DevOps Self-Service Centric Terraform Project Structure

How to structure Terraform code and its reusable modules

Photo by Craig Glantz

Before we dive into Terraform project structure, I'd like to share a fun fact to assure you we are in the right boat with Terraform!

Per GitHub's The State of the Octoverse survey of 2021–2022, the Hashicorp Configuration Language (HCL) took first place in the fastest-growing programming language category over the past year. This was largely driven by the growth in the

popularity of the Terraform tool and IaC practices to automate deployments in the cloud increasingly.

GROWTH IN PROGRAMMING LANGUAGES 2021-2022

01 HCL	**56.1%**
02 Rust	**50.5%**
03 TypeScript	**37.8%**
04 Lua	**34.2%**
05 Go	**28.3%**
06 Shell	**27.7%**
07 Makefile	**23.7%**
08 C	**23.5%**
09 Kotlin	**22.9%**
10 Python	**22.5%**

Image source: [The top programming languages | The State of the Octoverse (github.com)]

Guided by the DevOps self-service pipeline architecture, we will explore how to structure Terraform code and its reusable modules in this story. Notice the red highlighted rectangle in the diagram below. Let's dive in.

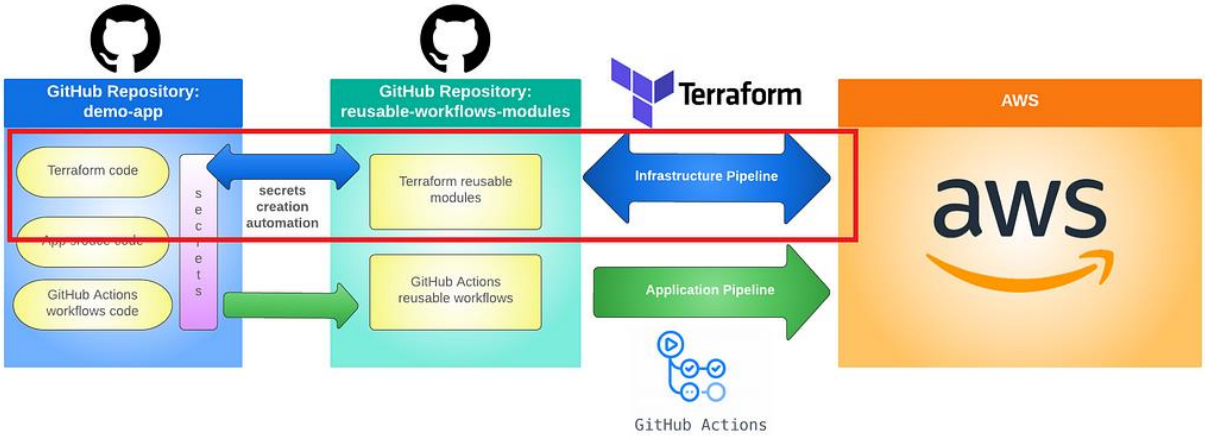


Diagram by author

Terraform Reusable Modules

In Terraform, a module is a container for multiple resources used together. Modules can create reusable components and are a good way to organize your infrastructure into logical units. Benefits of reusable Terraform modules include:

- DRY (Don't Repeat Yourself) Principle: Modules allow you to define your infrastructure in a reusable way, reducing the amount of code duplication and making it easier to manage.
- Reusability: Modules can be used across multiple environments and projects, allowing you to reuse your infrastructure code.
- Organization: Modules help you to organize your infrastructure code into logical units, making it easier to understand and maintain.

- Collaboration: Modules can be shared with other teams and organizations, making it easier to collaborate on infrastructure projects.
- Standardization: Help standardize compliance and security standards across various infrastructures and projects.

Terraform reusable modules should be kept in a centralized repository, which acts sort of like a private Terraform registry. For demo purposes, we can call this centralized repo `reusable-workflows-modules`. What does the reusable module structure look like? See the sample screenshot below:

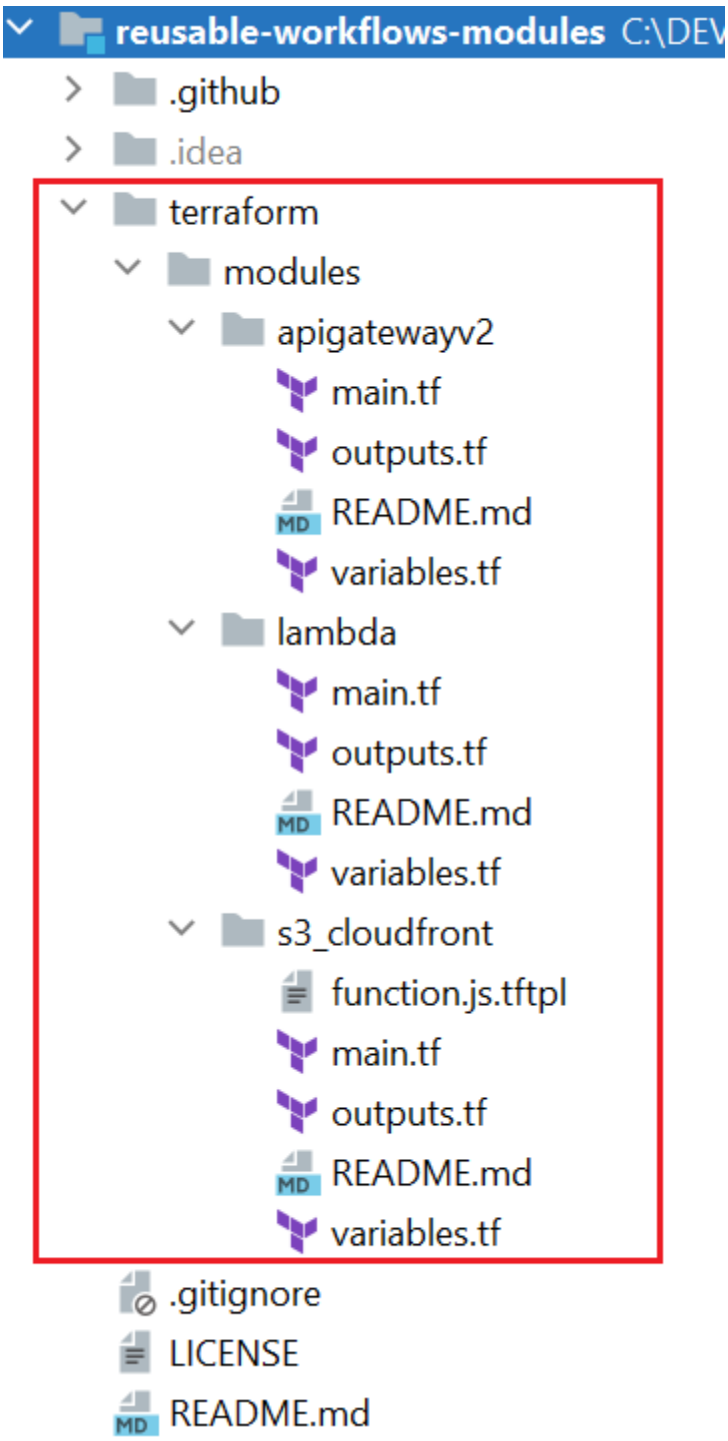


image by author

Under the `terraform/modules` directory, you can have one or multiple reusable modules. Notice each reusable module contains a list of files with standard naming conventions, which are pretty self-explanatory by their file names:

- `main.tf`
- `outputs.tf`
- `variables.tf`
- `README.md`

This file structure is merely a recommendation. Depending on the complexity of your reusable module(s), you may need to include other files in your file structure.

Project-Specific Terraform Code

Since we use GitHub Actions as our CI/CD tool, it makes sense for us to have Terraform code committed together with the project source code in the same repository. Suppose we have all Terraform code for all our projects committed in one standalone repo.

In that case, we cannot run GitHub Actions workflows to deploy Terraform to different environments/accounts in AWS for different projects, as it becomes extremely challenging for us to manage GitHub Environments/secrets and to match IAM credentials for many projects within the same repo.

It is best for Terraform code to reside in the same repo as its project source code. See the sample screenshot below, where in this project, `springboot-infracost-demo`, Terraform code is located under a `terraform` directory under root. There is a matching

GitHub Actions workflow `terraform.yml` under `.gitub/workflows` directory for running Terraform init/plan/apply, along with the workflows to deploy project code to its corresponding AWS resources.

Based on the 3–2–1 rule for DevOps self-service pipeline architecture, we can see that our three types of source code are highlighted in red rectangles below: GitHub Actions workflows, application source code, and Terraform code.

