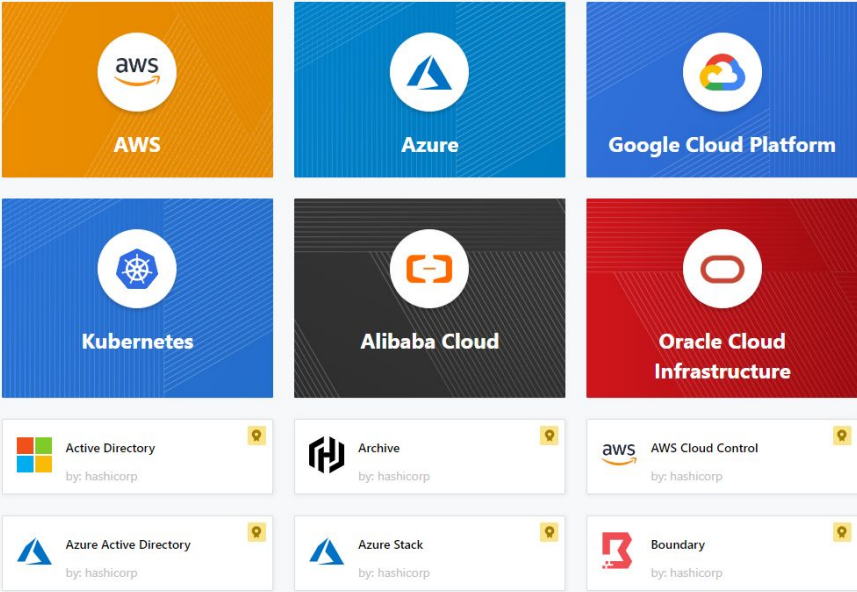




# Le state Terraform : providers

🔗 Providers

Providers are a logical abstraction of an upstream API. They are responsible for understanding API interactions and exposing resources.



The image displays a collection of Terraform providers. At the top, there are three large colored boxes: an orange box for AWS, a blue box for Azure, and a blue box for Google Cloud Platform. Below these are three more boxes: a blue box for Kubernetes, a dark grey box for Alibaba Cloud, and a red box for Oracle Cloud Infrastructure. At the bottom, there are six smaller white boxes, each representing a different provider: Active Directory, Archive, AWS Cloud Control, Azure Active Directory, Azure Stack, and Boundary. Each box contains the provider's logo and name, and most include a small yellow icon in the top right corner.

```
terraform {  
  required_providers {  
    aws = {  
      source = "hashicorp/aws"  
      version = "~> 4.0"  
    }  
  }  
}  
  
provider "aws" {  
  region = "eu-west-1"  
}
```



# Providers

Chaque bloc `terraform` peut contenir un certain nombre de paramètres liés au comportement de Terraform.

À l'intérieur d'un bloc `terraform`, seules des valeurs constantes peuvent être utilisées (pas de variables ni de fonctions)

```
# do
terraform {
  required_providers {
    aws = {
      source = "hashicorp/aws"
      version = "~> 4.0"
    }
  }
}

# don't
terraform {
  required_providers {
    aws = {
      source = "hashicorp/aws"
      version = "~> ${var.aws_provider_ver}"
    }
  }
}
```



# 03 — La CLI de Terraform



# Usage

## Les commandes essentielles à connaître

- **init**
- **validate**
- **fmt**
- **plan**
- **apply**
- **destroy**
- **state**
- **import**
- **workspace**

```
> 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.
```



# terraform init

Première commande à lancer quand on arrive sur un projet terraform (ou dans une CI) :

- Configuration du backend
- Téléchargement de tous les provider plugins
- Téléchargement de tous les modules
- Génère les fichiers nécessaires au bon fonctionnement
  - Répertoire ".terraform"
  - Fichier ".terraform.lock.hcl"

```
> terraform init

Initializing the backend...

Initializing provider plugins...
- Reusing previous version of hashicorp/aws from the dependency lock file
- Using previously-installed hashicorp/aws v3.76.1

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.
```



# terraform validate

Commande très pratique qui :

- Permet de valider la syntaxe et la cohérence de nos fichiers de configuration
- Peut-être utilisé avant un terraform plan / deploy
- Pre commit hook

Pour pouvoir s'exécuter, la commande à besoin d'un espace de travail initialisé au-préalable.

```
> terraform validate
Success! The configuration is valid.
```

```
> terraform validate

Error: Unsupported argument

   on main.tf line 12, in resource "aws_iam_account_alias" "account_alias":
   12:   invalide_attr      = aws.current_account

An argument named "invalide_attr" is not expected here.
```



# terraform refresh

## Commande exécuté en arrière plan la plupart du temps par Terraform

- Permet de récupérer la dernière version du state Terraform depuis notre backend
- Est systématiquement lancé lors d'un plan / apply / destroy



### Commande désormais dépréciée

Elle correspond en arrière plan à :

```
$> terraform apply -refresh-only -auto-approve
```

Et on peut examiner les changements que Terraform détecte lors de l'exécution de la commande :

```
$> terraform apply -refresh-only
```

**La bonne pratique étant de ne pas lancer cette commande nous même mais de faire confiance à Terraform (c.f. <https://www.terraform.io/cli/commands/refresh> en fin de document)**



# terraform plan

On peut découper cette commande en 3 étapes distinctes :

1. **Lecture du state courant pour être sûr qu'on est à jours**
  - fait un refresh
2. **Comparaison entre le state actuel et notre configuration**
  - diff entre le state et ce qu'on a changé
3. **Proposition d'actions pour remédier au(x) changement(s) de notre configuration**
  - change/delete/replace
  - ou ne rien faire si pas de changement

Exemple d'output d'un terraform plan :

```
# aws_api_gateway_method_settings.general_settings must be replaced
-/+ resource "aws_api_gateway_method_settings" "general_settings" {
  ~ id          = "jps08auax7-dev-*/*" -> (known after apply)
  ~ method_path = "*/*" -> "*/TEST" # forces replacement
    # (2 unchanged attributes hidden)

    # (1 unchanged block hidden)
}

Plan: 1 to add, 2 to change, 1 to destroy.
```





# terraform plan

Cette commande ne modifie pas l'état de votre infrastructure, elle est à titre d'information.

Le plan peut-être sauvegardé dans un fichier à l'aide de l'argument `-out=FILE`

Dans une approche DevOps, l'intégration du plan au processus de CI/CD peut-être intéressant :

- **Affichage du plan dans la merge request sur gitlab**
- **Automatisation du déploiement lorsque la relecture du nouveau plan est validé (humainement)**
  - Détection de texte dans le commentaire ...
  - Ajout d'emojis sur le message avec le plan



# terraform apply

Comme son nom l'indique, applique le résultat du plan à l'infrastructure actuelle :

- Exécute un terraform plan en amont de l'apply par défaut
- Si un chemin vers un plan est passé en argument de la commande, l'apply se fera sur ce plan

Si un plan est passé en argument de la commande, pas d'autorisation demandé par terraform avant d'exécuter la commande

Options utiles :

- **-auto-approve** : répond oui à toutes demandes de confirmation
- **-lock=false** : permet de passer outre le lock (**PAS RECOMMANDÉ**)
- **-no-color** : Supprime le format "terminal"
  - pratique si on souhaite inspecter le plan soit même dans un éditeur de texte



# terraform destroy

Une commande .... destructrice !

