clearpoint

Introduction to Terraform

24th November 2022

EROAD

Introduction

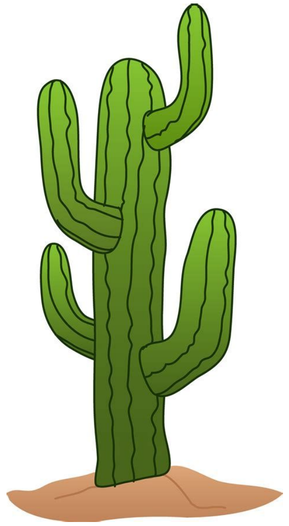
HashiCorp Terraform

- The history of infrastructure-as-code
- What is Terraform specifically
- Alternatives to Terraform
- Core concepts: providers, resources, backends, state
- Modules, variables, outputs, versioning
- CI/CD and development workflows
- Code repository structures
- Layering and the separation of responsibilities

Infrastructure as code

Back in the early wild-west days of cloud adoption we used to (attempt to) manage resources by-hand, but even the most meticulous of us are capable of making mistakes. Defining your infrastructure as code allows us to leverage the true potential of the cloud.

- Define it once, re-use it infinite times
- Review in one place
- Infrastructure code *becomes* your documentation
- Other teams and projects can benefit from your modular code
- It can even be versioned!



What is Terraform, and why would we use it?

Terraform is a platform-agnostic open source infrastructure-as-code tool.

It allows us to define the desired state for resources using a simple structured language, and will ensure that this state is maintained (by tracking changes).

- It is not bound to any specific cloud provider - GCP, Azure, AWS, Digital ocean, all sorts of things are supported
- It is the industry standard, and there are already many people who know it well and use it day-to-day
- It can run anywhere (not only SaaS)
- It lends itself well to modular design and reusability
- It is flexible enough to be of use for nearly any purpose

Alternatives

There are a number of other ways to do infrastructure-as-code:

- CloudFormation: AWS's native approach
- Bicep / ARM: Azure's native approach
- Pulumi: IaC integrated with actual programming languages
- CDK: IaC "Cloud Development Kit" for AWS. Similar to Pulumi, AWS specific

However in heterogenous / hybrid environments Terraform makes a lot of sense because the deployment pipelines and automation around IaC can be handled the same way.

In practise the simple declarative language "HCL" can often be more accessible to infrastructure and platform teams, who aren't necessarily software developers.

The basic workflow: code, plan, apply

The first basic workflow is really just a bunch of files on your local machine, and the Terraform binary. It will look like this:

01

Write some code

02

Run the `terraform plan` command

- Terraform will analyse your code to build a model of what you're asking for
- It will then interrogate the target environment to see if anything has changed since last time you ran Terraform
- If all looks good it will report back a detailed list of resources that need to be created / modified / removed

03

Run the `terraform apply` command

- Terraform will run the plan again (unless you give it a pre-made plan)
- After prompting to make sure you're OK it will attempt to make the changes detailed in the plan
- A state file will be written (or updated), which contains the detailed metadata of the resources controlled by your code

Concepts

Providers, Resources, Modules, Workspace, States and Backends

Providers

Terraform itself is modular: the core logic is responsible for parsing HCL code and managing dependencies etc, but the task of actually interacting with SaaS / PaaS / third parties is given over to a pluggable set of providers.

Hashicorp themselves support many (most) of the ones that you will ever need, but there are also many third party offerings, and you can even write your own. A public registry exists where you can search for providers and read the documentation for each.

Providers can be configured in-code, but most of them are able to be configured using environment variables too (such as the AWS keypairs which we wouldn't want to have in our code repositories).

Resources

Resources represent actual bits of infrastructure that will be provisioned by the providers, and are the smallest practical unit in Terraform.

- Resources belong to a provider
- As modules have input variables, resources also have required and optional *arguments* that determine the behaviour and configuration of the infrastructure created
- Also, as modules have outputs, resources have *attributes* which can be referenced to look up the details of the infrastructure that was created
- Dependencies between resources is automatically calculated by Terraform, which will ensure that resources are created in the correct order

Modules

A module is a self-contained collection of Terraform code which can be used from elsewhere (either as a “root” entrypoint, or a sub / sub-sub / sub-sub-sub module of one).

