What is IAC?

Infrastructure as Code it is the process of managing infrastructure in a file or files rather than manually configuring resources in a user interface. A resource in this instance is any piece of infrastructure in a given environment, such as a virtual machine, security group, network interface, etc.

Terraform is a tool for building, changing, and versioning infrastructure safely and efficiently. Terraform can manage existing and popular service providers as well as custom in-house solutions.

Configuration files describe to Terraform the components needed to run a single application or your entire datacenter. Terraform generates an execution plan describing what it will do to reach the desired state, and then executes it to build the described infrastructure. As the configuration changes, Terraform is able to determine what changed and create incremental execution plans which can be applied.

The infrastructure Terraform can manage includes low-level components such as compute instances, storage, and networking, as well as high-level components such as DNS entries, SaaS features, etc.

MANAGE INFRASTRUCTURE

VARIABLES TYPES

Strings, Numbers, Boolean, List, or Maps. We can define a `default` value, example

```
variable "vpcname" {
    type = string
    default = "myvpc"
}
```

```
variable "enabled" {
    default = true
}
```

```
variable "sshport" {
    type = number
    default = 22
}
```

Note: Number or integers don't need double quotes, but Terraform automatically converts number and bool values to strings when needed. For example 5 and "5" both are correct.

VARIABLES LIST

List is the same than an array. We can store multiple values Remember the first value is the 0 position. For example to access the 0 position is `var.mylist[0]`

```
variable "mylist" {
    type = list(string)
    default = ["Value1", "Value2"]
}
```

VARIABLE MAP

Is a Key:Value pair. We use the key to access to the value

```
variable "mymap" {
```

```
        Key2 = "Value2"
    }
```

```
}
```

Important: For example, if we need to access the value of Key1 (Value1) we can using the next example `var.mymap["Key1"]`

Note: Remember, we use [ ] for list, and we use { } for maps

INPUT VARIABLES

Is useful to permit the user to manually set a variable when we run Terraform plan, we can add a "description" and when we run a plan shows a message

It is useful to permit the user to set a variable manually when we run Terraform plan, we can add a "description," and when we run a plan, it shows a message

```
variable "vpc_name" {
    type = string
    description = "Set VPC name"
}
```

terraform plan example:

```
var.inputname
        Set VPC name
        Enter a value:
```

OUTPUTS

Is about the resource we created, when we run the `apply` we can see the value, not in the `plan` because in the next case for example, we need first the VPC for know the vpc.id

```
output "vpc_id" {
    value = "aws_vpc.myvpc.id"
}
```

If we run a `apply` we can see the next message:

```
Apply complete!
Outputs:
```

```
vpcid = vpc-099d9099f5faec2d9
```

LOCAL VALUES

A local value assigns a name to an expression, allowing it to be used multiple times within a module without repeating it.

Comparing modules to functions in a traditional programming language: if input variables are analogous to function arguments and outputs values are analogous to function return values, then *local values* are comparable to a function's local temporary symbols.

Note: For brevity, local values are often referred to as just "locals" when the meaning is clear from context.

Declaring a local value:

A set of related local values can be declared together in a single `locals` block

```
locals {
  service_name = "forum"
  owner        = "Community Team"
}
```

The expressions assigned to local value names can either be simple constants like the above, allowing these values to be defined only once but used many times, or they can be more complex expressions that transform or combine values from elsewhere in the module:

When to use local values:

Local values can be helpful to avoid repeating the same values or expressions multiple times in a configuration, but if overused they can also make a configuration hard to read by future maintainers by hiding the actual values used.

Use local values only in moderation, in situations where a single value or result is used in many places *and* that value is likely to be changed in future. The ability to easily change the value in a central place is the key advantage of local values.

ENVIRONMENT VARIABLES

We can create an export with our variable before execute `terraform plan`, and overwrite the value on the .tf files, for example `export TF_VAR_vpcname=envvpc`. This is useful for pass secrets or sensitive information in a secure form.

CLI VARIABLES

Another way to set variables is by using the command-line, for example `terraform plan -var="vpcname=cliname"`

TFVARS FILES

Passing variables inside a file, this is possible create a file called `terraform.tfvars` this file can be in a yaml or json notation, and is very simple, and also we can add maps, for example:

```
vpcname = "tfvarsname"
port = 22
policy = {
        test = 1
        debug = "true"
}
```

Note: The `terraform.tfvars` file is used to define variables and the `.tf` file declare that the variable exists.

