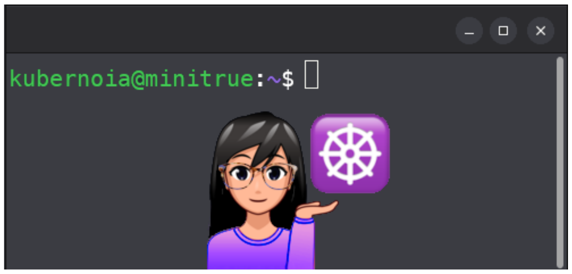
TERRAFORM INTRO

BY PAULOBA



```
kubernoia@minitrue:~$ 
```

Infrastructure as code

- More automation to reduce errors.
- Collaboration between developers and operations.
- More transparency.
- Traceability.
- Repeatable deployments.
- Agility.

Basic concepts

- All the `.tf` files in the directory are read.
- Virtually everything can be defined as a resource.
- The changes will be executed in parallel.
- Strings are the most complex and most widely used type of literal expression → recommended reading

Resources

Resources are the most important element in the Terraform language. Each resource block describes one or more infrastructure objects, such as virtual networks, instances, or DNS records.

- Resource blocks are used to declare resources.
- Resource behavior: applying a configuration with Terraform implies:
  - Create resources that exist in the config. and they are not associated with actual infrastructure in the state file.
  - Destroy resources that exist in the state file but no longer exist in the configuration.
  - Update (update in-place) resources whose arguments have changed.
  - Destroy and recreate resources whose arguments have changed and cannot be updated in-place due to API limitations.
- The meta-arguments section documents special arguments that can be used with each type of resource, including: depends_on, count, for_each, provider, lifecycle.
- Provisioners holds post-creation config. actions of a resource through the blocks
  - provisioner
  - connection

Providers

The providers are in charge of interacting with the APIs of the cloud providers GCP, AWS, etc.

```
provider "google" {
  version = "~> 2.1"
  region  = europe-west3
  project = my-test-project
}
```

Variables/inputs

For a terraform module to be reusable, it needs to be parameterizable. Variables are used for this.

```
variable "billing_account" {
  description = "The ID of the associated billing account"
  default   = "XXXXXXXXXX"
}
```

Outputs

Outputs are variables that a terraform module exposes to the user. Outputs are useful when we want to use the result of the execution of a terraform module as an input for another module.

Outputs are only rendered when Terraform applies your plan. Running `terraform plan` will not render outputs.

```
output "instance_ip_addr" {
  value = aws_instance.server.private_ip
}
```

Modules

Modules are self-contained terraform configurations that function as a reusable group.

State/backends

Terraform stores a deployment state in which it maps the actual objects to the resources defined in the configuration. By default the state is stored locally in the path where we have our configuration. For a deployment to be maintainable by a group of users, it is convenient to modify this behavior and have it be located in a shared remote storage. It is not recommended to keep tfstate in a local file stored in git repositories because `.tfstate` files contain sensitive information about the infrastructure managed by Terraform (secrets, passwords), including the current state of resources and the configurations used to create them.

You are likely to forget to commit and push your changes after running `terraform apply`, so your teammates will have out-of-date `.tfstate` files. Also, without any locking on these state files, if two team members run Terraform at the same time on the same `.tfstate` files, you may overwrite each other's changes.

To have a disaster recovery plan you can create a storage bucket with object versioning active that allows to have more than one version of a file stored so it's possible to recover older versions of the files in case of need.

The steps to configure a storage bucket as a backend for terraform are:

1. Add the code that deploys the storage bucket to the `main.tf`
   The code below deploys a multi-region bucket, standard type in Europe, that allows having two versions of the same file.

