

HashiCorp

Terraform



Semifir

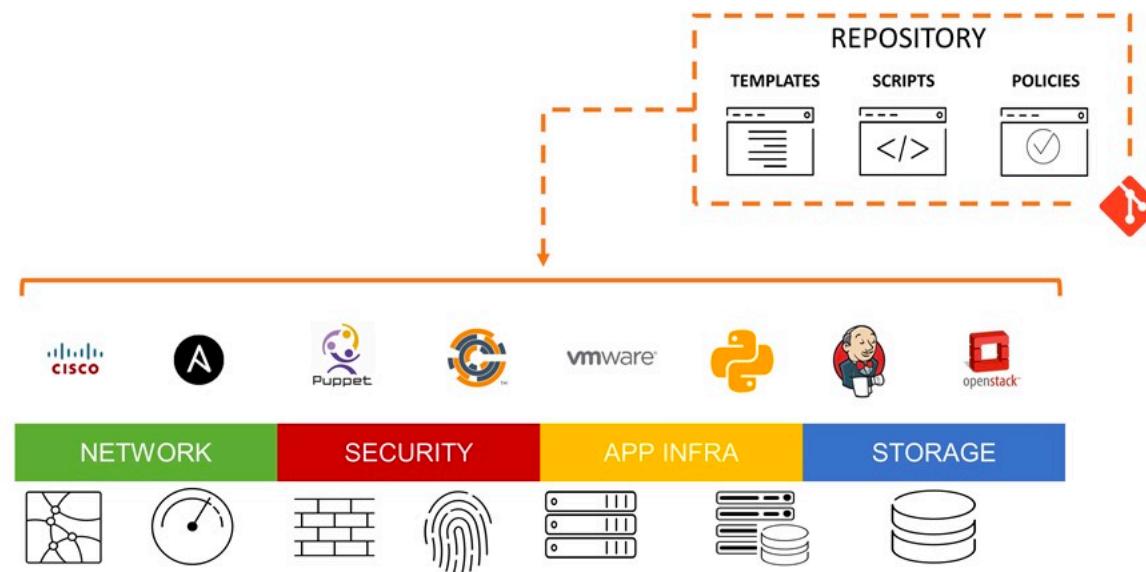


Terraform introduction à l'infrastructure-as-code (IaS)



Terraform est un outil qui permet de rendre possible l'**Infrastructure as Code**. C'est à dire qu'il permet de construire, modifier et versioner l'infrastructure que vous souhaitez déployer à l'aide de simple fichiers texte. Autrement dit, vous codez votre infrastructure, et vous pouvez suivre les mêmes bonnes pratiques de développement: versioning, tests, etc.

INFRASTRUCTURE as CODE





Les concepts principaux : Les Providers

Terraform est utilisé pour créer, gérer et mettre à jour des ressources d'infrastructure telles que des machines physiques, des machines virtuelles, des commutateurs de réseau, des conteneurs, etc. Presque tous les types d'infrastructure peuvent être représentés comme une ressource dans Terraform.

Un provider est responsable de la compréhension des interactions avec l'API et de l'exposition des ressources. Les providers sont généralement des services IaaS (AWS, GCP, Microsoft Azure, OpenStack), PaaS (Heroku, par exemple) ou SaaS (Terraform Cloud, DNSimple, CloudFlare, par exemple).



Les concepts principaux : Les Provisioners

Les provisioners sont exécutés après le démarrage ou la mise à disposition d'une machine virtuelle. Cela permet par exemple de lancer des commandes sur une instance EC2 fraîchement déployée, ou encore de bootstraper un agent Chef . Et pourquoi pas les deux en même temps !

```
resource "aws_instance" "web" {
  # ...

  provisioner "local-exec" {
    when      = "destroy"
    command  = "echo 'Destroy-time provisioner'"
  }
}
```



Les concepts principaux : Les Backends

Les backends de terraform sont utilisées pour stocker le fichier appelé “tfstate” qui résulte du lancement de terraform. Cela permet de stocker les informations de déploiement nécessaire au bon fonctionnement de terraform à différents endroits (en local, sur un bucket S3, sur Consul, etc). L'avantage est que certains backends permettent le travail collaboratif avec des systèmes de “lock” permettant de s'assurer qu'un seul processus d'exécution de terraform ne s'applique à un instant T.



Les concepts principaux : Les Modules

Toutes les intégrations existantes de Terraform avec différents providers (AWS, Azure et beaucoup d'autres sont réalisées sous la forme de plugins. L'outil étant open-source il est donc assez aisé (pour peu que l'on sache développer en Go) de développer son propre plugin pour des besoins spécifiques ou pas encore intégrés dans l'outil.



gruntwork.io

Pour résumer

Pour résumer, Terraform se rapproche fortement d'outils comme Amazon CloudFormation ou encore Azure Templates. Il permet donc de décrire des composants d'une infrastructure à déployer. Sa grande force est d'être agnostique de la technologie utilisée. Et de permettre le travail collaboratif assez simplement tout en étant facilement extensible.

Terraform bénéficie aussi d'une communauté très active et les releases sont très fréquentes.