Link: https://amazicworld.com/difference-between-variable-tf-and-variable-tfvars-in-terraform

AUTO TFVARS

This is for example using a file called `dev.auto.tfvars` (is the next file that look after look in the terraform.tfvars)

MULTIPLE VALUE FILES

We can create a specified `*.tvars` file and load for example with `terraform plan`, this is very useful to settings variables for different environments.

```
terraform plan -var-file=prod.tfvars
```

LOAD ORDER

- Any -var and -var-file options on the command line, in order they are provided. (This includes variables set by a Terraform Cloud workspace.)
- Any *.auto.tfvars or *.auto.tfvars.json files, processed in lexical order of their filenames.
- The tfvars.jsonfile, if present. `terraform.tfvars.json`

- The tfvarsfile, if present. `terraform.tfvars`
- Environment variables

Note: there is no mention of .tf file declaration in there, this is because variables declared in .tf files are concatenated into a single entity consisting of your variables.tf your main.tf and your output.tf files before being processed by Terraform. Hence this declaration have highest precedence in order of application.

VERSIONING

The `required_version` setting can be used to constrain which versions of the Terraform CLI can be used with your configuration. If the running version of Terraform doesn't match the constraints specified, Terraform will produce an error and exit without taking any further actions.

```
terraform {
  required_version = ">= 0.12"
}
```

The value for `required_version` is a string containing a comma-separated list of constraints. Each constraint is an operator followed by a version number, such as `> 0.12.0`. The following constraint operators are allowed:

- `=` (or no operator): exact version equality
- `!=`: version not equal
- `>`, `>=`, `<`, `<=`: version comparison, where "greater than" is a larger version number
- `~>`: pessimistic constraint operator, constraining both the oldest and newest version allowed. For example, `~> 0.9` is equivalent to `>= 0.9, < 1.0`, and `~> 0.8.4`, is equivalent to `>= 0.8.4, < 0.9`

We can also specified a provider version requirement

```
provider "aws" {
        region = "us-east-1"
        version = ">= 2.9.0"
}
```

Link: https://www.terraform.io/docs/configuration/terraform.html#specifying-a-required-terraform-version

PROVIDERS

A provider is responsible for understanding API interactions and exposing resources. If an API is available, you can create a provider. A provider user a plugin. In order to make a provider available on Terraform, we need to make a `terraform init` , this commands download any plugins we need for our providers. If for example we need to copy the plugin directory manually, we can do it, moving the files to `.terraform.d/plugins`

Note: Using `terraform providers` command we can view the specified version constraints for all providers used in the current configuration

Example configuration:

```
terraform {
  required_providers {
    aws = "~> 2.7"
  }
}
```

Check:

```
$ terraform providers
.
└── provider.aws ~> 2.7
```

When `terraform init` is re-run with providers already installed, it will use an already-installed provider that meets the constraints in preference to downloading a new version. To upgrade to the latest acceptable version of each provider, run `terraform init -upgrade` . This command also upgrades to the latest versions of all Terraform modules.

MULTIPLE PROVIDER SETUP

We can use for example multiple AWS providers with different regions, for this we need to create an `alias` and on the resource creation we need to specified the provider. For example

```
provider "aws" {
        region = "us-east-1"
}

provider "aws" {
        region = "us-west-1"
        alias = "ireland"
}
```

```
resource "aws_vpc" "irlvpc" {
        cidr_block = "10.0.0.0/16"
        provider   = "aws.ireland"
}
```

PROVISIONERS

Provisioners can be used to model specific actions on the local machine or on a remote machine in order to prepare servers or other infrastructure objects for service.

Note: Provisioners should only be used as a last resort. For most common situations there are better alternatives.

Example:

```
resource "aws_instance" "web" {
  # ...
  provisioner "local-exec" {
    command = "echo The server's IP address is ${self.private_ip}"
  }
}
```

The `local-exec` provisioner requires no other configuration, but most other provisioners must connect to the remote system using SSH or WinRM.

CREATION-TIME PROVISIONERS

By default, provisioners run when the resource they are defined within is created. Creation-time provisioners are only run during *creation*, not during updating or any other lifecycle. They are meant as a means to perform bootstrapping of a system. If a creation-time provisioner fails, the resource is marked as tainted. A tainted resource will be planned for destruction and recreation upon the next `terraform apply`

DESTROY-TIME PROVISIONERS

If `when = "destroy"` is specified, the provisioner will run when the resource it is defined within is *destroyed*.