   To store more versions in parallel adjust the parameter `num_newer_versions = x`

```
resource "google_storage_bucket" "terraform-state" {
  project    = "my-cloud-project"
  name       = "terraform-state"
  location   = "EU"
  versioning {
      enabled = true
  }
  lifecycle_rule {
      condition {
      num_newer_versions = 2
      }
      action {
      type = "Delete"
      }
 }
}
```

2. Create the bucket

```
$ terraform apply -target=google_storage_bucket.terraform-state
```

3. Create the tfstate.tf file with the following code

```
terraform {
backend "gcs" {
  bucket = "terraform-state"
  prefix = /dev/terraform/state"
  }
}
```

4. Apply the config. above running

```
$ terraform apply -target=backend.gcs
```

Configuring a storage bucket as a terraform state backend means that when you run terraform apply the tfstate is updated in the bucket, that means that the local tfstate will be empty.
In case you want to edit the remote tfstate you can run:

```
$ terraform state pull > terraform.tfstate
$ terraform state push terraform.tfstate
```

Plan

The `terraform plan` command shows the actions that terraform will perform over the infrastructure without applying any real change. This command acts as a dry-run command to perform before running `terraform apply` in order to analyze the output.

CLI

General commands

```
$ terraform init
```

Initializes a working directory containing Terraform configuration files. This is the first command that should be run after writing a new Terraform configuration or cloning an existing one from version control. It is safe to run this command multiple times.

```
$ terraform plan
```

Creates an execution plan that lets you preview the changes that Terraform plans to make to your infra. You can use the optional -out=FILE option to save the generated plan to a file on disk, which you can later execute by passing the file to terraform apply as an extra argument. This two-step workflow is primarily intended for when running Terraform in automation.

```
$ terraform apply
```

Executes the actions proposed in a terraform plan.

`Terraform logging`

In case you want to have extra verbosity in the output you can add the following variables to you `~/.bashrc` or `~/.zshrc`

Bash: `export TF_LOG="DEBUG"`

PowerShell: `$env:TF_LOG="DEBUG"`

Logging can be enabled separately for terraform itself and the provider plugins using the `TF_LOG_CORE` or `TF_LOG_PROVIDER` environment variables. These take the same level arguments as `TF_LOG`, but only activate a subset of the logs.

To persist logged output you can set `TF_LOG_PATH` in order to force the log to always be appended to a specific file when logging is enabled. Note that even when `TF_LOG_PATH` is set, `TF_LOG` must be set in order for any logging to be enabled.You can set `TF_LOG` to one of the log levels (in order of decreasing verbosity) `TRACE`, `DEBUG`, `INFO`, `WARN` or `ERROR` to change the verbosity of the logs.

Bash: `export TF_LOG="DEBUG"`

PowerShell: `$env:TF_LOG="DEBUG"`

Set the Terraform log location in your environment with the appropriate command (substituting the path to your preferred file):

Bash: `export TF_LOG_PATH="tmp/terraform.log"`

PowerShell: `$env:TF_LOG_PATH="C:\tmp\terraform.log"`

```
$ terraform apply -target module.foo
```

Executes the actions that relate only to the terraform module `module.foo`.

```
$ terraform force-unlock <lock-id-guid>
```

Deletes locks generated on the state for the current configuration, that may exist by a canceled execution or by two people trying to run terraform at the same time on the same state file.

```
$ terraform get -update
```

When used with the -update flag downloads updates, for example if there are references to terraform modules it downloads new versions of that module.

This command is useful, when for example you update the code in a terraform module, and you are calling that terraform module from the code you want to apply.

```
$ terraform fmt [options] [target...]
```

Rewrites the code in the configuration files with a canonical formal style.

This command applies a subset of terraform style conventions and other minor adjustments for better legibility.

Formatting decisions are always subjective and so you might disagree with the decisions that terraform fmt makes. This command is intentionally opinionated and has no customization options because its primary goal is to encourage consistency of style between different Terraform codebases, even though the chosen style can never be everyone's favorite.

Commands related with the state file

```
$ terraform state rm "module.foo.google_bigquery_dataset.bar"
```

The command above deletes from the `.tfstate` file the BigQuery dataset called `bar` that is created by the module `foo`.

