**How to Create Terraform Multiple Environments**

Don't repeat yourself (DRY) with Terraform to create your infrastructure in multiple environments/regions/cloud-providers

Table of Contents

1. [2 Strategies to Create Multiple Environments](https://getbetterdevops.io/terraform-create-infrastructure-in-multiple-environments/#2-strategies-to-create-multiple-environments)
   1. [1# Separated Directories](https://getbetterdevops.io/terraform-create-infrastructure-in-multiple-environments/#1-separated-directories)
   2. [2# Workspaces](https://getbetterdevops.io/terraform-create-infrastructure-in-multiple-environments/#2-workspaces)
2. [Comparison of the Strategies](https://getbetterdevops.io/terraform-create-infrastructure-in-multiple-environments/#comparison-of-the-strategies)
   1. [Separated directories](https://getbetterdevops.io/terraform-create-infrastructure-in-multiple-environments/#separated-directories)
   2. [Workspaces](https://getbetterdevops.io/terraform-create-infrastructure-in-multiple-environments/#workspaces)
3. [Conclusion](https://getbetterdevops.io/terraform-create-infrastructure-in-multiple-environments/#conclusion)

Terraform has simplified the way we provision infrastructure in the cloud and manage it as code. But best practices like separating infrastructure into multiple environments (*staging/QA/production*) don't change. Maybe even depending on your business needs, you need to extend the infrastructure across multiple geographical areas. Maybe you are even thinking about adopting a multi-cloud strategy.

If you are in this situation this means that you need to be able to create multiple environments in your code. The challenge here is to factorize the code as much as possible to the DRY (*Don't Repeat Yourself*) principle. There are many ways of achieving it using Terraform.

In this article, we will see two strategies for doing this with Terraform. Each has its strengths and weaknesses and we'll compare them at the end. Let's go!

**2 Strategies to Create Multiple Environments**

In both presented strategies, we're using modules included in projects for convenience. They can be versioned in distinct GIT repositories. Each module is called in a specific layer and terraform remote states are stored in a versioned S3 backend. Separated layers improve consistency and make easier rollback.

**1# Separated Directories**

Diagram, schematic

Description automatically generated

Project hierarchy to manage multiple environments with separated directories

In this project structure, each environment has its directory that contains layers. You can see below the network configuration of the staging environment :

terraform {

backend "s3" {

bucket = "terraform-remote-states"

# The key definition changes following the environment

key = "environments/staging/network.tf"

region = "us-east-1"

}

}

module "network" {

source = "../modules/network"

region = "us-east-1"

network\_cidr = "10.0.0.0/16"

private\_subnet\_cidrs = ["10.0.0.0/24", "10.0.1.0/24", "10.0.2.0/24"]

public\_subnet\_cidrs = ["10.0.3.0/24", "10.0.4.0/24", "10.0.5.9/24"]

}

As mentioned, we use a backend to persist remote states. The key definition, i.e. where the remote state is stored, varies according to the environment and the layer. In the module block, the different values can differ following the environment.

Dependencies between layers can be solved with [data sources](https://www.terraform.io/docs/language/data-sources/index.html). The existing resources in the cloud or outputs from the remote states can be fetched :

data "terraform\_remote\_state" "network" {

backend = "s3"

config = {

bucket = "terraform-remote-states"

key = "environments/staging/network.tf"

region = "us-east-1"

}

}

locals {

vpc\_id = data.terraform\_remote\_state.network.outputs.vpc\_id

}

For deployment, you have to perform a terraform init and apply on the layers in order :

$ terraform -chdir="./environments/staging/network" init

$ terraform -chdir="./environments/staging/network" apply

$ terraform -chdir="./environments/staging/database" init

$ terraform -chdir="./environments/staging/database" apply

$ terraform -chdir="./environments/staging/application" init

$ terraform -chdir="./environments/staging/application" apply

Initialization and deployment of the staging environment

**2# Workspaces**

Diagram

Description automatically generated

Project hierarchy to manage multiple environments with workspaces

Before we use an S3 backend to store remote states. Initially, there is only one workspace named default and only one associated with the state. Some backends as S3 support multiple workspaces. They allow multiple states to be associated with a single Terraform configuration.

