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
```

Copy

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

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

Copy

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 = "-"
# }
```


`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 = "-"
}
```

Copy

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
```

Copy

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
```

Copy

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

Workspace-separated environments use the same Terraform code but have different state files, which is useful if you want your environments to stay as similar to each other as possible, for example if you are providing development infrastructure to a team that wants to simulate running in production.

However, you must manage your workspaces in the CLI and be aware of the workspace you are working in to avoid accidentally performing operations on the wrong environment.

Update your Terraform directory

Note: If you ran the directory separation example, begin this section by removing your environment directories with `rm -rf dev/ prod/` in your root directory. Then, switch branches by running `git checkout file-separation` in your terminal.

All Terraform configurations start out in the `default` workspace. In your terminal, type `terraform workspace list` to see the list of your workspaces and which one is currently selected represented by `*`.

```
$ terraform workspace list
  *  default
```

Copy

Before you create a new workspace, you need to update your configuration files so that both environments can use the same one. In your root directory, remove the `prod.tf` file.

```
$ rm prod.tf
```

Copy

Update your variable input file to remove references to the individual environments.

In your text editor, open `variables.tf` and remove the environment references.

```
variable "region" {
  description = "This is the cloud hosting region where your webapp
}

- variable "dev_prefix" {
+ variable "prefix" {
  description = "This is the environment where your webapp is deplo
```

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

Rename `dev.tf` to `main.tf`.

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

Copy

Open this file in your text editor and replace the "dev" resource IDs and variables with the function of the resource itself. You are creating a generic configuration file that can apply to multiple environments.

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

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

- resource "aws_s3_bucket" "dev" {
+ resource "aws_s3_bucket" "bucket" {
-   bucket = "${var.dev_prefix}-${random_pet.petname.id}"
+   bucket = "${var.prefix}-${random_pet.petname.id}"
  acl    = "public-read"

  policy = <<EOF
{
    "Version": "2012-10-17",
    "Statement": [{
        "Sid": "PublicReadGetObject",
        "Effect": "Allow",
        "Principal": "*",
        "Action": [
            "s3:GetObject"
```

```
        ],
        "Resource": [
-       "arn:aws:s3:::${var.dev_prefix}-${random_pet.petname.id}/*"
+       "arn:aws:s3:::${var.prefix}-${random_pet.petname.id}/*"
        ]
    }]
}
EOF

  website {
    index_document = "index.html"
    error_document = "error.html"

  }
  force_destroy = true
}

- resource "aws_s3_bucket_object" "dev" {
+ resource "aws_s3_bucket_object" "webapp" {

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

}
```

Now that your workspace handles the resources as individual environments, only one output is expected. Open your `outputs.tf` file in your text editor and remove the `dev` environment reference in the output name. Change `dev` in the value to `bucket`.

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

Copy

Finally, replace `terraform.tfvars` with a `prod.tfvars` file and a `dev.tfvars` file to define your variables for each environment.

For your `dev` workspace, copy the `terraform.tfvars` file to a new `dev.tfvars` file.

```
$ cp terraform.tfvars dev.tfvars
```

Copy

Edit the variable definitions in your text editor. For your `dev` workspace, the prefix value should be `dev`.

```
region  = "us-east-2"
prefix  = "dev"
```

Copy

Create a new `.tfvars` file for your production environment variables by renaming the `terraform.tfvars` file to `prod.tfvars`.

```
$ mv terraform.tfvars prod.tfvars
```

Copy

Update `prod.tfvars` with your `prod` prefix.

```
region  = "us-east-2"
prefix  = "prod"
```

Copy

Now that you have a single `main.tf` file, initialize your directory to ensure your Terraform configuration is valid.

```
$ terraform init
```

Copy

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

Create a `dev` workspace

Create a new workspace in the Terraform CLI with the `workspace` command.

```
$ terraform workspace new dev
```

Copy

Terraform's output will confirm you created the workspace and are operating within that workspace.

```
Created and switched to workspace "dev"!

You're now on a new, empty workspace. Workspaces isolate their stat
so if you run "terraform plan" Terraform will not see any existing
for this configuration.
```

Any previous state files from your `default` workspace are hidden from your `dev` workspace, but your directory and file structure do not change.

Initialize the directory.

```
$ terraform init
```

Copy

Apply the configuration for your development environment in the new workspace, specifying the `dev.tfvars` file with the `-var-file` flag.

```
$ terraform apply -var-file=dev.tfvars
```

Copy

Terraform will create three resources and prompt you to confirm that you want to perform these actions in the workspace "dev".

```
## ...
```

```
Do you want to perform these actions in workspace "dev"?
  Terraform will perform the actions described above.
  Only 'yes' will be accepted to approve.

  Enter a value:

  ## ...
```

Enter `yes` and check your website endpoint in a browser.

Create a prod workspace

Create a new production workspace.

```
$ terraform workspace new prod
```

Copy

Terraform's output will confirm you created the workspace and are operating within that workspace.

```
Created and switched to workspace "prod"!

You're now on a new, empty workspace. Workspaces isolate their stat
so if you run "terraform plan" Terraform will not see any existing
for this configuration.
```

Any previous state files from your `dev` workspace are hidden from your `prod` workspace, but your directory and file structure do not change.

You have a specific `prod.tfvars` file for your new workspace. Run `terraform apply` with the `-var-file` flag and reference the file. Enter `yes` when you are prompted to accept the changes and check your website endpoint in a browser.

```
$ terraform apply -var-file=prod.tfvars
```

Copy

Your output now contains only resources labeled "production" and your single website endpoint is prefixed with `prod .`

State storage in workspaces

When you use the default workspace with the local backend, your `terraform.tfstate` file is stored in the root directory of your Terraform project. When you add additional workspaces your state location changes; Terraform internals manage and store state files in the directory `terraform.tfstate.d .`

Your directory will look similar to the one below.

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

Destroy your workspace deployments

To destroy your infrastructure in a multiple workspace deployment, you must select the intended workspace and run `terraform destroy -var-file=` with the `.tfvars` file that corresponds to your workspace.

Destroy the infrastructure in your `prod` workspace, specifying the `prod.tfvars` file with the `-var-file` flag.

```
$ terraform destroy -var-file=prod.tfvars
## ...

Plan: 0 to add, 0 to change, 3 to destroy.

Changes to Outputs:
  - website_endpoint = "http://prod-definitely-resolved-ghoul.s3-we

Do you really want to destroy all resources in workspace "prod"?
  Terraform will destroy all your managed infrastructure, as shown
  There is no undo. Only 'yes' will be accepted to confirm.

  Enter a value:
```

Copy

When you are sure you are running your `destroy` command in the correct workspace, enter `yes` to confirm the destroy plan.

Next, to destroy your development infrastructure, switch to your `dev` workspace using the `select` subcommand.

```
$ terraform workspace select dev
```

Copy

Run `terraform destroy` specifying `dev.tfvars` with the `-var-file` flag.

```
$ terraform destroy -var-file=dev.tfvars
```

Copy

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