

Terraform - Infrastructure-as-Code (IaC)

LOGIN UE NUAGE et DEVOPS

Eloi Perdereau, Hélène Coullon

<https://helene-coullon.fr/pages/ue-nuage-23-24/>

IMT Atlantique

Table of contents

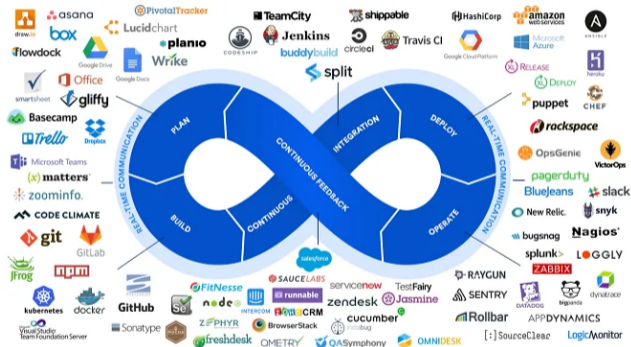
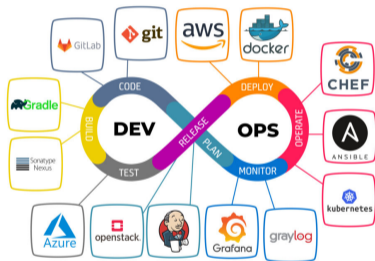
1. Introduction
2. Concepts of Terraform
3. Providers
4. Good practices
5. Your turn...

Introduction

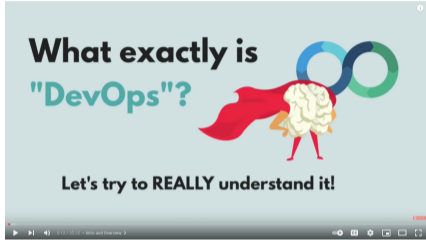
What you have seen so far

- **Virtualization and hypervisors**
 - Foundation of the Cloud computing
- **OpenStack**
 - The open-source operating system of the Cloud
 - Could be used for public and private Clouds
 - OVH Cloud (public)
 - IMT Atlantique (private)
- **Docker and Docker compose**
 - automate deployments
 - make deployments portable
- **Kubernetes**
 - orchestration of containerized applications
 - continuously organize, monitor, update, heal containers

DevOps and Cloud is a large ecosystem



DevOps (SRE) skills



Colors

- already known
- this course
- What is not studied in FIL

1. Concepts of development
2. Operating systems [DEVOS]
3. Networking and security
4. Containers
5. Automated CI/CD [HACKATON, PROCOM]
6. Cloud providers
7. Containers orchestration
8. Monitoring
9. Infrastructure as Code
10. Scripting
11. Version control



- This module is at the crossroads of **Cloud computing** and **Infrastructure-as-Code**
- Terraform is an **laC tool** initially made to **provision resources on Cloud providers**
 - Terraform is a provisioning tool (as Pulumi)
 - Terraform is not a configuration tool (Ansible is)
 - Terraform is not specific to containerized apps and systems (Docker, docker-compose and Kubernetes are)

Advantages of Terraform

- Terraform can manage infrastructure on **multiple Cloud platforms**.
- The **human-readable configuration language** helps you write infrastructure code.
- **Terraform's state** allows you to track resource changes throughout your deployments.
- You can commit your configurations to **version control** to collaborate safely.
- **1,000 providers** to manage resources on Amazon Web Services (AWS), Azure, Google Cloud Platform (GCP), etc.
- You can **compose resources from different providers** into reusable Terraform configurations called modules.
- Terraform's configuration language is **declarative**, meaning that it describes the desired end-state for your infrastructure, not how to get it.

Why Terraform?

I can do that through the graphical interfaces of Cloud providers! Yes but...

- Long and error-prone manual procedures.
- Difficult and error-prone when collaborating.
- Not scalable.

I can do that with Cloud providers CLIs and scripts! Yes but...

- You have to know as many CLIs as the number of Cloud providers you are using.
- A script is less specialized and structured than IaC, more difficult to write/read and maintain.
- You have to manually handle the state of your infrastructure which is difficult and error prone.

Concepts of Terraform

Declarative state : declare *what* not *how*

The desired state is written by the DevOps in `.tf` files.