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

  provisioner "local-exec" {
    when    = "destroy"
```

```
      command = "echo 'Destroy-time provisioner'"
    }
  }
```

Destroy provisioners are run before the resource is destroyed. If they fail, Terraform will error and rerun the provisioners again on the next `terraform apply`

Note: By default, a defined provisioner is a creation-time provisioner. You must explicitly define a provisioner to be a destroy-time provisioner

LOCAL VS REMOTE EXEC

With Terraform the plugins have 2 options to do the job:

- Local-Exec: From our local machine
- Remote-Exec: On the remote instance

One example of local-exec is create a ssh key in our machine.

```
resource "null_resource" "generate-sshkey" {
    provisioner "local-exec" {
        command = "yes y | ssh-keygen -b 4096 -t rsa -C 'terraform-vsphere-kubernetes' -N '' -f ${var.virtual_machine_kubernetes_controller.["private_key"]}"
    }
}
```

Another example for local-exec is execute a script for Download Lambda dependencies, and after that, make a zip.

One example for remote-exec is from the key create previously, we can configure on a deployed virtual machine

```
  provisioner "remote-exec" {
    inline = [
      "mv /tmp/authorized_keys /root/.ssh/authorized_keys",
      "chmod 600 /root/.ssh/authorized_keys",
    ]
    connection {
      type         = "${var.virtual_machine_template.["connection_type"]}"
    }
  }
```

Links:

- https://www.terraform.io/docs/provisioners/local-exec.html
- https://www.terraform.io/docs/provisioners/remote-exec.html

MASTER THE WORKFLOW

3 types of users, the workflow change according to the user

- Individual
  - Write: Create the Terraform files
  - Plan: Run Terraform plan and check
  - Create: Create the infrastructure
- Team
  - Write: Create the Terraform files and Checkout the latest code
  - Plan: Run Terraform Plan and raise a Pull-Request
  - Create: Merge and create
- Terraform Cloud
  - Write: Use Terraform Cloud as your `development` environment (statefiles, variables and secrets on Terrafom Cloud)
  - Plan: When a PR is raised, Terraform Plan is run
  - Create: Before merging a second plan is run before approval to create

TERRAFORM INIT

The `terraform init` command is used to initialize a working directory containing Terraform configuration files. It is safe to run this command multiple times, , this command will never delete your existing configuration or state. During init, the root configuration directory is consulted for backend configuration and the chosen backend is initialized using the given configuration settings.

Link: https://www.terraform.io/docs/commands/init.html

TERRAFORM VALIDATE

The `terraform validate` command validates the configuration files in a directory, referring only to the configuration and not accessing any remote services such as remote state, provider APIs, etc.

Validate runs checks that verify whether a configuration is syntactically valid and internally consistent, regardless of any provided variables or existing state.

It is thus primarily useful for general verification of reusable modules, including correctness of attribute names and value types.

It is safe to run this command automatically, for example as a post-save check in a text editor or as a test step for a re-usable module in a CI system.

Note: Validation requires an initialized working directory with any referenced plugins and modules installed.

Link: https://www.terraform.io/docs/commands/validate.html

TERRAFORM PLAN

The `terraform plan` command is used to create an execution plan. Terraform performs a refresh, unless explicitly disabled, and then determines what actions are necessary to achieve the desired state specified in the configuration files.

Link: https://www.terraform.io/docs/commands/plan.html

TERRAFORM APPLY

The `terraform apply` command is used to apply the changes required to reach the desired state of the configuration, or the pre-determined set of actions generated by a `terraform plan` execution plan.

Link: https://www.terraform.io/docs/commands/apply.html

TERRAFORM DESTROY

The `terraform destroy` command is used to destroy the Terraform-managed infrastructure.

Link: https://www.terraform.io/docs/commands/destroy.html

LEARN MORE SUBCOMMANDS

FMT

The `terraform fmt` command is used to rewrite Terraform configuration files to a canonical format and style. The canonical format may change in minor ways between Terraform versions, so after upgrading Terraform it is recommended to proactively run `fmt`

TAINT

The `terraform taint` command manually marks a Terraform-managed resource as tainted, forcing it to be destroyed and recreated on the next apply. Taint force the recreation. This command *will not* modify infrastructure, but does modify the state file in order to mark a resource as tainted. Once a resource is marked as tainted, the next plan will show that the resource will be destroyed and recreated and the next apply will implement this change.

Forcing the recreation of a resource is useful when you want a certain side effect of recreation that is not visible in the attributes of a resource. For example: re-running provisioners will cause the node to be different or rebooting the machine from a base image will cause new startup scripts to run.

Example:

```
$ terraform taint aws_vpc.myvpc
The resource aws_vpc.myvpc in the module root has been marked as
tainted.
```

Another example if we want taint the resource "aws_instance" "baz" resource that lives in module bar which lives in module foo.

