Terraform notes

Sunday 05 November 2023

00. References

Installation

— Downloads
— tfenv
— TFSwitch

IPPON

— Old slides
— New slides
— GitHub: Terraform base template

Udemy

— Great course - GitHub - http://kplabs.in/chat → `#terraform-associate` channel on slack
— Dedicated exam questions pack

HashiCorp certification

— https://www.hashicorp.com/certification/terraform-associate
— https://developer.hashicorp.com/terraform/tutorials/certification-003

01. CLI usage

`terraform init`

— The terraform init command is used to initialize a working directory containing Terraform configuration files.
— Plugins (Providers) code is downloaded inside `.terraform/` and `.terraform.lock.hcl` is used or updated
— Modules code is downloaded
— It will NOT create any sample files like `example.tf`
— `-upgrade` install the latest module and provider versions allowed within configured constraints, overriding the default behavior of selecting exactly the version recorded in the dependency lockfile.

`terraform validate`

The `terraform validate` command validates the configuration files in a directory.

Validate runs checks whether a configuration is syntactically valid including the 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 reusable module in a CI system. It can run before a terraform plan.

→ Validation requires an initialized working directory with any referenced plugins and modules installed

`terraform plan`

— The terraform plan command is used to create an execution plan.
— It will NOT modify things in infrastructure.
— Terraform performs a `refresh`, unless explicitly disabled (`-refresh=false`), and then determines what actions are necessary to achieve the desired state specified in the configuration files.

→ This command is a convenient way to check whether the execution plan for a set of changes matches your expectations without making any changes to real resources or to the state (for instance with a PR).

`terraform apply`

The terraform apply command is used to apply the changes required to reach the desired state of the configuration.

Terraform apply will also write data to the `terraform.tfstate` file.

Once apply is completed, resources are immediately available.

`terraform apply -auto-approve`

Safer apply

— `terraform plan -out=demopath`
— `terraform apply demopath`

Note : Terraform uses Parallelism to reduce the time it takes to create the resource. By default, this value is set to 10

`terraform destroy`

Used to destroy the Terraform-managed infrastructure.

```
terraform destroy -auto-approve
```

→ `terraform destroy` command is not the only command through which infrastructure can be destroyed. For instance, removing a resource and performing an apply...

`terraform state`

— `terraform state list` List resources within a Terraform state
— `terraform state show <resource>` Show the attributes of a single resource in the Terraform state.
— `terraform state pull` Manually download and output the state from remote state. This is useful for reading values out of state (potentially pairing this command with something like jq).
— `terraform state push` Update remote state from a local state file
— `terraform state mv` Rename the id used by terraform to reference an infrastructure element (prevents destroy / recreate)
— `terraform state rm` Remove instance from the state (forget that an existing resource is managed by terraform)

`terraform output`

The `terraform output` command is used to extract the value of an output variable from the state file.

```
terraform output myvar
```

`terraform graph`

```
terraform graph > graph.dot
```

```
cat graph.dot | dot -Tsvg > graph.svg
```

→ `dot` executable is installed using the `graphviz` package

`terraform fmt`

The `terraform fmt` command is used to rewrite Terraform configuration files to a canonical format and style.

For use-case, where the all configuration written by team members needs to have a proper style of code, `terraform fmt` can be used.

`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 does not modify infrastructure but does modify the state file.

— `resfreh` is executed implictly during `plan`, `apply`, `destroy`.
— Others like terraform `init`, `import` do not run `refresh` implictly.

Tfstate best practices

1 keep it secured!!! It's a FULL description of the infra and it contains a LOT of secrets, even if they are flaged as sensitive.
2 Use a backend

Troubleshooting and large infrastructures

Output

Terraform output can show an output recorded inside the state.

