Terraform

Alejandro Campos

June, 2024

Contents

1 Introduction

1.1 Necessity of Terraform

In the traditional IT model, infrastructure is provisioned as following (aprox.):

1. Bussiness comes up with the requirements for the new application.
2. Bussines analysts conver them into a set of high-lvl technical requirements.
3. Solution Architects designs the architecture to be followed for the deployment of this application, considering Front-End, Back-End, Databases, Load Balancers, etc.:
    - Number of servers needed.
    - Specifications of these servers.
    - Type of servers.
4. Once it is planned, it should be deployed on the Onpremise servers from the organization.
5. If additional harwar is needed, they would have to be ordered, which can take few days, weeks or even months for the hardware to be purchased and delivered to the datacenter.
6. Once the hardware is on the datacenter, Infrastructure Engineers will rack and stack all the equipment.
7. The System Administrators will perform initial configurations on all the servers.
8. The Network Adminisrtators will configure and wire all the equipment to connect them to the organization network.
9. The Storage Admin assign the storage to the servers and configure backups.
10. Then, the application is ready for the developer teams in order to deploy their applications.

This procedure has some disadvantages:

1. The whole process can take between few weeks to few months, and it is just to have the hardware in a ready state to begin the application deployment.
2. Saling up / scaling down the infrastructure on demand it is costly and time-consuming.
3. **Underutilization of compute resources:** the servers are usualy virtualized, and the VM's are always sized considering the peak utilization. So the inability to scale up or down easily means that most of these resources wouldn't be used during tha vast majority of the time (off-peak hours).

And they are some of the reasons why organizations have been moving to virtualization and cloud platforms, such as: Amazon AWS, Azure, IBM Cloud, Google Cloud, etc. By moving to cloud, the time to spin up the infrastructure when Bussiness comes up with the requirements for the new application is significantly reduced. So the time to market is significantly reduced as well.

Furthermore, another advantage on Cloud, is that organizations do not have to invest in or manage the hardware assets as in a traditional infrastructure model. The datacenter, the hardware assets and the services are managed by the cloud provider.

A VM can be provisioned in a matter of minutes in a Cloud Environment, and the time to market can be reduced in several months. Besides, infrastructure costs are reduced due to the no need of a datacenter neither human resources to manage it. Furthermore, it is easily scalable and reduce the underutilization of compute resources.

Cloud Infrastrucute come with API's and that opens a whole new world of opportunity for automation. As well as GUI where VM's can be provisioned with a few clicks (and a cash flow). But, when you have to handle with large infrastructure, manage it manually through the GUI's is not the ideal. That's when **Terraform** come in place, a tool for automate infrastructure provisioning to deploy environments faster in a consistent fashion using the different API's from the various cloud providers.

1.2 Infrastructure as Code (IaC)

The better way to provision cloud environment resources is codifying the entire provisioning process, this way we can write, execute and store code to: define, provision, configure, update, upgrade and destroy infrastructure resources. With IaC, any infrastructure component can be managed as code, such as: server, databases, networks, storage, app configurations. Tools like Ansible, Docker, Vagran, Puppet, Terraform are IaC examples, which are easy to learn languages to easily manage infrastructure.

With IaC we can define infrastructure resources using simple, human readable, high-level language. They can be categorized into 3 types:

- **Infrastructure Configuration Management:** used to install and manage existing infrastructure resources, such as: server, databases, networking devices, etc. They are designed to run into multiple instances at once but they best feature is that they are **idemportent**; so the same code can be run multiple times and the result will be the same, making just the necessary changes and not more. Examples:
  - Ansible
  - Puppet
- **Server Templating:** used to create custom static images from VM's or containers, containing all the required software and dependencies. They promote immutable infrastructure, unlike Infrastructure Configuration Management tools, so once the VM or the contaner is deployed, it is designed to remain unchanged (static). In case of changes to be made on the images, instead of updating the running instance, update on the image is needed and then re-deploy a new instance into a container.
  - Docker
  - Podman
  - Vagrant
  - Packer
- **Infrastructure Provisioning Tools:** used to provision infrastructure components using simple and declarative code, such as: servers, VM's, databases, VPC's (Private Cloud Network), subnets, security groups, storage, etc. They use to be **provider agnostic** (Terraform is), which means that they are compatible with almost all major cloud providers, having their plugins.
  - Terraform

1.3 Why Terraform?

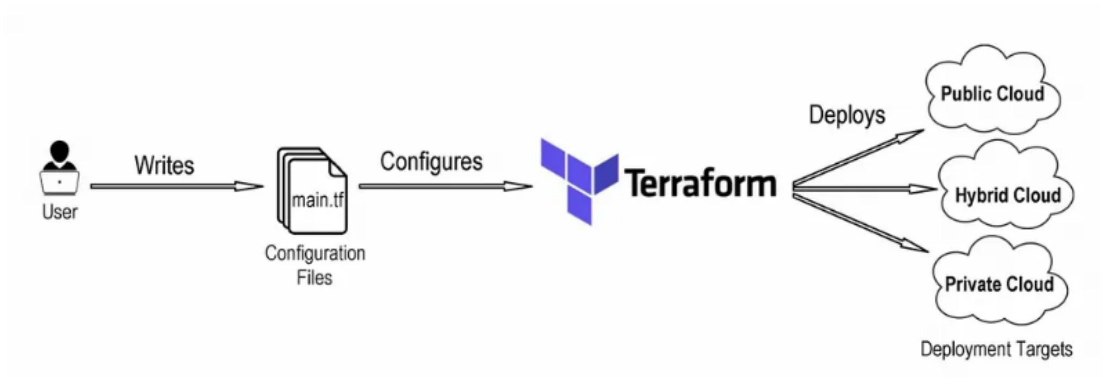
Terraform is one of the most populars IaC tools, beloging Infrastructure Provisioning type, open-source developer by HasiCorp. It is installed as a single binary which can be set up quickly allowing us to create, update, upgrade, destroy, configure and manage infrastructure in a matter of minutes. It is **provider agnostic**, so it can deploy infrastructure across multiple platforms, includin gprivate and public clouds.

Terraform can deploy in so many different cloud thanks to **provides**, they help Terraform to manage third-party plaftorms thourgh their API.

- Onpremise
- VMWare Machines
- AWS
- Google Cloud
- Azure
- ...

Terraform uses HCL (Hashicorp Configuration Language - .tf extension), which is a simple declarative language to define the infrastructure resources to be provisioned as blocks of code. Terraform will take care of what is required to go from the current state of the infrastructure to the desired state (the one defined on the HLC files).

Every object that Terraform manages is called a resource, which can be a compute instace, a server, a VM, a database, etc. Terraform will mage the lifecycle of the resource from its provisioning, to configuration, to decommissioning. It can ensure that the entire infrastructure is always in the defined state at all times.



2 Terraform Basics

2.1 Installing Terraform

1. Update the package list

   ```
   $ sudo apt-get update
   ```

2. Add the HashiCorp GPG key:

   ```
   $ curl -fsSL https://apt.releases.hashicorp.com/gpg \
   | sudo apt-key add -
   ```

3. Add the official HashiCorp Linux repository:

   ```
   $ sudo apt-add-repository \
   "deb [arch=amd64] https://apt.releases.hashicorp.com \
   $(lsb_release -cs) main"
   ```

4. Update the package list again:

   ```
   $ sudo apt-get update
   ```

5. Install Terraform:

   ```
   $ sudo apt-get install -y terraform
   ```

6. Check terraform is installed

   ```
   $ terraform verion
   ```

2.2 My first Terraform Provision

2.2.1 HCL (.tf - HashiCorp Configuration Language)

The HCL files contains block and arguments:

- **Blocks:** they contains information about the infrastructure platform and a set of resrouces within that platform that we want to create.
- **Arguments:** the configuration of them (inside `{}`), they are specific to the type of resource we are creating.

```
<block> <parameters> {
    key1 = value1
    key2 = value2
}
```

For example, imagine that we want to create a file in local host:

local.tf

```
resource "local_file" "cats" {
    filename = "/tmp/cats.txt"
    content  = "I love cats!"
}
```

- **Block Name:** resource
- **Resource Type:** `local_file`
  - **Provider:** local
  - **Resource Type:** file
- **Resource Name:** cats
- **Arguments:**
  - **filename:** "/tmp/cats.txt"
  - **content:** "I love cats!"

As another example here we have an HCL file to create an AWS S3 bucket:

aws-s3.tf

```
resource "aws_s3_bucket" "data-bucket" {
    bucket  = "webserver-bucket-org-2207"
    content = "private"
}
```

2.2.2 Terraform WorkFlow

The simple Terraform workflow consists of 4 steps:

1. Write the HCL (.tf) configuration files
2. Run the **terraform init** command, which will check the configuration files and initialize the working directory containing the .tf file.

   ```
   $ terraform init
   ```

3. Review the execution plan using **terraform plan** command, to see the execution plan that will be carried out by Terraform, which will show the actions that will be carried out by Terraform to create the resources, like:

   - `diff` command in `git`
   - `--checkout --diff` in `Ansible`

   This won't apply anything, just check

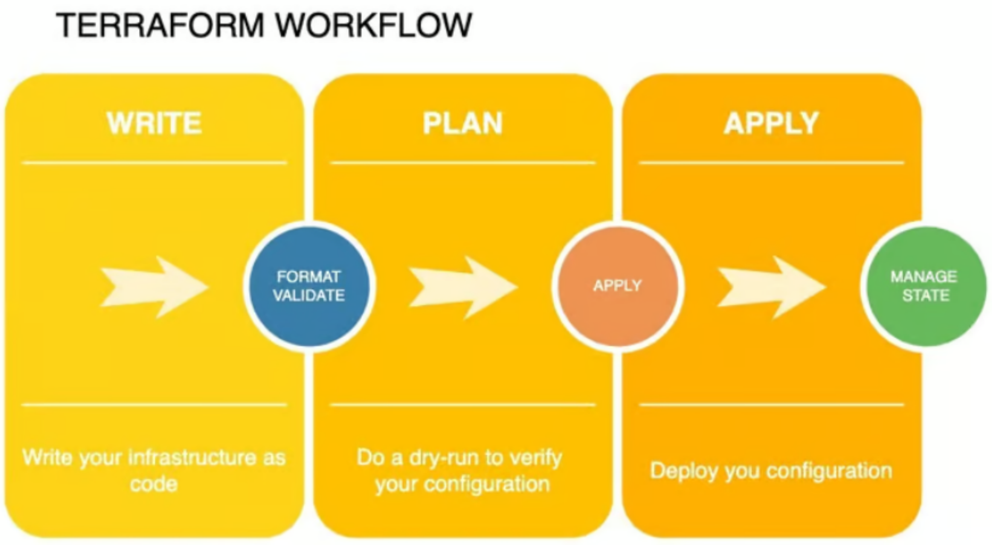
   ```
   $ terraform plan
   ```

4. Apply the changes using the **terraform apply** command, which will display the execution plan once again and it will ask to confirm by typing `Yes` to proceed. Once we confirm, it will proceed with the creation, update, deletion of the resources

```
$ terraform apply
```

Note

Terraform show command can also be run to see the deatils of the resources that we just created



2.2.3 Provider, Resource Types and their custom Arguments

How can we know the availabe resources types from a terraform provider, and how can we know the arguments we can use on this resource type? Terraform has hundreds of providers (they are like modules in Ansible), including the `local_provider` we have used in the example. Each provider has a unique list of resource types that can be created on that specific platform, and each resource type can have different (required or optional) arguments to create the resource.