- Permet de détruire les ressources contenues dans notre state
- Il est possible de créer un “plan de destruction” avant de faire appel à cette commande :
  - merci à cette commande **terraform plan -destroy**

En réalité, **terraform destroy** est un alias pour la commande : **terraform apply -destroy** (à partir de la version v0.15.2 de Terraform)



# terraform state

Une seconde commande.... destructrice (si mal utilisée)

Cette commande possède en réalité des sous-commandes :

- list
- mv
- pull
- push
- replace-provider
- rm
- show

```
> terraform state --help
Usage: terraform [global options] state <subcommand> [options] [args]

This command has subcommands for advanced state management.

These subcommands can be used to slice and dice the Terraform state.
This is sometimes necessary in advanced cases. For your safety, all
state management commands that modify the state create a timestamped
backup of the state prior to making modifications.

The structure and output of the commands is specifically tailored to work
well with the common Unix utilities such as grep, awk, etc. We recommend
using those tools to perform more advanced state tasks.

Subcommands:
  list          List resources in the state
  mv           Move an item in the state
  pull         Pull current state and output to stdout
  push         Update remote state from a local state file
  replace-provider Replace provider in the state
  rm           Remove instances from the state
  show         Show a resource in the state
```



# terraform state list

Commande qui permet de lister les ressources contenues dans notre state

- Lancer sans argument ⇒ permet de lister l'ensemble des ressources
- On peut filtrer par ressource ou module par exemple : `terraform state list module.elb`
- Si on connaît l'identifiant de notre ressource dans le state, on peut utiliser l'argument `-id=sg-1234abcd`

```
> terraform state list
data.aws_caller_identity.current
data.aws_default_tags.current
data.aws_iam_policy_document.
data.aws_iam_policy_document.
data.aws_iam_policy_document.
data.aws_iam_policy_document.
data.aws_iam_policy_document.
data.aws_iam_policy_document.
data.aws_iam_policy_document.
data.aws_iam_policy_document.
data.aws_iam_policy_document.
data.aws_iam_policy_document.
data.aws_iam_policy_document.
data.aws_iam_policy_document.
data.aws_iam_policy_document.
data.aws_iam_policy_document.
data.aws_iam_policy_document.
data.aws_iam_policy_document.
data.aws_iam_policy_document.
data.aws_iam_policy_document.
data.aws_iam_policy_document.ccoe_api
data.aws_iam_policy_document.deny_regions
data.aws_iam_policy_document.eda_deny_regions
data.aws_iam_policy_document.protect_ccoe_iam_roles_and_tags
data.aws_iam_policy_document.role_root_organization_admin
data.aws_iam_policy_document.sandbox_deny
aws_iam_account_alias.root_alias
aws_iam_role.organization_admin_role
aws_iam_role_policy_attachment.organization_admin_role
aws_iam_user.first
aws_iam_user_policy_attachment.admin
```

```
> terraform state list aws_iam_user.first
aws_iam_user.first
```

```
> terraform state list module.auth_nprd_account.null_resource.order
module.auth_nprd_account.null_resource.order
```

Erreur si la ressource n'existe pas :

```
> terraform state list aws_iam_user.firs
```

```
Error: Unknown resource
```

```
The current state contains no resource aws_iam_user.firs. If you've just added this resource to the configuration, you must run "terraform apply" first to create the resource's entry in the state.
```



# terraform state show

Commande qui permet de voir les attributs d'une ressource contenue dans notre state

**\$> terraform state show ADDRESS**

```
> terraform state show module.web_nprd_account.aws_organizations_account.subaccount
# module.web_nprd_account.aws_organizations_account.subaccount:
resource "aws_organizations_account" "subaccount" {
  arn                = "arn:aws:organizations::[REDACTED];"
  email              = "[REDACTED]"
  iam_user_access_to_billing = "ALLOW"
  id                 = "[REDACTED]"
  joined_method      = "CREATED"
  joined_timestamp   = "2020-09-11T18:40:18Z"
  name               = "[REDACTED]"
  parent_id         = "[REDACTED]"
  role_name          = "OrganizationAdmin"
  status             = "ACTIVE"
  tags               = {}
  tags_all           = {
    "account_category" = "nprd"
    "account_owner"    = "[REDACTED]"
  }
}
```



# terraform state pull

Permet de récupérer le state localement et de l'afficher

# terraform state push

Commande qui permet de modifier le fichier de state distant avec celui passé en argument

- **Ne doit quasiment jamais être utilisée**
- **Mécanismes de sécurités implémentés par Terraform pour éviter les actions non voulues**
  - voir la doc : <https://www.terraform.io/cli/commands/state/push>
- **Possibilité de désactiver ces mécanismes de sécurité à l'aide de l'argument -force**



# terraform state replace-provider

Commande permettant de remplacer un provider par un autre.

Exemple de cas pratique :

- **Migration de terraform v0.12.x vers la v0.13.x**
  - modification dans le nommage des providers
- **Switch vers l'utilisation d'un provider "maison" ou stocké en interne :**
  - switch du provider AWS d'HashiCorp vers un fork custom (*providerippon*) stocké dans un registre privé :

```
$> terraform state replace-provider hashicorp/aws registry.ippon.fr/providerippon/aws
```





# terraform state mv

Commande permettant de modifier l'adresse d'une ressource, le plus souvent dans le cadre d'un renommage

⇒ L'utilité est de surcharger le comportement de Terraform : cas d'exemple lors du renommage d'un bloc de ressource

- **Comportement de terraform par défaut**

- Lancement d'un terraform plan
- Voit que la ressource a été modifié
- Supprime l'ancienne
- Créer la nouvelle avec le nom voulu

- **Comportement souhaité**

- Utilisation de la commande **terraform state mv old\_adress.old\_name new\_adress.new\_name**
- Lancement d'un terraform plan
- Pas de changement donc pas de modification

L'utilisation de l'argument **-dry-run** permet de voir toutes les instances de ressources impactées sans appliquer la commande réellement



# terraform state rm

**Cette commande permet de supprimer une/des ressources du state : ATTENTION ! cela ne veut pas dire supprimer la ressource dans votre provider**

- **Cette commande permet en réalité de dire à Terraform “d’oublier” l’existence de cette ressource**
- **Si la configuration de cette ressource existe toujours, lors du prochain plan, terraform proposera de la recrée**
  - attention si le nom doit-être unique cela entraînera un conflit avec celle existante

**L’utilisation de l’argument `-dry-run` permet de voir toutes les instances de ressources impactées sans appliquer la commande réellement**



# terraform import

A l'aide ma ressource existe déjà dans mon compte AWS mais pas dans mon state terraform

La fonction d'import vient palier à ce problème :

- Consulter la doc de chaque ressource pour savoir comment l'importer
- Le fait d'importer la ressource ne fait que l'ajouter au state ! Il faut l'ajouter à la configuration manuellement

Terraform prévoit dans une prochaine version de générer la configuration automatiquement !

Exemple d'import d'un bucket S3 :

```
$ terraform import aws_s3_bucket.bucket bucket-name
```



# terraform workspace

Commande pratique qui permet d'avoir plusieurs state à l'intérieur d'un même backend

Cette commande possède en réalité des sous-commandes :

- **list** -> permet de lister l'ensemble workspaces disponible
- **select** -> permet de sélectionner un workspace
- **new** -> permet de créer un nouveau workspace
- **delete** -> permet de supprimer un workspace (il faut pour ceci qu'il existe, que ce ne soit pas votre workspace courant et que le state soit vide)
- **show** -> permet d'afficher le workspace dans lequel on se trouve



# Pour aller plus loin...

- **Une suite de 2 articles super intéressants sur la gestion de l'état Terraform et l'industrialisation de la manipulation des fichiers d'état :**
  - <https://blog.ippon.fr/2022/04/04/terraform-dans-tous-ses-etats-1-2/>
  - <https://blog.ippon.fr/2022/04/11/terraform-dans-tous-ses-etats-2-2/>
- **La documentation de Terraform state qui est très bien écrite :**
  - <https://www.terraform.io/language/state>



# TP1



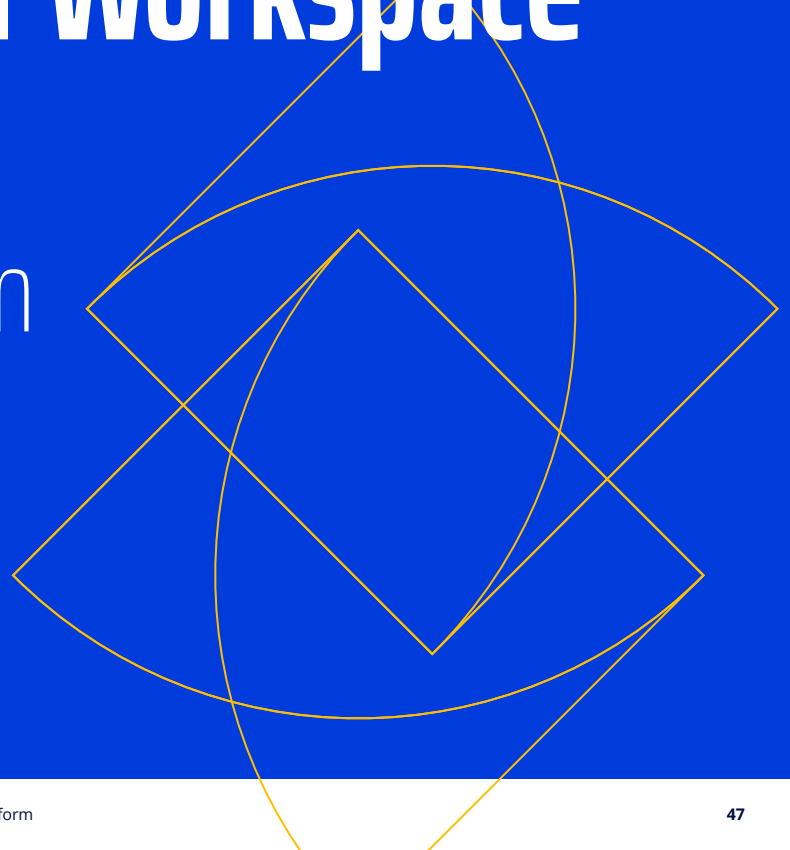
<https://gitlab.ippon.fr/formation/terraform/tps/tp1>