```
terraform taint module.foo.module.bar.aws_instance.baz
```

Link: https://www.terraform.io/docs/internals/resource-addressing.html

UNTAINT

The `terraform untaint` command manually unmark a Terraform-managed resource as tainted, restoring it as the primary instance in the state.

Note: This command *will not* modify infrastructure, but does modify the state file in order to unmark a resource as tainted.

```
$ terraform untaint aws_vpc.myvpc
Resource aws_vpc.myvpc2 has been successfully untainted.
```

IMPORT

The `terraform import` command is used to import existing resources into Terraform.

Import will find the existing resource from ID and import it into your Terraform state at the given ADDRESS.

ADDRESS must be a valid resource address. Because any resource address is valid, the import command can import resources into modules as well directly into the root of your state.

ID is dependent on the resource type being imported. For example, for AWS instances it is the instance ID (`i-abcd1234`) but for AWS Route53 zones it is the zone ID (`Z12ABC4UGMOZ2N`).

Usage: `terraform import [options] ADDRESS ID`

Example:

```
$ terraform import aws_vpc.vpcimport vpc-06f0e46d612
```

Note: `terraform import` command can import resources directly into modules

Link: https://www.terraform.io/docs/commands/import.html

SHOW

The `terraform show` command is used to provide human-readable output from a state or plan file. This can be used to inspect a plan to ensure that the planned operations are expected, or to inspect the current state as Terraform sees it.

Usage: `terraform show [options] [path]`

You may use `show` with a path to either a Terraform state file or plan file. If no path is specified, the current state will be shown.

WORKSPACE INTRODUCTION

The `terraform workspace` command is used to manage workspaces. It is useful to split and separate the statefiles.

This command is a container for further subcommands such as `list` , `select` , `new` , `delete` and `show`

Usage: `terraform workspace <subcommand> [options] [args]`

If we never create a workspace we use the default workspace

```
# terraform workspace list
* default
```

Note: For local state, Terraform stores the workspace states in a directory called `terraform.tfstate.d` . This directory should be treated similarly to local-only `terraform.tfstate`

Note: Terraform Cloud and Terraform CLI both have features called "workspaces," but they're slightly different. CLI workspaces are alternate state files in the same working directory; they're a convenience feature for using one configuration to manage multiple similar groups of resources.

CREATION A WORKSPACE

The `terraform workspace new` command is used to create a new workspace.

```
$ terraform workspace new dev
Created and switched to workspace "dev"!

You're now on a new, empty workspace. Workspaces isolate their state,
so if you run "terraform plan" Terraform will not see any existing state
for this configuration.
```

Now, if we execute a `terraform plan` in our environment, the plan will try to show many resources to create, because is a different state file.

SHOW WORKSPACE

The `terraform workspace show` command is used to output the current workspace.

```
$ terraform workspace show
dev
```

SELECT WORKSPACE

The `terraform workspace select` command is used to choose a different workspace to use for further operations. First we need to execute a `terraform workspace list` to know the workspaces names.

```
$ terraform workspace select default
Switched to workspace "default".
```

DELETE THE WORKSPACE

The `terraform workspace delete` command is used to delete an existing workspace.

```
$ terraform workspace delete dev
Deleted workspace "dev"!
```

STATE LIST

The `terraform state list` command is used to list resources within a Terraform state

The command will list all resources in the state file matching the given addresses (if any). If no addresses are given, all resources are listed.

Example list all resources:

```
$ terraform state list
aws_instance.foo
aws_instance.bar[0]
aws_instance.bar[1]
module.elb.aws_elb.main
```

Example Filtering by Resource