Modules generally contain the following:

- Variables: the input parameters to configure the modules behaviour
- Resources: the infrastructure that will be provisioned by the module
- Outputs: useful information that is offered to the caller (IDs of resources etc)

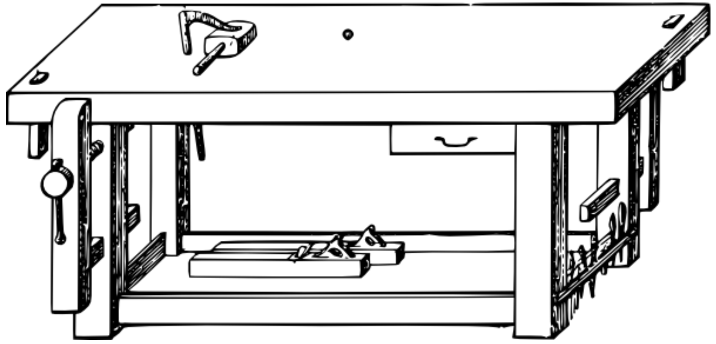
A module could be thought of as a *function* in a programming language. If we are doing well then our modules are clean, well documented, consistent with the other modules in our code-bases, and are generic enough to be of use elsewhere.

Workspace

This is where we ask Terraform to run against a module, with a given set of input variables, configuration for providers, and a backend for state management. This entrypoint module is sometimes called a "root" module.

Workspaces are useful in the following ways:

- Allows sets of infrastructure to be maintained by different groups of people
- Breaks up large deployments into chunks that run quickly and minimises risk
- Outputs (and state) from one workspace can become inputs to another



State

This is one of the potentially most troubling aspects of Terraform, and one that certainly needs to be well understood to operate safely at scale in a production environment.

As mentioned elsewhere, each TF “workspace” has an associated state file which keeps track of the detailed metadata of provisioned resources.

The state is read each time a plan is run, and compared to what *actually* exists in the target infrastructure.

In this way Terraform can alert us to things that may have been changed through some other means, and knows what needs to be done to bring things back into line.

State is absolutely crucial to a Terraform workspace, and must be safeguarded in a production environment (against accidental loss, as well as concurrent access).

Concept: Backends

As mentioned previously, TF state is important data that needs to be looked after. Furthermore once we start running Terraform from different locations we also need to safeguard against parallel execution, and provide distributed access to it.

This is where pluggable backends come in.

- In its most basic (default) form, state is stored in a json file in the current dir
- Various forms of remote-state is also supported (S3 etc), which solves the *distribution* and *backup* problems
- Locking is also supported, for example using DynamoDB (which protects against concurrent access)

Patterns

Layering and separation of responsibilities

In larger / more complex production environments there will be different teams managing different parts of the infrastructure (eg security / networking / platforms / DBs / applications). Take this example:

- **Platform:** networking and Kubernetes
- **Application:** databases and message queues

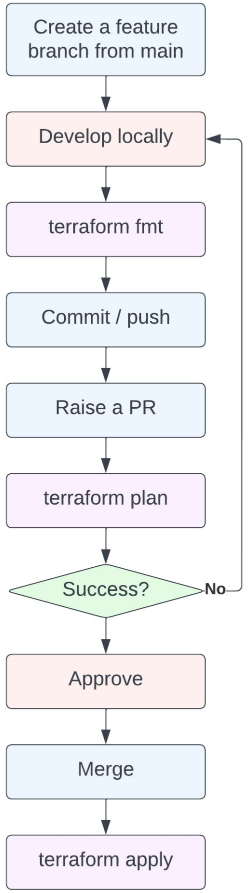
If the platform team manages their resources in one workspace then the application team can build on top of this in a different one (without running the risk of being able to accidentally modify it).

Various solutions exist for exposing the metadata from one workspace to another.