# Terraform basics



# — B. Variables et Workspace

Gestion et utilisation



# Variables Terraform

## Les différents types

- Type primitif :
  - **string** : Chaîne de caractère alphanumérique + "\_" et "-"
  - **number** : Valeur numérique (entier ou non) tel que **15** ou **6.1234**
  - **bool** : Booléen parmi **true** ou **false**
- Type complex, structural ou collections :
  - **list** (ou **tuple**) : Séquence de valeur (parmi les types possibles)
    - exemple : `["formation_terraform_1", "formation_terraform_2"]`
  - **map** (ou **object**): Groupe de valeur identifié par une clé
    - exemple : `{type = "formation", duration = 2}`
- Autre :
  - **set** : liste d'éléments uniques et non ordonnés





# Variables Terraform

## Déclaration

Chaque variable d'entrée acceptée par un module doit être déclarée à l'aide d'un bloc **"variable"**

Mots clés à ne pas utiliser en tant que nom de variables :

- source
- version
- providers
- count
- for\_each
- lifecycle
- depends\_on
- locals

```
variable "image_id" {
  type = string
}

variable "availability_zone_names" {
  type    = list(string)
  default = ["us-west-1a"]
}

variable "docker_ports" {
  type = list(object({
    internal = number
    external = number
    protocol = string
  }))
  default = [
    {
      internal = 8300
      external = 8300
      protocol = "tcp"
    }
  ]
}
```



# Variables Terraform

## Arguments pour une variable

- **type** : définit le type de la variable attendue parmi les types possibles
- **description** : permet de savoir à quoi sert la variable (utile pour documentation)
- **default** : valeur par défaut si aucune valeur n'est spécifié lors de l'appel au module
- **sensitive** : indique à Terraform s'il s'agit d'un contenu sensible (pas d'affichage en output)
- **nullable** : autorise ou non que la valeur spécifiée soit "null"
- **validation** : permet d'effectuer une validation de la valeur souhaité

```
variable "image_id" {  
  type      = string  
  description = "Image ID to use when creating an instance"  
  default   = "id-xxxx"  
  sensitive  = false  
  nullable   = false  
  
  validation {  
    condition     = length(var.image_id) > 3 && substr(var.image_id, 0, 3) == "id-"  
    error_message = "The image_id value must be a valid Image id, starting with \"id-\"."  
  }  
}
```



# Variables Terraform

## Définition

Plusieurs façon de définir des variables lors d'un appel de module :

- input demandé par terraform si non renseigné
  - les inputs obligatoire sont à rentrer un par un dans la console
- variables d'environnements
  - nom des inputs à préfixer par **"TF\_VAR\_"**
    - ***export TF\_VAR\_image\_id=id\_12345***
    - ***export TF\_VAR\_name\_key=name\_value***
- ligne de commande en précisant le nom de la variable
  - ***terraform command -var="image\_id=id-12345" -var="name\_key=name\_value"***
- fichier (format HCL ou Json)
  - ***terraform command -var-file="simpleVars.tfvars"***
  - ***terraform command -var-file="simpleJsonVars.tfvars.json"***
  - ***terraform command -var-file="common.tfvars" -var-file="env1.tfvars"***



# Variables Terraform

## Ordre de chargement

Tous les mécanismes pour définir les variables peuvent être utilisés ensemble dans n'importe quelle combinaison.

**Remarque** : Si plusieurs valeurs sont attribuées à la même variable, Terraform utilise la dernière valeur trouvée, en écrasant toutes les valeurs précédentes.

Terraform charge les variables dans l'ordre suivant :

1. valeur par défaut définie lors de la définition de la variable
2. variable d'environnement si définie
3. **terraform.tfvars** si présent
4. **terraform.tfvars.json** si présent
5. **\*.auto.tfvars** ou **\*.auto.tfvars.json** (charger ordre alphabétique du nom de fichier)
6. les variables définies par ligne de commande : **-var** ou **-var-file**



# Workspaces

## Pourquoi ?

**Terraform** permet de mettre rapidement en place une infrastructure.

→ Mais comment gérer plusieurs environnements avec les mêmes fichiers de configuration ?

**Rappel** : Chaque configuration **Terraform** a un **backend** associé qui définit comment les opérations sont exécutées et où les données persistantes telles que l'état **Terraform** ou les variables **Outputs** sont stockées.

Les données persistantes stockées dans le **backend** appartiennent à ce qu'on appelle un **workspace**. Initialement, le **backend** n'a qu'un seul espace de travail, nommé **default** et donc il n'y a qu'un seul état Terraform associé à cette configuration.



# Workspaces

## Comment ?

Certains **backends** (la plupart des remotes) prennent en charge plusieurs espaces de travail (que l'on peut nommer), permettant d'associer plusieurs états à une même configuration **Terraform**.

→ On peut donc créer plusieurs environnements (test, prod, id unique,...) avec principalement la même configuration mais en gardant différents états et variables

Le but des workspaces est donc de **créer des états différents et indépendants sur la même configuration**.

Les différents environnements devront utiliser les mêmes **credentials** et le même **backend**.

La variable `terraform.workspace` permet d'utiliser le nom du workspace dans les fichiers de configuration.

Cela permet d'utiliser des conditions sur la configuration en fonction du **workspace**.



# Workspaces

## CLI

Principales commandes :

- `terraform workspace show`
  - Permet d'afficher le workspace dans lequel on se trouve
- `terraform workspace list`
  - Permet de lister l'ensemble workspaces disponible
- `terraform workspace new`
  - Permet de créer un nouveau workspace
- `terraform workspace select [workspace_name]`
  - Permet de sélectionner un workspace
- `terraform workspace delete [workspace_name]`
  - Permet de supprimer un workspace



# Workspaces

## Alternative

Certains clients veulent une séparation **complète** entre leurs environnements. Cela nécessite de séparer les fichiers **state** et les **credentials** (pour déployer sur plusieurs comptes AWS par exemple).

Une solution efficace pour déployer plusieurs environnements est l'intégration dans des pipelines des différentes étapes de l'apply d'un code terraform (gestion du backend, des variables d'entrées, des steps de déploiement...)

Il y a plusieurs solutions pour la gestion des variables, notamment :

- Utiliser des fichiers **.tfvars** différents et associés à chaque environnement. La CI sélectionne le bon fichier de variable avec ***-var-file="prod.tfvars"***
- Utiliser **terragrunt** avec des fichiers différents pour chaque environnement. La CI exécutera le fichier **terragrunt** qui exécutera les fichiers **terraform** associés





# TP2



<https://gitlab.ippon.fr/formation/terraform/tps/tp2>

# Environnement et workspace



# — C. Manipulation de données

Utiliser les fonctions utiles et façonner les structures de données



# On veut tirer le meilleur de la variabilisation

En terraform, les fonctions sont là pour nous aider à

- Conditionner
- Sélectionner
- Modifier

Se référer à la très bonne doc officielle :  
<https://www.terraform.io/language/functions>

Des fonctions essentielles

- concat, merge, format, lookup, contains

Des itérateurs à connaître

- count, for\_each



# Un point sur les fonctions essentielles

## Listes

- **concat**
  - merger deux listes
- **contains**
  - vérifier si un élément est présent
- **formatlist**
  - Formater toutes les strings d'une liste avec une/des variable(s)

## Utiles

- **try**
  - va essayer de récupérer une valeur, sinon lui donner une variable par défaut => évite que terraform tombe en erreur

## Maps

- **merge**
  - merger deux map
- **lookup**
  - aller chercher une variable dans une map (et valeur par défaut si non trouvée)

## Strings

- **format**
  - formater une string avec une/des variable(s)
- **upper/lower/title**
  - formater de manière homogène une string



# count vs for\_each

qui choisir ?

