Refactor Monolithic Terraform Configuration

17 MIN | PRODUCTS USED Terraform

Some Terraform projects start as a *monolith*, a Terraform project managed by a single main configuration file in a single directory, with a single state file. Small projects may be convenient to maintain this way. However, as your infrastructure grows, restructuring your monolith into logical units will make your Terraform configurations less confusing and safer to make changes to.

These tutorials are for Terraform users who need to restructure Terraform configurations as they grow. In this tutorial, you will provision two instances of a web application hosted in an S3 bucket that represent production and development environments. The configuration you use to deploy the application will start in as a monolith. You will modify it to step through the common phases of evolution for a Terraform project, until each environment has its own independent configuration and state.

Prerequisites

Although the concepts in this tutorial apply to any module creation workflow, this tutorial uses Amazon Web Services (AWS) modules.

To follow this tutorial you will need:

- An AWS account Configure one of the authentication methods described in our AWS Provider Documentation. The examples in this tutorial assume that you are using the Shared Credentials file method with the default AWS credentials file and default profile.
- The AWS CLI
- The Terraform CLI

> *If you don't have an AWS account, the AWS CLI installed locally, or Terraform installed locally, complete this tutorial in an interactive lab from your web browser. Launch it here.*

Show Terminal

Apply a monolith configuration

In your terminal, clone the example repository. It contains the configuration used in this tutorial.

```
$ git clone https://github.com/hashicorp/learn-terraform
```

Copy

Tip: Throughout this tutorial, you will have the option to check out branches that correspond to the version of Terraform configuration in that section. You can use this as a failsafe if your deployment is not working correctly, or to run the tutorial without making changes manually.

Navigate to the directory.

```
$ cd learn-terraform-code-organization
```

Copy

Your root directory contains four files and an "assets" folder. The root directory files compose the configuration as well as the inputs and outputs of your deployment.

- `main.tf` - configures the resources that make up your infrastructure.
- `variables.tf` - declares variables to mark the dev and prod environments, along with a region to deploy your infrastructure in.
- `terraform.tfvars.example` - will define your region and environment prefixes.
- `outputs.tf` - specifies the two website endpoints for your dev and prod buckets.
- `assets` - houses your webapp HTML file.

In your text editor, open the `main.tf` file. The file consists of a few different resources:

- The `random_pet` resource creates a string to be used as the unique name of your S3 bucket.
- Two `aws_s3_bucket` resources designated `prod` and `dev` , which each create an S3 bucket with a read policy. Notice the resource argument `bucket` , which defines the S3 bucket name by interpolating the environment prefix and the `random_pet` resource name.
- Two `aws_s3_bucket_object` resources designated `prod` and `dev` , which upload `content` from the local `assets` directory (using the built in `file()` function).

Terraform requires unique identifiers - in this case `prod` or `dev` for each `s3` resource - to create separate resources of the same type.

Open the `terraform.tfvars.example` file in your repository and edit it with your own variable definitions. Change the region to your nearest location in your text editor.

```
region = "us-east-1"
prod_prefix = "prod"
dev_prefix = "dev"
```

Copy

Save your changes in your editor and rename the file to `terraform.tfvars` . Terraform automatically loads variable values from any files that end in `.tfvars` .

```
$ mv terraform.tfvars.example terraform.tfvars
```

Copy

In your terminal, initialize your Terraform project.

```
$ terraform init
```

Copy

Then, apply the configuration.

```
$ terraform apply
```

Copy

Accept the apply plan by entering `yes` in your terminal to create the 5 resources.

Navigate to the web address from the Terraform output to display the deployment in a browser. Your directory now contains a state file, `terraform.tfstate` .

Separate configuration

Defining multiple environments in the same `main.tf` file may become hard to manage as you add more resources. The HashiCorp Configuration Language (HCL), which is the language used to write configurations in Terraform, is meant to be human-readable and supports using multiple configuration files to help manage your infrastructure.

You will organize your current configuration by separating the configurations into two separate files — one root module for each environment. To split the configuration, first make a copy of `main.tf` and name it `dev.tf`.

```
$ cp main.tf dev.tf
```

Rename the `main.tf` file to `prod.tf`.

```
$ mv main.tf prod.tf
```

You now have two identical files. Open `dev.tf` and remove any references to the production environment by deleting the resource blocks with the `prod` ID. Repeat the process for `prod.tf` by removing any resource blocks with the `dev` ID.