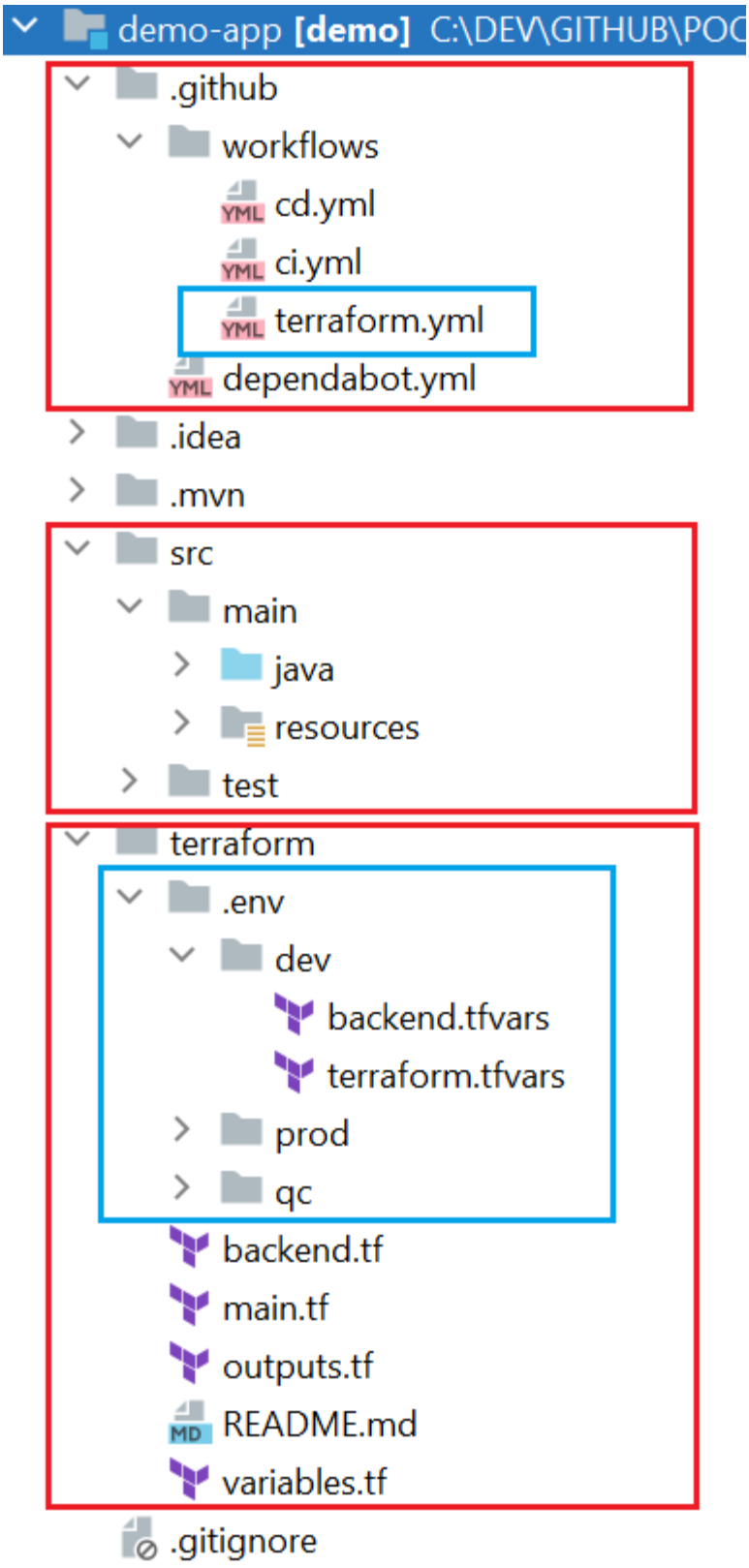


image by author

How to call reusable modules from project-specific Terraform code

Since Terraform modules reside in `reusable-workflows-modules` repo, project-specific Terraform code only carries project-specific logic, such as environment-specific configurations. We are not repeating the reusable module logic. We can call the reusable Terraform modules located in `reusable-workflows-modules` repo from `main.tf` under the project root.

It's best practice always to pin Terraform module source to a particular version, or `main` in our sample below, referring to the latest version of the branch `main` of `reusable-workflows-modules` repo.

Our Terraform reusable modules reside in a subdirectory of `reusable-workflows-modules` repo: `github.com/wenqiglantz/reusable-workflows-modules/terraform/modules`. To specify the Terraform module subdirectory, we need to place a special double-slash syntax right after the repo name. Terraform interprets the double-slash to indicate that the remaining path after that point is a subdirectory within the repo.