	<b>count</b>	<b>for_each</b>
<b>type d'itérateur</b>	list (index)	set, map (clés)
<b>création conditionnelle</b>	oui	oui +
<b>usecases complexe</b>	limité	oui
<b>rétro-compatibilité</b>	bonne	limitée



# Retrouver un élément dans une collection

## Quand en a-t-on besoin ?

- Retrouver un élément dans une liste (index)
- Retrouver un élément dans une map (clé)

## Comment l'utiliser ?

- Pour les listes
  - `var.my_list[<index>]`
  - `element(var.my_list, <index>)`
- Pour les maps
  - `var.my_map.my_key`
  - `var.my_map["my_key"]`
  - `lookup(var.my_map, "my_key", <default_value>)`



# L'expression clé : le "splat"

## Quand en a-t-on besoin ?

- Itérer sur une collection d'éléments

## Comment l'utiliser ?

- Le symbole spécial [\*] itère sur tous les éléments de la liste donnée à sa gauche et accède à partir de chacun d'eux au nom de l'attribut donné à sa droite.
  - Exemple : `var.my_object_list[*].id`
- Une expression splat peut également être utilisée pour accéder aux attributs et aux index de listes de types complexes en étendant la séquence d'opérations à la droite du symbole

```
variable "my_object_list" {
  type = list(object({
    id   = string
    age  = number
  }))
  default = [
    {
      age = 38
      id  = "first"
    },
    {
      age = 27
      id  = "second"
    }
  ]
}

output "all_ids" {
  value = var.my_object_list[*].id
}

# [for o in var.my_object_list : o.id]
```



# Un point sur les dynamic statements

## Quand en a-t-on besoin ?

- Création de block conditionnels dans une ressource terraform
- Très utilisé lorsqu'on veut modulariser

## Comment les mettre en place ?

- Avec le bloc spécial "dynamic"
- Et une itération sur une map ou un set. Ce qu'il nous faut à minima, c'est une clé !





```
resource "aws_security_group" "test_1" {
```

```
  name      = "my-security-group"
  ...
  ingress {
    description = "Open SSH for ip
250.250.250.250/32"
    from_port   = 22
    to_port     = 22
    protocol    = "tcp"
    cidr_blocks = ["250.250.250.250/32" ]
  }

  ingress {
    description = "Open SSH for ip
251.251.251.251/32"
    from_port   = 22
    to_port     = 22
    protocol    = "tcp"
    cidr_blocks = ["251.251.251.251/32" ]
  }

  ingress {
    description = "Open SSH for ip
252.252.252.252/32"
    from_port   = 22
    to_port     = 22
    protocol    = "tcp"
    cidr_blocks = ["252.252.252.252/32" ]
  }
  ...
}
```

# Variabiliser avec les dynamic statements

```
variable "ingress_ip_list" {
  type = list(string)
  default = [
    "250.250.250.250/32",
    "251.251.251.251/32",
    "252.252.252.252/32"
  ]
}

resource "aws_security_group" "test_1" {
  name      = "my-security-group"
  ...
  dynamic "ingress" {
    for_each = toset(var.ingress_ip_list)
    content {
      description = "Open SSH for ip ${ingress.key}"
      from_port   = "22"
      to_port     = "22"
      protocol    = "tcp"
      cidr_blocks = [ingress.key]
    }
  }
  ...
}
```



# TP3



<https://gitlab.ippon.fr/formation/terraform/tps/tp3>

# Manipulation de données



# — D. Modules terraform

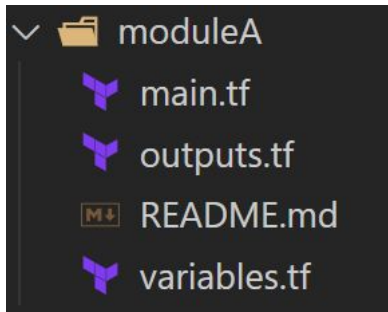
L'intérêt de modulariser  
et comment bien le faire



# Modules terraform

## Un module c'est quoi ?

- Un module est simplement un répertoire avec un ou plusieurs fichiers tf à l'intérieur
- Il définit un groupe de ressources terraform partageant un même cycle de vie, dont les ressources sont interdépendantes
- On parle souvent de **root module** (terraform apply) et de **modules terraform** destinés à être appelés par un ou des root module. Ici, un *module* est un *module terraform*.
- Un module terraform est voué à être instancié plusieurs fois.
- Possibilité de gérer ses modules, d'en récupérer sur <https://registry.terraform.io/>



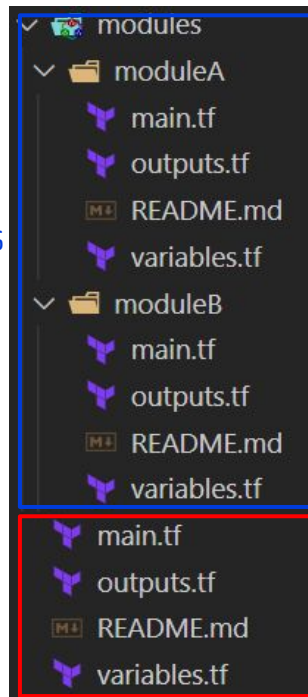
# Modules terraform

## Exemple de répertoire terraform avec des modules locaux

```
module "stack1" {  
    source = "./modules/moduleA"  
}  
  
module "stack2" {  
    source = "./modules/moduleB"  
}
```

Appel des modules depuis le  
main.tf du root module

dossier comportant les  
modules locaux



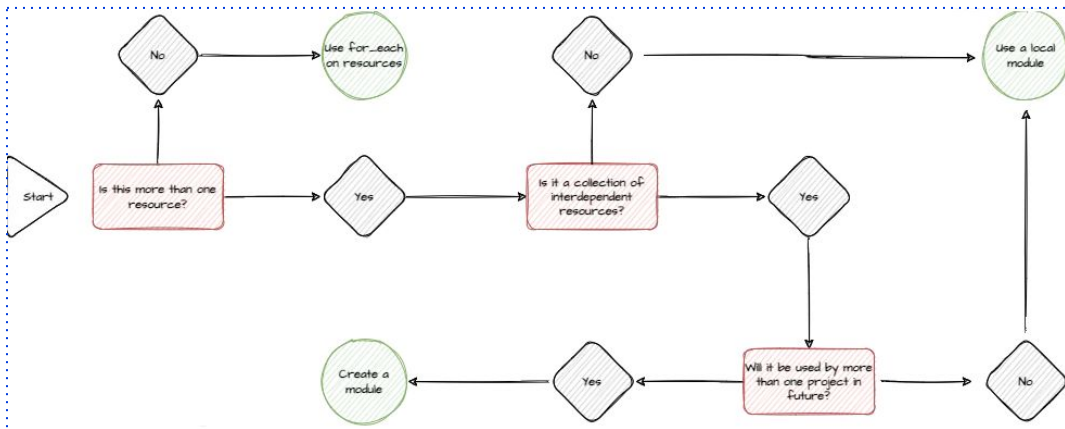
root module



# Modules terraform

## Quand utiliser un module ?

- DRY : Don't Repeat Yourself
- Les usecases sont souvent les mêmes, on veut capitaliser sur l'écriture de code terraform et sa variabilisation
- Dans le cas de modules externes : vouloir gérer un cycle de vie d'une partie de la stack indépendamment