IMPORTANT: output must be defined inside the tf file to be part of the state !

```
# inside the tf file
output "arns" {
  value = aws_instance.myec2[*].arn
}
```

Then `terraform output arns`

Don't refresh state. This is a hack when the infra is huge

`terraform plan -refresh=false`

Plan a specific update

`terraform plan -refresh=false -target=aws_security_group.allow_ssh_conn`

Destroy a specific resource

`terraform destroy -target aws_instance.myec2`

Recreate (replace) a specific resource

`terraform apply -replace="aws_instance.web"` # see also: taint

Debug

You can set `TF_LOG` to one of the log levels `TRACE`, `DEBUG`, `INFO`, `WARN` or `ERROR` to change the verbosity of the logs.

`export TF_LOG=TRACE`

`export TF_LOG_PATH=/tmp/crash.log`

Comments

`# Standard single line comment`

`// single line C style comment. Prefer #`

`/* multiline C style comment */`

02. Provider

Plugins

— Plugins are executable binaries written in Go that communicate with Terraform Core over an RPC interface. **Terraform currently supports only one type of plugin called providers.**
— Plugins are stored inside `.terraform` folder

Introduction

A provider is responsible for understanding API interactions and exposing resources.

Most of the available providers correspond to one cloud or on-premises infrastructure platform, and offer resource types that correspond to each of the features of that platform. You can explicitly set a specific version of the provider within the provider block.

To upgrade to the latest acceptable version of each provider, run `terraform init -upgrade`

Each Terraform module must declare which providers it requires, so that Terraform can install and use them. Provider requirements are declared in a `required_providers block`.

Version definition

— `version = "2.7"`
— `version = ">= 2.8"`
— `version = "<= 2.8"`
— `version = ">=2.10,<=2.30"`
— `version = "~> 2.0"`

Example 1: AWS provider usage

```
# terraform settings: no variable can be used inside this block
terraform {
  # if needed
  required_version = "> 0.12.0"
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 5.0"
    }
  }
}
provider "aws" {
  region = "eu-west-1"
}
resource "aws_instance" "myec2" {
  ami           = "ami-00c39f71452c08778"
  instance_type = "t2.micro"
}
```

Example 2: Vault provider usage

```
provider "vault" {
  address = "http://127.0.0.1:8200"
}
data "vault_generic_secret" "demo" {
  path = "secret/db_creds"
}
output "vault_secrets" {
  value     = data.vault_generic_secret.demo.data_json
  sensitive = "true"
}
```

Versions management

See: Dependency lock file - Tutorial

`terraform init` installs providers

— With releases matching the one registered inside `.terraform.lock.hcl` file
— Or with the release matching constraints. Then, the installed release is registered inside `.terraform.lock.hcl`

→ sharing `.terraform.lock.hcl` inside git ensure everyone is using the same provider releases.

Note

— `terraform init -upgrade` installs the latest provider releases that are matching constraints, ignoring `.terraform.lock.hcl`
— `.terraform.lock.hcl` is NOT used for locking the state

Multiple provider configurations

See: alias: Multiple Provider Configurations

```
# A provider block without an alias argument is the default
   configuration for that provider.
provider "aws" {
  region = "eu-west-1"
}
provider "aws" {
  alias  = "backup_region"
  region = "eu-west-2"
  # for having different credentials
  profile = "disaster_reco"
}
resource "aws_instance" "main_ec2" {
  ami           = "ami-00c39f71452c08778"
  instance_type = "t2.micro"
}
resource "aws_instance" "backup_ec2" {
  provider      = "aws.backup_region"
  ami           = "ami-00c39f71452c08778"
  instance_type = "t2.micro"
}
```

03. Variables

Variables

```
variable "image_id" {
  type        = string
  description = "Image ID to use when creating an instance"
  default     = "id-xxxx"
  sensitive   = false
  nullable    = false
  validation {
    condition     = length(var.image_id) > 3 && substr(var.image_id,
   0, 3) == "id-"
    error_message = "The image_id value must be a valid Image id,
   starting with \"id-\"."
  }
}
```

Terraform CLI defines the following optional arguments for variable declarations:

- `type` This argument specifies what value types are accepted for the variable.
  - `string`
  - `number`
  - `bool` : can be `true` or `false`
  - `list` : for instance, `type = list(string)`
  - `map`
- `default` A default value which then makes the variable optional.
- `description` This specifies the input variable's documentation.
- `validation` A block to define validation rules, usually in addition to type constraints.
- `sensitive` Limits Terraform UI output when the variable is used in configuration. **IMPORTANT: the variable remains plain text readable inside the state.**
- `nullable` Specify if the variable can be `null` within the module.