```
$ terraform state list aws_instance.bar
aws_instance.bar[0]
aws_instance.bar[1]
```

Link: https://www.terraform.io/docs/commands/state/list.html

STATE PULL

The `terraform state pull` command is used to manually download and output the state from remote state. This command also works with local state (but is not very useful because we can see the local file)

This command will download the state from its current location and output the raw format to stdout.

This is useful for reading values out of state (potentially pairing this command with something like jq). It is also useful if you need to make manual modifications to state.

STATE MV

The `terraform state mv` command is used to move items in a Terraform state. This command can move single resources, single instances of a resource, entire modules, and more. This command can also move items to a completely different state file, enabling efficient refactoring.

Usage: `terraform state mv [options] SOURCE DESTINATION`

This can be used for simple resource renaming, moving items to and from a module, moving entire modules, and more. And because this command can also move data to a completely new state, it can also be used for refactoring one configuration into multiple separately managed Terraform configurations.

Example:

```
terraform state mv 'packet_device.worker' 'packet_device.helper'
```

Link: https://www.terraform.io/docs/commands/state/mv.html

STATE RM

The `terraform state rm` command is used to remove items from the Terraform state. This command can remove single resources, single instances of a resource, entire modules, and more.

Usage: `terraform state rm [options] ADDRESS...`

Items removed from the Terraform state are *not physically destroyed*. Items removed from the Terraform state are only no longer managed by Terraform. For example, if you remove an AWS instance from the state, the AWS instance will continue running, but `terraform plan` will no longer see that instance.

There are various use cases for removing items from a Terraform state file. The most common is refactoring a configuration to no longer manage that resource (perhaps moving it to another Terraform configuration/state).

Example remove a resource:

```
$ terraform state rm 'packet_device.worker'
```

Example remove a module:

```
$ terraform state rm 'module.foo'
```

Link: https://www.terraform.io/docs/commands/state/rm.html

USE AND CREATE MODULES

A module is a simple directory that contains other .tf files. Using modules we can make the code reusable. Modules are local or remote.

TERRAFORM REGISTRY

Is the place to find modules, theses modules are verified by Hashicorp

Link: https://registry.terraform.io/

MODULES INPUTS/OUTPUTS

For make modules inputs we use inputs variables. Example module code:

```
variable "dbname" {
        type = string
}
resource "aws_instance" "myec2db" {
        ami = "ami-01a6e"
        tags = {
                Name = var.dbname
        }
}
```

Call to the module example:

```
module "dbserver" {
        source = "./db"
        dbname = "mydbserver"
}
```

Module outputs are very similar to module inputs, an example in a module output:

```
output "privateip" {
        value = aws_instance.myec2db.private_ip
}
```

When we use a module with an output, to use the output we need to specified in the call to our module, for example:

```
module "dbserver" {
        source = "./db"
        dbname = "mydbserver"
}

output "dbprivateip" {
        value = module.dbserver.privateip
}
```

CHILD MODULES

Terraform allow having child modules, modules within modules. Basically is a directory with tf files, with others directories with others sub modules. After add a subdirectory, remember to execute again `terraform init`

READ AND WRITE CONFIGURATION

DYNAMIC BLOCKS

Within top-level block constructs like resources, expressions can usually be used only when assigning a value to an argument using the `name = expression` form. This covers many uses, but some resource types include repeatable *nested blocks* in their arguments.

You can dynamically construct repeatable nested blocks like `setting` using a special `dynamic` block type, which is supported inside `resource`, `data`, `provider`, and `provisioner` blocks:

```
resource "aws_elastic_beanstalk_environment" "tfenvtest" {
  name                = "tf-test-name"
  application         =
"${aws_elastic_beanstalk_application.tftest.name}"
  solution_stack_name = "64bit Amazon Linux 2018.03 v2.11.4 running Go
1.12.6"

  dynamic "setting" {
    for_each = var.settings
    content {
      namespace = setting.value["namespace"]
```

```
          name = setting.value["name"]
          value = setting.value["value"]
        }
      }
    }
```

A `dynamic` block acts much like a `for` expression, but produces nested blocks instead of a complex typed value. It iterates over a given complex value, and generates a nested block for each element of that complex value.

Link: https://www.terraform.io/docs/configuration/expressions.html

TUPLES

The difference between a tuple and a list, is on the list we need to specified one type (string or numbers), and using tuple we can use multiple data-types

```
variable "mytuple" {
    type = tuple([string, number, string])
    default = ["cat", 1, "dog"]
}
```

OBJECTS

Is the same case of tuples with list, but in this case is with maps. Using Objects we can use multiple data-types

```
variable "myobject" {
    type = object({name = string, port = list(number)})
    default = {
        name = "TJ"
        port = [22, 25, 80]
    }
}
```

Note: Using objects and tuples allows us to have multiple values of several distinct types to be grouped as a single value.

DEPENDENCIES

Explicitly specifying a dependency is only necessary when a resource relies on some other resource's behavior but *doesn't* access any of that resource's data in its arguments.