Installer Terraform sur Linux





Nous allons voir ensemble comment installer TerraForm sur un CentOS 7

Tous d'abord, mettez à jours votre système :

```
$ sudo yum update
```

Ensuite, nous installerons wget et décompresserons les paquets s'ils ne sont pas déjà installés

```
$ sudo yum install wget unzip
```

Nous sommes maintenant prêts à télécharger le fichier zip Terraform pour Linux à partir du site officiel.

```
$ wget https://releases.hashicorp.com/terraform/0.11.13/terraform_0.11.13_linux_amd64.zip
```



Ensuite, nous allons décompresser l'archive dans **/usr/local/bin/**

```
$ sudo unzip ./terraform_0.11.13_linux_amd64.zip -d /usr/local/bin/
```

Terminé. La seule chose qui reste à présent est de vérifier si terraform est installé avec succès, avec la commande suivante:

```
[centos@localhost ~]$ terraform -v  
Terraform v0.11.13
```

```
Your version of Terraform is out of date! The latest version  
is 0.12.7. You can update by downloading from www.terraform.io/downloads.html
```

Si vous souhaitez mettre à jour votre version de Terraform, il suffit de recommencer la manipulation avec la dernière version disponible, ici la 0.12.7

Installer Terraform sur Windows





Nous allons voir ensemble comment installer TerraForm sur un Windows

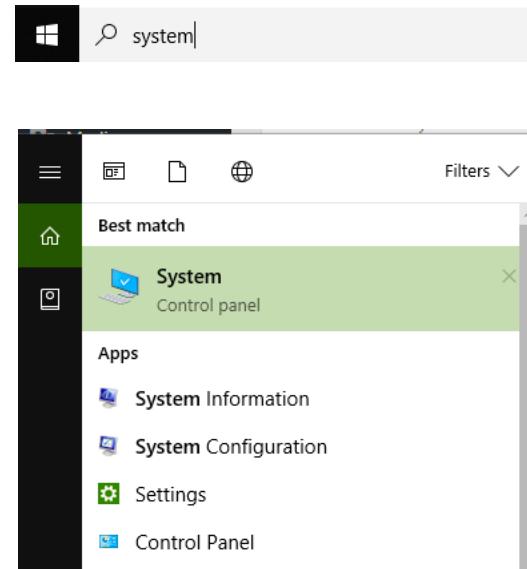
Téléchargez terraform pour Windows

Remarque: Terraform est présenté sous forme d'archive zip. Par conséquent, après avoir téléchargé Terraform, décomptez le package. Terraform s'exécute sous la forme d'un seul binaire nommé terraform. Tous les autres fichiers du package peuvent être supprimés en toute sécurité et Terraform fonctionnera toujours.

Copiez les fichiers du zip vers “c: \ terraform” par exemple. C'est notre **chemin terraform** .
La dernière étape consiste à vérifier que le binaire terraform est disponible sur **PATH** .



Dans Rechercher, recherchez, puis sélectionnez: Système (Panneau de configuration).

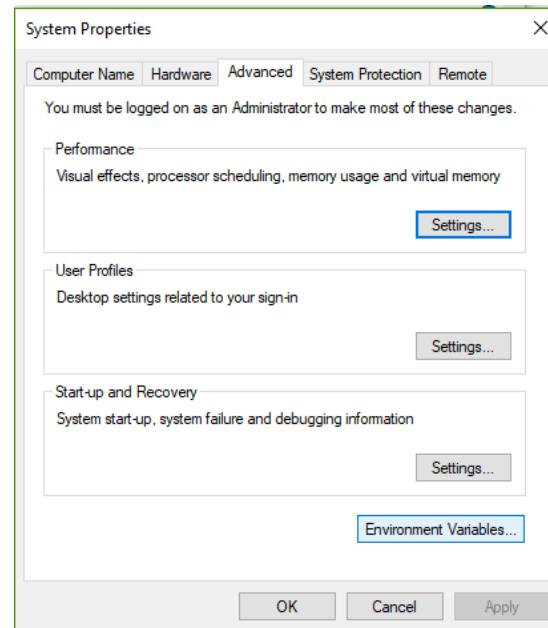


Cliquez sur le lien **Paramètres système avancés** .

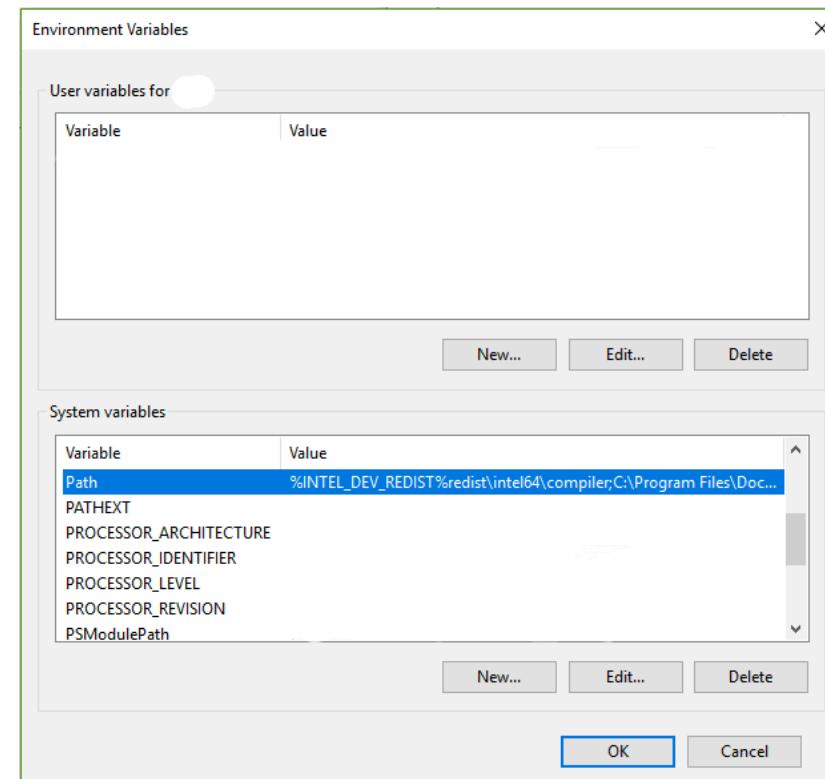




Cliquez sur Variables d'environnement .