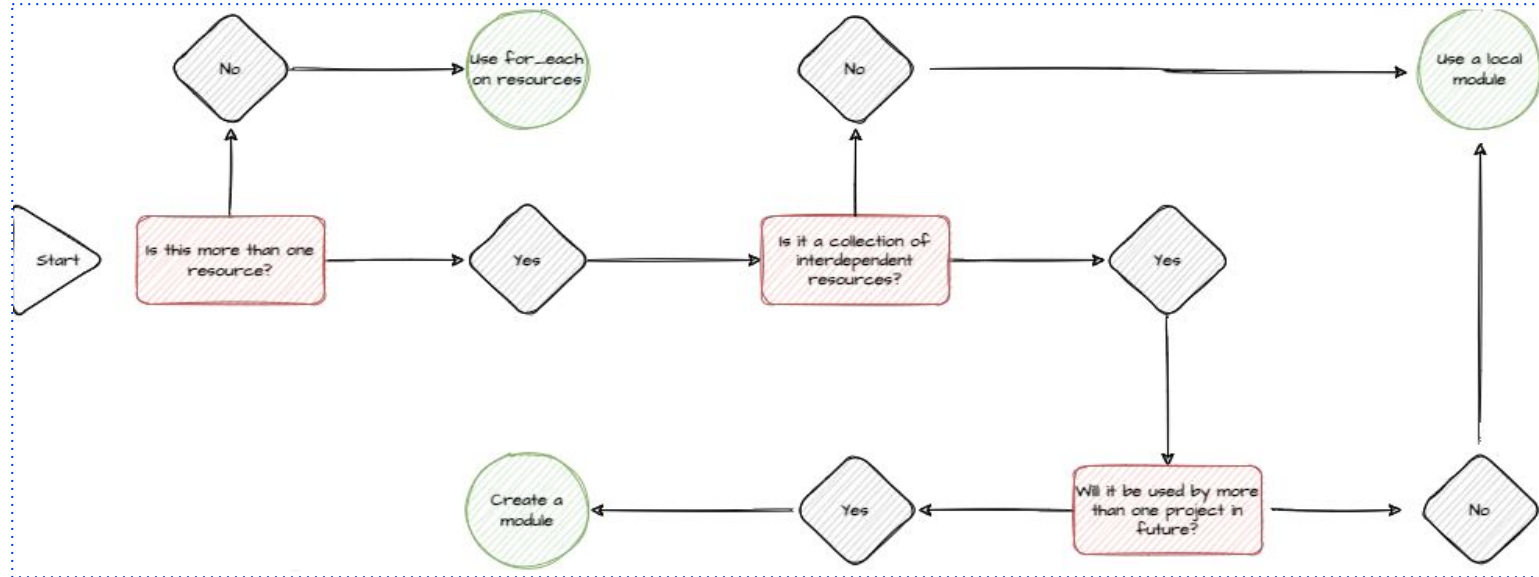


<https://medium.com/version-1/terraform-in-real-life-writing-modules-c0b6b5335218>



# Modules terraform

Quoi choisir : for\_each, module dédié, module local ?



<https://medium.com/version-1/terraform-in-real-life-writing-modules-c0b6b5335218>



# Modules terraform

## Les questions à se poser

- Regroupement:
  - Ressources qui seront systématiquement déployées ensemble
- Privilèges:
  - Les droits pour déployer les ressources sont identiques (ex : permissions liées à une équipe)
- Cycle de vie et longévité communes des ressources:
  - court terme (déploiement/destruction fréquente, ex : applicatif) ou long terme (déploiement moins fréquent, ex : réseaux, base de données)

## Exemple de modularisation d'une stack





# Modules terraform

## Comment utiliser un module ?

Pour être utilisé, un module doit simplement être référencé par sa **source**. Il peut contenir un certain nombre de variables obligatoires, ou optionnelles.

```
module "my_local_module" {  
  source = "./module-rep"  
  input_1 = "my-input-variable-1"  
}
```

```
module "instance1" {  
  source = "git::https://corporate-gitlab.com/tf-aws-vpc.git"  
  
  input_1 = "my-input-variable-1"  
  # ... any number of input variables ...  
  input_n = "my-input-variable-n"  
}
```

La référence au module peut être :

- **locale**
- **distante**



# Modules terraform

## Comment utiliser un module ?

- Possibilité d'instancier plusieurs fois un même module (en plusieurs blocs, ou en utilisant une boucle for\_each)

```
module "instance1" {  
    source = "./module-ec2"  
  
    instance_type = "t2.micro"  
}  
  
module "instance2" {  
    source = "./module-ec2"  
  
    instance_type = "t3.large"  
}
```

- On passe des variables (ou input variables) au module pour le personnaliser lors de l'exécution



# Modules terraform

## Comment utiliser un module ? (focus providers 1/2)

Par défaut, le module utilise le provider par défaut. Mais il est également possible de lui "passer des providers"

```
module "instance1" {  
  source = "git::https://corporate-gitlab.com/tf-aws-vpc.git "  
  
  providers = {  
    aws      = aws  
    aws-euw2 = aws.euw2  
  }  
  
  # ... any number of input variables  
}
```

main.tf

providers.tf

```
# The default "aws" configuration is used for AWS  
# resources in the root module where no explicit provider  
# instance is selected.  
provider "aws" {  
  region = "eu-west-1"  
}  
  
# An alternate configuration is also defined for a different  
# region, using the alias "euw2".  
provider "aws" {  
  alias   = "euw2"  
  region = "eu-west-2"  
}
```

At the root module level



# Modules terraform

## Comment utiliser un module ? (focus providers 2/2)

On peut donc définir à l'intérieur du module terraform quels alias de providers il attend en input du root module, afin de les utiliser en son sein.

```
terraform {  
  required_providers {  
    aws = {  
      source = "hashicorp/aws"  
      version = ">= 2.7.0"  
    }  
  }  
}
```

Inside the terraform module



# Modules terraform

## Bonnes pratiques

- Nommage des modules :
  - **terraform-[provider]-[fonction/scope]**
- Eviter l'**encapsulation** des modules : 2 niveau max sauf exception (module utilitaire type tag ou convention de nommage)
- On met dans un module des **ressources** partageant le **même cycle de vie** et qui sont **interdépendantes**
  - Il faut voir un module comme une “petite stack de ressources qui sont vouées à être liées de leur création à leur suppression”
- On ne **spécifie pas de block provider** dans un module qui n'est pas un *root* module (i.e. amené à être appelé par d'autres modules).
- On ne spécifie que les **versions minimales des providers** dans un module ( $\geq$ )
  - $\sim$  dans le “root module” pour éviter les breaking changes



# Modules terraform

## Bonnes pratiques (la suite)

- Lorsque l'on référence des **modules distants** (disponibles sur un autre repository git, qu'il soit privé ou publique), nous les référençons **toujours avec des versions spécifiques taggées**.

```
# refer to a specific (tagged) version on a terraform registry repo
module "consul" {
  source = "terraform-aws-modules/vpc/aws"
  version = "v3.18.1"
}

# refer to a specific tag on corporate repo
module "vpc" {
  source = "git::https://corporate-gitlab.com/tf-aws-vpc.git?ref=v1.2.0"
}
```



# Modules terraform

## Les faire vivre

- Définir une roadmap pour le module
- Recueillir les besoins utilisateurs et les classer par priorité
  - Plus le module coche les cases d'un besoin, plus il est utilisé
  - Démarche 80/20
- Documentation explicite sur les décisions / évolutions ([CHANGELOG.md](#) + [README.md](#))
- Adopter des principes open source dans le cycle de vie du module. Tips :
  - Créez une communauté,
  - Avoir un guide de contribution clairement défini et publié. ([CONTRIBUTING.md](#))
  - À terme, vous pourriez permettre aux membres de confiance de la communauté de posséder l'"ownership" de certains modules.



# TP4



<https://gitlab.ippon.fr/formation/terraform/tps/tp4>

# Modules terraform





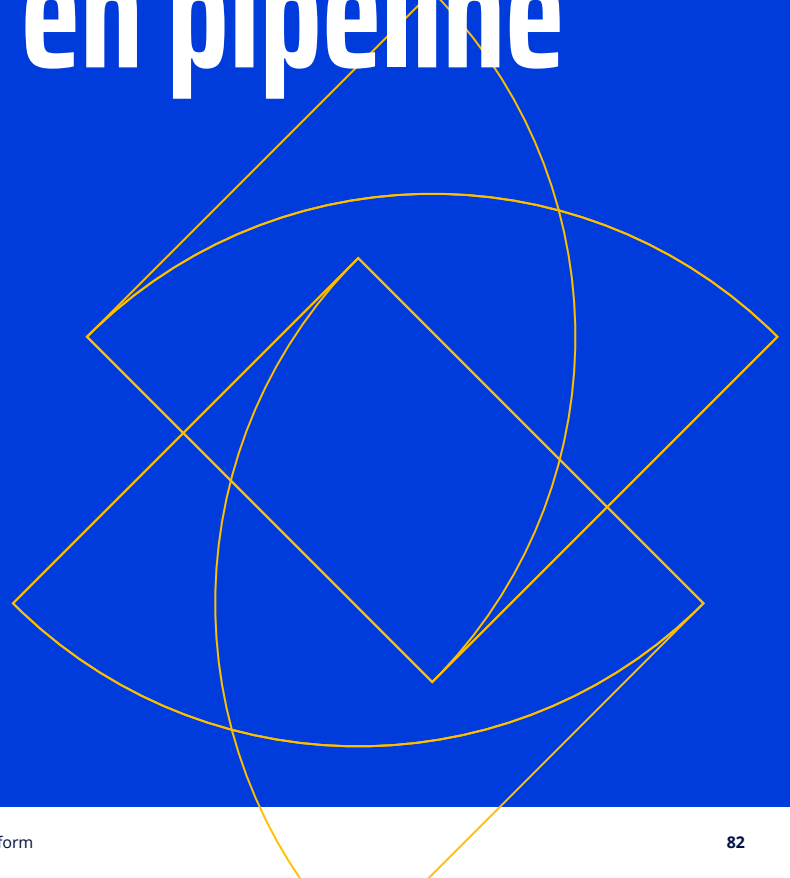
# Journée 2



- A. Intégration pipeline
- B. Paramètres, secrets
- C. Tests & tools
- D. Au quotidien