Examples of other providers:

- local: used to manage local resources, such as files.
- aws: provider used to lifecycle management of AWS resources, including EC2, Lambda, EKS, ECS, VPC, S3, RDS, DynamoDB, and more. This provider is maintained internally by the HashiCorp AWS Provider team.
- azurerm: provider used to lifecycle management of Microsoft Azure using the Azure Resource Manager APIs. maintained by the Azure team at Microsoft and the Terraform team at HashiCorp.
- google: provider used to lifecycle management of GCP resources, including Compute Engine, Cloud Storage, Cloud SDK, Cloud SQL, GKE, BigQuery, Cloud Functions and more. This provider is collaboratively maintained by the Google Terraform Team at Google and the Terraform team at HashiCorp.

- helm: provider used to deploy software packages in Kubernetes using Helm.
- kubernetes: provider used to manage of all Kubernetes resources, including Deployments, Services, Custom Resources (CRs and CRDs), Policies, Quotas and more.
- alicloud: provider used to interact with the many resources supported by Alibaba Cloud. The provider needs to be configured with the proper credentials before it can be used.
- ...

To know all the available providers, check the Official Terraform Registry Providers

2.3 Update and Destroy Infrastructure

Taking the previous example of the file, let's see how can we update and destroy this resources.

2.3.1 Update

For update is very easy, we just need to modify the field we want and then run again the 2 last steps:

```
$ terraform plan
```

Even though the update is trivial, as **file resource type** is **immutable infrastructure** Terraform will **always delete the old resource and then create a new resource** when updating an **immutable resource**. So taking this into account, if you are agree with the changes:

```
$ terraform apply
```

2.3.2 Delete

To remove completely the resources within a configuration directory, run:

```
$ terraform destroy
```

You will be asked to confirm the destroy, so type `yes` in the prompt and it will delete all the resources in the current configuration directory.

2.4 More details on Terraform Providers

The first action Terraform does when we run `terraform init` within a directory containing the configuration files, it downloads and installs all the plugins needed to provision the resources defined on the repository. Terraform providers are distributed by HashiCorp and they are open-source available at Terraform Registry.

There are 3 tiers of providers:

- **Official Providers:** owned and mantained by HashiCorp, and include the major cloud providers such as AWS, GCP, Azure, etc.
- **Partner Providers:** owned and mantianed by a third-party technology company that has gone through a partner provider process with HashiCorp.

- **Community Providers:** published, owned and mantainer by individual contributors of the HashiCorp community.

The `terraform init` command, shows the version of the plugins it is installing, and then it downloads and store them into the hidden directory:

```
/path/to/terraform/directory/.terraform/providers/registry.terraform.io/hashicorp/
```

By default, Terraform installs the latest version of the provider plugins. And almost all the plugins but specially official providers plugins are updated frequently, to enable new features, patch bugs, ease their usage, etc. It is a good feature, but also can introduce breaking changes into your code if you don't specify the provider version.

2.5 Terraform Directories

In Terraform directories we should have more than one configuration files, in order to configure and manage different resources. **Terraform will consider any file with the .tf extension within the directory running it**. Also, you can have the number of resources configuration you want in a single configuration file.

The most common repository structure is the following:

- `main.tf`: main configuration file containing the resource definition
- `variables.tf`: configuration file containing variables declaration
- `outputs.tf`: file containing outputs from resources
- `providers.tf`: configuration file containing Providers definition

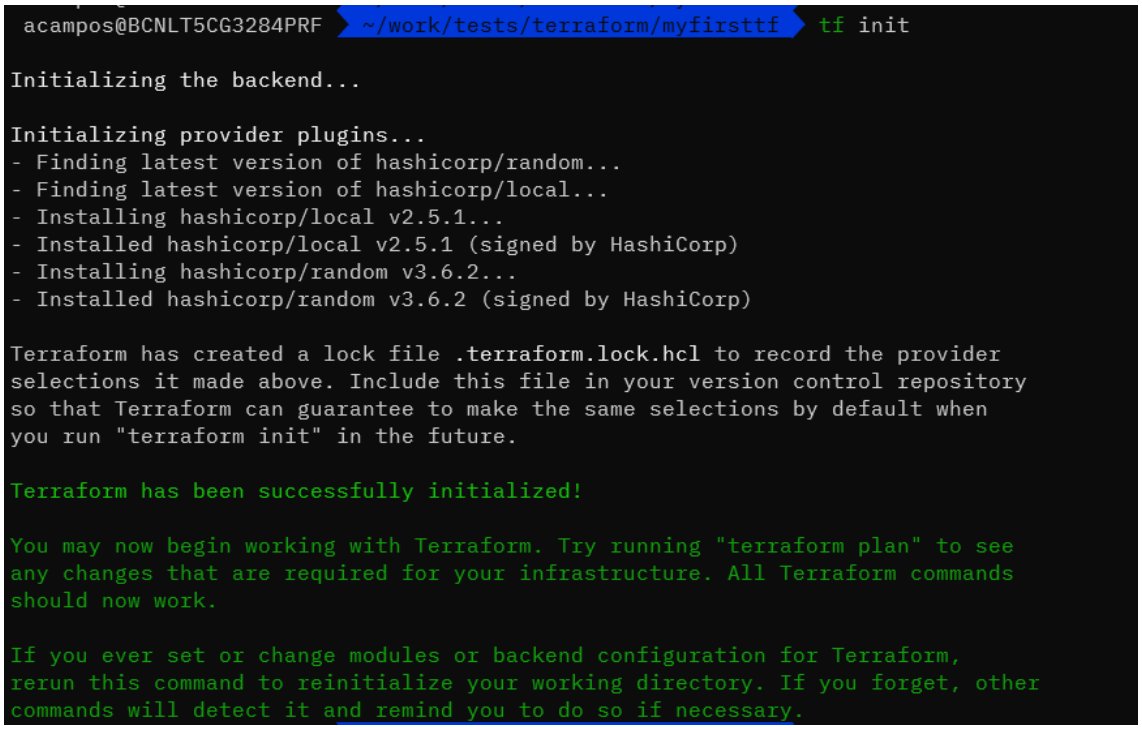
2.6 Multiple Providers

Until now, we have been just talking about a single provider ("local"), but Terraform supports the use of multiple providers within the same configuration, let's see an example:

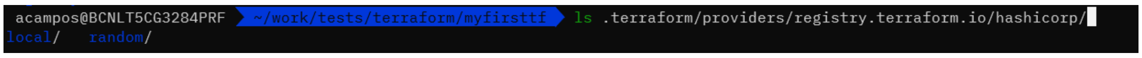
main.tf

```
resource "local_file" "cats" {
    filename = "/tmp/cats.txt"
    content  = "I love cats!"
}

resource "random_pet" "my-cat" {
    prefix    = "Mrs"
    separator = "."
    length    = "1"
}
```

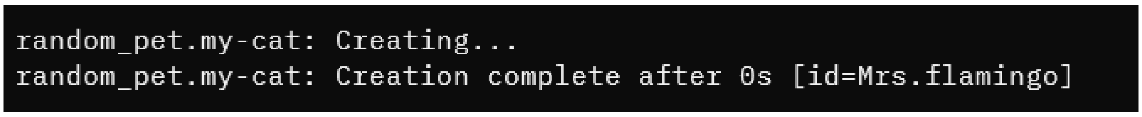
The `main.tf` file now has resource definition for 2 different providers: one resource uses the `local` provider using the `file` resource type and other the `random` provider using the `pet` resource type.



```
 acampos@BCNLT5CG3284PRF  ~/work/tests/terraform/myfirsttf  tf init

Initializing the backend...

Initializing provider plugins...
- Finding latest version of hashicorp/random...
- Finding latest version of hashicorp/local...
- Installing hashicorp/local v2.5.1...
- Installed hashicorp/local v2.5.1 (signed by HashiCorp)
- Installing hashicorp/random v3.6.2...
- Installed hashicorp/random v3.6.2 (signed by HashiCorp)

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

Terraform has been successfully initialized!

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

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



```
 acampos@BCNLT5CG3284PRF  ~/work/tests/terraform/myfirsttf  ls .terraform/providers/registry.terraform.io/hashicorp/
local/   random/
```



```
random_pet.my-cat: Creating...
random_pet.my-cat: Creation complete after 0s [id=Mrs.flamingo]
```

2.7 Variables in Terraform

2.7.1 Basic Variables

Use variables in code is one of the best practices to can reuse code again and again just configuring a few variables. In case of Terraform, we use variable to configure and deploy resources based on a set of variables that can be provided during the execution. For that purpose, we need to create a new configuration file called `variables.tf`

The `variables.tf` file, jsut like the `main.tf` file, consists of blocks and arguments. Let's see an example:

variables.tf

```
variable "cats_filename" {
    default = "/tmp/cats.txt"
}

variable "cats_content" {
    default = "I love cats!"
}

variable "my-cat_prefix" {
    default = "Mrs"
}
...
```

How to use them? Just replace them by

```
var.cats_filename
```

For example:

main.tf

```
resource "local_file" "cats" {
    filename = var.cats_filename
    content  = var.cats_content
}

resource "random_pet" "my-cat" {
    prefix = var.my-cat
...
}
```

WARNING

When using variables, you do not have to enclose the values into double quotes ("") as in other IaC programming languages

Appart from `default` we can define more arguments for the variables:

- `default`: variable value
- `description` (optional): enforces the type of variable
- `type` (optional): what the variable is used for, available types:
  - string
  - number
  - bool
  - any
  - list: `["cat", "dog"]`
  - map: `{pet1 = cat pet2 = dog}`
  - object
  - tuple

Simple example:

variables.tf

```
variable "length" {
    default     = 2
    type        = number
    description = "length of the pet name"
}

variable "enable_pets" {
    default     = false
    type        = bool
    description = "variable to enable pets"
}
```

Note

As the type is optional, if nothing is specified it is set to type **any** by default

2.7.2 Lists

Numbered collection of values.

- Definition

variables.tf

```
variable "names" {
    default     = ["Mixi", "Nina", "Soda"]
    type        = list
    description = "list of my pet favourite names"
}
```

- Usage

main.tf

```
resource "random_pet" "my-cat" {
    prefix = var.names[0]
}
```

- List of numbers: `type = list(number)`
- List of strings: `type = list(string)`

2.7.3 Maps

Numbered collection of values.

- Definition

variables.tf

```
variable "names" {
    default = {
        "favourite_name": "Nina"
        "other_name": "Mixi"
    }
    type        = map
    description = "my pet favourite names in map"
}
```

- Usage

main.tf

```
resource "random_pet" "my-cat" {
    prefix = var.names["favourite_name"]
}
```

- Map of numbers: `type = map(number)`
- Map of strings: `type = map(string)`

2.7.4 Sets

Like a list, but without duplicate elements (it is forbiden and `tf apply` will fail)

- Definition

variables.tf

```
variable "names" {
    default     = ["Mixi", "Nina", "Soda"]
    type        = set(string)
    description = "set of my pet favourite names"
}
```

- Usage

main.tf

```
resource "random_pet" "my-cat" {
    prefix = var.names[0]
}
```

- Set of numbers: `type = set(number)`
- Set of strings: `type = set(string)`

2.7.5 Objects

Complex data structures by combining all the variable types that we have seen so far

- Definition

```
variables.tf

variable "nina" {
    type = object({
        name          = string
        color         = string
        age           = number
        food          = list(string)
        favourite_pet = bool
    })

    default = {
        name          = "nina"
        color         = "white"
        age           = 3
        food          = ["fish", "meat"]
        favourite_pet = true
    }

    description = "nina object definition"
}
```

2.7.6 Tuples

Similar to a list; sequence of elements. But the difference is that list uses variables of the same type, such as list of strings, list of numbers, etc. In case of tuple, we can make use of all the elements types.

- Definition