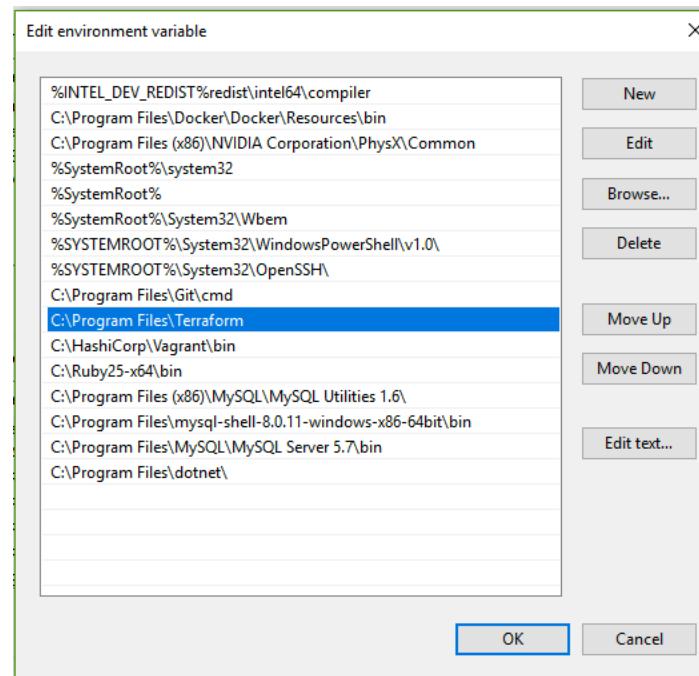


Dans la section **Variables système**, recherchez la variable d'environnement **PATH** et sélectionnez-la. Cliquez sur **Edit**. Si la variable d'environnement **PATH** n'existe pas, cliquez sur **Nouveau**.





Dans la fenêtre **Edit System Variable** (ou **New System Variable**), ajoutez à la fin de la variable d'environnement **PATH** la valeur du chemin terraform ex. "C:\ terraform;".



Cliquez sur **OK**. Fermez toutes les fenêtres restantes en cliquant sur **OK**.
Rouvrez la fenêtre d'invite de commande et exécutez terraform.

Le langage





Terraform utilise son propre langage de configuration, conçu pour permettre une description concise de l'infrastructure. Le langage Terraform est déclaratif, décrivant un objectif visé plutôt que les étapes permettant d'atteindre cet objectif.

La syntaxe du langage Terraform consiste uniquement en quelques éléments de base :

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

<BLOCK TYPE> "<BLOCK LABEL>" "<BLOCK LABEL>" {
  # Block body
  <IDENTIFIER> = <EXPRESSION> # Argument
}
```

- Les **blocks** sont des conteneurs pour d'autres contenus et représentent généralement la configuration d'un type d'objet, tel qu'une ressource. Les blocs ont un *type de bloc*, peuvent avoir zéro ou plusieurs *étiquettes* et un *corps* contenant un nombre quelconque d'arguments et de blocs imbriqués. La plupart des fonctionnalités de Terraform sont contrôlées par des blocs de niveau supérieur dans un fichier de configuration.
- Les **identifier** assignent une valeur à un nom. Ils apparaissent dans des blocs.
- Les **expressions** représentent une valeur, littéralement ou en référençant et en combinant d'autres valeurs. Ils apparaissent sous forme de valeurs pour les arguments ou dans d'autres expressions.



Les ressources

Les ressources sont l'élément le plus important du langage Terraform. Chaque bloc de ressources décrit un ou plusieurs objets d'infrastructure, tels que des réseaux virtuels, des instances de calcul ou des composants de niveau supérieur tels que les enregistrements DNS

Les déclarations de ressources peuvent inclure un certain nombre de fonctionnalités avancées, mais seul un petit sous-ensemble est requis pour une utilisation initiale.

Exemple :

```
resource "aws_instance" "web" {
    ami           = "ami-a1b2c3d4"
    instance_type = "t2.micro"
}
```

Ce ressource bloc déclare une ressource d'un type donné ("`aws_instance`") avec un nom local donné ("`web`"). *Le nom est utilisé pour faire référence à cette ressource depuis ailleurs dans le même module Terraform, mais n'a aucune signification en dehors de la portée d'un module.*

Le type et le nom de la ressource servent ensemble d'identifiant pour une ressource donnée et doivent donc être uniques dans un module.

Dans le corps du bloc (entre { et }) se trouvent les arguments de configuration de la ressource elle-même. La plupart des arguments de cette section dépendent du type de ressource et, dans cet exemple `ami` et `instance_type` sont tous deux des arguments définis spécifiquement pour aws.

```
resource "aws_iam_role" "example" {
  name = "example"

  # assume_role_policy is omitted for brevity in this example. See the
  # documentation for aws_iam_role for a complete example.
  assume_role_policy = "..."