Examples

```
# this is valid!
variable "image_id" {}
```

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

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

```
variable "docker_ports" {
  type = list(object({
    internal = number
    external = number
    protocol = string
  }))
  default = [
    {
      internal = 8300
      external = 8300
      protocol = "tcp"
    }
  ]
}
```

Reference attributes

```
resource "aws_eip" "lb" {
  domain = "vpc"
}
resource "aws_security_group" "allow_tls" {
  name        = "allow_tls"
  description = "Allow TLS inbound traffic"
  ingress {
    description = "TLS from VPC"
    from_port   = 443
    to_port     = 443
    protocol    = "tcp"
    cidr_blocks = ["${aws_eip.lb.public_ip}/32"]
  }
}
```

Syntax

- `<resourcetype>.<resourcename>.<attribute>`
- `data.<name>`
- `var.<name>`
- `local.<name>`
- `module.<instancename>.<outputname>`

Output values

They can be used to

- Display information
- Register data inside the state
- Send information back to a caller module

```
output "public_url" {
  value = "https://${aws_eip.lb.public_ip}:8080"
}
```

Output can be referenced from a caller module

```
# Here, we are in the root module
module "my_endpoint" {
  source = ../module/endpoint
}
# module.my_endpoint.public_url
```

Ordre des précédance des variables

1 Default values
2 Environnement `export TF_VAR_str="env"` → defines str=env
3 `terraform.tfvars` file with lines such as `key=value`
4 `terraform.tfvars.json` file
5 `*.auto.tfvars` and `*.auto.tfvars.json` files
6 CLI using `-var` or `-var-file`

→ Storing credentials as part of environment variables is also a much better approach than hard coding it in the source files.

Basic map usage

```
resource "aws_instance" "myec2" {
  ami           = "ami-082b5a644766e0e6f"
  instance_type = var.types["us-west-2"]
}
variable "types" {
  type = map(any)
  default = {
    us-east-1  = "t2.micro"
    us-west-2  = "t2.nano"
    ap-south-1 = "t2.small"
  }
}
```

Access

— `var.my_map.my_key`
— `var.my_map["my_key"]`
— `lookup(var.my_map, "my_key", <default_value>)`

→ Note: access using zero based index is NOT possible.

Map

— merge takes an arbitrary number of maps or objects, and returns a single map or object that contains a merged set of elements from all arguments.
— lookup retrieves the value of a single element from a map, given its key. If the given key does not exist, the given default value is returned instead.

Basic list usage

`list(...)`: a sequence of values identified by consecutive whole numbers starting with zero. The keyword `list` is a shorthand for `list(any)`, which accepts any element type as long as every element is **the same type**.

```
resource "aws_instance" "myec2" {
  ami           = "ami-082b5a644766e0e6f"
  instance_type = var.list[0]
}
```

```
variable "list" {
  type    = list(string)
  default = ["m5.large", "m5.xlarge", "t2.medium"]
}
```

Access

— `var.my_list[<index>]`

— `element(var.my_list, <index>)`

Key functions

— concat takes two or more lists and combines them into a single list.
— contains determines whether a given list or set contains a given single value as one of its elements.
— formatlist produces a list of strings using a string template and a list of variables
— index finds the element index for a given value in a list.

list + list → map : use zipmap !

See: zipmap

```
zipmap(["a", "b"], [1, 2])
{
  "a" = 1
  "b" = 2
}
```

Local Values

Constants that are defined within a module.