To delete elements with indexes, in Unix-like systems it's needed to add single quotes around the element to delete as explained in this github issue at the time of writing this.

Example:

```
terraform state rm 'google_project_iam_member.gke-cluster["roles/storage.objectViewer"]'
```

```
$ terraform state pull > terraform.tfstate
```

Downloads the remote state file to a local file called `terraform.tfstate`

```
$ terraform state push terraform.tfstate
```

It's used to manually upload the contents of a local state file to the remote state file, it's useful when you are manually editing in local and you want to update the remote state that is stored in a remote storage.

```
$ terraform import
```

Imports infrastructure resources that exist in a project into the terraform state file. It can be useful when you create a resource manually via the cloud provider API and then you want to add this resource to the state file. Note that importing a resource into the state requires you to add the terraform code into the terraform script to avoid inconsistencies the next time you run `terraform apply`.

Example 0 - terraform import:

```
$ terraform import aws_instance.foo i-abcd1234
```

imports an aws instance in the resource called foo

Example 1 - terraform import:

```
# LIST PACKAGES DEPLOYED WITH HELM

# SEARCH FOR THE ONE YOU WANT TO IMPORT INTO THE tfstate

# CHECK IN WHAT HELM NAMESPACE IT'S DEPLOYED

$ helm list | grep my-app

NAME   NAMESPACE  REVISION  UPDATED     STATUS    CHART        APP VERSION
my-app default    5         2023-03-09  deployed  my-app-3.6   3.6.0

# RUN TERRAFORM IMPORT COMMAND
# $ terraform import
module.[MODULE-NAME].helm_release.[COMPONENT-NAME]
[HELM-NAMESPACE]/[HELM-DEPLOYMENT-NAME]

$ terraform import module.my-module.helm_release.my-app

default/my-namespace
```

Post on how to push tfstate to a bucket→https://www.scalr.com/blog/terraform-state-update

Example 2 - terraform import:

```
$ terraform import

module.foo.module.redis-memorystore.google_redis_instance.redis_instance[0]

projects/my-project-bar/locations/us-central1/instances/redis-memorystore-name
```

Example 3 - terraform import:

Example import from a resource created within a `for_each` loop.

Download the remote Terraform state to a local file called state:

```
$ terraform state pull > state
```

Edit the local state, then search for the resource to import and create a skeleton with minimum fields like name, id. The rest of the fields will be populated later when running terraform import.

Example: import a `pub/sub` topic created with a `for_each` loop.

The state code (without the resource we want to import) looks like this:

Before importing, create a skeleton for the resource to be imported later.

Next, upload the local state to the remote bucket, running:

```
$ terraform state push -force state
```

The next step is to run terraform import.

Usage: terraform [global options] import [options] ADDR ID

- The resource ADDR is the resource name given by the provider (e.g. module.modulename.google_pubsub_topic.modulename-svcs-communication)
- The resource ID is available within the output of a terraform plan

```
$ terraform import module.my-module-name.google_pubsub_topic.my-topic
projects/my-project-name/topics/my-topic-name
```

Best practices

- Add a .gitignore file to avoid pushing by mistake all the local content from .terraform
- Best practices to use Terraform.

  Source → *https://cloud.google.com/docs/terraform/best-practices-for-terraform*

  - Follow a standard module structure
  - Adopt a naming convention
  - Use variables carefully
  - Expose outputs
  - Use data sources

Tips

- tfswitch it's a CLI tool that allows install Terraform versions in a quick way and change between the Terraform binary versions just by running the command

  ```
  $ tfswitch
  ```

- terraform-docs allows you to generate documentation for your terraform code in an automatic way; the docu integrates easily with GitHub actions and it's generated in a wide variety of formats: `markdown, AsciiDoc, JSON, tfvars hcl, tfvars json, toml, xml, yaml`.

  Example:

  ```
  $ terraform-docs markdown table
  "/home/user/src/my-terraform-module" --output-file "README.md"
  ```