```
variables.tf

variable "kity" {
    type        = tuple([string, number, bool])
    default     = ["Mixi", 7, bool]
    description = "my kity var definition"
}
```

WARNING

The order of the elements type should be exactly the same as defined, if not, `tf apply` will fail

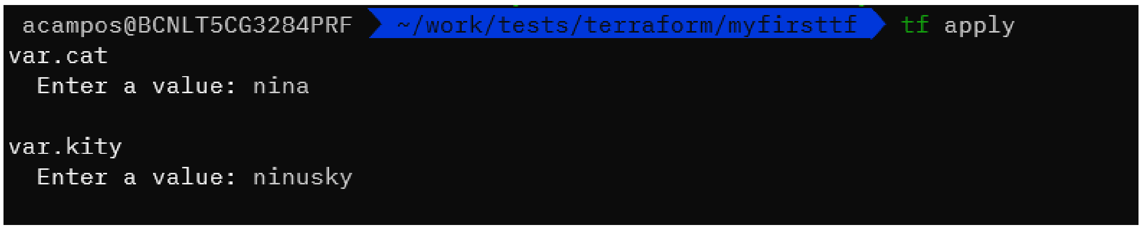
- Usage

main.tf

```
resource "random_pet" "my-cat" {
    prefix = var.names[0]
}
```

2.7.7 Variables Empty Values Allowed?

The answer to this question is **yes**! If you define empty variables when **tf apply** it will ask you to enter values for each empty variabled defined



```
 acampos@BCNLT5CG3284PRF  ~/work/tests/terraform/myfirsttf  tf apply
var.cat
  Enter a value: nina

var.kity
  Enter a value: ninusky
```

2.7.8 Variables in Flags

It is also allowed in Terraform to pass variables as arguments using flags:

```
$ tf apply -var "filename=/tmp/cats.txt" -var "content=I love cats"
```

2.7.9 Terraform Variable Definition Files

Furthermore, you can define your variables in a file called:

- `<file_name>.tfvars`
- `<file_name>.tfvars.json`

terraform.tfvars

```
filename = "/tmp/cats.txt"
content = "I love cats!"
...
```

WARNING

The file can be named as you want, but it is mandatory to have one of these both extensions:

- `.tfvars`
- `.tfvars.json`

Then you can pass it as argument to terraform:

```
$ terraform apply -var-file variables.ini
```

2.7.10 Terraform Variables as Linux env vars

Terraform is amazing, it is also allowed, you need to define variables as following:

```
$ export TF_VAR_<my_var_name>=<my_var_value>
```

2.7.11 Terraform Variables Definition Precedence

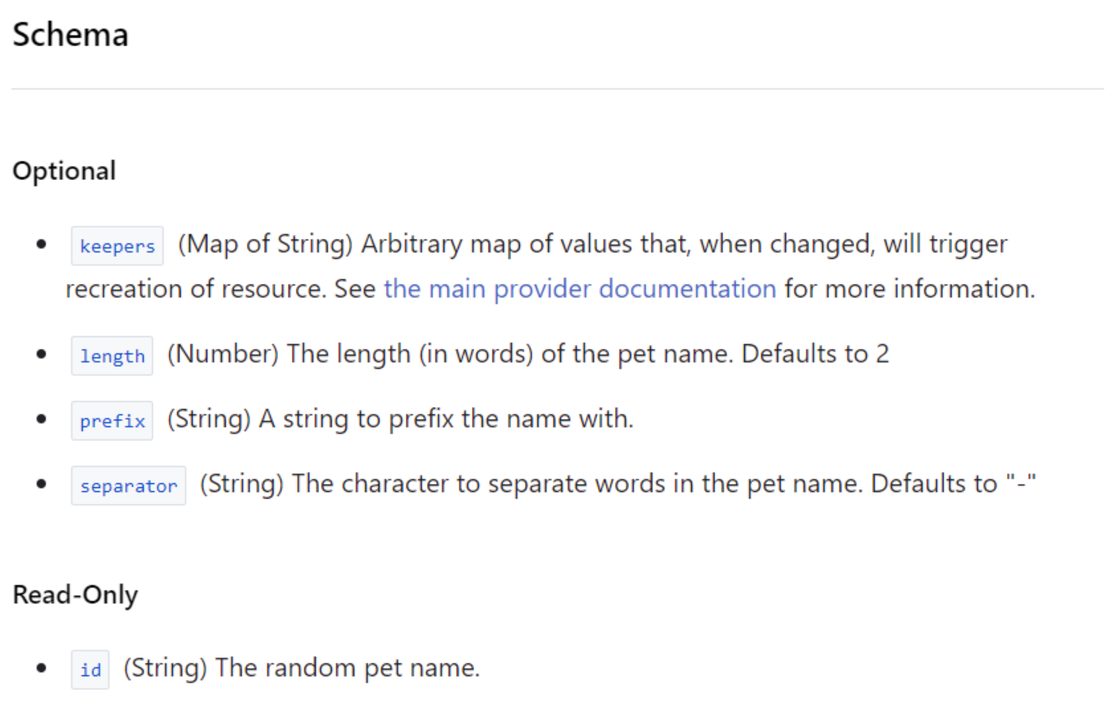
1. Variables in flags.
2. Terraform Variable Definition File (specified in `-var-file`).
3. Terraform Variable Definition File (default ones).
4. Linux Enviroment Variables.

2.8 Reference Attributes

There are a lot of resources that are dependent on each other, for example imaginte that we want to make use of the output of one resource and use it as an input for another one?

If we go to the documentation of the providers, we will see a section called **Schema**, where we will see a list of attributes returned back from the resource after you run `tf apply`.

If we look at the example before, `random_pet` resource returns just one attribute called **id (string)**



Schema

Optional

- `keepers` (Map of String) Arbitrary map of values that, when changed, will trigger recreation of resource. See the main provider documentation for more information.
- `length` (Number) The length (in words) of the pet name. Defaults to 2
- `prefix` (String) A string to prefix the name with.
- `separator` (String) The character to separate words in the pet name. Defaults to "-"

Read-Only

- `id` (String) The random pet name.

So, we can link one resource to another one using reference attributes:

```
main.tf

resource "local_file" "cats" {
    filename = var.cats_filename
    content  = random_pet.my-cat.id
}

resource "random_pet" "my-cat" {
    prefix    = var.names[0]
    separator = var.separator
    length    = var.length
}
```

If we want to include it into an string:

```
main.tf

resource "local_file" "cats" {
    filename = var.cats_filename
    content  = "My pet is called ${random_pet.my-cat.id}"
}

resource "random_pet" "my-cat" {
    prefix    = var.names[0]
    separator = var.separator
    length    = var.length
}
```

If you run `tf apply` you will see the variable replaced!

2.8.1 Resource Dependency

When Terraform creates these resoruces, it knows about the dependencies of the resources. So if we follow the example before, Terraform knows that `local_file` depends on `random_pet`, so as a result, it uses the appropiate order to provision resources. Besides, when resources are deleted, Terraform does it in the inverse order.

As well, if we don't have any dependency relation but we want to stablish it, in order to force the order Terraform creates the resources, we can do it using **explicit dependencies**:

```
main.tf

resource "local_file" "cats" {
    filename   = var.cats_filename
    content    = "My pet is called Nina"
    depends_on = [
        random_pet.my-cat
    ]
}

resource "random_pet" "my-cat" {
    prefix    = var.names[0]
    separator = var.separator
    length    = var.length
}
```

Resource dependency is a very important feature in Terraform, so it is common to need some order in the configuration / deployment of your infrastructure, so you need one resource to not be created until another one is. If dependencies are not explicit or implicit defined, Terraform will do it randomly parallelizing as much as he can with the resource creation / update.

2.8.2 Output Variables

Along with input variables, Terraform also supports output variables, this variables can be used to store the value of an expression in Terraform. So we can assign the value of the output of a function to a variable:

main.tf

```
resource "random_pet" "my-cat" {
    prefix    = var.names[0]
    separator = var.separator
    length    = var.length
}

output "pet_name" {
    value = random_pet.my-cat.id
    description = "Record the value of pet ID generated
                    by the random_pet resource"
}
```

To se the values of the output variables we can run:

- To see al the output variables

```
$ terraform output
```

- To see an specific variable

```
$ terraform output <output_variable_name>
```

- When you run `tf apply` it also display the output variable values

We already saw that dependent resources can make use of reference expressions to get the output from one resource block as an input to another block. As such, output variables are not really required here. The mainly usage of output variables is when you want to quickly display details about a provision resource on the screen, or to feed the output variables to other IaC tools.

WARNING

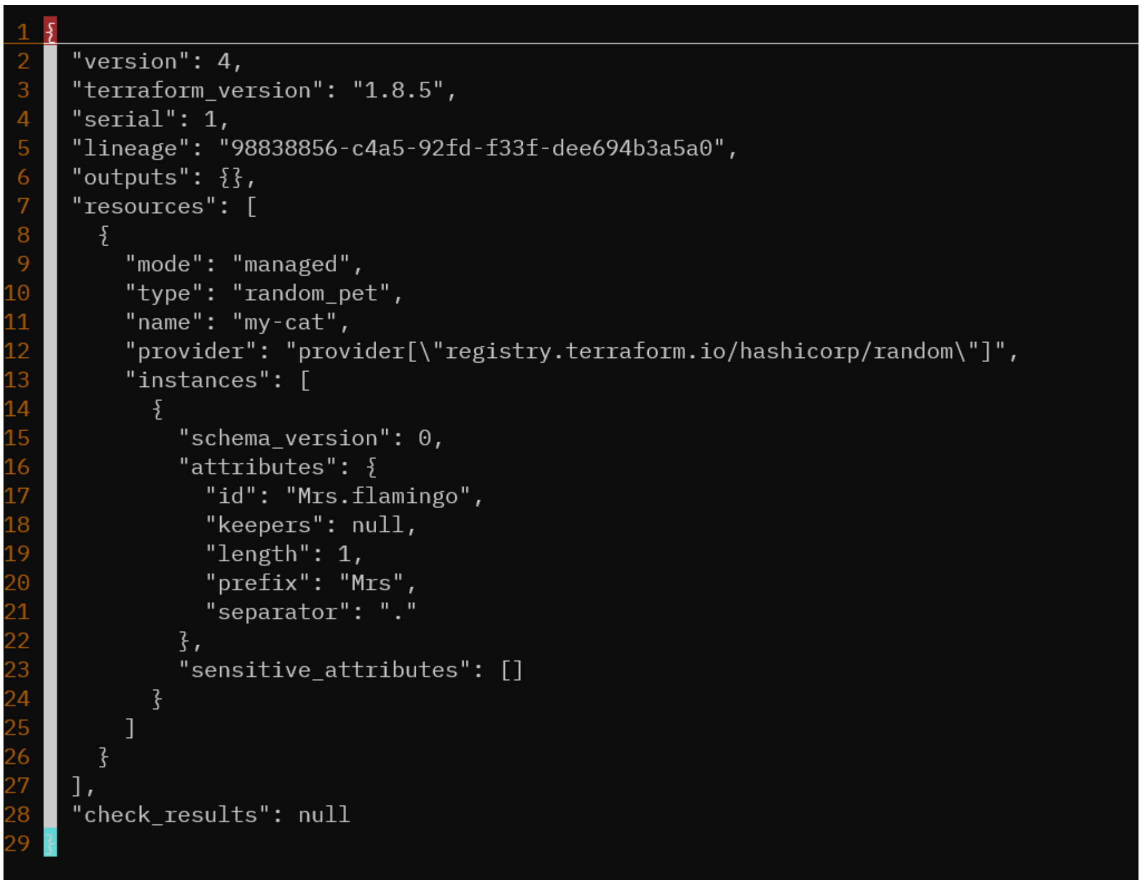
We need to first `tf apply`, if not, any variable will be printed on the `tf output`

3 Terraform State

3.1 Introduction

In this section we are going to discover what Terraform State is and see what really happens under the hoods when we run terraform commands. When we run `terraform apply` the first action Terraform does is to check if there are some past states of this resource stored in-memory. If it does not find any on the memory it will implement the creation of the resource.

As well, `terraform apply` tries to refresh the in-memory state, and when applied, it assigns to the resources unique **id's**, so next time we try to apply changes using terraform, it will know the configuration on the resource. It is done creating a file inside the directory called: `terraform.tfstate`, this file is a JSON data structure that maps the real-world infrastructure resources to resource definition in the configuration files, it has all the record of the infrastructure created by this Terraform directory:



```
{
  "version": 4,
  "terraform_version": "1.8.5",
  "serial": 1,
  "lineage": "98838856-c4a5-92fd-f33f-dee694b3a5a0",
  "outputs": {},
  "resources": [
    {
      "mode": "managed",
      "type": "random_pet",
      "name": "my-cat",
      "provider": "provider[\"registry.terraform.io/hashicorp/random\"]",
      "instances": [
        {
          "schema_version": 0,
          "attributes": {
            "id": "Mrs.flamingo",
            "keepers": null,
            "length": 1,
            "prefix": "Mrs",
            "separator": "."
          },
          "sensitive_attributes": []
        }
      ]
    }
  ],
  "check_results": null
}
```

`terraform.tfstate` file contains every little detail belonging to the infrastructure that was created by Terraform on this directory. It uses it as a single source of truth when using commands such as `terraform plan` and `terraform apply`. So each time we apply new changes using Terraform:

1. Terraform check the state of the resources checking this file
2. Terraform change this file to record new state of the resources, if the resource is unmutable, it will generate a new **id** for the resource, if not, it will keep the old one.

So Terraform uses `.tfstate` file to map the resources configuration with the real world infrastructure, and this mapping allows Terraform to create execution plans when a drift is identify between the resources configuration file and the statefile. It also tracks resource metadata such as resource dependencies.

3.2 Forcing Terraform to believe in .tfstate file

When dealing with a handful number of resources, it may be feasible for Terraform to reconcile state with the realworld infrastructure after every single terraform command (pla/apply). But Terraform use to manage a lot of resources, and resources distributed for different providers. So it is not feasible for Terrafomr to reconcile state for every terraform resource, that's why it can take long times to perform the actions.

So in the cases that Terraform takes a lot to perform `tf plan` or `tf apply` you can make the `.tfstate` file as the only record of truth, **without having to reconcile**, improving the performance significantly.

To do this:

```
$ terraform plan -refresh=false
```

3.3 How to share State File between team members?

As you already know, Terraform stores the changes applied to the `terraform.tfstate` file, which is stored locally in the same directory you run Terraform. That's good for starting projects, but what about shared projects working with more than one person? Every user in the team should always have the latest state data before running Terraform, and make sure that nobody else runs Terraform at the same time.

So it is highly recommended to save the `terraform.tfstate` file in a remote data store rather than to rely on a local copy. This allows the state to be shared between all members of the team.

But we must consider that `terraform.tfstate` contains sensitive information, within it contains every little detail about our infrastructure: as private and public IP's, cpu's, size and type of disks, ssh keypad, initial database passwords, etc. And as you can see, it is stored as a basic JSON format file, so it should be encrypted and stored in a secure storage, **so it is NOT recommended to store it on git public repositories**. Instead, you should store the state in remote backend systems like:

- AWS S3
- Google Cloud Storage
- HasiCorp Consul
- Terraform Cloud

4 Deeping on Terraform

4.1 Terraform Commands

To check syntax:

```
$ terraform validate
```

To scan the configuration files and formats the code into a canonical format:

```
$ terraform fmt
```

To take a look on the current state of the infrastructure as seen by Terraform:

```
$ terraform show [-json]
```

To list all providers used in the configuration files: To check syntax:

```
$ terraform providers
```

To print all the output variables:

```
$ terraform output [var_name]
```

To sync Terraform with the real world infrastructure, to take into account changes made to a resource created by Terraform outside its control (as manual changes). This reconciliation is useful to determine what action to take during the next apply. It doesn't modify any resource but it modifies the state file:

```
$ terraform refresh
```

Note

The `tf refresh` command is run automatically when running other Terraform commands such as `tf plan` and `tf apply`. But it can be bypassed as we have seen before using the `-refresh=false` flag.

To create a visual representation of the dependencies and a Terraform configuration or an execution plan:

```
$ terraform graph
```

Note

The output make no much sense, but we can visualize it doing:

1. Install visualized in Linux like graphviz

   ```
   $ apt install graphviz -y
   ```

2. The execute the terraform

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

3. Download the file and open it in a browser

4.2 Mutable vs Immutable Infrastructure

mutable infrastructure refers to a system where components can be updated or modified after their initial creation. This traditional approach involves making changes directly to the existing infrastructure. For instance, if a server needs a software update or configuration change, these modifications are applied directly to the live server.

However, this approach can lead to configuration drift, where the actual state of the infrastructure diverges from the defined configuration over time, potentially leading to inconsistencies and hard-to-debug issues.

mmutable infrastructure, on the other hand, promotes a paradigm where components are never modified after their creation. Instead of updating an existing resource, a new instance is created with the desired changes, and the old one is replaced. This ensures that the infrastructure's state remains consistent and predictable, aligning closely with the declarative nature of Terraform. With immutable infrastructure, if an application or server needs an update, a new version of the server is created with the updates, and the old version is decommissioned. This approach significantly reduces the risk of configuration drift and makes rollbacks straightforward, as reverting to a previous state is simply a matter of redeploying the previous version of the infrastructure.

WARNING

Furthermore, by default Terraform destroys the resource first before creating a new one in its place!! So if the creation failed, the previous resource would be removed and not replaced XD

But, we can change this behaviour with **LifeCycle Rules**

4.3 LefeCycle Rules

Previously we have seen that when Terraform updates an immutable resource first deletes the resource before creating a new one with the updated configuration. But this may not be a desirable approach in all cases, and omsetimes you may want the updated version of the resource to be created first before the older one is deleted, or you may not want the resource to be deleted at all, even if there was a change made in its local configuration. This can be achieved in Terraform by using Lifecycle Rules.

Lifecycle Rules make use of the same block syntax that we have seen many times so far, and they go directly inside the resource block whose behavior we want to change. For example:

main.tf

```
resource "local_file" "cats" {
    filename = var.cats_filename
    content  = "My pet is called Nina"

    lifecycle {
        create_before_destroy = true
    }
}
```

To create the resource replacement before removing:

main.tf

```
lifecycle {
    create_before_destroy = true
}
```

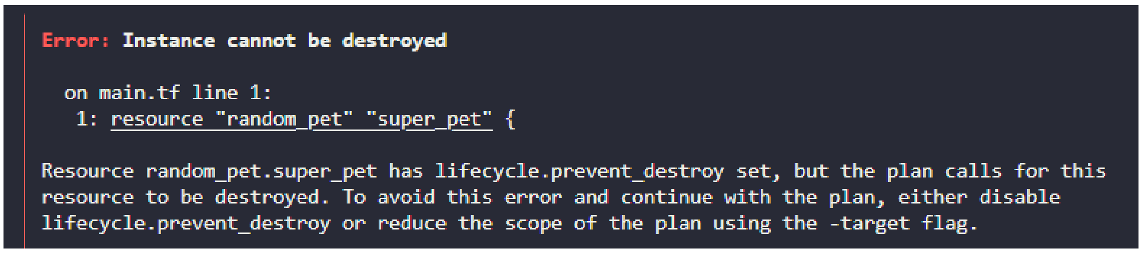
WARNING

Be carefull applying this lifecycle rule, because if we apply it for example to a `local_file`, the local file, we are saying to Terraform to create first a file in a location, and then the same file to be destroyed in the same location during the recreate operation. This goes to show that it is not always advisable to use this rule!

To avoid the destruction of a resource:

main.tf

```
lifecycle {
    prevent_destroy = true
}
```



```
Error: Instance cannot be destroyed

  on main.tf line 1:
   1: resource "random_pet" "super_pet" {

Resource random_pet.super_pet has lifecycle.prevent_destroy set, but the plan calls for this
resource to be destroyed. To avoid this error and continue with the plan, either disable
lifecycle.prevent_destroy or reduce the scope of the plan using the -target flag.
```

WARNING

This resources are not destroyed if we use `tf apply` command, but they can be destroyed running the `tf destroy` command

To prevent a resource from being updated:

main.tf

```
lifecycle {
    ignore_changes = all
}
```

To prevent a resource from being updated if one or more atributes are modified (but not all):

main.tf

```
lifecycle {
    ignore_changes = [
        <atribute1>,
        <atribute2>,
    ]
}
```

4.4 Datasources

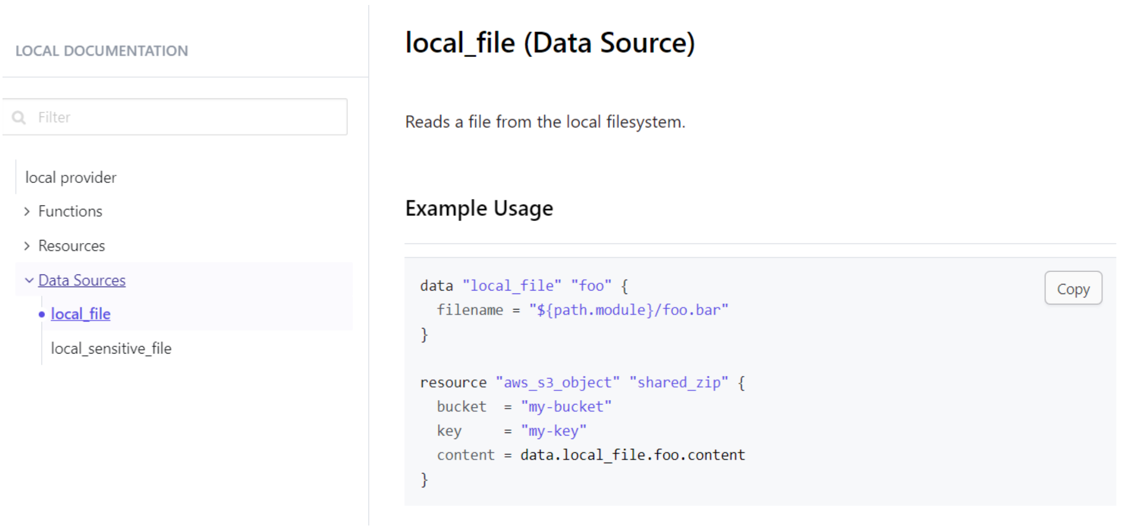
Imagine that a databse instance was provisioned manually in the AWS Cloud, although Terraform does not manage this resource, it can read atributes such as the database name, host address, or the DB user. And can use this data to privison an application resource that is managed by Terraform using **datasources**

Let's see an example of how the content of the resource in `/tmp/dog.txt` not managed by Terraform can be used to fulfill the `/tmp/cat.txt`

main.tf

```
resource "local_file" "cats" {
    filename = var.cats_filename
    content  = data.local_file.dog.content
}

data "local_file" dog" {
    filename = "/tmp/dog.txt"
}
```

The datasources you can extract of each resource can be found on the documentation of each provider and resource type:



local_file (Data Source)

Reads a file from the local filesystem.

Example Usage