Local values can be helpful to avoid repeating the same values or expressions multiple times in a configuration.

```
locals {
  common_tags = {
    Owner   = "DevOps Team"
    service = "backend"
  }
}
resource "aws_instance" "app-dev" {
  ami           = "ami-082b5a644766e0e6f"
  instance_type = "t2.micro"
  tags          = local.common_tags
}
```

Locals can help defining helpers

```
locals {
  name_prefix = "${var.name != "" ? var.name : var.default_name}"
}
```

For expression

See documentation

Have a deep look at it since there are questions on the exam about this statement.

04. Loops, iterations and conditions

Count and count index

In resource blocks where the count is set, an additional count object (`count.index`) is available in expressions, so that you can modify the configuration of each instance.

```
resource "aws_instance" "myec2" {
  count          = 3
  ami            = "ami-0dab0800aa38826f2"
  instance_type = "t2.micro"
  tags = {
    Name = "myinstance-${count.index}"
  }
}
```

```
output "arns" {
  value = aws_instance.myec2[*].arn
}
```

List iteration

```
variable "users" {
  type        = list(string)
  description = "user names"
  default     = ["alice", "bob"]
}
resource "aws_iam_user" "myuser" {
  count = length(var.users)
  name  = element(var.users, count.index)
  path  = "/system/"
  tags = {
    tag-key = "tag-value"
  }
}
```

Count vs for_each

— If your resources are almost identical, `count` is appropriate.
— If distinctive values are needed in the arguments, usage of `for_each` is recommended.

Set

SET items are **unordered** and **no duplicates** allowed.

Convert a list to a set : toset function (which removes ordering and duplicates)

for_each

With a set → `each.key` **(which equals** `each.value` **)**

```
resource "aws_iam_user" "iam" {
  for_each = toset(["user-0", "user-01", "user-02", "user-03"])
  name     = each.key
}
```

`each.key` value is used to reference a given instance: `aws_iam_user.iam["user-0"]` .

With a map → `each.key` / `each.value`

```
resource "aws_instance" "myec2" {
  ami = "ami-0cea098ed2ac54925"
  for_each = {
    little_instance = "t2.micro"
    big_instance    = "t2.medium"
  }
  instance_type = each.value
  tags = {
    Name = each.key
  }
}
```

`each.key` value is used to reference a given instance: `aws_instance.myec2["little_instance"]` .

Conditional Expression

```
variable "istest" {}
resource "aws_instance" "dev" {
  ami           = "ami-082b5a644766e0e6f"
  instance_type = "t2.micro"
  count         = var.istest == true ? 3 : 0
}
resource "aws_instance" "prod" {
  ami           = "ami-082b5a644766e0e6f"
  instance_type = "t2.large"
  count         = var.istest == false ? 1 : 0
}
```

Dynamic block

Dynamic Block allows to dynamically construct repeatable nested blocks **inside** resource, data, provider, and provisioner blocks.

Note: this feature is a bit difficult to read and therefore, alternatives should be used when it is possible.

```
variable "sg_ports" {
  type        = list(number)
  description = "list of ingress ports"
  default     = [8200, 8201, 8300, 9200, 9500]
}
resource "aws_security_group" "dynamicsg" {
  name        = "dynamic-sg"
  description = "Ingress for Vault"
  # with an iterator
  dynamic "ingress" {
    for_each = var.sg_ports
    iterator = port
    content {
      from_port   = port.value
      to_port     = port.value
      protocol    = "tcp"
      cidr_blocks = ["0.0.0.0/0"]
    }
  }
  # without iterator, the variable is the label
  dynamic "egress" {
    for_each = var.sg_ports
    content {
      from_port   = egress.value
      to_port     = egress.value
      protocol    = "tcp"
      cidr_blocks = ["0.0.0.0/0"]
    }
  }
}
```

Remember!

`count` → `count.index`

`for_each`

— with a set → `each.key` == `each.value`
— with a map → `each.key` / `each.value`

`dynamic_bloc`

— by default: label of the dynamic block
— with an `iterator` : the iterator name

05. Advanced concepts

Functions

See: functions

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

Use `terraform console` to try functions!

Data sources

Data sources allow data to be fetched or computed for use elsewhere in Terraform configuration.