```
resource "aws_instance" "example" {
  ami           = "ami-a1b2c3d4"
  instance_type = "t2.micro"
  depends_on = [aws_iam_role_policy.example]
}
```

The `depends_on` meta-argument, if present, must be a list of references to other resources in the same module.

DATA SOURCES

Data Sources are the way Terraform can query AWS and return results (Api request to get information)

Use of data sources allows a Terraform configuration to make use of information defined outside of Terraform, or defined by another separate Terraform configuration.

For example we can information to an EC2 created on AWS without being created using Terraform,

Example using a data source to know the AZ of an Instance created without Terraform, a print on the output command.

```
# Find the latest available AMI that is tagged with Component = "DB server"
data "aws_instance" "dbsearch" {
filter {
        name = "tag:Name"
        values = ["DB server"]
 }
}
output "dbserver" {
 value = data.aws_instance.dbsearch.availability_zone
}
```

A data block request that Terraform read from a given data source and export the result under the give local name.

Note: Data source attributes are interpolated with the general syntax *data.TYPE.NAME.ATTRIBUTE*. The interpolation for a resource is the same but without the *data.* prefix (TYPE.NAME.ATTRIBUTE).

Link: https://www.terraform.io/docs/providers/aws/d/instance.html

BUILT-IN FUNCTIONS

The Terraform language includes a number of built-in functions that you can call from within expressions to transform and combine values. The general syntax for function calls is a function name followed by comma-separated arguments in parentheses:

```
max(5, 12, 9)
```

Note: The Terraform language does not support user-defined functions, and so only the functions built in to the language are available for use.

A very useful function is the `file` function, using that, we can read the contents of a file and returns them as a string. Another important function is the `flatten` function, this takes a list and replaces any elements that are list with a flattened sequence of the list contents.

The last important function is `lookup` , this retrieves the value of a single element from a map, given its key. If the given key does not exist, a the given default value is returned instead.

```
lookup(map, key, default)
```

Examples:

```
> lookup({a="ay", b="bee"}, "a", "what?")
ay
> lookup({a="ay", b="bee"}, "c", "what?")
what?
```

Links:

- https://www.terraform.io/docs/configuration/functions.html
- https://www.terraform.io/docs/configuration/expressions.html#function-calls

SECURING KEYS

In the case of using AWS provider the best practice for store credentials is having the keys in the AWS config file `.aws/credentials` , and not having the credentials in the Terraform code

SENTINEL

Sentinel is an embedded policy-as-code framework integrated with the HashiCorp Enterprise products. It enables fine-grained, logic-based policy decisions, and can be extended to use information from external sources.

SECRET INJECTION

Vault allows you to manage secrets and protect sensitive data. Secure, store and tightly control access to tokens, passwords, certificates, encryption keys for protecting secrets and other sensitive data using a UI, CLI, or HTTP API.

FILES TO UPLOAD

The following Terraform files should be ignored by Git when committing code to a repo:

The `terraform.tfstate` or `.auto.tfvars` files contains the terraform state of a specific environment and doesn't need to be preserved in a repo.
The `terraform.tfvars` file may contain sensitive data, such as passwords or IP addresses of an environment that you may not want to share with others.

MANAGE STATE

LOCAL STATE

The local backend stores state on the local filesystem, locks that state using system APIs, and performs operations locally.

```
terraform {
  backend "local" {
    path = "relative/path/to/terraform.tfstate"
  }
}
```

PARTIAL CONFIGURATION

You do not need to specify every required argument in the backend configuration. Omitting certain arguments may be desirable to avoid storing secrets, such as access keys, within the main configuration. When some or all of the arguments are omitted, we call this a *partial configuration*.

With a partial configuration, the remaining configuration arguments must be provided as part of the initialization process. There are several ways to supply the remaining arguments:

- Interactively: Terraform will interactively ask you for the required values
- File: A configuration file may be specified via the `init` command line. To specify a file, use the `-backend-config=PATH` option when running `terraform init`
- Command-line key/value pairs: Key/value pairs can be specified via the `init` command line.

TERRAFORM REFRESH

The `terraform refresh` command is used to reconcile the state Terraform knows about (via its state file) with the real-world infrastructure. This can be used to detect any drift from the last-known state, and to update the state file.

This does not modify infrastructure, but does modify the state file. If the state is changed, this may cause changes to occur during the next plan or apply.

Usage: `terraform refresh [options] [dir]`

For example, if we change some configuration using the web console, we need to do:

```
$ terraform refresh
$ terraform plan
```

`refresh` Is good to sync changes, in some cases is better to use `terraform import` depending of the complexity

Link: https://www.terraform.io/docs/commands/refresh.html

REMOTE BACKENDS

Stores the state as a given key in a given bucket on Amazon S3. This backend also supports state locking and consistency checking via Dynamo DB, which can be enabled by setting the `dynamodb_table` field to an existing DynamoDB table name. A single DynamoDB table can be used to lock multiple remote state files. Terraform generates key names that include the values of the `bucket` and `key` variables.

Example:

```
terraform {
  backend "s3" {
    bucket = "mybucket"
    key    = "state/terraform.tfstate"
    region = "us-east-1"
  }
}
```

Note: This assumes we have a bucket created called `mybucket`. The Terraform state is written to the key `state/terraform.tfstate`.

Important: If we have a local state, and after that we change to the S3 backend, when we execute `terraform init` the system ask us if we want to copy existing state to the new backend.

Similar to if we delete a remote backend from the configuration, when we execute the re-initialization, Terraform will ask if we would like to migrate our state back down to normal local state.

Link: https://www.terraform.io/docs/backends/types/s3.html

REVERT TO LOCAL BACKEND

If we want to change from S3 backend to Local backend, only we need to do `terraform destroy` after that delete `backend.tf` file, and run `terraform init`

BACKEND LIMITATIONS & SECURITY

When we use Terraform is only allowed one backend.

The state cannot store secrets, for that reason we need to encrypt at rest.

TERRAFORM FORCE UNLOCK

Manually unlock the state for the defined configuration.

This will not modify your infrastructure. This command removes the lock on the state for the current configuration. The behavior of this lock is dependent on the backend being used. Local state files cannot be unlocked by another process.

Usage: terraform force-unlock LOCK_ID [DIR]

This is useful for example when another person write the remote state file, and something happening and the state file is lock, and we need to release.

Link: https://www.terraform.io/docs/commands/force-unlock.html

TERRAFORM CLOUD

If we need the best solution for encryption and backups, the answer is Terraform Cloud. Terraform Cloud encrypt the state file at rest, and also encrypt using TLS.

STATE IN MEMORY

When we use remote states, Terraform save all the temporal information on memory, nothing persistent on disk, this is another Terraform security implementation.

BACKEND AND STRING INTERPOLATION

Backend cannot have any interpolations or use any variables. This need to be manually hardcoded

STATE PUSH

The `terraform state push` command is used to manually upload a local state file to remote state. This command also works with local state.

Usage: `terraform state push [options] PATH`

Link: https://www.terraform.io/docs/commands/state/push.html

TYPES OF BACKEND

- Standard: state management, storage and locking
- Enhanced: Only on Terraform Cloud, standard + can run operations remotely

Note: Backend that support state lockings: Azurerm, Consul, S3

RECAP OF STATE

- Only ONE backend allowed
- Secrets are stored in state
- State locking when working with teams
- State is stored in memory when using a remote backend
- Standard and Enhanced backends
- No interpolation allowed in backend setup
- Terraform refresh will attempt to resync the state
- Terraform state push will override the state

DEBUG IN TERRAFORM

Terraform has detailed logs which can be enabled by setting the `TF_LOG` environment variable to any value. This will cause detailed logs to appear on stderr.

Is not about to Terraform Client Debugging, panic errors, go errors, etc. Is useful to provide the logs to Hashicorp.