We are going to use this feature to define multiple environments. Each environment will be a workspace. In the backend definition, we add the workspace\_key\_prefix parameter. It is specific to the S3 backend and it will define the S3 state path as /workspace\_key/workspace\_key\_prefix/workspace\_name :

terraform {

backend "s3" {

bucket = "terraform-remote-states"

workspace\_key\_prefix = "environments"

key = "network"

region = "us-east-1"

}

}

The remote states will look as follow in S3 :

Diagram

Description automatically generated

Organization of Terraform remote states in S3 using workspaces

The network layer stays the same as before :

terraform {

backend "s3" {

bucket = "terraform-remote-states"

# The key definition changes following the environment

key = "environments/staging/network.tf"

region = "us-east-1"

}

}

module "network" {

source = "../modules/network"

region = "us-east-1"

network\_cidr = "10.0.0.0/16"

private\_subnet\_cidrs = ["10.0.0.0/24", "10.0.1.0/24", "10.0.2.0/24"]

public\_subnet\_cidrs = ["10.0.3.0/24", "10.0.4.0/24", "10.0.5.9/24"]

}

The Terraform command manages the workspace. The first step is to create the new workspace:

$ terraform -chdir="./network" workspace new staging

Creation of the staging workspace

Then you have to select it :

$ terraform -chdir="./network" workspace select staging

Selection of the staging workspace

The vars directory presented in the diagram above contains variable files to customize the environment. You have to load the good one when you apply the configuration :

$ terraform init -chdir="./network"

$ terraform apply -chdir="./network" -var-file="./vars/staging.tfvars"

Another way is to use locals instead and play with terraform.workspace :

terraform {

backend "s3" {

bucket = "terraform-remote-states"

workspace\_key\_prefix = "environments"

key = "network"

region = "us-east-1"

}

}

variable "network\_cidr" {

type = list(string)

default = {

staging = "10.0.0.0/16"

qa = "10.1.0.0/16"

production = "10.2.0.0/16"

}

}

variable "private\_subnet\_cidrs" {

type = list(string)

default = {

staging = ["10.0.0.0/24", "10.0.1.0/24", "10.0.2.0/24"]

qa = ["10.1.0.0/24", "10.1.1.0/24", "10.1.2.0/24"]

production = ["10.2.0.0/24", "10.2.1.0/24", "10.2.2.0/24"]

}

}

variable "public\_subnet\_cidrs" {

type = list(string)

default = {

staging = ["10.0.3.0/24", "10.0.4.0/24", "10.0.5.0/24"]

qa = ["10.1.3.0/24", "10.1.4.0/24", "10.1.5.0/24"]

production = ["10.2.3.0/24", "10.2.4.0/24", "10.2.5.0/24"]

}

}

locals {

network\_cidr = lookup(var.network\_cidr, terraform.workspace, null)

private\_subnet\_cidrs = lookup(var.private\_subnet\_cidrs, terraform.workspace, null)

public\_subnet\_cidrs = lookup(var.public\_subnet\_cidrs, terraform.workspace, null)

}

module "network" {

source = "../modules/network"

region = "us-east-1"

network\_cidr = local.network\_cidr

private\_subnet\_cidrs = local.private\_subnet\_cidrs

public\_subnet\_cidrs = local.public\_subnet\_cidrs

}

**Comparison of the Strategies**

**Separated directories**

Pros:

* Environments are separated and identifiable
* More granularity: you can customize environment layers
* Less chance of applying a configuration in a bad environment

Cons:

* Need to duplicate a piece of file structure to create a new environment
* Several directory levels in the project

**Workspaces**

Pros:

* Scalability with repeatable environments
* Simplicity

Cons:

* More possibility to make errors by selecting a wrong workspace
* The customization of an environment layer is less obvious

**Conclusion**

There is no single solution for managing multiple environments in a Terraform project. The two approaches we have discussed both have their qualities and their shortcomings. It depends on your project expectations.

Do you need to scale quickly or is the pace of environment creation more extended in time? Do you want to have file isolation between environments or rely on the workspace abstraction mechanism?

In any case, you can compensate for the shortcomings of this choice with other solutions. By using a continuous deployment pipeline you can greatly reduce the error rate of selecting the wrong workspace. The creation of a new environment in a file structure can be generated on the fly using templating.

Also, the two methods we have seen are not exhaustive. You can take inspiration from them and create a new one that best fits your use case.