```
data "local_file" "foo" {
  filename = "${path.module}/foo.bar"
}

resource "aws_s3_object" "shared_zip" {
  bucket  = "my-bucket"
  key     = "my-key"
  content = data.local_file.foo.content
}
```

Note

Do not confuse resources with datasources!

- Resources are created with the `resource` block, they are use to create, update and destroy infrastructure.

- Datasources are crerated with the `data` block, they are used to read information from a specific resource.

4.5 Meta Arguments

So far, we have seen how to create individual resources, but what if we want to create multiple instance of the same resource? So using **Meta Arguments**. We have defined two Meta Arguments during this course:

4.5.1 Depends On

main.tf

```
depends_on = [
    random_pet.my-pet
]
```

4.5.2 Lifecycle

main.tf

```
lifecycle {
    prevent_destroy = true
}
```

4.5.3 Count

main.tf

```
count = #Number_of_instances
```

In the following example terraform will try to create the same file 3 times:

main.tf

```
resource "local_file" "cats" {
    filename = "/tmp/cats.txt"
    content  = "My pet is called Nina"
    count    = 3
}
```

To create 3 different resources:

main.tf

```
resource "local_file" "cats" {
    filename = var.filename[count.index]
    content  = "My pet is called Nina"
    count    = length(var.filename)
}

variable "filename" {
    type    = list(string)
    default = [
        "/tmp/cats.txt",
        "/tmp/dogs.txt",
        "/tmp/pets.txt
    ]
}
```

Note

Also we have used a built in Terraform function: `length()` which will return the length of a list, set or map.

WARNING

Use `count` is not a good practice in Terraform, because if we want to remove one of the elements on the list it will probably destroy and recreate all the following elements. Because it detects that are not on the same possition on the list. So avoid as much as you can use this and use `for_each` instead.

4.6 For Each

This is the best way to create multiple instance of the same resource in Terraform. Because it will not remove any resource extra if we just want to remove one of the resources and you can add as many resources as you want.

main.tf

```
resource "local_file" "cats" {
    for_each = var.filename

    filename = each.value
}

variable "filename" {
    type    = set(string)
    default = [
        "/tmp/cats.txt",
        "/tmp/dogs.txt",
        "/tmp/pets.txt
    ]
}
```

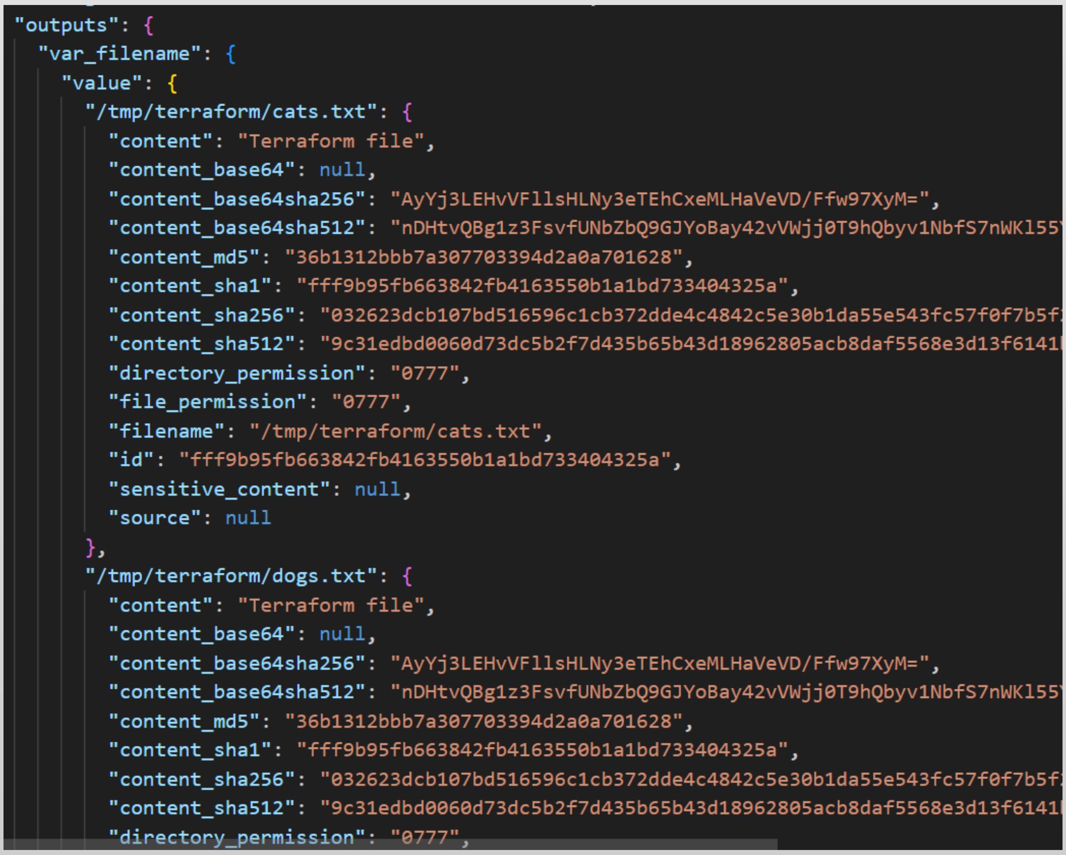
WARNING

The `for_each` Meta Argument only works with **maps** or **sets**. Because using `for_each` Terraform is converting the variable into a `map` so the resources are no longer identified by the index, they are identified by its value, to avoid the same missbehaviour as `count`. That's why we changed the `filename` variable to a type set.

Also we could do the following using the built in Terraform function: `toset()` which will convert a list to a set.
If we look now to the variable, it is converted to a map:

main.tf

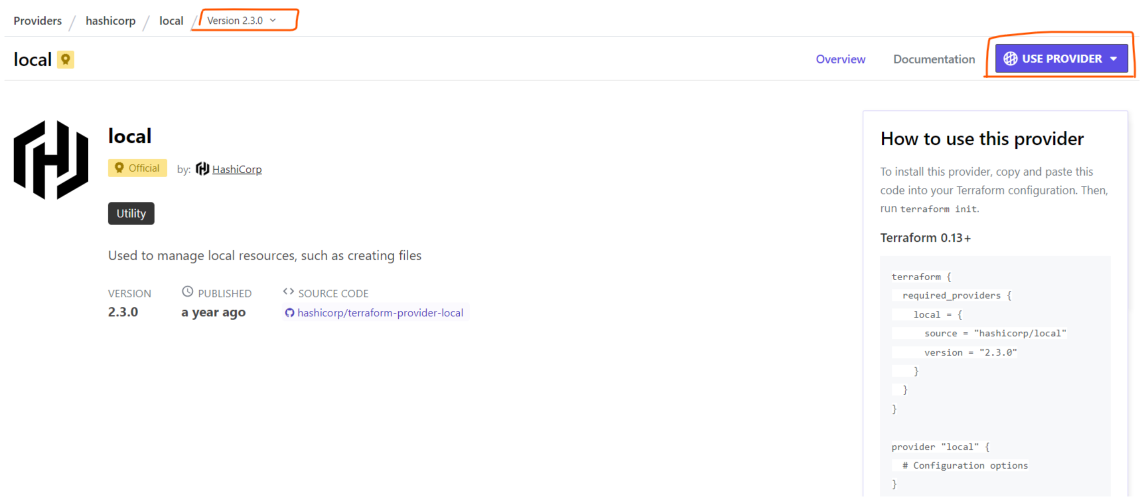
```
resource "local_file" "cats" {
    for_each = toset(var.filename)

    filename = each.value
}

variable "filename" {
    type    = list(string)
    default = [
        "/tmp/cats.txt",
        "/tmp/dogs.txt",
        "/tmp/pets.txt"
    ]
}
```



```
"outputs": {
  "var_filename": {
    "value": {
      "/tmp/terraform/cats.txt": {
        "content": "Terraform file",
        "content_base64": null,
        "content_base64sha256": "AyYj3LEHvVFllsHLNy3eTEhCxeMLHaVeVD/Ffw97XyM=",
        "content_base64sha512": "nDHtvQBg1z3FsvfUNbZbQ9GJYoBay42vVWjj0T9hQbyv1NbfS7nWKl55
        "content_md5": "36b1312bbb7a307703394d2a0a701628",
        "content_sha1": "fff9b95fb663842fb4163550b1a1bd733404325a",
        "content_sha256": "032623dcb107bd516596c1cb372dde4c4842c5e30b1da55e543fc57f0f7b5f
        "content_sha512": "9c31edbd0060d73dc5b2f7d435b65b43d18962805acb8daf5568e3d13f6141
        "directory_permission": "0777",
        "file_permission": "0777",
        "filename": "/tmp/terraform/cats.txt",
        "id": "fff9b95fb663842fb4163550b1a1bd733404325a",
        "sensitive_content": null,
        "source": null
      },
      "/tmp/terraform/dogs.txt": {
        "content": "Terraform file",
        "content_base64": null,
        "content_base64sha256": "AyYj3LEHvVFllsHLNy3eTEhCxeMLHaVeVD/Ffw97XyM=",
        "content_base64sha512": "nDHtvQBg1z3FsvfUNbZbQ9GJYoBay42vVWjj0T9hQbyv1NbfS7nWKl55
        "content_md5": "36b1312bbb7a307703394d2a0a701628",
        "content_sha1": "fff9b95fb663842fb4163550b1a1bd733404325a",
        "content_sha256": "032623dcb107bd516596c1cb372dde4c4842c5e30b1da55e543fc57f0f7b5f
        "content_sha512": "9c31edbd0060d73dc5b2f7d435b65b43d18962805acb8daf5568e3d13f6141
        "directory_permission": "0777",
```

4.7 Version Constraints

In this section we will see how to make use of specific provider versions in Terraform. Without specific configuration, `tf init` command downloads the latest version of the provider plugins that are needed by the configuration files. The instructions to use a specific version of a provider is available in the provider documentation in the registry.



Used to manage local resources, such as creating files

VERSION 2.3.0
PUBLISHED a year ago
SOURCE CODE hashicorp/terraform-provider-local

How to use this provider

To install this provider, copy and paste this code into your Terraform configuration. Then, run `terraform init`.

Terraform 0.13+

```
terraform {
  required_providers {
    local = {
      source = "hashicorp/local"
      version = "2.3.0"
    }
  }
}

provider "local" {
  # Configuration options
}
```

So of you have seen, to define the version of a specific provider we will need to use a new block: `required_providers` under `terraform` block:

providers.tf

```
terraform {
    required_providers {
        local = {
           source = "hashicorp/local"
           version = "2.3.0"
        }
    }
}
```

We can use more version constrains:

providers.tf

```
terraform {
    required_providers {
        local = {
           source = "hashicorp/local"
           version = "!= 2.3.0"
        }
    }
}
```

Note

To configure providers, it is common to create a new file to write this configuration:

```
providers.tf
```

providers.tf

```
terraform {
    required_providers {
        local = {
            source = "hashicorp/local"
            version = "< 2.3.0"
        }
    }
}
```

providers.tf

```
terraform {
    required_providers {
        local = {
            source = "hashicorp/local"
            version = "> 2.3.0"
        }
    }
}
```

providers.tf

```
terraform {
    required_providers {
        local = {
            source = "hashicorp/local"
            version = "~> 2.3.0"
        }
    }
}
```

providers.tf