}

resource "aws_iam_instance_profile" "example" {
  # Because this expression refers to the role, Terraform can infer
  # automatically that the role must be created first.
  role = aws_iam_role.example.name
}

resource "aws_iam_role_policy" "example" {
  name   = "example"
  role   = aws_iam_role.example.name
  policy = jsonencode({
    "Statement" = [
      # This policy allows software running on the EC2 instance to
      # access the S3 API.
      "Action" = "s3:*",
      "Effect" = "Allow",
    ],
  })
}

resource "aws_instance" "example" {
  ami          = "ami-a1b2c3d4"
  instance_type = "t2.micro"

  # Terraform can infer from this that the instance profile must
  # be created before the EC2 instance.
  iam_instance_profile = aws_iam_instance_profile.example

  # However, if software running in this EC2 instance needs access
  # to the S3 API in order to boot properly, there is also a "hidden"
  # dependency on the aws_iam_role_policy that Terraform cannot
  # automatically infer, so it must be declared explicitly:
  depends_on = [
    aws_iam_role_policy.example,
  ]
}
```

Les Méta-arguments d'une ressource

depends_on

Utilisez le méta-argument **depends_on** pour gérer les dépendances de ressources cachées que Terraform ne peut pas déduire automatiquement.

La spécification explicite d'une dépendance n'est nécessaire que lorsqu'une ressource repose sur le comportement d'une autre ressource mais *n'accède pas* aux données de cette ressource dans ses arguments.

Cet argument est disponible dans tous les resourceblocs, quel que soit le type de ressource.

Les Méta-arguments d'une ressource

count et for_each

Par défaut, un bloc de resource configure un seul objet d'infrastructure. Cependant, vous souhaitez parfois gérer plusieurs objets similaires, tels qu'un pool fixe d'instances de calcul. Terraform a deux façons de le faire:

count et for_each.

Quand utiliser for_each au lieu de count ?

Si vos instances de ressources sont presque identiques, count est approprié. Si certains de leurs arguments nécessitent des valeurs distinctes qui ne peuvent pas être directement dérivées d'un entier, il est plus sûr de les utiliser for_each.



Le count

Le count méta-argument accepte un nombre entier et crée autant d'instances de la ressource. Chaque instance est associée à un objet d'infrastructure distinct et chacune est créée, mise à jour ou détruite séparément lors de l'application de la configuration.

```
resource "aws_instance" "server" {
    count = 4 # create four similar EC2 instances

    ami          = "ami-a1b2c3d4"
    instance_type = "t2.micro"

    tags {
        Name = "Server ${count.index}"
    }
}
```



Le for_each

Le méta-argument `for_each` accepte une carte ou un ensemble de chaînes et crée une instance pour chaque élément de cette carte ou de cet ensemble.

```
resource "azurerm_resource_group" "rg" {
    for_each = {
        a_group = "eastus"
        another_group = "westus2"
    }
    name      = each.key
    location = each.value
}
```

```
# default configuration
provider "google" {
  region = "us-central1"
}

# alternative, aliased configuration
provider "google" {
  alias  = "europe"
  region = "europe-west1"
}

resource "google_compute_instance" "example" {
  # This "provider" meta-argument selects the google provider
  # configuration whose alias is "europe", rather than the
  # default configuration.
  provider = google.europe

  # ...
}
```

Les Méta-arguments d'une ressource : Les providers

Terraform permet éventuellement de définir plusieurs configurations alternatives ("aliasées") pour un seul fournisseur, afin de permettre la gestion des ressources de différentes régions dans des services multi-régions, etc. Le méta-argument provider annule le comportement par défaut de Terraform. de sélectionner une configuration de fournisseur basée sur le nom du type de ressource.

Par défaut, Terraform prend le mot initial dans le nom du type de ressource (séparé par des traits de soulignement) et sélectionne la configuration par défaut pour ce fournisseur nommé. Par exemple, le type de ressource `google_compute_instance` est automatiquement associé à la configuration par défaut du fournisseur nommé `google`.

En utilisant le méta-argument provider, une configuration de fournisseur avec alias peut être sélectionnée.



Les providers

Bien que les ressources constituent la structure principale du langage Terraform, les *comportements* des ressources dépendent des types de ressources qui leur sont associés et ces types sont définis par les **providers**.

La plupart des fournisseurs disponibles correspondent à une plate-forme d'infrastructure en nuage ou locale et offrent des types de ressources correspondant à chacune des fonctionnalités de cette plate-forme.

Les fournisseurs ont généralement besoin de leur propre configuration pour spécifier les URL, les régions, les paramètres d'authentification, etc. Tous les types de ressources appartenant au même fournisseur partagent la même configuration, évitant ainsi la répétition de ces informations communes dans toutes les déclarations de ressources.



Une configuration d'un provider est créée à l'aide d'un bloc provider :

