Terraform - Infrastructure-as-Code (IaC)

FIL A3 Cloud Computing

Eloi Perdereau, Hélène Coullon

https://helene-coullon.fr/pages/ue-terraform-23-24/

IMT Atlantique

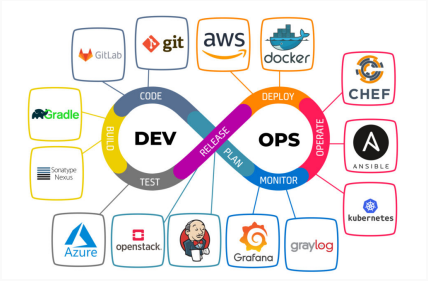
Table of contents

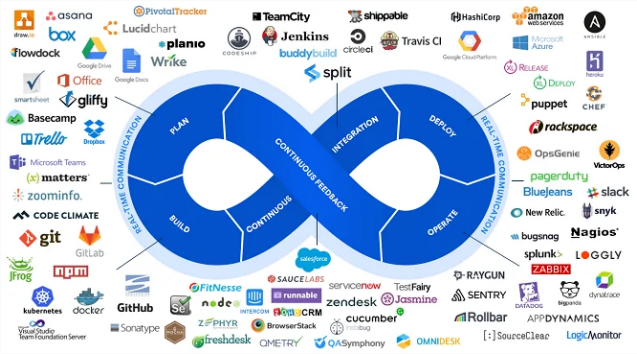
Introduction

Cloud computing - what you have seen so far

- Virtualization and hypervisors
  - Foundation of the Cloud computing
- AWS
  - One possible public Cloud provider
- OpenStack
  - The open-source operating system of the Cloud
  - Could be used for public and private Clouds
    - OVH Cloud (public)
    - IMT Atlantique (private)

What you have seen in the DevOps course





- Ansible: a configuration tool (Infrastructure as Code)
- Docker and docker-compose: containers
- Kubernetes: a containers orchestrator

DevOps (SRE) skills

Colors

- already known
- DevOps course
- This year
- What is not studied in FIL

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

Terraform

- This module is at the crossroads of Cloud computing and Infrastructure-as-Code
- Terraform is an IaC 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.

Organization

1. A bit of theory to understand the language and the concepts properly
2. A tutorial to learn how to use Terraform from concrete examples
3. A project to learn how to search and find information to write Terraform codes

Evaluated skills

- CG2 project (by 2)

Concepts of Terraform

Automatic ordering

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.

Automatic ordering

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.

Automatic ordering

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).

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.

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 ⇒ 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 ⇒ 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 refresh`

Updates the state file by querying the provider.

Commands for different stages

`terraform refresh`

Updates the state file by querying the provider.

`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 refresh`

Updates the state file by querying the provider.

`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 refresh`

Updates the state file by querying the provider.

`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 deletes 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.

- `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.

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

Multiple definitions of an identifier are forbidden. Attributes are *single assignment*.

In addition to arguments within a block, there are a few meta attributes that have special semantics, e.g. the `for_each` attribute.

HCL Language Syntax (2): Blocks

Blocks

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

- Have a key identifier: here ***resource***. It has a meaning in the context where it is defined
- String identifiers attached: here ***docker_image*** and ***redis***.
- They can represent a type or name identifier in order to refer to them in another part of the ***.tf*** configuration.

HCL Language Syntax (2): Blocks

Blocks

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

- Have a key identifier: here `resource`. It has a meaning in the context where it is defined
- String identifiers attached: here `docker_image` and `redis`.
- They can represent a type or name identifier in order to refer to them in another part of the `.tf` configuration.

Block can have embedded blocks that, again, have meaning only in the context of the current block. e.g. `docker_image` block can embed a `build` block.

HCL Language Syntax (2): Blocks

Blocks

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

- Have a key identifier: here `resource`. It has a meaning in the context where it is defined
- String identifiers attached: here `docker_image` and `redis`.
- They can represent a type or name identifier in order to refer to them in another part of the `.tf` configuration.

Block can have embedded blocks that, again, have meaning only in the context of the current block. e.g. `docker_image` block can embed a `build` block.

Multiple embedded block with the same keyword are sometimes allowed. It usually results in a list of objects.

Kinds of top-level blocks

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

`resource`, `data`, `provider`

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

- These are the main state declarations of resources managed by Terraform.

Kinds of top-level blocks

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

`resource`, `data`, `provider`

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

- These are the main state declarations of resources managed by Terraform.
- 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`

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

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

There are a few other top-level blocks, e.g. *`locals`*, *`module`*, etc.

Variables

User variables

The DevOps can declare three kinds of user variables: Input, Output, and Local.

They are declared in the *`variable`*, *`output`*, and *`locals`* blocks respectively.

Variables

User variables

The DevOps can declare three kinds of user variables: Input, Output, and Local.

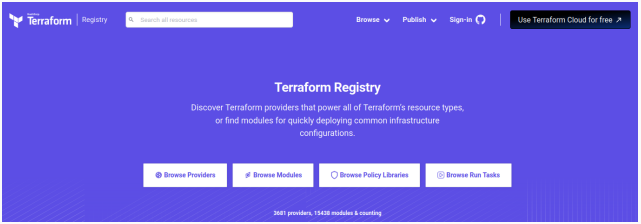
They are declared in the `variable`, `output`, and `locals` blocks respectively.

Terraform's keywords (on the RHS)

- Resources attributes are referenced with type and name of the resource, e.g. `docker_image.my-redis.image_id`
- To reference a *data source*, we prefix the reference as above with *data.*.
- For input and local variables, we use `var.my-inputvar` and `local.my-localvar`
- Terraform has other such special variable keywords, e.g. `each` and `module`.

Providers

Provider registry



Good practices

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...

hcoullon@researcher ~ $

UE Cloud FIL A3 2023-2024 - Module Terraform - IMT Atlantique

2023-11-20

Séance 1 - Introduction, cours et tutoriels

- Slides
- Tutoriel sur le provider Docker
- Tutoriel sur le provider OpenStack
- Tutoriel sur le provider GCP
- Tutoriel sur le provider Kubernetes

Séance 2 - TP noté

- Sujet de TP