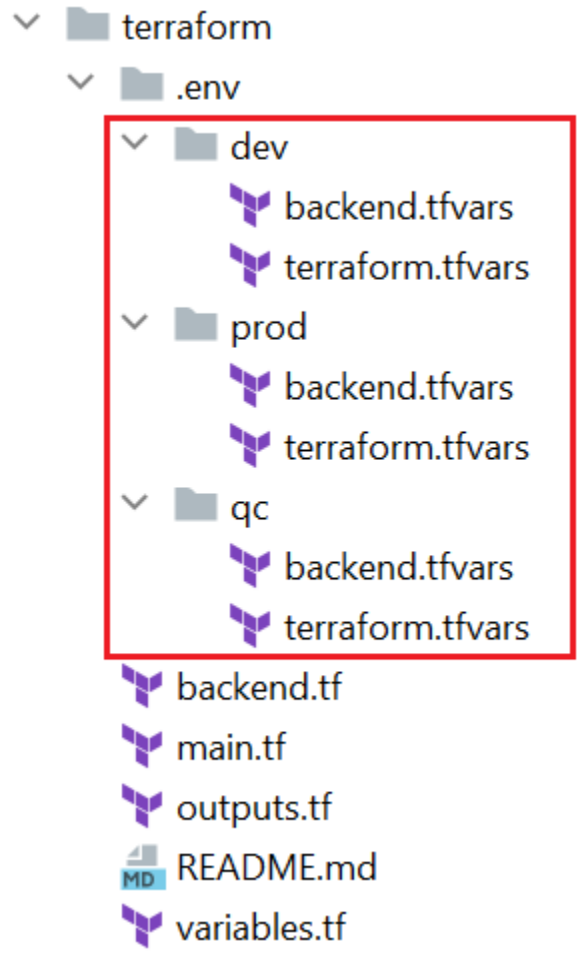
To pin Terraform module source to a particular branch, the `main` branch of `reusable-workflows-modules` repo, we add `?ref=main` at the end of the module name. Here's a sample code snippet of how `springboot-infracost-demo` app's `main.tf` calls the reusable module `lambda`:

```
module "lambda" {
  source           = "github.com/wenqiglantz/reusable-workflows-
modules//terraform/modules/lambda?ref=main"
  s3_bucket_name   = var.s3_bucket_name
  s3_object_key    = var.s3_object_key
```

```
  lambda_functions = var.lambda_functions
}
```

Group .tfvars files under the environment-related directories

Notice the screenshot below from an application repo, we have `terraform.tfvars` file located under the environment-related directories such as `dev`, `qc`, and `prod` under `.env`. These are the files holding environment-specific values for the variables.



Remote State Management

There is a known open issue on the limitation of not being able to pass in variables in `backend.tf` for remote state management in S3. As `backend.tf` is called during `terraform init`, and there is no way for terraform to parse variables in `backend.tf` before it's initialized. Fortunately, there is a workaround. Here's three steps to follow:

- We can keep the `backend.tf` file at the terraform root, as shown in the screenshot above, but this file will hold no S3-specific details. Here's a code snippet of `backend.tf`:

```
terraform {
  backend "s3" {
    //details in .env/{env}/backend.tfvars
  }
}
```

- We add a `backend.tfvar` file under the `.env` directory under its respective environment folder. This file contains the S3 bucket name, key, region, and the encrypt flag. See the following snippet for `.env/dev/backend.tfvar`:

```
bucket         = "terraform-remote-backend-dev"
key            = "demo/state.tfstate"
region         = "us-east-1"
encrypt        = "true"
```

- We then run `terraform init` by passing in this `backend.tfvar`. Here's the snippet to run that locally:

```
terraform init -backend-config='./.env/dev/backend.tfvars'
```

When running the GitHub Actions workflow, we get the following from `terraform.yml`:

```
terraform init -backend-config='./.env/${{
github.event.inputs.environment || 'dev' }}/backend.tfvars' -
upgrade=true -no-color -input=false
```