```
provider "google" {  
  project = "acme-app"  
  region  = "us-central1"  
}
```

Le nom donné dans l'en-tête de bloc (« google » dans cet exemple) est le nom du provider à configurer. Terraform associe chaque type de ressource à un provider en prenant le premier mot du nom du type de ressource (séparé par des traits de soulignement). Le provider "google" est donc supposé être le provider du nom du type de ressource **google_compute_instance**.

Le corps du bloc (entre { et }) contient des arguments de configuration pour le provider lui-même. La plupart des arguments de cette section sont spécifiés par le fournisseur lui-même; dans cet exemple à la fois **project** et **regions** ont spécifiques au google provider.



Voici la liste des providers actuellement disponible pour Terraform :

[ACMÉ](#)

[Akamai](#)

[Alibaba Cloud](#)

[Archiver](#)

[Arukas](#)

[Avi Vantage](#)

[Aviatrix](#)

[AWS](#)

[Azur](#)

[Azure Active Directory](#)

[Azure Stack](#)

[Bitbucket](#)

[Brightbox](#)

[CenturyLinkCloud](#)

[Chef](#)

[Circonus](#)

[Cisco ASA](#)

[Cisco ACI](#)

[UltraDNS](#)

[Voûte](#)

[VMware NSX-T](#)

[VMware vCloud Director](#)

[VMware vRA7](#)

[VMware vSphere](#)

[Yandex](#)

[UCloud](#)

[Cloudflare](#)

[CloudScale.ch](#)

[CloudStack](#)

[Cordonnier](#)

[Consul](#)

[Datadog](#)

[DigitalOcean](#)

[DNS](#)

[DNSSimple](#)

[DNSMadeEasy](#)

[Docker](#)

[Dyn](#)

[Exoscale](#)

[Externe](#)

[F5 BIG-IP](#)

[Rapidement](#)

[Moteur souple](#)

[FortiOS](#)

[TelefonicaOpenCloud](#)

[Modèle](#)

[TencentCloud](#)

[Terraform](#)

[Terraform Cloud](#)

[TLS](#)

[Triton](#)

[StatusCake](#)

[GitHub](#)

[GitLab](#)

[Google Cloud Platform](#)

[Grafana](#)

[Échelle de grille](#)

[Hedvig](#)

[Barre](#)

[Heroku](#)

[Hetzner Cloud](#)

[HTTP](#)

[HuaweiCloud](#)

[Icinga2](#)

[Allumage](#)

[InfluxDB](#)

[JDCloud](#)

[Kubernetes](#)

[Livret](#)

[Linode](#)

[Local](#)

[Logentries](#)

[LogicMonitor](#)

[Mailgun](#)

[MongoDB Atlas](#)

[MySQL](#)

[Naver Cloud](#)

[Netlify](#)

[Nouvelle relique](#)

[Nomade](#)

[NS1](#)

[Nul](#)

[Nutanix](#)

[1 & 1](#)

[Pile ouverte](#)

[OpenTelekomCloud](#)

[OpsGenie](#)

[Infrastructure cloud Oracle](#)

[Oracle Cloud Platform](#)

[Oracle Public Cloud](#)

[OVH](#)

[Paquet](#)

[PagerDuty](#)

[Réseaux Palo Alto](#)

[PostgreSQL](#)

[PowerDNS](#)

[ProfitBricks](#)

[RabbitMQ](#)

[Propriétaire de ranch](#)

[Rancher2](#)

[au hasard](#)

[Échelle de droite](#)

[Rundeck](#)

[RunScope](#)

[Scaleway](#)

[Selectel](#)

[SignalFx](#)

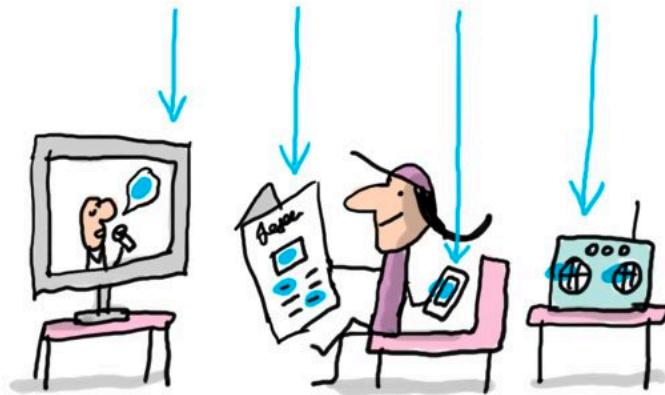
[Skytap](#)

[Couche souple](#)

[Spotinst](#)



INFORMATIONS



Initialisation d'un provider

Chaque fois qu'un nouveau provider est ajouté à la configuration - explicitement via un bloc provider ou en ajoutant une ressource de ce fournisseur - Terraform doit initialiser le fournisseur avant de pouvoir être utilisé. L'initialisation télécharge et installe le plug-in du fournisseur afin qu'il puisse être exécuté ultérieurement.

L'initialisation du fournisseur est l'une des actions de **terraform init**. L'exécution de cette commande téléchargera et initialisera tous les fournisseurs qui ne sont pas déjà initialisés.

Les variables

Les variables d'entrée servent de paramètres pour un module Terraform, permettant de personnaliser certains aspects du module sans modifier le code source du module et de partager les modules entre différentes configurations.

Lorsque vous déclarez des variables dans le module racine de votre configuration, vous pouvez définir leurs valeurs à l'aide des options de la CLI et des variables d'environnement. Lorsque vous les déclarez dans des modules enfants , le module appelant doit transmettre des valeurs dans le bloc module.

Déclaration d'une variable d'entrée

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

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

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

L'étiquette après le mot-clé variable est un nom pour la variable, qui doit être unique parmi toutes les variables du même module. Ce nom permet d'affecter une valeur à la variable de l'extérieur et de référencer la valeur de la variable depuis le module.



Le nom d'une variable peut être n'importe quel identifiant valide, *à l'exception* des suivants:

source / version / providers / count / for_each / lifecycle / depends_on / locals

Ces noms sont réservés aux méta-arguments dans les blocs de configuration de module et ne peuvent pas être déclarés en tant que noms de variables.

Utilisation des valeurs de variable d'entrée

Dans le module qui a déclaré une variable, sa valeur est accessible à partir d'expressions telles que `var.<NAME>`, où `<NAME>` correspond à l'étiquette donnée dans le bloc de déclaration:

