Ranjana Jha Follow
Oct 27, 2021 · 6 min read · Listen

Infrastructure as a code best practices : Terraform

In this post we will go over what a typical infrastructure would look like for an enterprise product and the best practices for the same.

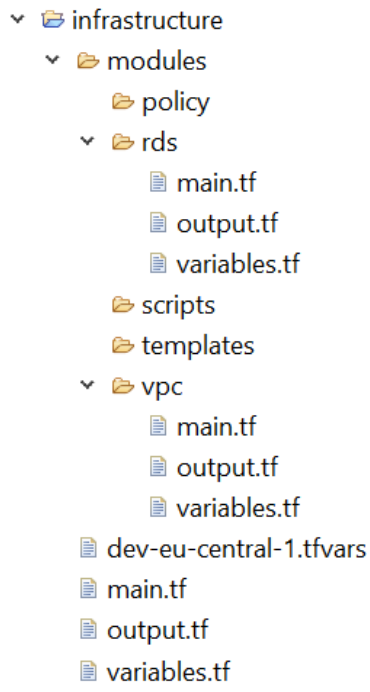
1. High level infrastructure overview :

A typical infrastructure of a product can be categorized into the following three categories :

- *Global infra :* This infrastructure contains the components which acts as the foundation of the overall infra across the product. The infra at this layer should mostly consist of setting up the networking — vpc ,subnets ,acl etc , dns and so on.
- *Common Infra :* This layer consists of the infra which is needed by more than one service. Examples would be Redis Cluster , EKS Cluster etc.
- *Service Infra :* This layer consists of infra which is service specific and should reside alongside the code repository of the service itself. Example: rds for a service would reside alongside the code of the service.

2. Consistent code structure :

All the terraform code written should follow a consistent code structure. Below can be an example :



iaac code structure

- *modules :* This folder should contain terraform code for resource creation . Examples : vpc,rds,subnets etc. Creating modules promotes reusability, hence reducing code duplication. Also each of the sub-folders/resources in modules should contain a structure like : main.tf , variables.tf , output.tf
- *policy :* This folder should contain policy documents such as IAM role policies as json files .So this folder should act as a collection of policies which would be used by the modules folder. Examples : rds_iam_role_polcy.json , etc.
- *scripts :* The folder should contain any scripts such as shells scripts or python scripts used for any resource handling or creation . Hence it acts a common place of any kind of script we write for our infra.
- *templates :* In terraform we can use “.tpl” files for various purpose , in such a case we can keep those files in this folder for clarity.
- *main.tf :* This file acts the entry point when we call terraform commands like init, validate , plan ,deploy and destroy.
- *output.tf :* The outputs which need to be written to the state should be present here.

For dev and region — eu-central-1 , the expected file name should be : dev-eu-central-1.tfvars .

For int and region — eu-central-1 , the expected file name should be : int-eu-central-1.tfvars and so on.

The files contain initialized values for the variables declared in variables.tf file.

3. Use terraform modules :

Use of terraform modules wherever possible should be done and such modules should be kept in a common place in gitlab/code repository . Whenever we need a new resource for a service, a repo for such a resource should be created containing the terraform code. Such modules can then be directly referred by the service infra code . In case another service needs the same resource in future , we don't need to duplicate the terraform code and can use the tried and tested version present .

4. Consistent Naming convention :

Terraform recommends the following naming conventions :

- *General :* Use `_` (underscore) instead of `-` (dash) in all: resource names, data source names, variable names, outputs. Only use lowercase letters and numbers.
- *Resources :* Do not repeat resource type in resource name (not partially, nor completely):

Good: `resource "aws_route_table" "public" {}`

Bad: `resource "aws_route_table" "public_route_table" {}`

Bad: `resource "aws_route_table" "public_aws_route_table" {}`

Use " `-` " inside arguments values and in places where value will be exposed to a human (eg, inside DNS name of RDS instance).