```
terraform {
    required_providers {
        local = {
            source = "hashicorp/local"
            version = "> 2.3.0, < 3.0.0, != 2.5.0"
        }
    }
}
```

5 Terraform with AWS

AWS is one of the most popular cloud computing platforms in the world offering hundreds of services, such as:

- Compute
- DB's
- Storage
- Machine Learning
- Analytics
- Iot

Having all this infrastructure options it make the companies easier to build their applications on AWS or migrate its on-premiss infrastructure to AWS. Also, it has the most extensive global cloud infrastructure, this allows us to deploy services in a number of different globally distribution regions, and within multiple datacenters known as availability zones.

5.1 Introduction to IAM (Identity and Access Management)

The account you created at frist time using AWS is an administrator account, you can manage any service within AWS using this account, but this is not the recommended approach. This user account can be compared with `root` account in Linux. So the best practice is to create new users with different permissions using this root account, and do not use the root account for any other purpose.

When we create users in AWS there are 2 types of access that we can configure:

- Access to the management console (user and password) to access the GUI
- Programmatic access (key value ID-secret) to use terminal

With IAM, you can set permissions for other users to access, view, edit, the resources. And as well roles can be managed to enable the access between different services on AWS:

5.2 Accessing AWS Programmaticaly

1. Create a programatic access through the GUI
2. Install aws binary

   ```
   $ curl "https://awscli.amazonaws.com/awscli-exe-linux-x86_64.zip" -o "awscliv2.zip"
     unzip awscliv2.zip
     sudo ./aws/install
   ```

3. Configure aws binary, it will ask for the key-secret generated on the GUI

   ```
   $ aws configure
   ```

Note

As output format you can choose JSON, YAML, TXT or Table, but it is recommended to choose JSON.

The configuration is stored under `~/.aws/config`

Basic Usage of aws cli:

```
$ aws <command> <subcommand> [parameters]
```

```
$ aws help
```

```
$ aws <command> help
```

```
$ aws <command> <subcommand> help
```

Some examples:

- Create a user

```
$ aws iam create-user --user-name lucy
```

- Create a group

```
$ aws iam create-group --group-name lucy
```

- Create a group

```
$ aws iam create-group --group-name lucy
```

Note

When you create a resource on AWS, an `ARN` will be assigned to it, and it is the unique id assigned to each resource on AWS.

5.3 IAM resources with Terraform

Official AWS Provider Documentation

The first thing we need to do is configure the provider:

```
provider "aws" {
    region = "us-west-2"
    access_key = "fdjhsakfhas"
    secret_key = "fhhashfakshl"
}
```

WARNING

This is not a good practice, it is better to store this data in your machine using `aws configure`:

- `~/.aws/config`
- `~/.aws/credentials`

```
[default]
aws_access_key_id =
aws_secret_access_key =
```

Another option is to define the following variables:

```
$ export AMS_ACCESS_KEY_ID=
  export AMS_SECRET_ACCESS_KEY_ID=
  export AMS_REGION=
```

The we should define the iam resource we want to add:

```
resource "aws_iam_user" "lucy_user" {
    name = "lucyr"
    path = "/system/"

    tags = {
        description = "Technical Team Lead"
    }
}
```

5.4 IAM Policies with Terraform

In this section we will se how to create IAM policies with Terraform and attach them to users and groups.

Note

All users in AWS start with the least privileges. To add permissions we need to add policies to groups and users.

For example:

```
resource "aws_iam_policy" "admin_policy" {
    name   = "AdminUsers"
    policy = <<EOF
    {
        Version   = "2012-10-17"
        Statement = [
          {
            Action = [
              "ec2:Describe*",
            ]
            Effect   = "Allow"
            Resource = "*"
          },
        ]
    }
    EOF
}

resource "aws_iam_user_policy_attachment" "lucy_admin_access" {
    user       = aws_iam_user.lucy_user.name
    policy_arn = aws_iam_policy.admin_policy.arn
}
```

Another way to specify the policy is:

admin-policy.json

```
{
    Version = "2012-10-17"
    Statement = [
        {
        Action = [
            "ec2:Describe*",
        ]
        Effect   = "Allow"
        Resource = "*"
        },
    ]
}
```

```
resource "aws_iam_policy" "admin_policy" {
    name   = "AdminUsers"
    policy = file("admin-policy.json")
}

resource "aws_iam_user_policy_attachment" "lucy_admin_access" {
    user       = aws_iam_user.lucy_user.name
    policy_arn = aws_iam_policy.admin_policy.arn
}
```

5.5 Introduction to S3 in AWS

S3 stands for a Simple Storage Service in AWS, insted it is simple, it provides infinitely scalable storage solution from AWS. Also offering high availablility of data storing it in across multiple devices and availability zones in a region. S3 is an object base storage, it allows us to store objects or flat files, such as documents, images and videos in the AWS cloud. It is equivalent to a file share storage, such as NFS, which are suitable to install flat files but not to install OS or DB's.

Data in S3 is stored in the form of an S3 bucket, a bucket can be considered to be a container of a directory which stores all your files, and you can create as many as this buckets as you wish in your AWS account. Everything within a bucket is an object, so imagine that we have an S3 bucket called "pets".

```
pets

lolo.txt
pictures/
    |-- nina.mp4
    |-- set.mp4
videos/
    |-- nina.mp4
    |-- set.mp4
```

Every item (files and folders) is an object, so the files are objects, and the files inside folders are objects wiht the fullpath, and also the folders are objects.

When you create a bucket on AWS, the **bucket name** must be unique, because once it is created, AWS creates also a DNS name for it, accessible from anywhere in the world. So, the name should be as well dns compliant: no uppercases, no underscores and between 3 and 63 characters long. When the bucket is created it will be accessible through the DNS name which looks like this:

```
https://<bucket_name>.<region>.amazonaws.com
```

Also any object inside the bucket is accessible through:

```
<bucket_dns>/path
```

Any object in an S3 bucket consists of an object data and metadata:

- **Data**
  - **key:** actual name of the object
  - **value:** actual data referenced by that object
- **Metadata:** details about the object stored in the bucket
  - The time when the object was created
  - The owner of the object
  - The size of the object
  - ...

When an object is uploaded to an AWS bucket, it provides it the least permissions. So by default, no one can access the objects in the bucket with the exception of the bucket owner. Access to the bucket permissions are managed through **bucket policies**, and the bucket objects specific permissions are manage through **access control lists**

5.6 S3 with Terraform

Let's create 2 resources in AWS: a bucket and a bucket object:

main.tf

```
resource "aws_s3_bucket" "terraform_learning" {
    bucket = "terraform-learning-242384230"
    tags   = {
        description = "Files regarding terraform learning"
    }
}

resource "aws_s3_bucket_object" "terraform_basis" {
    content = "/path/to/your/local/file"
    key     = "terraform_basis.txt"
    bucket  = aws_s3_bucket.terraform.id
}
```

Now define the policies and set them:

main.tf

```
data "aws_iam_group" "terraform_students" {
    group_name = "terraform_students"
}

resource "aws_s3_bucket_policy" "terraform_students_policy" {
    bucket  = aws_s3_bucket.terraform.id
    policy = <<EOF
    {
        Version   = "2012-10-17"
        Statement = [
          {
            "Action" = "*",
            "Effect"   = "Allow",
            "Resource" = "arn:aws:s3:::${aws_s3_bucket.terraform.id}/*"
            "Principal": {
                "AWS": [
                    "${data.aws_iam_group.terraform_students.arn}"
                ]
            }
          }
        ]
    }
    EOF
}
```

5.7 Introduction to DynamoDB in AWS

DynamoDB is a no SQL database solution provided by AWS, it is a highly scalable database that can cope with millions of requests from applications such as mobile, web, gaming, iot, etc. It is a fully managed service from AWS. It provides high availability (because the data is replicated across different AWS regions), easy scalability and low latency data access.

A no SQL database are those which store the data in key-value pairs and documents (as mongodb, elasticsearch, etc.).

5.7.1 DynamoDB with Terraform

main.tf

```
resource "aws_dynamodb_table" "motorbikes" {
    name         = "motorbikes"
    hash_key     = "Plate" # The primary key for the table (mandatory for all items)
    billing_mode = "PAY_PER_REQUEST"
    attribute {
        name = "Plate"
        type = "S" # String
    }
}

resource "aws_dynamodb_table_item" "motorbikes-items" {
    table_name = aws_dynamodb_table.motorbikes.name
    hash_key   = aws_dynamodb_table.motorbikes.hash_key
    item = <<EOF
    {
        "Brand": {"S": "Suzuki"},
        "Year": {"N": "2024"},
        "Model": {"S": "xt"},
        "Plate": {"S": "0000AAA"}
    }
    EOF
}
```

6 Remote State

6.1 Introduction

In previous section we have seen how terraform uses the `.tfstate` file to matches the real world infrastructure with the state configured on the file and then with the changes to apply when `tf plan` or `tf apply`. The `.tfstate` file is created automatically when we do `tf plan` or `tf apply`.

In the examples we have seen so far, the `.tfstate` is created and stored locally, in the terraform directory. But this can be a big problem when we are working as a team, because the file is only available on the client machine. However, it is also not a good idea to store the file in a version control system, so it contain sensitive data belonging to out infrastructure.

Furthermore, if the file is stored locally when executing, if 2 persons try to to `tf apply` at the same time, it may lead to unindtended consequences such as the corruption of the `.tfstate`. But, if the `tf apply` runs over the same `.tfstate` file (the same local-remote version), Terraform protects itself from getting into this situation, where concurrent operations are run against the same state file. So you cannot do it, you get an error, while the first operation is in progress, Terraform locks the `.tfstate` file, so we won't be able to run another terraform operation into another terminal until the first operation finishes. It is a very importeant Terraform feature called **state locking**.

Version controllers as GitHub, GitLab, Bitbucket, etc. Do not support **state locking**, so the `.tfstate` may be corrupted using this systems. And as the repositories are pulled, multiple users can use the "same" `.tfstate` at the same time, because it is not the "same". Which can result in issues like conflicts, data loss, corrumption of the state file, etc. Besides, if someone forget about pull the last `.tfstate` version, it can result in disastrous ethics, destroying some resources.

For all these reasons, it is much better option to store Terraform state in a secured, shared storage by making use of **remote backends**. With this option, the `.tfstate` no longer resides in the configuration directory or version control systems. When a **remote backend** is configured, Terraform will automatically load the `.tfstate` from the **shared storege** every time it is required by `tf apply`, as well as provide with **state locking** feature. In addition, **remote backends** provide different ways to secure the storage, such as encryption at rest and in transit, to make sure that all the sensitive information stored inside is secured.

6.2 Remote Backend with S3 Bucket

In this section we are going to see how to use S3 Bucket as **remote backend** and dynamodb which will be used to implement the **state locking** and **consistency checks**.

providers.tf

```
terraform {
    backend "s3" {
        bucket         = "name-of-the-bucket"
        key            = "/pth/to/folder/terraform.tfstate"
        region         = "us-west-1"
        dynamodb_table = "state-locking"
    }
}
```

Note

As a best practice, `backend` configuration should go inside:

- `terraform.tf`
- `providers.tf`

To use and configure the new `backends` we should run `tf init`. If we have a previous `.tfstate` file it will give us the option to upload the current state file to the bucket, we need to type "yes" if we want to. So, from now, when we run `tf apply` it will load the remote `.tfstate` file and if we apply, any changes to the state will be automatically uploaded to the backend instantaneously. And once the operation is completed, the lock will be released. So the `.tfstate` file will not be stored in the local configuration directory anymore.

6.3 Terraform State Commands

In this section we will learn how to list and manipulate the `terraform.tfstate` file, using the `tf state` command.

```
$ tf state <subcommand> [options] [args]
```

Available subcommands:

- `list`: list all the resources recorded within the `terraform.tfstate` file, just the resource address, no other details about the resource.

  ```
  $ tf state list
  ```

- `show`: show the attributes of a single resource in the `terraform.tfstate` file.