```
resource "aws_instance" "example" {
    instance_type = "t2.micro"
    ami           = var.image_id
}
```

La valeur attribuée à une variable est accessible uniquement à partir d'expressions du module dans lequel elle a été déclarée.

Contraintes de type

L' argument type dans un bloc variable vous permet de restreindre le type de valeur qui sera accepté comme valeur pour une variable. Si aucune contrainte de type n'est définie, une valeur de tout type est acceptée.

Bien que les contraintes de type soient facultatives, il est recommandé de les spécifier. Ils servent de rappels faciles aux utilisateurs du module et permettent à Terraform de renvoyer un message d'erreur utile si le type incorrect est utilisé.

Les mots-clés de type pris en charge sont:

string / number / bool

list(<TYPE>)

set(<TYPE>)

map(<TYPE>)

object({<ATTR NAME> = <TYPE>, ... })

tuple([<TYPE>, ...])

Les valeurs de sortie

Les valeurs de sortie ressemblent aux valeurs de retour d'un module Terraform et ont plusieurs utilisations.

Un module enfant peut utiliser les sorties pour exposer un sous-ensemble de ses attributs de ressources à un module parent.

Un module racine peut utiliser les sorties pour imprimer certaines valeurs dans la sortie de la CLI après exécution `terraform apply`.

Déclaration d'une valeur de sortie

Chaque valeur de sortie exportée par un module doit être déclarée à l'aide d'un output bloc:

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

L'étiquette située immédiatement après le mot-clé output est le nom, qui doit être un identifiant valide . Dans un module racine, ce nom est affiché à l'utilisateur. dans un module enfant, il peut être utilisé pour accéder à la valeur de la sortie.

L'argument value prend une expression dont le résultat doit être retourné à l'utilisateur. Dans cet exemple, l'expression fait référence à l' attribut private_ip exposé par une ressource aws_instance définie ailleurs dans ce module (non représenté). Toute expression valide est autorisée en tant que valeur de sortie.

Le CLI





Les commandes Terraform (CLI)

```
$ terraform
Usage: terraform [-version] [-help] <command> [args]

The available commands for execution are listed below.
The most common, useful commands are shown first, followed by
less common or more advanced commands. If you're just getting
started with Terraform, stick with the common commands. For the
other commands, please read the help and docs before usage.

Common commands:
  apply      Builds or changes infrastructure
  console    Interactive console for Terraform interpolations
  destroy    Destroy Terraform-managed infrastructure
  env        Workspace management
  fmt        Rewrites config files to canonical format
  get        Download and install modules for the configuration
  graph     Create a visual graph of Terraform resources
  import    Import existing infrastructure into Terraform
  init      Initialize a Terraform working directory
  output    Read an output from a state file
  plan      Generate and show an execution plan
  providers Prints a tree of the providers used in the configuration
  refresh   Update local state file against real resources
  show      Inspect Terraform state or plan
  taint     Manually mark a resource for recreation
  untaint  Manually unmark a resource as tainted
  validate  Validates the Terraform files
  version   Prints the Terraform version
  workspace Workspace management