# — A. Intégration en pipeline




# — B. Paramètres, secrets



# Secrets

## Comment gérer les secrets ?

- Terraform ne gère pas les secrets pour vous. Il faudra passer par des outils tiers, tels que Hashicorp Vault, AWS Secret Manager / Parameter Store pour faire interagir terraform avec ces secrets
  - exemple : en retrouver la valeur pour initialiser / configurer une ressource (bdd)
- On peut utiliser terraform pour générer des valeurs aléatoires pour nos secrets
-  Tout mot de passe utilisé par terraform (qu'il l'ait créé ou non) est stocké en clair dans son state
  - d'où la nécessité de le chiffrer



# Secrets

## Comment créer directement les mots de passe sur AWS ?

- Secrets Manager permet de stocker des mots de passe sur AWS
- Créer un secret sous format JSON:

```
{
  "username": "admin",
  "password": "password"
}
```

- Récupérer les informations dans le terraform

```
data "aws_secretsmanager_secret_version" "creds" {
  # Fill in the name you gave to your secret
  secret_id = "db-creds"
}

locals {
  db_creds = jsondecode(
    data.aws_secretsmanager_secret_version.creds.secret_string
  )
}

resource "aws_db_instance" "example" {
  engine           = "mysql"
  engine_version  = "5.7"
  instance_class  = "db.t2.micro"
  name            = "example"
  # Set the secrets from AWS Secrets Manager
  username = local.db_creds.username
  password = local.db_creds.password
}
```



# — C. Tests & tools



# Terraform tools

## Des outils pour des besoins spécifiques

De nombreux outils gravitant autour de Terraform existent, il existe un projet [github](#) qui liste une grande partie d'entre eux.

Parmi les nombreux outils présent ont peut citer parmi les plus utiles/populaires :

- tfenv/tfswitch ⇒ permet de switcher de version de terraform facilement
- terraform-docs ⇒ générer la documentation à partir du code
- tflint ⇒ linter spécifique pour Terraform
- tfsec ⇒ Scan le code à la recherche de potentiel problème de sécurité
- checkov ⇒ Comme tfsec mais permet aussi de scanner les dockerfile
- infracost ⇒ Évalue le coût engendré par les modifications de la plateforme



# Testing terraform

## Pourquoi ?

- Terraform s'appuie sur un langage de programmation (HCL)
  - Pourquoi ne pas s'inspirer de nos confrères développeurs et écrire des tests unitaires sur son *infrastructure as code* ?
- Difficulté de tester une infrastructure complète :
  - dépendances entre composants
  - gros volumes de ressources

Solution ⇒ Effectuer les tests à l'échelle d'un module ou d'un ensemble de module de taille raisonnable.

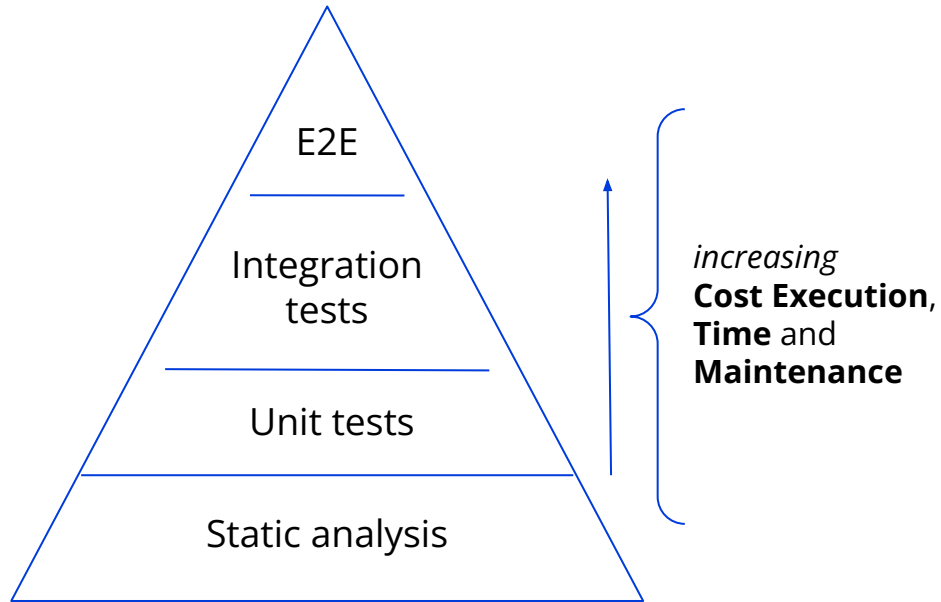




# Testing terraform

## Pyramide de test pour IaC

- **End-to-End**
  - terraform deploy on dev/non-production environnement
- **Integration**
  - terraform apply : deploy to a sandbox environnement
- **Units**
  - terraform plan
- **Statics**
  - terraform validate / tflint / tfsec...



# Testing terraform

## Solutions existantes

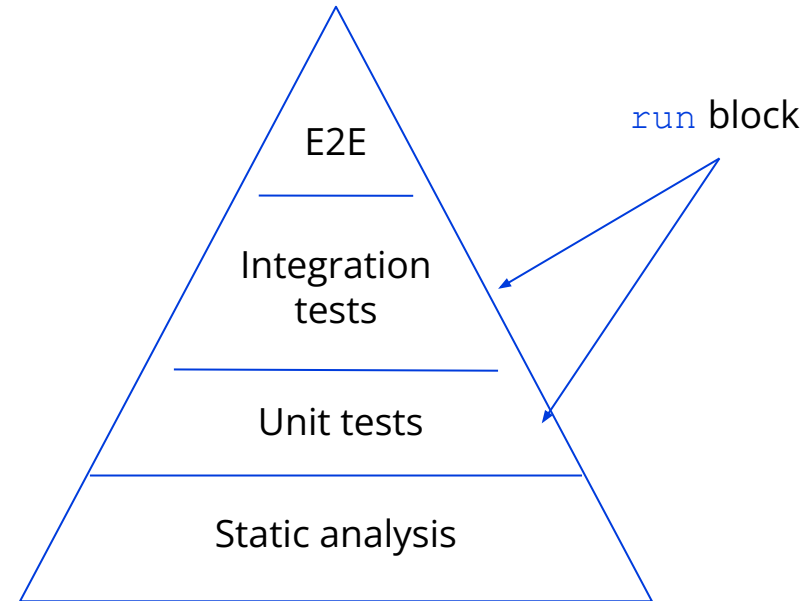
	<u>Terratest</u>	<u>Kitchen-Terraform</u>	<u>Terraform compliance</u>	<u>Native Terraform Tests</u>
Depuis	02/2016	06/2016	04/2017	10/2023
Popularité	++++	++	+	++ (new et grandissant)
Licence	Apache	Apache	MIT	Même que terraform
Langage framework	GoLang	Ruby	Python/HCL/Gherkin	GoLang
Langage à connaître	Notion en GoLang	Connaître le framework Kitchen Configuration par YAML Notion en Ruby	Behaviour Driven Development (BDD) basé sur <u>radish</u>	HCL
Principe	Test automatisé de bout en bout et pas seulement du contrôle de valeurs attendues	Plugins Kitchen pour tester terraform + contrôle avec <u>InSpec</u>	S'appui sur le pattern de "negative-testing".	Utiliser les nouveaux blocks <i>run</i> et <i>check</i> introduits par terraform 1.6.x



# Testing terraform

## Que propose nativement terraform ?

- Des tests
  - au plan (unitaires)
  - à l'apply (d'intégration)
- Des fichiers HCL dédiés pour nos tests
- Des backends dédiés pour les tests
- La commande `terraform test`
- Utilisation de modules et providers de test



# Advanced tf blocks

## Le block “check”

- “By using `check{}` blocks, we are able to continuously assert the health of our infrastructure.” – [source](#)
- Dispo à partir de terraform 1.5



# — D. Terraform au quotidien



# Use your IDE plugins

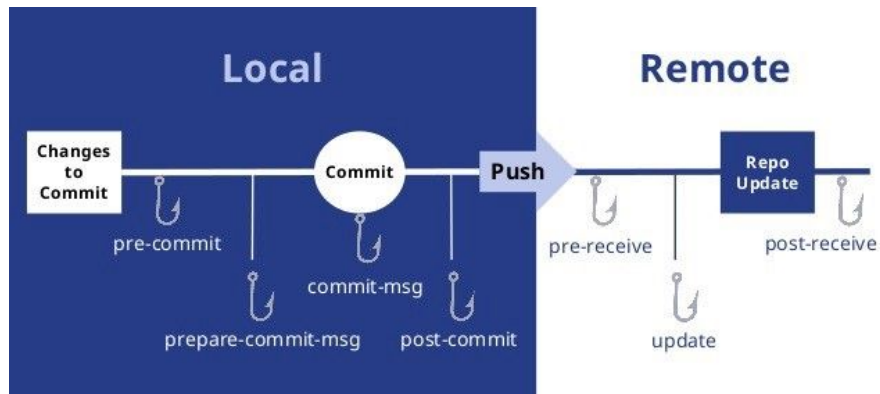
- Pour VSCode
  - <https://github.com/hashicorp/vscode-terraform>
- IntelliJ
  - <https://plugins.jetbrains.com/plugin/7808-terraform-and-hcl>



# Pre-commit

Simplifier le développement, améliorer la qualité de code et réduire la boucle de feedback

Qu'est ce qu'un git hook ? (.git/hooks)



Qu'est ce que pre-commit ?