use "count" as the first argument in a resource

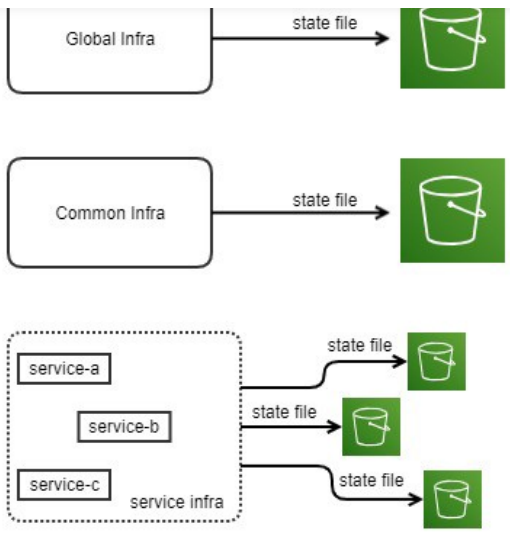
use "tag" as the last argument followed by "depends" and "lifecycle" .

- *Variables :* Use plural form in name of variables of type list and map. When defining variables order the keys as : description , type, default . Always include description for all variables even if you think it is obvious.
- *outputs.tf :* The general recommendation for the names of outputs is that it should be descriptive for the value it contains and be less free-form than you would normally want. If the returned value is a list it should have plural name. Always include `description` for all outputs even if you think it is obvious.

Naming conventions
Edit description
www.terraform-best-practices.com

5. Organizing state file :

The state should be stored remotely in an s3 bucket and each of the infra layers : Global, Common and Service should have their own state file. Further more , each of the services in the service layer should have their own state file as well. Organizing state files like this ensure loose coupling. Each service per env should have its own state file . Following shows how the state file should be organized.



Organizing State File

6. Use an automated testing framework to write unit and functional tests that validate your terraform modules :

Automated testing is every bit as important for writing infrastructure code as it is for writing application code. Example of such a framework is terratest.

Terratest | Automated tests for your infrastructure code.

Test infrastructure code with Terratest in 4 steps Create a file ending in Write test code using Go _test.go and run...

terratest.gruntwork.io

7. Infrastructure service names :

A typical infrastructure created using terraform can follow the below naming style for consistency purpose :

For a single resource :
infra_name = ${name}-${resource_type}
name = ${env}-${service_name}

Examples :
env = dev , service_name = customer-svc , resource_type = db-cluster
rds_cluster_name : dev-customer-svc-db-cluster

For multiple instance of a resource :
infra_name = ${name}-${resource_type}-${count.index}
name = ${env}-${service_name}

Examples :
env = dev , service_name = customer-svc , resource_type = db-cluster-instance , count.index = 0
rds_instance_name : dev-customer-svc-db-cluster-instance-0

env = dev , service_name = customer-svc , resource_type = db-cluster-instance , count.index = 1
rds_instance_name : dev-customer-svc-db-cluster-instance-1

The reasons for the variables used are :

env : we want env to be a part of the infra name so that we can use a single account for creating multiple envs. i.e in case we want to have dev and test env resources within a single aws account the value of this variable we help us distinguish the resources of different env.

service_name : we want the service which owns the resource in the infra name because it becomes easier to search/find resource by service names when there are lot of similar resources . For example if we have lots of queues in our account and if we want to look for a particular queue and we know the service which owns the queue we can search that queue by just typing the service name.

Name = “The actual resource name . This should be the ${infra_name} stated above. The key “Name” is case sensitive. In some of the resources this will help to give the resource a name in the aws console . Example in case of subnets or security groups.”
environment = “The env like : dev , int, prod. Useful in case we have multiple env in one account.”
owner-service = “The service which uses this resource and is the owner of the resource. Useful to list resources used by a particular service.”
infra-region = “The region of the resource.”
team-name = “ The team name who is the owner for the resource.”
time-zone = “ The primary time of the operating resource.”
start-time = “The start time of the resource.”
stop-time = “The stop time of the resource. The start and stop times helps a scheduler script know when to start and stop a resource.”
contact = “The team email to contact in case of any queries related to the resource.”

These key-value pairs can be utilized for billing, ownership, automation, access control, and many other use cases.

———————————————————————————————————