All other commands:
  0.12upgrade  Rewrites pre-0.12 module source code for v0.12
  debug        Debug output management (experimental)
  force-unlock Manually unlock the terraform state
  push         Obsolete command for Terraform Enterprise legacy (v1)
  state        Advanced state management
```

Terraform est contrôlé via une interface de ligne de commande (CLI) très facile à utiliser.

Pour afficher une liste des commandes disponibles à tout moment, exécutez simplement `terraform` sans argument:



Commande: apply

La commande **terraform apply** est utilisée pour appliquer les modifications nécessaires pour atteindre l'état souhaité de la configuration ou l'ensemble prédéfini d'actions générées par un **terraform plan**

Commande: console

La commande **terraform console** fournit une console interactive pour évaluer les expressions .

Commande: destroy

La commande **terraform destroy** est utilisée pour détruire l'infrastructure gérée par Terraform.



Commande: env

La commande **terraform env** est obsolète. La commande **terraform workspace** devrait être utilisée à la place.

Commande: workspace

La commande **terraform workspace** est utilisée pour gérer les espaces de travail.

Commande: get

La commande **terraform get** est utilisée pour télécharger et mettre à jour les modules mentionnés dans le module racine.



Commande: fmt

La commande **terraform env** est obsolète. La commande **terraform workspace** devrait être utilisée à la place.

Commande: graph

La commande **terraform graph** est utilisée pour générer une représentation visuelle d'une configuration ou d'un plan d'exécution. La sortie est au format DOT, qui peut être utilisé par [GraphViz](#) pour générer des graphiques.

Commande: import

La commande **terraform import** est utilisée pour importer des ressources existantes dans Terraform.



Commande: init

La commande **terraform init** est utilisée pour initialiser un répertoire de travail contenant les fichiers de configuration Terraform. Il s'agit de la première commande à exécuter après l'écriture d'une nouvelle configuration Terraform ou le clonage d'une configuration existante à partir du contrôle de version. Vous pouvez exécuter cette commande plusieurs fois en toute sécurité.

Commande: output

La commande **terraform output** permet d'extraire la valeur d'une variable de sortie du fichier d'état.

Commande: plan

La commande **terraform plan** est utilisée pour créer un plan d'exécution. Terraform effectue une actualisation, sauf si explicitement désactivé, puis détermine les actions nécessaires pour atteindre l'état souhaité spécifié dans les fichiers de configuration.

Cette commande est un moyen pratique de vérifier si le plan d'exécution d'un ensemble de modifications correspond à vos attentes sans apporter de modification aux ressources réelles ni à l'état. Par exemple, **terraform plan** peut être exécuté avant de valider une modification du contrôle de version, afin de garantir qu'il se comportera comme prévu.



Commande: providers

La commande **terraform providers** imprime des informations sur les fournisseurs utilisés dans la configuration actuelle.

Commande: refresh

La commande **terraform refresh** est utilisée pour réconcilier l'état de Terraform (via son fichier d'état) avec l'infrastructure du monde réel. Cela peut être utilisé pour détecter toute dérive par rapport au dernier état connu et pour mettre à jour le fichier d'état.

Cela ne modifie pas l'infrastructure, mais modifie le fichier d'état. Si l'état est modifié, cela peut entraîner des modifications lors du prochain plan ou de son application.

Commande: show

La commande **terraform show** est utilisée pour fournir une sortie lisible par l'homme à partir d'un fichier d'état ou d'un fichier de plan. Cela peut être utilisé pour inspecter un plan afin de s'assurer que les opérations planifiées sont prévues ou pour inspecter l'état actuel tel que le voit Terraform.



Commande: providers

La commande **terraform providers** imprime des informations sur les fournisseurs utilisés dans la configuration actuelle.

Commande: refresh

La commande **terraform refresh** est utilisée pour réconcilier l'état de Terraform (via son fichier d'état) avec l'infrastructure du monde réel. Cela peut être utilisé pour détecter toute dérive par rapport au dernier état connu et pour mettre à jour le fichier d'état.

Cela ne modifie pas l'infrastructure, mais modifie le fichier d'état. Si l'état est modifié, cela peut entraîner des modifications lors du prochain plan ou de son application.

Commande: show

La commande **terraform show** est utilisée pour fournir une sortie lisible par l'homme à partir d'un fichier d'état ou d'un fichier de plan. Cela peut être utilisé pour inspecter un plan afin de s'assurer que les opérations planifiées sont prévues ou pour inspecter l'état actuel tel que le voit Terraform.

Commande: state

La commande **terraform state** est utilisée pour la gestion avancée des états. À mesure que votre utilisation de Terraform devient plus avancée, vous devrez peut-être parfois modifier l' état de Terraform. Plutôt que de modifier directement l'état, les commandes terraform state peuvent être utilisées dans de nombreux cas.
Cette commande est une sous-commande imbriquée, ce qui signifie qu'elle contient d'autres sous-commandes.

Commande: validate

La commande **terraform validate** permet de valider la syntaxe des fichiers terraform. Terraform effectue une vérification de la syntaxe sur tous les fichiers terraform du répertoire et affiche une erreur si l'un des fichiers ne se valide pas.
Cette commande **ne vérifie pas le formatage** (par exemple, tabulations / espaces, nouvelles lignes, commentaires, etc.).

Les éléments suivants peuvent être signalés:

Syntaxe [HCL](#) non valide (par exemple, guillemet de fin ou signe égal manquant)
références HCL non valides (par exemple, nom de variable ou attribut inexistant)
même providerdéclaré plusieurs fois
même moduledéclaré plusieurs fois
même resourcedéclaré plusieurs fois
modulenom invalide

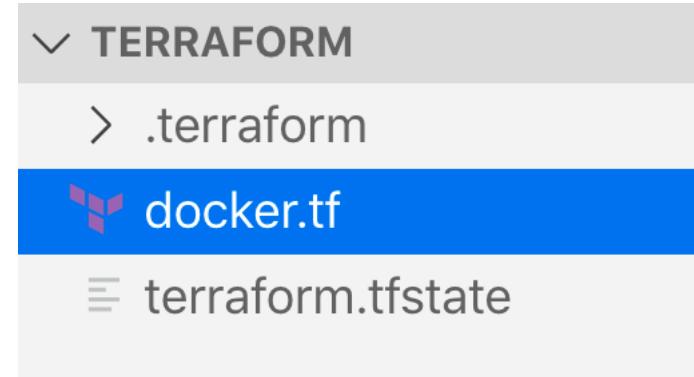
interpolation utilisé dans des endroits où il est non pris en charge (par exemple variable, depends_on, module.source, provider)
valeur manquante pour une variable (aucun -var foo=...indicateur, -var-file=foo.varsindicateur, TF_VAR_foovariable d'environnement terraform.tfvarsou valeur par défaut dans la configuration)

Mise en pratique simple avec docker



Utilisation de Terraform pour l'automatisation de la création d'une infrastructure avec Docker

Tous d'abord nous allons créer un dossier pour Terraform et créer un fichier main.tf





On va d'abord initialiser un provider pour notre ressource

```
1 provider "docker" {  
2   host = "unix:///var/run/docker.sock"  
3 }
```

Ensute nous allons créer une première ressource qui correspondra à notre conteneur

```
# Create an Nginx container
resource "docker_container" "nginx" {
    image = "${docker_image.nginx.latest}"
    name  = "enginecks"
    ports {
        internal = 80
        external = 80
    }
}
```

Puis une deuxième ressource qui correspondra à notre image

```
resource "docker_image" "nginx" {  
    name = "nginx:latest"  
}
```



Par la suite dans notre terminal on se rend dans notre dossier de travail

```
antoine: ~ $ cd /Users/antoine/Desktop/terraform □
```



On initialise Terraform

```
antoine: ~/Desktop/terraform $ terraform init  
Initializing the backend...  
Initializing provider plugins...
```

Mise en pratique simple avec docker



```
[antoine: ~/Desktop/terraform $ terraform plan
Refreshing Terraform state in-memory prior to plan...
The refreshed state will be used to calculate this plan, but will not be
persisted to local or remote state storage.

docker_image.nginx: Refreshing state... [id=sha256:5a3221f0137beb960c34b9cf4455424b6210160fd618c5e79401a07d6e5a2cednginx:latest]
docker_container.nginx: Refreshing state... [id=3ea557da31b52cc59afad94b28b672f8a00173b43d0365e6c6404176e3ce9ab4]