  ```
  $ tf state show aws_dynamodb_table.alex_friends
  ```

- `mv`: move items in `terraform.tfstate` file, from their resource address to another == renaming a resource

  ```
  $ tf state mv [options] <source> <destination>
  ```

  ```
  $ tf state mv aws_dynamodb_table.alex aws_dynamodb_table.clau
  ```

- `pull`: pull the `terraform.tfstate` file from the **remote backend**.
- `rm`: delete items from the `terraform.tfstate` file. it is removed from the `terraform.tfstate` but not from the real world infrastructure.
- ...

7 Terraform Provisioners

7.1 AWS EC2

One of the most commonly used services from AWS or any cloud provider are VM's in the Cloud. This VM's provide scalable compute that can be deployed in a matter of minutes. In AWS they are called **EC2** (Elastic Compute Cloud) instances. And just like any compute, virtual or physical, and EC2 instance would run an operatin gsystem such as a distribution of Linux or Windows.

AWS EC2 provdes pre-configured templates known as **AMIs** (Amazon Machine ImageS), these templates contain software configuration such as the operating system, any additional software to be deployed on these EC2 instances, and **Instance Types**. **Instance Types** are the different types of instance that we can choose on AWS depending on our CPU, memory, networking capacity requirements. Some examples of Instance Types:

- General Purpose
- Compute Optimized
- Memory Optimized
- ...

Also they are configured, for example in the General Purpose Instance Type we can configure:

- t2.nano
- t2.micro
- t2.small
- t2.large
- ...

Persistent Storage for these instances is provided by another service called **EBS** (Elastic Block Storage). There are different types of EBS volumes:

- io1
- io2
- gp2

- st1
- ...

As well, you can configure the EC2 to pass user data as user, password or pubkey. As well, you can run scripts when the instances start, for example if we want to install NGINX we can pass a shell script to it.

7.2 AWS EC2 with Terraform

provider.tf

```
terraform {
    provider "aws" {
        region = ...
    }
}
```

main.tf

```
resource "aws_instance" "webserver" {
    ami           = "ami-0edbfdasñfjafadsf..."
    instance_type = "t2.micro"
    tags          = {
        Name        = "webserver"
        Description = "An NGINX WebServer on Ubuntu"
    }

    key_name  = aws_key_pair.pem_web.id
    vpc_security_groups_ids = [ aws_security_group.ssh-access.id ]
    user_data = <<EOF
    !#!/bin/bash
    sudo apt uptade
    sudo apt install nginx -y
    sytemctl enable nginx
    systemctl start nginx
    EOF

}

output publicip {
    value = aws_instance.webserver.public_ip
}

resource "aws_key_pair" "pem_web" {
    public_key = file("/root/.ssh/pem_web.pub")
}

resource "aws_security_group" "ssh-access" {
    name = "ssh-access"
    description = "Allow SSH access from the Internet"
    ingress {
        from_port = 22
        to_port   = 22
        protocol  = "tcp"
        cidr_blocks = ["0.0.0.0/0"]
    }
}
```

7.3 Terraform Provisioners

Provisioners provide a way for us to carry out tasks such as running commands or scripts on remote resources or locally on the machine where Terraform is installed. For example, to run a bash script after a resource is created, we can make use of the `remote-exec` provisioner:

main.tf

```
resource "aws_instance" "webserver" {
    ami           = "ami-0edbfdasñfjafadsf..."
    instance_type = "t2.micro"
    tags          = {
        Name        = "webserver"
        Description = "An NGINX WebServer on Ubuntu"
    }

    key_name  = aws_key_pair.pem_web.id
    vpc_security_groups_ids = [ aws_security_group.ssh-access.id ]
    provisioner "remote-exec" {
        inline = [ "sudo apt update",
                   "sudo apt install nginx -y",
                   "sudo systemctl enable nginx",
                   "sudo systemctl start nginx"
        ]
    }

}
```

WARNING

For the `remote-exec` provisioner to work, we need:

- `ssh connection`: if the server is linux
- `winrm connection`: if the server is windows

And this can be achieved by making use of the appropiate security groups while creating the resources.

As well as the security group, we will need to add the connection:

main.tf

```
resource "aws_instance" "webserver" {
    ami           = "ami-0edbfdasñfjafadsf..."
    instance_type = "t2.micro"
    tags          = {
        Name        = "webserver"
        Description = "An NGINX WebServer on Ubuntu"
    }

    key_name  = aws_key_pair.pem_web.id
    vpc_security_groups_ids = [ aws_security_group.ssh-access.id ]
    provisioner "remote-exec" {
        inline = [ "sudo apt update",
                   "sudo apt install nginx -y",
                   "sudo systemctl enable nginx",
                   "sudo systemctl start nginx"
        ]
    }

    connection {
        type        = "ssh"
        host        = self.public_ip
        user        = "Ubuntu"
        private_key = file("~/.ssh/web_key")
    }

}

resource "aws_key_pair" "pem_web" {
    public_key = file("/root/.ssh/pem_web.pub")
}
```

Note

Using the **self.** we can retrieve output values of the same resource we are creating, because by default, provisioners are executed after the resource is created.

`local_exec` provisioner is used to run tasks in the local machine, for example write the output of a resource in a file locally:

main.tf

```
resource "aws_instance" "webserver" {
    ami           = "ami-0edbfdasñfjafadsf..."
    instance_type = "t2.micro"
    tags          = {
        Name        = "webserver"
        Description = "An NGINX WebServer on Ubuntu"
    }

    provisioner "local-exec" {
        command = "echo ${self.public_ip} >> output.txt"
    }
}
```

We can configure provisioners to run before a resource is destroyed.

main.tf

```
provisioner "local-exec" {
    when    = destroy
    command = "echo ${self.public_ip} destroyed! >> output.txt"
}
```

If the provider fails, the resource creation will fail as well, but we can configure a variable like in ansible to create the resource even if the providers fail:

main.tf

```
provisioner "local-exec" {
    on_failure = continue
    command    = "echo ${self.public_ip} destroyed! >> output.txt"
}
```

Note

Terraform recommeds to use provisioners sparingly, because:

- Making use of provisioners increase the complexity of the configuration.
- `tf plan` has no way to accurately model the actions of a provisioner.
- A connection block must be defined for some provisioners to work, this means that the network connectivity form the local machine and authentication must be stablished before the provisioner is run.

So, in order to avoid that, Terraform recommends to make use of provisioners which are native to the resource, for example `user_data` creating EC2 instance, which is a native feature of AWS EC2, and there are arguments for the other Cloud Providers which can be used in the same way.:

main.tf

```
resource "aws_instance" "webserver" {
    ami           = "ami-0edbfdasñfjafadsf..."
    instance_type = "t2.micro"
    tags          = {
        Name        = "webserver"
        Description = "An NGINX WebServer on Ubuntu"
    }

    user_data     = <<EOF
        #!/bin/bash
        sudo apt update
        ...
        EOF

    provisioner "local-exec" {
        command = "echo ${self.public_ip} >> output.txt"
    }
}
```

8 Terraform Import

8.1 Terrafom Taint

There would be cases when a resource creation fails when we run `tf apply`, when this happens, Terraform marks the resource as tainted, and it will be shown when we run `tf apply` again, and it will try to recreate the entire resource because the old one is tainted.

This feature can be useful if, for some reason, we want to force a particular resource to be recreated, although it didn't fail when created last time.

- To taint a resource

```
$ tf taint <resource_type>.<resource_name>
```

- To untaint a resource

```
$ tf untaint <resource_type>.<resource_name>
```

8.2 Debugging in Terraform

To make terraform to have verbose, we can make use of a system environment variable:

```
$ export TF_LOG=<log_lvl>
```

Terraform accept 4 `log_lvl`:

- INFO: less verbosity
- WARNING
- ERROR
- DEBUG
- TRACE: most verbosity

To make terraform persist logged output in order to force the log to always be appended to a specific file when logging is enabled:

```
$ export TF_LOG_PATH=/path/to/local/file/for/tf/logs
```

8.3 Terraform Import

8.3.1 Introduction

In this section, we will learn how to input existing infrastructure into the Terraform configuration using `tf import` command. So far, we have seen how to create and manage resources using Terraform. Ideally, we want from the begining to initialize, create, manage and delete all infrastructure resources using Terraform, but in the real world, it is not always like that. And people like so much to use the Cloud Providers Consoles.

But what if we want to bring resources created by hand (or other methods like ansible) into the control of Terraform? This is where `tf import` comes into play. So far, we have seen how to read data from other resources that are not managed by the current Terraform repository using `datasources`. For example:

```
data "aws_instance" "newserver" {
    instance_id = "i-fdalkjsadfhasdhhkl03483209"
}
output newserver {
    value = data.aws_instance.newserver.public_ip
}
```

8.3.2 Import Command

But we cannot upgrade or delete this resource using Terraform, because it is not actually managed by it, just checked. So, to bring a resource completely in the management and control of Terraform, we have top **import it**, making use of the `tf import` command.

```
$ tf import <resource_type>.<resource_name> <instance_id>
```

So, using the same example as before but instead of using `datasource` using `tf import`:

```
$ tf import aws_instance.newserver i-971399e63faa4804a
```

WARNING

The execution of the previous command will fail, because the resouce does not exist on the configuration. It is normal, because Terraform just "import" the configuration into the `terraform.tfstate` file, but it does not add the resource configuration into your `main.tf`, that's something you need to do manually.

In order to fix this error, what we can do is add to our `main.tf` an empty resource block, although the configuration is not the desired one, the command `tf import` won't fail.

8.3.3 Create the resource

Then, we can modify the `main.tf` completing the resource block by defining all the resource arguments and their values, checking its configuration on the Cloud Provider Console, inspecting the `terraform.tfstate` file or using the `tf state show` command.

8.3.4 Check the results

Now, it-s time to check the resource imported, run the `tf plan` command and verify there is no changes to apply.

```
$ tf plan
```

WARNING

Sometinmes, can ocur that with actual object configuration we can not import the resource exactly in the same way it was when we created it, so we will need to manage with that accepting the `upgrades` or trying to modify the resource via Cloud Provider Console.

9 Terraform Modules

9.1 Introduction

While progressing through the course, you would have noticed that out terraform configuration files may become increasingly long. When dealing with large infrastructure projects with hundreds of resources, we can do it on a single file, because Terraform does not impose any limit on the number of resources per project or configuration file.

However, this would mean that out configuration file will contain hundreds to thousands lines of code. Alternatively, we can also split the configuration into multiple files within the configuration directory and things will work the same way, Terraform will consider any file with `.tf` extension as a configuration file. Although it would organize our Terraform code a bit better as compared to writting it all in a single configuration file, there are still a lot of disadvantages in following this approach when implementing Terraform. As for example:

- The complexity of the configuration files is the same.
- We can have a lot of duplicate code.
- Difficult to update a resource, because updates in one part of the configuration can have unintended effects on other resources.
- It serverely limits the reusability.

But... All your problems can be solved by making use of **modules** in Terraform. Any configuration directory containing a set of configuration files is called a **module**. So let's see an example of how to take advantage of using modules:

- Directory `terraform-projects/development`

terraform-projects/development/main.tf

```
module "dev-webserver" {
    source = "../aws-instance" # It can be relative path from terraform
                               # root module or full system path
}
```

- Directory `terraform-projects/aws-instance`

terraform-projects/aws-instance/main.tf