Example

```
data "aws_ami" "app_ami" {
  most_recent = true
  owners      = ["amazon"]
  filter {
    name   = "name"
    values = ["amzn2-ami-hvm*"]
  }
}
resource "aws_instance" "instance-1" {
  ami           = data.aws_ami.app_ami.id
  instance_type = "t2.micro"
}
```

Note: a lot of filters can be used. See ec2 instances

Data sources can be used to connect a state (with its outputs)

→ Note using terraform_remote_state is too permissive... prefer tfe_outputs which only gives access to output.

```
data "terraform_remote_state" "eip" {
  backend = "s3"
  config = {
    bucket = "kplabs-terraform-backend"
    key    = "network/eip.tfstate"
    region = "us-east-1"
  }
}
```

Dependency

Implicit

```
resource "aws_eip" "my_eip"{
  vpc = "true"
}
resource "aws_instance" "my_ec2" {
  instance_type = "t2.micro"
  public_ip     = aws_eip.myeip.private_ip
}
```

Explicit

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_s3_bucket" "example" {
  acl = "private"
}
resource "aws_instance" "myec2" {
  instance_type = "t2.micro"
  depends_on    = [aws_s3_bucket.example]
}
```

Update order

— Create resources that exist in the configuration but are not associated with a real infrastructure object in the state.
— Destroy resources that exist in the state but no longer exist in the configuration.
— Update in-place resources whose arguments have changed. → for instance EC2 tags
— Destroy and re-create resources whose arguments have changed but which cannot be updated in-place due to remote API limitations. → for instance EC2 AMI

Lifecycle meta argument

`create_before_destroy`

By default, resources are destroyed and then created.

With `create_before_destroy = true`, the new replacement object is created first, and the prior object is destroyed after the replacement is created.

→ Great to always have resources up on prod.

`prevent_destroy`

Prevents resources from being destroyed with `terraform destroy`

→ Great for databases, S3 buckets,... use `prevent_destroy = true` to enable

VERY IMPORTANT: if the entire resource definition is removed from the terraform file, IT WILL BE DESTROYED.

`ignore_changes`

Ignore certain changes to the live resource that does not match the configuration.

— `ignore_changes = [arg1, arg2]`
— `ignore_changes = all` → does what it say!

```
resource "aws_instance" "myec2" {
  ami           = "ami-00c39f71452c08778"
  instance_type = "t2.micro"
  tags = {
    Name = "Hello world"
  }
  # ignore changes on tags !
  lifecycle {
    ignore_changes = [tags]
  }
}
```

`replace_triggered_by`

Replaces the resource when any of the referenced items change

Taint

A resource can be taint. This means that it will be recreated during the next apply.

— Explicit taint: `terraform taint aws_instance.my_instance`
— Implicit taint: if a local exec or remote exec fails

`terraform taint` can also be used to taint resources within a module.

`terraform taint module.couchbase.aws_instance.my_instance`

06. Provisioners

Provisioners are used to execute scripts on a local or remote machine as part of resource creation or destruction.

Scripts can be executed on resource creation (default) or destruction.

Important: Use provisioners as a last resort. There are better alternatives for most situations.

See: Provisioners syntax

Remote exec

The `remote-exec` provisioner invokes a script on a remote resource after it is created.

The connection can use `ssh` or `WinRM`.

```
resource "aws_instance" "myec2" {
  ami           = "ami-0ca285d4c2cda3300"
  instance_type = "t2.micro"
  key_name      = "terraform-key"
  # default sg used here. Be sure that it allows port 80 and 22
  connection {
    type        = "ssh"
    user        = "ec2-user"
    private_key = file("./terraform-key.pem")
    host        = self.public_ip
  }
  provisioner "remote-exec" {
    inline = [
      # Updating with the latest command for Amazon Linux machine
      "sudo yum install -y nginx",
      "sudo systemctl start nginx"
    ]
  }
}
```

Local exec

The `local-exec` provisioner invokes a local executable after a resource is created.