*A framework for managing and maintaining multi-language pre-commit hooks*

— [pre-commit.com](https://pre-commit.com)



# Pre-commit

## Un fonctionnement simple, un fichier de configuration

### installation

Un fichier : `myrepo/.pre-commit-config.yaml`  
Une commande : `pre-commit install`  
Des binaires : python pip, binaires divers

### usage

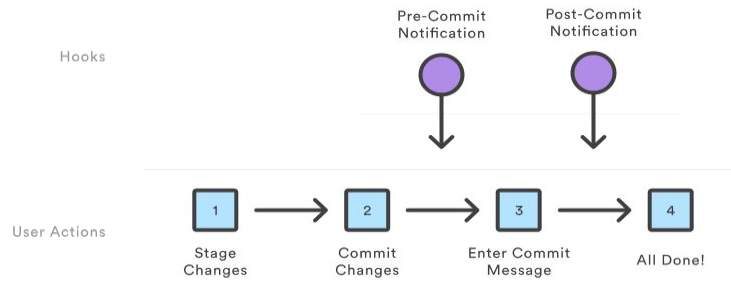
Par la commande

- `pre-commit run [name_of_your_hook]`
- `pre-commit run --all`

ou directement avec git:

- `git commit -am "feat: my pre-committed commit"`

```
1 repos:
2   - repo: git://github.com/antonbabenko/pre-commit-terraform
3     rev: v1.46.0
4   hooks:
5     - id: terraform_fmt
6     - id: terraform_tflint
7     - id: terraform_validate
8     - id: terraform_tfsec
9     args: ["."]
10    - id: terraform_docs
```





# Pre-commit

## Quels hooks, exécutés à quel endroit ?

### Choisir ses hooks :

- Par défaut
- Que l'on crée soit même
- Proposés et maintenus par la communauté

### Où exécuter pre-commit ?

- Sur le poste du développeur
- Rarement dans la CI/CD où on choisira de run les hooks, pas pre-commit



# Pre-commit

## Demo time

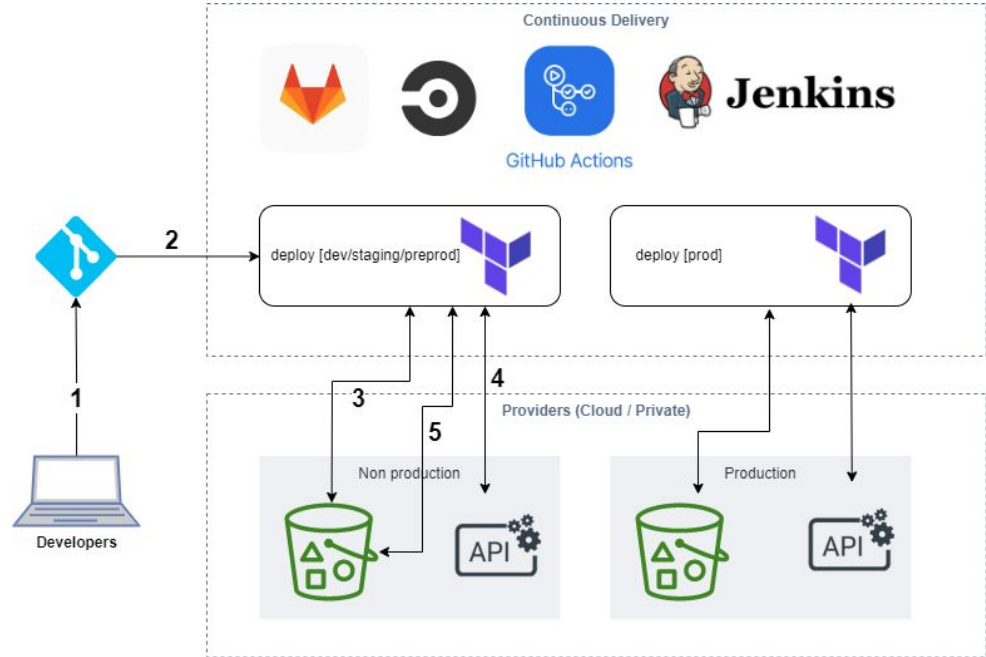
<https://github.com/alexis-renard/pre-commit-terraform-demo>



# Tips

## Développement / Déploiement standard

1. Les développeurs écrivent le code
2. Soumission d'une MR/PR
3. MR/PR validé  
⇒ le pipeline CI/CD exécute le code et commence par une validation du state (init)
4. Le plan est généré : appel APIs au travers des providers
5. Exécution du plan (apply) : modification de l'infrastructure et mise à jour du state en conséquence



# Tips

## Opérations courantes (1/2)

Il est essentiel d'établir un processus stable et sécurisé permettant de sécuriser au maximum son infrastructure et d'appliquer les mise à jour Terraform.

1. Toujours planifier en premier
  - a. Commencer toujours par générer un plan d'exécution et enregistrez le dans un fichier de sortie (option **-out**)
2. Mettre en œuvre un pipeline automatisé (Jenkins / GitlabCI / ...)
  - a. Garantie un contexte d'exécution cohérent
3. Utiliser des identifiants de type "service account" pour la CI
  - a. Évite à l'utilisateur de saisir ses identifiants
  - b. Facilite l'adoption du "least privileges"



# Tips

## Opérations courantes (2/2)

Il est essentiel d'établir un processus stable et sécurisé permettant de sécuriser au maximum son infrastructure et d'appliquer les mise à jour Terraform.

4. Terraform import
  - a. Pratique pour intégrer des ressources existantes créées hors terraform
  - b. Attention aux ressources complexes (type VM) qui comportent potentiellement des configurations faites à leur création, ce ne sera pas forcément pris en compte au moment de l'apply après l'import
  - c. Lorsqu'il n'y a pas d'effet de bord, privilégier la destruction/recreation à l'import
5. Ne **jamais** modifier le state Terraform à la main, utiliser les commande "**terraform state <sub-command>**"
6. Si jamais vous avez des scripts / outils externes, pensez à vérifier les dépendances et leurs mises à jour (patch de sécurité, etc...)
  - a. Idée : automatisation de ce processus possible avec l'outil [Dependabot](#)
7. Penser à faire des alias sur les commandes courantes :)



# Tips

## Terraform State

Terraform nécessite d'avoir un accès sensible à votre infrastructure pour fonctionner correctement. Il faut donc penser à sécuriser tout ce qui peut être exposé pour réduire les risques de sécurité, notamment le state qui peut contenir des informations sensibles pour votre organisation

1. Stocker le state dans un remote backend (pas en local)
  - a. Seuls la CI/CD et les administrateurs peuvent accéder au backend
2. Chiffrer le state
  - a. Ajoute une couche de sécurité en cas de compromission du lieu de stockage remote state.
3. Eviter de stocker des secrets dans le state
  - a. Préférez l'utilisation de providers dédiés si possible comme par exemple :
    - i. [vault](#) ⇒ gestion des identifiants, mot de passe , certificats, ...
    - ii. [tls](#) ⇒ génération de clés privés (PEM + OpenSSH)
    - iii. service spécifique d'un fournisseur cloud ⇒ (e.g. [Openstack Barbican](#) (/\), AWS Secret Manager)



# Tips

## Sécurité

1. Penser à verrouiller l'accès aux outputs sensibles
  - a. Via Terraform et l'attribut d'output sensitive, la valeur ne s'affiche pas en console (donc dans des logs) mais reste utilisable pour le reste des ressources.
2. Séparer les tâches/périmètres des modules
  - a. Délimiter les accès par périmètres
    - i. pour éviter qu'un seul identifiant puissent effectuer toutes les actions sur l'infrastructure
      1. exemple ⇒ module network : identifiant dédié au réseau avec des droits restreints à son périmètre
    - ii. pour éviter d'avoir un tfstate-for-all qui prendra beaucoup de temps à s'exécuter et dont les dépendances prendront du temps à être calculées
3. En environnement CI/CD, ne pas faire d' "apply" automatique sur les ressources sensibles pour éviter d'éventuelles régressions / effets de bords.
4. Penser à lancer des jobs de check / mise en conformité de ressources régulièrement.
  - a. exemple ⇒ une stack de sécurité (e.g. nouvelles règles config sur AWS)



# Tips

## Debug

Terraform dispose de logs détaillés qu'on peut activer en définissant la variable d'environnement **TF\_LOG** à la valeur souhaitée.

