Using Terraform Modules for reusable and maintainable Infrastructure

Code reuse and avoiding code duplication

Vidhya Chari
Feb 18, 2020 · 3 min read ★

Image by pxfuel.

Why Terraform modules ?

We, the DevOps team at Boxed, create identical copies of our cloud infrastructure for different environments, staging vs production. The challenge was to

be able to pass different parameters/attributes to various infrastructure resources for staging vs production. As our infrastructure grew things became more complicated and lead to code duplication. This affected our ability to maintain/extend our codebase. We wanted to avoid having to copy/paste every piece of infrastructure for all our environments.

Solution → **Terraform modules** !

How ?

The terraform configuration files that contain the resources definitions are packaged as a terraform module. The modules live inside a common folder that can be accessed by all folders/projects. This lets us write reusable and maintainable infrastructure code. We maintain multiple modules that are reused for all our environments.

Terraform modules in Action

Following is the file layout of our Terraform project:

```
terraform-infrastructure
  -- staging
     |-- main.tf
     |-- provider.tf
     |-- backend.tf
     |-- variables.tf
  -- production
     |-- main.tf
     |-- provider.tf
     |-- backend.tf
     |-- variables.tf
  -- terraform
     -- modules
        |-- common-infrastructure
           |-- gcloud_cdn.tf
           |-- gcloud_memorystore.tf
           |-- gcloud_bucket.tf
           |-- gcloud_sql.tf
           |-- variables.tf
```

As you can see above, we have separate folders for ***staging*** and ***production***. Our modules are located inside the `/terraform/modules/common-infrastructure` folder. Here is an example of a google cloud bucket resource definition in the `gcloud_bucket.tf` file.

Any parameter values that are expected to be replaced per environment is being extracted into variables. For instance, *${var.environment}* and *${var.storage_class}*.

```
# Google Cloud bucket

resource "google_storage_bucket" "picture-bucket" {
  name    = picture-bucket-${var.environment}"
  storage_class = ${var.storage_class}
  location      = "US"
}
```

Below is an example of google cloud sql resource definition defined in the `gcloud_sql.tf` file. Again any parameter and attribute values are being extracted into separate variables. For instance, *${var.environment}*, *${var.cloud_sql_db_tier}* etc.

```
# Google Cloud SQl

resource "google_sql_database_instance" "master-db" {
  provider   = google-beta
  name       = master-${var.environment}"
  database_version = "POSTGRES_9_6"
  region     = var.gcp_region

  depends_on = [
google_service_networking_connection.private_sql_connection_0,
]

settings {
  tier              = ${var.cloud_sql_db_tier}
  availability_type = ${var.cloud_sql_db_availability_type}
  disk_autoresize   = true
  disk_size         = ${var.cloud_sql_disk_size}
  disk_type         = "PD_SSD"

  backup_configuration {
    enabled    = true
    start_time = "06:00"
  }

maintenance_window {
  day          = 7
  hour         = 7
  update_track = "stable"
}

ip_configuration {
  private_network = var.vpc_network_self_link
  ipv4_enabled    = false
  }
```

```
  }
}
```

Here is the definition for the `variables.tf` file. The default values can be set to empty string.

```
variable "storage_class" {
  default = ""
}

variable "cloud_sql_db_tier" {
  default = ""
}

variable "cloud_sql_disk_size" {
  default = ""
}

variable "cloud_sql_db_availability_type" {
  default = ""
}
```

The `main.tf` file inside the staging/production folder makes the call to the terraform module folder. The `source = "../terraform/modules/common-infrastructure"` points to the directory of the terraform modules. The `module "staging"` and `module "production"` serves as an identifier for the module. The variables defined above are assigned values per environment. The ***terraform init*** command downloads the modules to the *.terraform* folder. These variable values are passed into the terraform modules at the time of creating the execution **plan** followed by **apply**. Here is a code snippet of `main.tf` file located inside staging folder.

```
module "staging" {
  source = "../terraform/modules/common-infrastructure"
  environment = "staging"
  cloud_sql_db_tier = "db-custom-16-61440"
  cloud_sql_disk_size = "20"
  cloud_sql_db_availability_type = "ZONAL"
  storage_class = "REGIONAL"
}
```

Below is the code snippet for `main.tf` located inside the production folder.

```
module "production" {
  source = "../terraform/modules/common-infrastructure"
  environment = "production"
  cloud_sql_db_tier = "db-custom-2-13312"
  cloud_sql_disk_size = "50"
  cloud_sql_db_availability_type = "REGIONAL"
  storage_class = "MULTI_REGIONAL"
}
```

We maintain separate google cloud projects for our staging and production environments. The `provider.tf` file inside the staging/production folder points to the staging/production google projects. The `backend.tf` file points to the google cloud bucket with our terraform state backend.

Potential Use Cases

Let's say your organization has multiple business units/portfolios or even maintains infrastructure for multiple customers in a multi-tenant architecture. Terraform modules are the way to go! You can share your modules within your organization. Invest more time in building robust modules rather than having to copy/paste configuration files across multiple folders/projects.

Conclusion

It is quite evident how powerful the terraform modules are. We are constantly incorporating best practices and standards to our modules. We have code reviews in place that helps us validate any changes/updates to our infrastructure. We are working towards a more DRY infrastructure codebase!

Terraform Terraform Modules