```
export TF_LOG=TRACE
```

You can set `TF_LOG` to one of the log levels `TRACE`, `DEBUG`, `INFO`, `WARN` or `ERROR` to change the verbosity of the logs. `TRACE` is the most verbose and it is the default if `TF_LOG` is set to something other than a log level name.

Note: When `TF_LOG_PATH` is set, `TF_LOG` must be set in order for any logging to be enabled, and `TF_LOG_PATH` point to a specific file (not directory), for example `TF_LOG_PATH=./terraform.log`

Link: https://www.terraform.io/docs/internals/debugging.html

UNDERSTAND TERRAFORM CLOUD AND ENTERPRISE

TERRAFORM CLOUD

Terraform Cloud (TFC) is a free to use, self-service SaaS platform that extends the capabilities of the open source Terraform CLI and adds collaboration and automation features.

Terraform Cloud enables connecting to common VCS platforms (GitHub, GitLab, Bitbucket) and triggering Terraform runs (plan and apply) from changes to configuration within the VCS. TFC manages state for the user, including keeping a history of changes. Terraform Cloud exposes an HTTP API that anyone can integrate with to build more automation around infrastructure change.

TERRAFORM CLOUD SETUP

First we need to create an account on https://app.terraform.io/signup/account and create a new Git repository, and upload our `.tf` files.

After that on Terraform Cloud web page we need to create an Organization, only we need to specified the name, and one email address, after that we can connect to our Git repository (Github, Gitlab, Bitbucket and Azure DevOps are available), also we need to authorize Terraform Cloud in our Git repository, is only one click.

Also we can implement a Private module registry in Terraform Cloud.

Now we need to create a Workspace on Terraform Cloud (is different to the Workspace on Terraform), this is for create a separate environment, for example dev, prod, etc.

Right now we need to Configure Variables, because we used AWS provider, we need to configure the AWS credentials (Also we can configure using Vault)

For the last, we need to configure a Queue Plan, this allow us to execute a `Terraform plan` after this complete, we can Confirm & Apply

Note: The remote backend stores Terraform state and may be used to run operations in Terraform Cloud. When using full remote operations, operations like `terraform plan` or `terraform apply` can be executed in Terraform Cloud's run environment, with log output streaming to the local terminal.

Note: Workspaces, managed with the `terraform workspace` command, aren't the same thing as Terraform Cloud's workspaces. Terraform Cloud workspaces act more like completely separate working directories; CLI workspaces are just alternate state files.

Note: Terraform Cloud always encrypts state at rest and protects it with TLS in transit.

TERRAFORM CLOUD DESTROY

If we want to do a Destroy, we only need to click on the option "Queue destroy plan"

TERRAFORM CLOUD COMPARISON OSS VS CLOUD

Component	Local Terraform	Terraform Cloud
Terraform Configuration	On Disk	In linked version control repository, or periodically updated via API/CLI
Variable values	As .tfvars file, as CLI arguments, or in shell environment	In workspace
State	On disk or in remote backend	In workspace
Credential and secrets	In shell environment or entered at prompts	In workspace, stored as sensitive variables

TERRAFORM CLOUD PAID FEATURES

Some paid features are Roles & Team management, Cost Estimation and Sentinel.

Up to 5 users we can have the free tier, with VCS integration, Workspace Management, Secure Variable Storage, Remote Runs & Applies, Full API Coverage and Private Module Registry.

Link: https://www.hashicorp.com/products/terraform/pricing/

TERRAFORM CLOUD COMPARISON CLOUD VS ENTERPRISE

Terraform Enterprise is a hosted version of Terraform Cloud, but using Terraform Enterprise we can have the following features:

- SAML/SSO
- Audit Logs
- Private Network Connectivity
- Clustering

  Sentinel, VCS Integration are offered also in Terraform Cloud. Everything that is in Terraform cloud is already included in Terraform Enterprise.

EXTRA NOTES

- A Terraform Enterprise install that is provisioned on a network that does not have Internet access is generally known as an air-gapped install. These types of installs require you to pull updates, providers, etc. from external sources vs. being able to download them directly.
- Terraform Enterprise requires a PostgresSQL for a clustered deployment.
- Some Backends supported: Terraform Enterprise, Consul, S3, Artifactory.
- Terraform Cloud supports the following VCS providers: GitHub, Gitlab, Bitbucket and Azure DevOps
- The existence of a provider plugin found locally in the working directory does not itself create a provider dependency. The plugin can exist without any reference to it in Terraform configuration.
- Function `index` finds the element index for a given value in a list starting with index 0.
- HashiCorp style conventions suggest you that align the equals sign for consecutive arguments for easing readability for configurations

  ```
  ami           = "abc123"
  instance_type = "t2.micro"
  ```

- Terraform can limit the number of concurrent operations as Terraform walks the graph using the `-parallelism=n` argument. The default value for this setting is `10`. This setting might be helpful if you're running into API rate limits.
- HashiCorp style conventions state that you should use 2 spaces between each nesting level to improve the readability of Terraform configurations.

- Terraform supports the #, //, and /../ for commenting Terraform configuration files. Please use them when writing Terraform so both you and others who are using your code have a full understanding of what the code is intended to do.
- The `terraform console` command provides an interactive console for evaluating expressions such interpolations. https://www.terraform.io/docs/commands/console.html