- The **order** of provisioning is determined automatically.
- Terraform will **create** infrastructure in the right order.

Declarative state : declare *what* not *how*

The desired state is written by the DevOps in `.tf` files.

- The **order** of provisioning is determined automatically.
- Terraform will **create** infrastructure in the right order.
- The order is defined when resources **refer** to each other.

Declarative state : declare *what* not *how*

The desired state is written by the DevOps in `.tf` files.

- The **order** of provisioning is determined automatically.
- Terraform will **create** infrastructure in the right order.
- The order is defined when resources **refer** to each other.
- **Changes** in the declared state are compared against the state file.

We can create multiple versions of the same replicated infrastructure (e.g. dev, prod).

Resources

- A resource can represent anything. e.g. VM, docker image, virtual network, ip, user, account, role, etc.
- Providers furnish an API that lists
 1. Available resource **types**.
 2. For each of them their **parameters**.

Resources

- A resource can represent anything. e.g. VM, docker image, virtual network, ip, user, account, role, etc.
- Providers furnish an API that lists
 1. Available resource **types**.
 2. For each of them their **parameters**.

Terraform Core

- Configuration : every **.tf** files \Rightarrow resources declarations.
- The current directory constitutes the **root module**.
- State file : contains the **current state** of resources under Terraform's management.
- Upon each CLI call, the state file is **refreshed** with the actual resources.

Basic Architecture

Resources

- A resource can represent anything. e.g. VM, docker image, virtual network, ip, user, account, role, etc.
- Providers furnish an API that lists
 1. Available resource **types**.
 2. For each of them their **parameters**.

Terraform Core

- Configuration : every **.tf** files \Rightarrow resources declarations.
- The current directory constitutes the **root module**.
- State file : contains the **current state** of resources under Terraform's management.
- Upon each CLI call, the state file is **refreshed** with the actual resources.

Terraform detects changes in the configuration and **plan** API calls accordingly.

Commands for different stages

```
terraform init
```

Initialize the working directory and download providers.

Commands for different stages

`terraform init`

Initialize the working directory and **download** providers.

`terraform plan`

Produce an execution **plan** with details on what to add/delete/change by comparing the '.tf' configurations and the state file.

Plans can be stored to be applied in the future.

Commands for different stages

terraform init

Initialize the working directory and **download** providers.

terraform plan

Produce an execution **plan** with details on what to add/delete/change by comparing the '.tf' configurations and the state file.

Plans can be stored to be applied in the future.

terraform apply

Produce a plan and **execute** it. A planned execution may fail if the provider doesn't agree with Terraform's API calls.

Commands for different stages

terraform init

Initialize the working directory and download providers.

terraform plan

Produce an execution plan with details on what to add/delete/change by comparing the '.tf' configurations and the state file.

Plans can be stored to be applied in the future.

terraform apply

Produce a plan and execute it. A planned execution may fail if the provider doesn't agree with Terraform's API calls.

terraform destroy

Calls the provider to delete managed resources.

HCL Language Syntax (1): Attributes

aka "argument", "parameter", "field", "property", "key-value pair", "entry"

Attributes are distinguished with the **equal** sign = meaning *assignment*.

The value can be any expressions: function calls, lists, objects, references, etc.

- `name = "redis server"`
- `credentials = file("./creds.json")`
- `labels = { app = "redis" }`
- `image = docker_image.myimage.name`
- etc.

HCL Language Syntax (1): Attributes

aka "argument", "parameter", "field", "property", "key-value pair", "entry"

Attributes are distinguished with the **equal** sign = meaning *assignment*.

The value can be any expressions: function calls, lists, objects, references, etc.

- `name = "redis server"`
- `credentials = file("./creds.json")`
- `labels = { app = "redis" }`
- `image = docker_image.myimage.name`
- etc.

Multiple definitions of an attribute are **forbidden**. They are *single assignment*.

In addition to arguments within a block, there are a few **meta attributes** that have special semantics, e.g. `count`, `for_each` and `depends_on`.

HCL Language Syntax (2): Blocks

Blocks

e.g. `resource "docker_image" "redis" { ... }`

- Have a mandatory **key** identifier, here `resource`. It have a meaning in the context where it is defined, like standard attributes.
- Strings can be attached, here `docker_image` and `redis`.
- They can be **referred** in another part of the `.tf` configuration.