```
resource "aws_instance" "myec2" {
  ami           = "ami-082b5a644766e0e6f"
  instance_type = "t2.micro"
  provisioner "local-exec" {
    command = "echo ${aws_instance.myec2.private_ip} >> ips.txt"
  }
}
```

→ With the `local-exec`, the attribute name is `command`, which is a `string`.

On destroy

```
# I agree, this example is a bit silly
provisioner "remote-exec" {
  when = destroy
  inline = [
    "sudo yum -y remove nano"
  ]
}
```

→ With the `remote-exec`, the attribute name is `inline`, which is a `list(string)`.

Failure management

By default, provisioners that fail will also cause the terraform apply itself to fail.

The `on_failure` setting can be used to change this. The allowed values are:

— `continue` Ignore the error and continue with creation or destruction.
— `fail` Raise an error and stop applying (the default behavior)

07. Modules

Way to organize source code, allowing reusability.

Module sources

See: Module Sources

```
# On the Terraform registry repo
module "myVpc1" {
  source  = "terraform-aws-modules/vpc/aws"
  version = "5.1.2"
}
```

```
# On a Git repository 'git::https://' or 'git::ssh://'
# 'ref' can reference a tag name, a hash or a branch
module "myVpc2" {
  source = "git::https://corporate-gitlab.com/tf-aws-vpc.git?ref=v1
   .2.0"
}
```

```
# Local module ALWAYS starts with './' or '../'
module "instance1" {
  source = "./module-ec2"
}
```

→ A lot of sources can be used, such as S3 (`s3::https://`), GCS buckets (`gcs::https://`), GitHub (`github.com/...`),...

Key points

A module can be instantiated several times, with a different module instance name

```
module "myVpc1" {
  source  = "terraform-aws-modules/vpc/aws"
  version = "5.1.2"
}
module "myVpc2" {
  source  = "terraform-aws-modules/vpc/aws"
  version = "5.1.2"
}
```

Parameters can be provided to a module instance. They are mapped to the module variables

```
module "myVpc1" {
  source  = "terraform-aws-modules/vpc/aws"
  version = "5.1.2"
  # input variables
  var1 = "val1"
  var2 = "val2"
}
```

A module can send informations back to its caller using `output`

```
# inside the module
output "foo_name" {
  description = "Name of ssth"
  value       = my_object.name
}
```

```
# inside the caller
module "module_instance1" {
  source = "../module/my_module"
}
# later, use module.module_instance1.foo_name
```

Standard module structure

See: Module structure

```
$ tree minimal-module/
.
+-- README.md
+-- main.tf
+-- variables.tf
+-- outputs.tf
```