.gitignore

Be sure to revise your `.gitignore` file at your project root to instruct GitHub to ignore certain terraform files/directories. Here's a sample of the Terraform-related files/directories to be ignored in `.gitignore`:

```
########################################
# Terraform files/directory to ignore #
########################################
# Local .terraform directories
**/.terraform/*

# .tfstate files
*.tfstate
*.tfstate.*

# Crash log files
crash.log
crash.*.log

# Ignore override files as they are usually used to override
resources locally and so
# are not checked in
override.tf
override.tf.json
```

```
*_override.tf
*_override.tf.json

# Include override files you do wish to add to version control
using negated pattern
# !example_override.tf

# Include tfplan files to ignore the plan output of command:
terraform plan -out=tfplan
# example: *tfplan*

# Ignore CLI configuration files
.terraformrc
terraform.rc

# Lock
.terraform.lock.hcl

# Plan files
*.tfplan
```

GitHub Actions Workflow for Terraform

GitHub Actions workflow is used to run `terraform init`, `plan`, `apply`, and `destroy` for your project. First, let's get to the housekeeping items to ensure that GitHub secrets are properly configured.

Secrets configuration

The only GitHub environment secrets needed to run the terraform workflow are `TERRAFORM_ROLE_TO_ASSUME` and `AWS_REGION`. Configure them accordingly per your AWS account setup.

Terraform reusable workflow

A reusable GitHub Actions workflow for Terraform deployment has been extracted into `reusable-workflows-modules` repo, details of this reusable workflow `terraform.yml` are as follows:

A few important highlights:

- Notice line 69 above, `git config --global url."https://oauth2:${{ secrets.NPM_TOKEN }}@github.com".insteadOf https://github.com.` This is important if your reusable modules reside in a private repo. We are passing a `NPM_TOKEN` that has access to the private repo as the client app doesn't pass such credentials when calling Terraform reusable module (credit: Unable to init with private repo modules · Issue #33 · hashicorp/setup-terraform). This line is critical. Without this line, the workflow will fail at `terraform init` step due to not being able to access the `reusable-workflows-modules` private repo without passing in a valid token.
- Also, notice we are passing in `terraform.tfvars` with `-var-file` to the Terraform commands. Depending on the environment passed in from the `workflow_dispatch` manual trigger, Terraform workflow checks in the environment folder under `.env`, runs for that chosen environment with the respective `.tfvars` file passed in.
- `terraform apply` step only executes on `apply-branch` which is passed in from the calling workflow, it defaults to the default branch in

your repo, such as `main`, depending on the branching strategy for your repo.

- To run `terraform destroy`, create a feature branch, `destroy`, and trigger this workflow from that branch to destroy.

How To Call Terraform Reusable Workflow

When deploying Terraform files, your app's workflow should be calling the above Terraform reusable workflow. Here's a sample `terraform.yml` from an app that calls Terraform's reusable workflow:

With this, you have a fully functional infrastructure pipeline up and running to provision your cloud resources for your application.

Summary

We explored DevOps self-service centric Terraform project structure in this article. We looked into what Terraform reusable modules are, their benefits, and how they are structured in the GitHub repo.

We then moved on to examine the code structure of the three types of source code in an application repo and how to trigger a call from the application repo's Terraform code to the reusable module in the centralized repo.

Finally, we dived deep into the details of Terraform GitHub Actions workflow, both the reusable workflow in the centralized repo and the calling workflow from the application repo. I hope you find this story helpful on your path toward DevOps self-service.

The sample code for this story can be found in the following GitHub repos:

- reusable-workflows-modules
- springboot-infracost-demo

I welcome you to check out the rest of the four parts in my five-part "The Path to DevOps Self-Service" series:

DevOps Self-Service Pipeline Architecture and Its 3–2–1 Rule
A high-level architectural overview of the self-service pipeline
betterprogramming.pub

DevOps Self-Service-Centric GitHub Actions' Workflow Orchestration
How to orchestrate GitHub Actions' workflows that are driven by image immutability
betterprogramming.pub

DevOps Self-Service Centric Pipeline Security and Guardrails
A list of hand-picked actions for security scans and guardrails for your pipelines, infrastructure, source code, base…
betterprogramming.pub

DevOps Self-Service Centric Pipeline Integration
Secrets management as the glue of pipeline integration
betterprogramming.pub

Happy coding!

References

The top programming languages

Languages After nearly 30 years of Java, you might expect the language to be showing some signs of wear and tear, but…
octoverse.github.com

Terraform Modules: Create Reusable Infrastructure As Code | Build5Nines
Terraform, being an Infrastructure as Code (IaC) tool, enables you to write declarative code that is then used to…
build5nines.com

Unable to init with private repo modules · Issue #33 · hashicorp/setup-terraform
I have a terraform project that uses private repo modules. All of the repositories are under my organization's control…
github.com