HCL Language Syntax (2): Blocks

Blocks

e.g. `resource "docker_image" "redis" { ... }`

- Have a mandatory **key** identifier, here `resource`. It have a meaning in the context where it is defined, like standard attributes.
- Strings can be attached, here `docker_image` and `redis`.
- They can be **referred** in another part of the `.tf` configuration.

Block can be **embedded**. e.g. in a container resource:

```
mounts { volume_options { no_copy = true }}
```


HCL Language Syntax (2): Blocks

Blocks

e.g. `resource "docker_image" "redis" { ... }`

- Have a mandatory **key** identifier, here `resource`. It have a meaning in the context where it is defined, like standard attributes.
- Strings can be attached, here `docker_image` and `redis`.
- They can be **referred** in another part of the `.tf` configuration.

Block can be **embedded**. e.g. in a container resource:

```
mounts { volume_options { no_copy = true }}
```

Multiple embedded blocks with the same keyword are sometimes **allowed**. It usually results in a list of objects. e.g. to refer to a particular mount option:

```
mounts[0].volume_options[0].no_copy
```

Kinds of top-level blocks

Terraform has concepts for each kind of block that can be declared at the **top-level**.

resource, data, provider and variables

The references for those blocks are found in the provider's documentation at <https://registry.terraform.io/providers/>.

- The *resource* block is the main state declarations of **managed resources**.

Kinds of top-level blocks

Terraform has concepts for each kind of block that can be declared at the **top-level**.

resource, data, provider and variables

The references for those blocks are found in the provider's documentation at <https://registry.terraform.io/providers/>.

- The *resource* block is the main state declarations of **managed resources**.
- The *data* block is for **read-only** resource.

Kinds of top-level blocks

Terraform has concepts for each kind of block that can be declared at the **top-level**.

resource, data, provider and variables

The references for those blocks are found in the provider's documentation at <https://registry.terraform.io/providers/>.

- The *resource* block is the main state declarations of **managed resources**.
- The *data* block is for **read-only** resource.
- The *provider* block sets configuration parameters for a provider.

Kinds of top-level blocks

Terraform has concepts for each kind of block that can be declared at the **top-level**.

resource, data, provider and variables

The references for those blocks are found in the provider's documentation at <https://registry.terraform.io/providers/>.

- The *resource* block is the main state declarations of **managed resources**.
- The *data* block is for **read-only** resource.
- The *provider* block sets configuration parameters for a provider.
- Blocks for module variables

There are a few other top-level blocks, e.g. *module*, *check*, *import*.

Variables and References

Variables

Modules (including the root module) can have **three kinds** of user variables: *Inputs*, *Outputs*, and *Locals*.

Declared with the *variable*, *output*, and *locals* top-level blocks respectively.

Variables and References

Variables

Modules (including the root module) can have **three kinds** of user variables: *Inputs*, *Outputs*, and *Locals*.

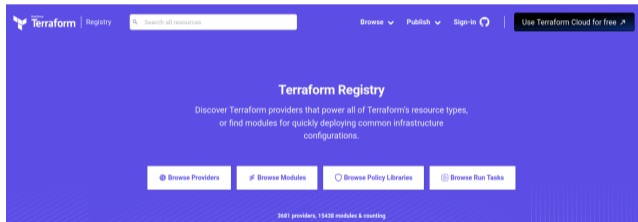
Declared with the *variable*, *output*, and *locals* top-level blocks respectively.

References to resources, data sources and variables

- Resources attributes are referenced with **type** and **name** of the resource, e.g. *docker_image.redis.image_id*
- To reference a *data source*, we use the keyword **data**, e.g. *data.docker_image.redis.image_id*.
- For input and local variables, we use **var** and **local** keywords, e.g. *var.my_input_var*
local.my_local_var
- Terraform has other such special variable keywords, e.g. *module*, *each*, *path*.

Providers

Provider registry



Good practices

Objective: avoid troubleshooting

- Read and understand carefully each declarations and plan.
- Version control your Terraform codes. Beware not to commit secrets.
- CI/CD on your Terraform infrastructure.
- Store the Terraform state files on remote storages with lock mechanisms.

Your turn...