-----
```

```
An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
+ create
```

```
Terraform will perform the following actions:
```

```
# docker_container.nginx will be created
+ resource "docker_container" "nginx" {
  + attach          = false
  + bridge          = (known after apply)
  + container_logs = (known after apply)
  + exit_code       = (known after apply)
  + gateway         = (known after apply)
  + id              = (known after apply)
  + image            = "sha256:5a3221f0137beb960c34b9cf4455424b6210160fd618c5e79401a07d6e5a2ced"
  + ip_address      = (known after apply)
  + ip_prefix_length = (known after apply)
  + log_driver      = "json-file"
  + logs            = false
  + must_run        = true
  + name             = "enginecks"
  + network_data    = (known after apply)
  + restart          = "no"
  + rm               = false
  + start            = true

  + ports {
    + external = 80
    + internal = 80
    + ip       = "0.0.0.0"
    + protocol = "tcp"
  }
}
```

```
Plan: 1 to add, 0 to change, 0 to destroy.
```

```
-----
```

Note: You didn't specify an "-out" parameter to save this plan, so Terraform
can't guarantee that exactly these actions will be performed if
"terraform apply" is subsequently run.

On test notre configuration

Mise en pratique simple avec docker



```
[antoine: ~/Desktop/terraform $ terraform apply
docker_image.nginx: Refreshing state... [id=sha256:5a3221f0137beb960c34b9cf4455424b6210160fd618c5e79401a07d6e5a2cednginx:latest]
docker_container.nginx: Refreshing state... [id=3ea557da31b52cc59afad94b28b672f8a00173b43d0365e6c6404176e3ce9ab4]
```

An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
+ create

Terraform will perform the following actions:

```
# docker_container.nginx will be created
+ resource "docker_container" "nginx" {
    + attach          = false
    + bridge          = (known after apply)
    + container_logs = (known after apply)
    + exit_code       = (known after apply)
    + gateway         = (known after apply)
    + id              = (known after apply)
    + image            = "sha256:5a3221f0137beb960c34b9cf4455424b6210160fd618c5e79401a07d6e5a2ced"
    + ip_address      = (known after apply)
    + ip_prefix_length = (known after apply)
    + log_driver      = "json-file"
    + logs            = false
    + must_run        = true
    + name             = "enginecks"
    + network_data    = (known after apply)
    + restart          = "no"
    + rm               = false
    + start            = true

    + ports {
        + external = 80
        + internal = 80
        + ip       = "0.0.0.0"
        + protocol = "tcp"
    }
}
```

Plan: 1 to add, 0 to change, 0 to destroy.

Do you want to perform these actions?

Terraform will perform the actions described above.
Only 'yes' will be accepted to approve.

Enter a value: yes

```
docker_container.nginx: Creating...
docker_container.nginx: Creation complete after 1s [id=f7bac7af00d498ba1dce02171790a1a6c4331a179a6e80dd9184c356a4007927]
```

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

Puis on lance notre infrastructure



Si on regarde nos conteneur Docker, on constate que Nginx a bien été lancé

```
[antoine: ~/Desktop/terraform $ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              PORTS          NAMES
f7bac7af00d4        5a3221f0137b      "nginx -g 'daemon of..."   About a minute ago   Up About a minute   0.0.0.0:80->80/tcp   enginecks
```

Mise en pratique simple avec docker

```
antoine: ~/Desktop/terraform $ terraform destroy
docker_image.nginx: Refreshing state... [id=sha256:5a3221f0137beb960c34b9cf4455424b6210160fd618c5e79401a07d6e5a2cednginx:latest]
docker_container.nginx: Refreshing state... [id=f7bac7af00d498ba1dce02171790a1a6c4331a179a6e80dd9184c356a4007927]

An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
- destroy

Terraform will perform the following actions:

# docker_container.nginx will be destroyed
resource "docker_container" "nginx" {
  - attach          = false -> null
  - gateway         = "172.17.0.1" -> null
  - id              = "f7bac7af00d498ba1dce02171790a1a6c4331a179a6e80dd9184c356a4007927" -> null
  - image           = "sha256:5a3221f0137beb960c34b9cf4455424b6210160fd618c5e79401a07d6e5a2ced" -> null
  - ip_address      = "172.17.0.2" -> null
  - ip_prefix_length = 16 -> null
  - log_driver       = "json-file" -> null
  - logs             = false -> null
  - must_run         = true -> null
  - name             = "enginecks" -> null
  - network_data     = [
    {
      - gateway        = "172.17.0.1"
      - ip_address     = "172.17.0.2"
      - ip_prefix_length = 16
      - network_name   = "bridge"
    },
  ] -> null
  - restart          = "no" -> null
  - rm               = false -> null
  - start            = true -> null

  - ports {
    - external = 80 -> null
    - internal = 80 -> null
    - ip       = "0.0.0.0" -> null
    - protocol = "tcp" -> null
  }
}

# docker_image.nginx will be destroyed
resource "docker_image" "nginx" {
  - id      = "sha256:5a3221f0137beb960c34b9cf4455424b6210160fd618c5e79401a07d6e5a2cednginx:latest" -> null
  - latest = "sha256:5a3221f0137beb960c34b9cf4455424b6210160fd618c5e79401a07d6e5a2ced" -> null
  - name   = "nginx:latest" -> null
}

Plan: 0 to add, 0 to change, 2 to destroy.

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

Enter a value: yes
```

On peu maintenant supprimer notre infrastructure