Best practice CI/CD and development workflows

An example of trunk-based development with Terraform:

- PRs can be linted with \`terraform fmt\`, tested with \`terraform plan\`
- In production the main branch should be protected so only code-owners can merge PRs (once approved)
- Applies should only be run from the main / trunk branch (otherwise someone can easily end up having their resources automatically deleted)
- If applies are automated then you MUST use some form of locking for your state files (to avoid concurrent execution)



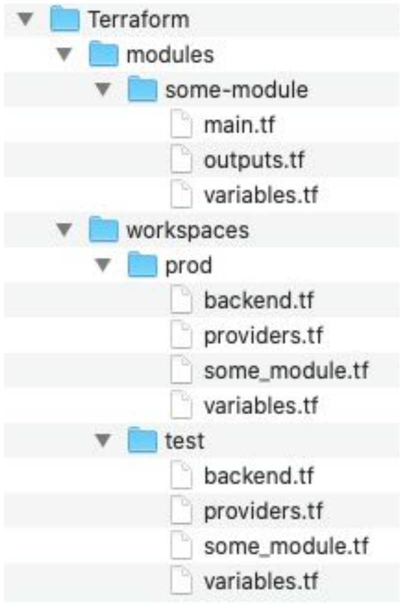
Code repo structure

This is an example of a monorepo for Terraform code:

- Modules and workspaces exist in one repo (this allows local / relative imports)
- Modules can be used by any workspace
- Even better, keep your application code in the same repo and version the infrastructure alongside it!

What about versioning your modules?

This is a big subject, and gladly there are several approaches - the correct one depends on each situation. Further reading is advised.



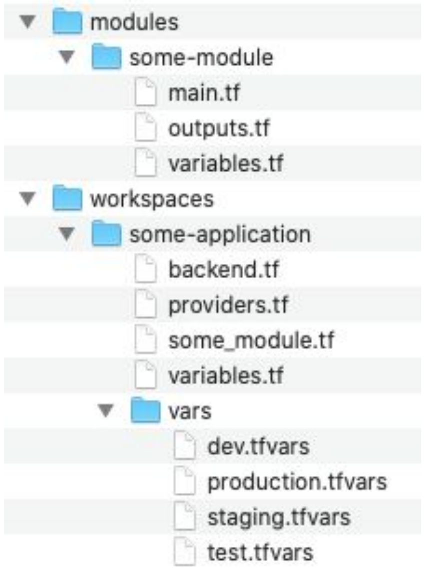
```
▼ Terraform
    ▼ modules
        ▼ some-module
              main.tf
              outputs.tf
              variables.tf
    ▼ workspaces
        ▼ prod
              backend.tf
              providers.tf
              some_module.tf
              variables.tf
        ▼ test
              backend.tf
              providers.tf
              some_module.tf
              variables.tf
```

Another way - tfvars files

In the previous monorepo example we had a directory for each workspace, but that isn't the only way to realise this idea of "one root module being applied to several workspaces".

Terraform allows us to specify variables in another format called tfvars. We then tell Terraform which particular tfvars file to load values from when we execute.

This allows us to cut down on the amount of *boilerplate* code required by the statically-defined workspaces in the previous example: we just provide these once, and apply a different set of variables to them each time.



```
modules
    some-module
        main.tf
        outputs.tf
        variables.tf
workspaces
    some-application
        backend.tf
        providers.tf
        some_module.tf
        variables.tf
        vars
            dev.tfvars
            production.tfvars
            staging.tfvars
            test.tfvars
```

Terraform Cloud

HashiCorp Terraform Cloud

Running your own Terraform CI/CD pipelines (well) is not easy, and will be a full time job for somebody (or a team of somebodies) at sufficient scale. Hashicorp have solved this problem for us with Terraform Cloud, which is a specifically designed SaaS tool for Terraform.

- Integrates with your VCS (GitHub, BitBucket etc)
- Secure secret storage
- Multiple workspaces
- Backend and locking for state
- Automated plans (and applies if you like)
- SSO and group/team support
- The basic tier is free!

What is next?

Having covered some of the basic concepts and patterns, the next thing is to go try it out!

Next we will build on what we've discussed here with some practical exercises designed to demonstrate some of these ideas in action.

clearpoint.

Thank you.