```
$ tree complete-module/
.
+-- README.md
+-- main.tf
+-- variables.tf
+-- outputs.tf
+-- ...
+-- modules/
|   +-- nestedA/
|   |   +-- README.md
|   |   +-- variables.tf
|   |   +-- main.tf
|   |   +-- outputs.tf
|   +-- nestedB/
|   +-- .../
+-- examples/
|   +-- exampleA/
|   |   +-- main.tf
|   +-- exampleB/
|   +-- .../
```

Terraform registry

See: https://registry.terraform.io/browse/modules

Requirements for Publishing Module

— **GitHub:** The module must be on GitHub and must be a public repo. This is only a requirement

for the public registry.

— **Named:** Module repositories must use this three-part name format `terraform-<PROVIDER>-<NAME>`
— **Repository description:** The GitHub repository description is used to populate the short description of the module.
— **Standard module structure:** The module must adhere to the standard module structure.
— **x.y.z tags for releases:** The registry uses tags to identify module versions. Release tag names must be a semantic version, which can optionally be prefixed with a `v`. For example, `v1.0.4` or `0.9.2`

08. Workspaces

See: Workspaces

Workspaces allow multiple state files of a single configuration. Most of the time, each workspace has a different set of environment variables associated.

Not suitable for isolation for strong separation between workspace (stage/prod)

→ One codebase, several environments (dev1, dev2,...)

→ State File Directory: `terraform.tfstate.d/dev1` , `terraform.tfstate.d/dev2` ,...

Commands

- `terraform workspace new` Create a new workspace
- `terraform workspace list` List Workspaces
- `terraform workspace select` Select a workspace
- `terraform workspace show` Show the name of the current workspace
- `terraform workspace delete` Delete a workspace. The `default` workspace can NOT be deleted.

→ Variable `${terraform.workspace}`

09. Backends

A backend defines where Terraform stores its state data files (`terraform.tfstate`). By default, the local backend is used... but it does not allow collaboration between users. A few backends are available out of the box : S3, PostgreSQL, Consul, Kubernetes,...

The local backend stores state on the local filesystem, locks that state using system APIs, and performs operations locally. By default, Terraform uses the "local" backend.

When configuring a backend for the first time (moving from no defined backend to explicitly configuring one), Terraform will give you the option to migrate your state to the new backend.

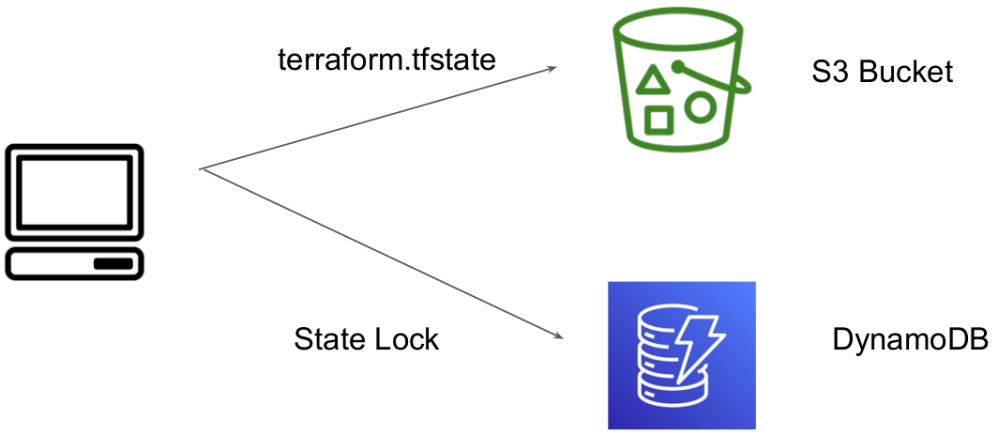
S3 backend

S3 backend is one of the most popular backends.

```
terraform {
  backend "s3" {
    bucket = "mybucket"
    key    = "path/to/my/key"
    region = "eu-west-1"
  }
}
```

Important

— By default, S3 does not support State Locking functionality.
— You need to make use of a DynamoDB table to achieve state locking functionality.



External backend configuration

```
terraform {
  backend "consul" {}
}
```

```
terraform init \
  -backend-config="address=demo.consul.io" \
  -backend-config="path=example_app/terraform_state" \
  -backend-config="scheme=https"
```

State locking

In real use cases, it is important to prevent concurrent updates on the state to prevent the file from becoming corrupted.

— If state locking fails, Terraform will not continue
— Not all backends support locking. The documentation for each backend includes details on whether it supports locking or not.
— Failure recovery (use it at your own risks!) `terraform force-unlock` manually unlock the state.

10. Import

BEFORE Terraform 1.5: the CLI

See: import CLI

The `terraform import` CLI command can only import resources into the state. Importing via the CLI does not generate configuration.

Before you run `terraform import` you must manually write a resource configuration block for the resource. The resource block describes where Terraform should map the imported object.

WITH Terraform 1.5.0: the import Block

See: import block

`terraform import` can create the `resource.tf` file from the remote infrastructure element to be imported !

import.tf