Tip: To fast-forward to this file separated configuration, checkout the branch in your example repository by running `git checkout file-separation`.

Your directory structure will look similar to the one below.

```
.
├── README.md
├── assets
│   └── index.html
├── dev.tf
├── outputs.tf
├── prod.tf
├── terraform.tfstate
├── terraform.tfvars
└── variables.tf
```

Although your environments are in separate configurations, your `variables.tf` and `terraform.tfvars` files contain all the variable declarations and definitions for both environments. You now have resources split between environments in `prod.tf` and `dev.tf` and your environments have unique identifiers to distinguish the region you are deploying your infrastructure to.

Terraform loads all configuration files within a directory and appends them together, which means that any resources or providers with the same name in the same directory will cause a validation error. If you were to run a terraform command now, your `random_pet` resource and `provider` block would cause errors.

Edit the `prod.tf` file by commenting out the `terraform` block, the `provider` block, and the `random_pet` resource.

```
# terraform {
#   required_providers {
#     aws = {
#       source = "hashicorp/aws"
#     }
#   }
# }

# provider "aws" {
#   region = var.region
# }

# resource "random_pet" "petname" {
#   length    = 3
#   separator = "-"
# }
```

With your `prod.tf` shared resources commented out, your production environment will still inherit the value of the `random_pet` resource in your `dev.tf` file.

Simulate a hidden dependency

You may want your development and production environments to share bucket names, but the current configuration is particularly dangerous because of the hidden resource dependency built into it. Imagine that you want to test a random pet name with four words in development (instead of three). Update your `random_pet` resource in `dev.tf` with a `length` attribute of `4`.

```
resource "random_pet" "random" {
  length    = 4
  separator = "-"
}
```

You might think you are only updating the development environment because you only changed `dev.tf`, but remember, production now inherits its values from development. Save and apply the configuration.

```
$ terraform apply
```

Enter `yes` when prompted to apply the changes. The operation updates *all* five of your resources by destroying and recreating them.

Notice that Terraform destroyed and recreated all the infrastructure in both development and production. In this scenario, you encountered a hidden resource dependency because the configurations share state.

Any configuration changes in Terraform should first go through a `terraform plan` and be carefully reviewed before applying. If an operator does not carefully review the plan output or if this change is auto-applied in a CI/CD pipeline, you could accidentally apply unnecessary breaking changes to other environments.

Destroy your resources before moving on. Respond to the confirmation prompt with a `yes`.

```
$ terraform destroy
```

Separate states

The destroy you just ran got rid of resources from both development and production. While you could use the `terraform apply` command with the `-replace` flag to specify which resources you need to recreate individually, that approach requires more work. To avoid having to individually replace resources, you need to separate your development and production state.

State separation signals more mature usage of Terraform; with additional maturity comes additional complexity. There are two primary methods to separate state between environments: directories and *workspaces*.

To separate environments with potential configuration differences, use a directory structure. Use workspaces for environments that do not greatly deviate from one another, to avoid duplicating your configurations. Try both methods in the sections below to help you understand which will serve your infrastructure best.

Directories Workspaces

By creating separate directories for each environment, you can shrink the blast-radius of your Terraform runs and ensure you will only touch intended infrastructure. Your state files are stored on disk in their corresponding configuration directories and do not touch the other directories by default, to help ensure deployments will not impact one another.

Directory separated environments rely on duplicate Terraform code, which may be useful if your deployments need differ, for example to test infrastructure changes in development. But they can run the risk of creating drift between the environments over time. If you want to move a project with an existing single state file to directory-separated states, you *must* destroy your previous state by destroying the infrastructure in Terraform before splitting your configuration into directories, which will cause disruptions in your deployment.