```
resource "aws_instance" "webserver" {
    ami           = var.ami
    instance_type = var.instace_type
    key_name      = var.key
}
```

terraform-projects/aws-instance/variables.tf

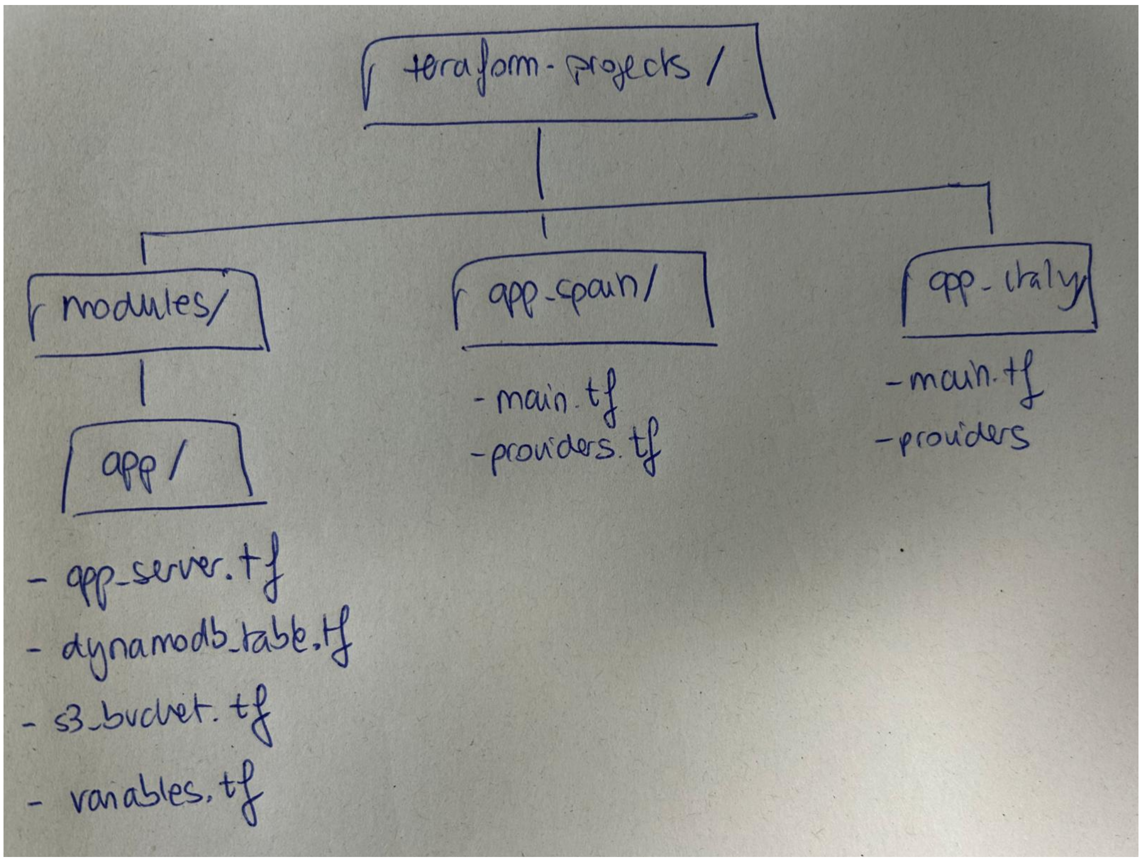
```
variable "ami" {
    type = string
    default = "ami-fasdfk"
    ...
}

variable "instance_type" {
....
}
```

So we need to run `terraform` from the `development` folder, which will be out **root module**, meanwhile the `aws_instance` will be a **child module**

9.2 Creating and Using Modules

So, imagine a project where you need to provision X resources for an application, and we want to have this application replicated Y times. So a good aproach for that should be the following:

- Structure



- `modules/app/`

terrafom-projects/modules/app/app_ server.tf

```
resource "aws_instance" "app_server" {
    ami           = var.ami
    instance_type = "t2.medium" # Hardcoded value
    tags          = {
        Name = "${var.app_region}-app-server"
    }

    depends_on = [ aws_dynamodb_table.payroll_db,
                   aws_s3_bucket.payroll_data
                 ]
}
```

terrafom-projects/modules/app/s3_ bucket.tf

```
resource "aws_s3_bucket" "payroll_data" {
    bucket = "${var.app_region}-${var.bucket}"
}
```

terrafom-projects/modules/app/dynamo_ table.tf

```
resource "aws_dynamodb_table" "payroll_db" {
    name         = "user_data"         # Hardcoded value
    billing_mode = "PAY_PER_REQUEST"   # Hardcoded value
    hash_key     = "EmployeeID"        # Hardcoded value

    attribute {
        name = "EmployeeID"
        type = "N"
    }
}
```

terrafom-projects/modules/app/variables.tf

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

variable "bucket" {
    type    = string
    default = "flexit-payroll-apha-ffasdf"
}

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

WARNING

There are variables hardcoded because we dont want them to be changed from the root module

- `app_spain/`: root module

terraform-projects/app_ spain/main.tf

```
module "app_spain" {
    source     = "../modules/app"
    app_region = "us-east-1"
    ami        = "..."
}
```

- `app_italy/`: root module

terraform-projects/app_ italy/main.tf

```
module "app_spain" {
    source     = "../modules/app"
    app_region = "us-east-2"
    ami        = "..."
}
```

Modules are comparable to libraries or packages that are used in most programming languages, or comparable to roles in ansible.

Benefits:

- Simpler configuration ocmpared to single configuration file with hundreds to thousands of lines.
- Our root module now it is very short and easy to understand and manage.
- Using a pre-configured module that has been tested and validated also decreases PBCAKs.
- Enables code reusability.
- Fix some variables of the configuration, to ensure they are not modified by the root module.

9.3 Using Modules from Terraform Registry

So far, we know that Terraform Registry is used to provide "providers", but it also used as a public repository that stores modules. The modules ono the Terraform Registry are sorted based on the provider for which they are created. They can be categorized into 2 types:

- **Verified Modules:** tested and mantainer by HashiCorp.
- **Community Modules:** published by users on the community but not validated by HashiCorp.

In some cases, the modules can contain several sub-modules that can be used for different use cases.

To use a remote module:

main.tf

```
module "security_group_ssh" {
    source     = "check/in/the/documentation"
    version    =
    vpc_id     =
    name       =
    ...
}
```

Note

It is a good idea to specify the version of the module, if you don't specify the version, it will always download the latest version available of this module on the registry.

To just download the needed remote modules from registries:

```
$ tf get
```

10 Terraform Functions

10.1 Already Learned Terraform Functions

We have already used a few Terraform functions throughout this course:

- `file()`: to read data from a file.
- `length()`: to determine the number of elements of a given list or map.
- `toset()`: to convert a list into a set.

Terraform provides an interactive console that can be used for testing functions as well as interpolations. To get into this interactive console, we need to use the command:

```
$ tf console
```

The interactive console loads the state associated with the configuration directory by default allowing us to load any values that is currently stored on it. It also loads variables that are stored within configuration files.

Usage examples inside the console:

tf console

```
> file("/root/terraform-projects/main.tf")
> len(var.region)
```

10.2 Basic New Terraform Functions

We can categorize functions in the following groups:

- Numeric Functions

- String Functions
- Collection Functions
- Type Conversion Functions

10.2.1 Numeric Functions

To operate and manipulate numbers:

- `max(-1, 2, -10, 200)` or `max(var.my_list)`
- `min(-1, 2, -10, 200)` or `min(var.my_list)`
- `ceil(10.3)`: to ceil a number.
- `floor(10.3)`: to ceil a number.
- ...

10.2.2 String Functions

To transform or manipulate strings:

- `split(" ", "Hola que ase")`: separate a string in a list of strings making use of the separator.
- `lower(string)`
- `upper(string)`
- `title(string)`: camel case.
- `substr(string, offset, length)`: get the substring between the positions "offset" and "offset + length".
- `join(",", ["hola", "que", "ase"])`: gather all the elements of a list in one single string.
- ...

10.3 Collection Functions (lists and sets)

- `lenght(collection)`
- `index(collection, "element_value")`: to find the index of a matching element.
- `element(collection, index)`: to retrieve an element in a collection located at a specific index.
- `contains(collection, "string to check")`: to check if an element is inside a list or not.
- ...

10.4 Map Functions

- `keys(map)`: to get the keys of a map into a list.
- `values(map)`: to get the keys of a map into a list.
- `lookup(map, "key")`: lookup the value for specific key, if it is not on the map, return an error.
- `lookup(map, "key","default_return")`: if it is not on the map, return "`default_return`"
- ...

10.5 Usage example

variables.tf

```
variable "cloud_users" {
    type = string
    default = "andrew:ken:faraz:mutsumi:peter:steve:braja"
}
```

main.tf

```
resource "aws_iam_user" "cloud" {
    name = split(":", var.cloud_users)[count.index]
    count = length(split(":", var.cloud_users))
}
```

10.6 Operators

Examples of automatic operations:

- `1 + 2`, `1 - 2`.
- `3 * 4`, `1 / 2`.
- `==`: equal operator returning a boolean.
- `!=`: not equal operator returning a boolean.
- `1 < 2, 1 > 2, 1 >= 2, 1 <= 2`: returning booleans.
- `&&`: local "and" operation returning a boolean.
- `||`: local "or" operation returning a boolean.
- `!`: not operator before an other operator.
- ...

10.7 Conditionals

```
condition ? true_val : false_val
```

For example:

main.tf

```
resource "aws_instance" "mario_servers" {
    ami           = var.ami
    instance_type = var.name == "tiny" ? "t2.nano" : "t2.2xlarge"

    tags = {
        Name = var.name
    }
}
```

11 Workspaces

Erlier on this documentation, we learned about state and its use when running Terraform operatins. Wheter stored locally or in remote backend such as S3 or another Terraform Provider option. And, as we know, state is essential in mapping the real-world infrastructure allowing Terraform to model what changes it needs to apply based on the configuration defined in the state files.

All the `terraform.tfstate` files we have seen so far have one-to-one mapping between the configuration directory and the `.tfstate` file. This means that we have one `terraform.tfstate` file per configuration directory.

Imagine the example that we have 2 environments, PRO and PRE and we want to deploy the same object, just changing the ami. So far, the only option we know is using modules and two different directories for each environment. However, the goal of using Terraform or any IaC tool for that matter is to eliminate repeatable steps and efficiently make use of existing code. For that purpose, Terraform offer a feature that allows configuration files within a directory to be reused multiple times for different use cases, called **workspaces**.

With **workspaces** we can use the same configuration directory to create multiple infrastructure environments; such as PRO, PRE, TST, QA, etc.

To create a **workspace**:

```
$ tf workspace new <workspace_name>
```

Note

Once a **workspace** is created, Terraform will immediately switch to it as well (not as git).

To list the **workspaces** that have been created:

```
$ tf workspace list
```

Note

default workspace is always created when you init a Terraform project, but it is hardly recommended to use **custom workspaces** for large terraform projects.

To switch to an specific **workspaces** that have been already created:

```
$ tf workspace select <workspace_name>
```

So, in order to use different configurations for each **workspace** we will need to remove all the **hardcoded** values from `main.tf` and put them all in `variables.tf` file. For example:

variables.tf

```
variable region {
    type    = string
    default = "ca-central-1"
}

variable instance_type {
    type    = string
    default = "t2.micro"
}

variable ami {
    type    = map
    default = {
        "PRO" = "ami-fkdapsñfkasñdj"
        "PRE" = "ami-0f80983409820p"
    }
}
```

main.tf

```
resource "aws_instance" "PRO" {
    ami = lookup(var.ami, terraform.workspace)
    instance_type = var.instance_type
    tags          = {
        Name = terraform.workspace
    }
}
```

Then, when we run `tf apply` it will generate 2 different `terraform.tfstate` files, one for each **workspace**, stored in a different directory:

```
/path/to/terraform-project/terraform.tfstate.d/
```