```
provider "aws" {
  region = "eu-west-1"
}
# After importing, you can optionally remove import blocks from your
   configuration
# or leave them as a record of the resource's origin.
import {
  to = aws_security_group.mysg
  id = "sg-07f13feb262ba8b6f"
}
```

Command To Autogenerate Code for Imported Resource

`terraform plan -generate-config-out=mysg.tf`

Note : `-generate-config-out` flag may change in future releases. See documentation.

Command To Generate the Final State file

`terraform apply -auto-approve`

11. Terraform Cloud

https://www.hashicorp.com/cloud

— Supported OS: everything except Unix and Android
— VCS providers: GitHub, BitBucket, Azure DevOps server, Azure DevOps services

Workspace management

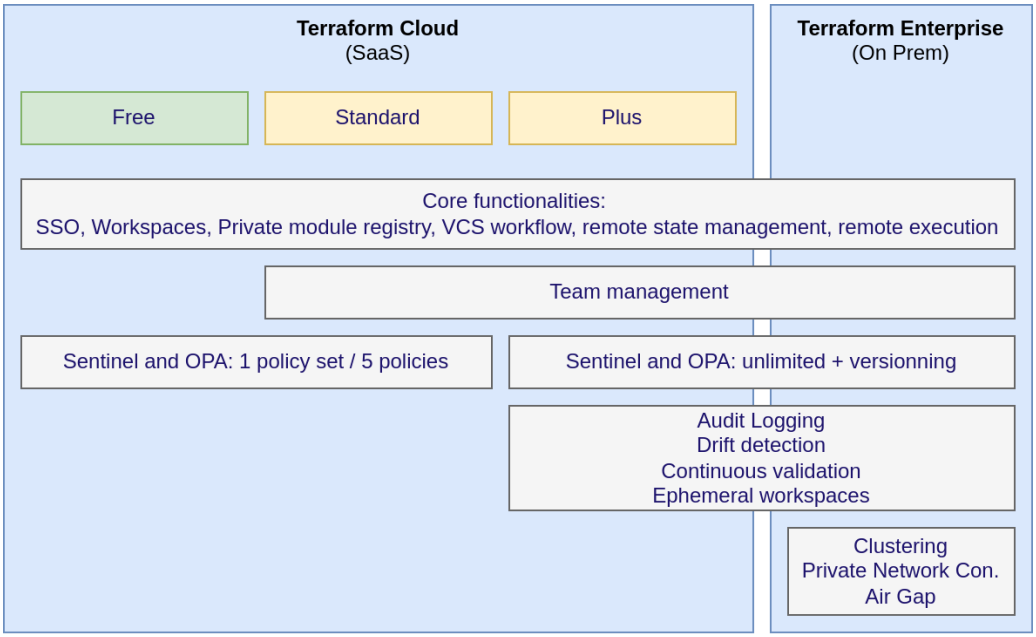
— NOT the same as the open source flavor
— Integration with GIT repositories
— Automatic plan on commit (and one click apply!)
— Variables management

Sentinel

Sentinel is an embedded policy-as-code framework

— Policy controls to make apply fail
— Only available with paid plans!

→ IMPORTANT: policies are only enforced at Terraform Cloud level... we still can do what we want on AWS directly...



NOTE: HashiCorp is shifting from a user based pricing to a consumption based pricing.
The first 500 resources are free. Then, pricing per hour per resource.

12. Ecosystem

Gitignore

https://github.com/github/gitignore/blob/main/Terraform.gitignore

— `terraform.tfstate` file can include sensitive information
— `*.tfvars` may contain sensitive data like passwords

Atlantis

https://www.runatlantis.io

Terragrunt

Wrapper arround terraform, in order to templetize `*.tf` files.

Misc

— terraform-docs generate documentation from Terraform modules in various output formats
— tfsec static analysis security scanner for Terraform code.
— checkov same as tfsec and can scan other type of sources
— infracost computes the cost impact of infrastructure changes before launching resources.

Tests

— Terratest
— Native Terraform Tests: requires Terraform v1.6+

13. Final notes

This documentation was created in order to prepare HashCorp Terraform certification-003. Knowing all this content should be enough in order to get certified (no guarantee of course, but I was certified in 2023).

TODO

— A dedicated section explaining state in details should be created based on https://developer.hashicorp.com/terr
— For expression should be explained inside **??** section
— **??** section is a bit weak, even if some knowlede is mandatory in order to be certified