When you are finished separating these environments into directories, your file structure should look like the one below.

```
.
├── assets
|   ├── index.html
├── prod
|   ├── main.tf
|   ├── variables.tf
|   ├── terraform.tfstate
|   └── terraform.tfvars
└── dev
    ├── main.tf
    ├── variables.tf
    ├── terraform.tfstate
    └── terraform.tfvars
```

Create `prod` and `dev` directories

Create a directory corresponding to each environment named `prod` and `dev`.

```
$ mkdir prod && mkdir dev
```

Move the `dev.tf` file to the `dev` directory, and rename it to `main.tf`.

```
$ mv dev.tf dev/main.tf
```

Copy the `variables.tf`, `terraform.tfvars`, and `outputs.tf` files to the `dev` directory

```
$ cp outputs.tf terraform.tfvars variables.tf dev/
```

Copy

Your environment directories are now one step removed from the `assets` folder where your webapp lives. Open the `dev/main.tf` file in your text editor and edit the file to reflect this change by editing the file path in the `content` argument of the `aws_s3_bucket_object` resource with a `/..` before the `assets` subdirectory.

```
resource "aws_s3_bucket_object" "dev" {
  acl          = "public-read"
  key          = "index.html"
  bucket       = aws_s3_bucket.dev.id
-  content      = file("${path.module}/assets/index.html")
+  content      = file("${path.module}/../assets/index.html")
  content_type = "text/html"
}
```

You will need to remove the references to the `prod` environment from your `dev` configuration files.

First, open `dev/outputs.tf` in your text editor and remove the reference to the `prod` environment.

```
- output "prod_website_endpoint" {
-   value = "http://${aws_s3_bucket.prod.website_endpoint}/index.ht
- }
```

Next, open `dev/variables.tf` and remove the reference to the `prod` environment.

```
- variable "prod_prefix" {
```

```
-   description = "This is the environment where your webapp is dep
- }
```

Finally, open `dev/terraform.tfvars` and remove the reference to the `prod` environment.

```
region      = "us-east-2"
- prod_prefix = "prod"
dev_prefix  = "dev"
```

Create a `prod` directory

Rename `prod.tf` to `main.tf` and move it to your production directory.

```
$ mv prod.tf prod/main.tf
```

Move the `variables.tf`, `terraform.tfvars`, and `outputs.tf` files to the `prod` directory.

```
$ mv outputs.tf terraform.tfvars variables.tf prod/
```

Repeat the steps you took in the `dev` directory, and uncomment out the `random_pet` and `provider` blocks in `main.tf`.

First, open `prod/main.tf` and edit it to reflect new directory structure by adding `/..` to the file path in the `content` argument of the `aws_s3_bucket_object`, before the `assets` subdirectory.

Next, remove the references to the `dev` environment from `prod/variables.tf`, `prod/outputs.tf`, and `prod/terraform.tfvars`.

Finally, uncomment `terraform` block, the `provider` block, and the `random_pet` resource in `prod/main.tf`.

```
terraform {
  required_providers {
    aws = {
      source = "hashicorp/aws"
    }
  }
}

provider "aws" {
  region = var.region
}

resource "random_pet" "petname" {
  length    = 3
  separator = "-"
}
```

Tip: To fast-forward to this configuration, run `git checkout directories`.

Deploy environments

To deploy, change directories into your development environment.

```
$ cd dev
```

Copy

This directory is new to Terraform, so you must initialize it.

```
$ terraform init
```

Copy

Run an apply for the development environment and enter `yes` when prompted to accept the changes.

```
$ terraform apply
```

Check your website endpoint in a browser.

You now have only one output from this deployment. Repeat these steps for your production environment.

```
$ cd ../prod
```

This directory is new to Terraform, so you must initialize it first.

```
$ terraform init
```

Run your apply for your production environment and enter `yes` when prompted to accept the changes. Check your website endpoint in a browser.

```
$ terraform apply
```

Now your development and production environments are in separate directories, each with their own main configuration file and state.

Destroy infrastructure

Before moving on to the second approach to environment separation, destroy both the `dev` and `prod` directories.

```
$ terraform destroy
```

To learn about another method of environment separation, navigate to the "Workspaces" tab.

Next steps

In this tutorial, you started with a monolithic Terraform configuration that deployed two environments. You separated those environments by creating different directories or workspaces, and state files for each.

- To combat drift, you should start to identify the resources that can be bundled as modules.
- For more information on state management and using Terraform as a team, consider trying Terraform Cloud as a remote backend and migrate your configuration.

Was this tutorial helpful?

Yes No

‹ BACK TO COLLECTION

HashiCorp

System Status Cookie Manager Terms of Use Security Privacy

stdin: is not a tty