Les valeurs possible sont les suivantes (par ordre décroissant de verbosité):

- **TRACE**
- **DEBUG**
- **INFO**
- **WARN**
- **ERROR**

Il est possible soit en définissant cette variable (et sa valeur) au niveau de l'environnement d'exécution (export ou set), ou directement à l'exécution de la commande : **TF\_LOG=<VALEUR> terraform plan**

Il est possible de séparer les logs Terraform (core) des logs providers. Pour cela il suffit d'utiliser les variables :

- **TF\_LOG\_CORE**
- **TF\_LOG\_PROVIDER**

Pour persister les logs dans un fichier, il faut utiliser la variables **TG\_LOG\_PATH**

Bonus: **JSON** ⇒ produit des logs avec un niveau TRACE encodé au format JSON (expérimental pour le moment)





# Open TOFU

## Quand Hashicorp attaque, la communauté contre attaque

Le **soucis** : changement de licence vers la *Business Source License (BSL)*

Les **conséquences** :

- perte de confiance de la communauté en Hashicorp à garder ses solutions open source
- création d'un fork communautaire vigoureusement soutenu
  - pour protéger les business ayant fait un go to terraform
  - car plus de limitations liées à la roadmap d'Hashicorp (par rapport à son offre payante)



# Bonus



# Lock your providers

## Le fameux fichier `.terraform.lock.hcl`

Permet de savoir quelles versions des providers ont été utilisées lors du dernier `plan/apply` d'une stack terraform.

Les contraintes de version spécifiées dans le fichier `versions.tf` déterminent quelles versions des dépendances sont potentiellement compatibles (minimales et maximales), *mais* lors du `plan/apply`, Terraform va prendre la version la plus à jour sur les versions spécifiques à utiliser. Il écrit ensuite automatiquement les versions qu'il a sélectionnées dans le fichier `.terraform.lock.hcl` afin de pouvoir (par défaut) prendre à nouveau les mêmes décisions à l'avenir.

!/ en cas d'OS différents utilisés

```
+provider "registry.terraform.io/hashicorp/azurerem" {
+ version   = "2.30.0"
+ constraints = "~> 2.12"
+ hashes = [
+   "h1:FJwsuowaG5CIdZ0WQyFZH9r6kIJeRkKts9+GcRsTz1+Y=",
+   "h1:c/nt5XrDYm1mUir2KuFijYebPcwKqS9CRGd3duDSGfY=",
+   "h1:yre4Ph76g9H84MbuH2z25MulDjSA4FsrX653807PccY=",
+   "zh:04f0a50bb2ba92f3bea6f0a9e549ace5a4c13ef0cbb6975494cac0ef7d4acb43",
+   "zh:2082e12548ebcdd6fd73580e83f626ed4ed13f8cdfd51205d8696ffe54f30734",
+   "zh:246bcc449e9a92679fb30f3c0a77f05513886565e2dcc66b16c4486f51533064",
+   "zh:24de3930625ac9014594d79bfa42d600eca65e9022b9668b54bfd0d924e21d14",
+   "zh:2a22893a576ff6f268d9bf81c4a56406f7ba79f77826f6df51ee787f6d2840a",
+   "zh:2b27485e19c2aaa9f15f29c4cff46154a9720647610171e30fc6c18ddc42ec28",
+   "zh:435f24ce1fb2b63f7f02aa3c84ac29c5757cd29ec4d297ed0618423387f7bd4",
+   "zh:7d99725923de5240ff8b34b5510569aa4ebdc0bdb27b7bac2aa911a8037a3893",
+   "zh:7e3b5d0af3b7411dd9dc65ec9ab6caee8c191aee0fa7f20fc4f51716e67f50c0",
+   "zh:da0af4552bef5a29b88f6a0718253f3bf71ce471c959816eb7602b0dad469ca",
+ ]
+}
```



# Un terraform-base-template

La bonne manière de centraliser les conventions et bien partir de scratch

<https://github.com/ippontech/terraform-base-template>



# Custom Provider

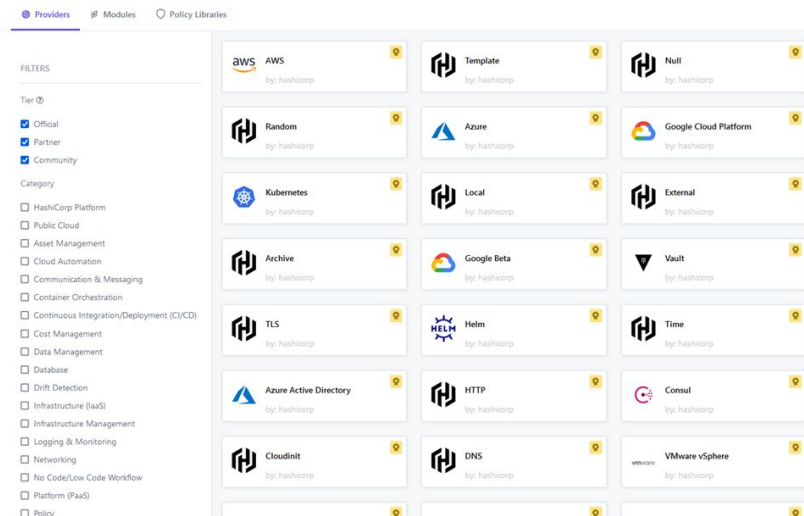
## Rappel

### Provider

- Garants du processus de création et de maintenance des ressources
  - Multitude de provider disponible
    - utilitaire :
      - template, null, random, local, external,...
    - Hashicorp :
      - vault, consul...
    - Cloud provider public :
      - aws, gcp, azure, vmware (vSphere)
    - Container :
      - Kubernetes, Helm

### Custom Provider

- Pas de provider pour votre fournisseur de service
  - Cloud privé interne
  - Fonctionnalités propriétaire non open-sourcable
- Extension d'un provider existant
  - Besoin spécifique



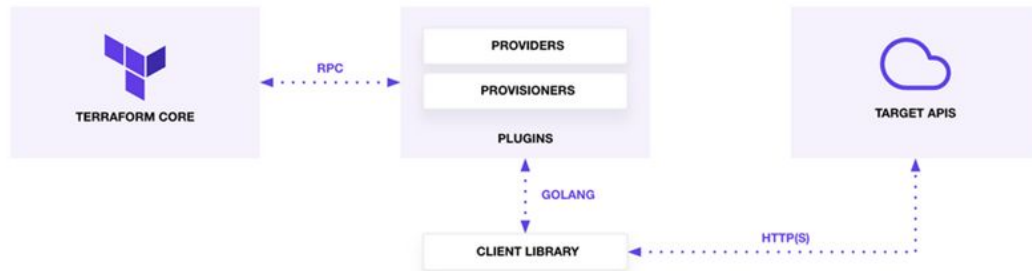
# Custom Provider

## Pré-requis

- Simplifier l'interaction avec une API custom
- Permet d'avoir un langage unique et commun pour toutes les opérations sur les infrastructures

### Prérequis:

- Une API représentant une « ressource » implémentant les méthodes :
  - create
  - read
  - delete
  - update (optionnel)
- GoLang
- [Semantic Versioning](#)
- Documentation



# Custom Provider

## Hands-on ?

Ippon live twitch to develop a terraform provider

- <https://www.twitch.tv/ippontech/v/1958401141?sr=a&t=0s>





**[www.ippon.fr](http://www.ippon.fr)**

contact@ippon.fr — +33 1 46 12 48 48 — @ipponTech





# Template code

```
#=====
# Count pour création conditionnelle d'une ressource
#=====

variable "example_1_create_security_group" {
    type    = bool
    default = true
}

resource "aws_security_group" "test1" {
    count          = var.example_1_create_security_group ? 1 : 0
    description   = "My test security group 1"
}
```

