# Optimizing Terraform projects with Terragrunt and a Terraform Registry

## Introduction

Defining your cloud computing infrastructure with code (IAC) is becoming an industry standard for enterprise IT teams so they can scale effectively as their development teams and applications grow.

Terraform, by Hashicorp, is an infrastructure as code tool that lets you define both cloud and on-prem resources using human-readable configuration files that you can version, reuse, and share across your various projects and development teams.

This post will focus on several methods and patterns to take Terraform even further, specifically focusing on keeping your infrastructure code and configurations organized and easy to maintain. As a result, this post assumes you've worked with Terraform before and have a general understanding of how to use it and how it works. If you want to know more about Terraform and what it can offer, take a look at Hashicorp's [website](). Also check out this [video](https://www.youtube.com/watch?v=h970ZBgKINg) by Hashicorp’s cofounder for a short summary of what Terraform can offer.

## Modularization & Terraform Registry

Terraform has a few different offerings that each provide different features - the most basic of which is open source, the Terraform CLI. Although Terraform CLI is a great tool on its own, it is significantly more useful when best practices are implemented.

One of the very first things you should do to organize your Terraform code is to modularize it by breaking it apart into child modules or components that encapsulate smaller pieces of your infrastructure; instead of having one main.tf file that provisions all of your resource types, break up your architecture into several components (i.e. appservice plan, appservice, storage account, redis cache) and reference them in your encapsulating module (ex. main.tf).

Hashicorp also gives you access to their public Terraform registry for various providers like AWS and Azure. The public registry already has re-usable modules for the simplest cloud resource blocks for you to extend and utilize.

However, although these pre-defined modules exist, to standardize things like resource naming conventions, cloud computing sizing/scaling, and other restrictions you may want to impose on your development teams, you should create your own modules that utilize modules in the public Terraform Registry and create them with distinctly defined inputs and outputs.

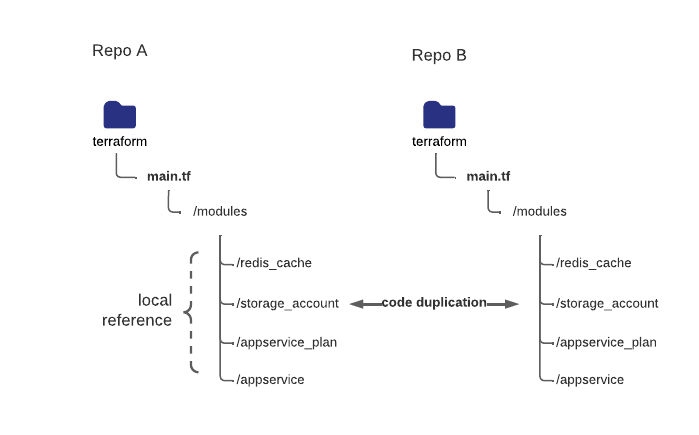
Now the question arises: where do I keep the code for my child modules so that they are easily distributable, re-usable, and maintainable? Creating these child modules and storing the code within the application code repository itself creates some disadvantages. What if you want to re-use these sub modules for other repositories or different projects? You'd have to duplicate the child module code and track it in several places. Not ideal.

To address this, you should utilize a shared registry or storage location for your child modules so that you can reference them from multiple repositories and even distribute different versions to different projects. This would involve moving each sub module to its own individual repo to be maintained independently and then uploading it to your registry or central storage location. Acceptable methods that Terraform can work with are:

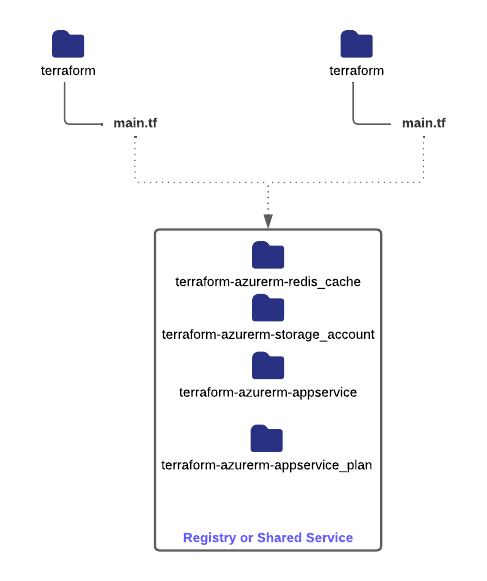
* GitHub
* Bitbucket
* Generic Git, Mercurial repositories
* HTTP URLs
* S3 buckets
* GCS buckets

See the [Terraform documentation on module sources](https://www.terraform.io/language/modules/sources) for more information.

Utilizing these methods allow you to maintain your sub modules independently and distribute different versions that larger applications can choose to inherit. You might go from terraform projects like this:



to this:



## Terragrunt

### Preface

Utilizing registries to modularize your infrastructure is only a small part of the improvements you can make to your Terraform code.

One of the major concepts of Terraform is how it tracks the state of your infrastructure with a state file. In Terraform, you need to define a "remote state" for each grouping of infrastructure you are trying to deploy/provision. This remote state could be stored an S3 bucket or an Azure Storage account, Terraform Cloud, or another applicable service. This is how Terraform knows where to track the state of your infrastructure to determine if any changes need to be applied or if the configuration has drifted away from baseline that is defined in your source code. The organization of your state files is very important, especially if you are managing it yourself and not using Terraform Cloud to perform your infrastructure runs yet.

In addition, you must keep in mind that when your development teams and applications grow in size, you will frequently need to manage development, testing, quality assurance, and production environments for several projects/components simultaneously.

How do you maintain all this infrastructure reliably? Do you track all the applications for one tier in one state file? Or do you break them up and track them separately?

How do you efficiently share variables across environments and applications? How do you successfully apply these terraform projects in a continuous deployment pipeline in a way that is consistent and repeatable for different types of terraform projects?

At one of our current clients, we were involved with onboarding Terraform as an infrastructure as code (iac) tool. However, we ran into many challenges when trying to deploy multiple tiers (dev, test, stage, production etc.) across several workstreams, specifically in a continuous manner within a deployment pipeline

The client I work for has the following requirements for the web UI portion of the services they offer (consider Azure Cloud provider for context):

* each tier has *\*six applications\** for different areas of the United States
* *\*Each application\** has a web server, a redis cache, app service plan, a storage account, and a key vault access policy to access a central key store
* Development and test tiers are deployed to a single region
* Stage and production environments are deployed to two regions, east and central

Stage and Production tiers have up to 48 resources respectively! Our client also had several other IT services that needed similar architectural setups; most projects involved deploying six instances of the application (for each service area of the Unites States), each of which was configured differently through application settings.

Initially, our team decided to use the Terraform CLI and to track our state files using an Azure Storage Account. Within the application repository, we would store several backend.tf files alongside our terraform code for each tier and pass them dynamically to terraform init --backend-config=<PATH>. We also passed variable files dynamically to terraform plan --var-file=<PATH> to combine common and tier specific application setting template files. We adopted this process in our continuous deployment pipeline by ensuring the necessary authentication principals and terraform cli packages were available on our deployment agents and then running the appropriate constructed terraform command in the appropriate directory on that agent.

This is great, but it caused us a few problems when scaling out our process. The process we used initially allowed developers to create their own terraform modules specific to their application which utilized either local or shared modules in our private registry. One of the major problems came when trying to apply these modules in a continuous integration pipeline. Each project had their own unique terraform project and their own configurations that our continuous deployment pipeline needed to adhere to.

Let’s consider the naming convention of an application. Usually, you would want the same named prefix on your resources (or apply it as a resource tag) to visually identify what projects the resources belong to. Since we had to maintain multiple tiers of environments for these projects (dev, test, stage, prod), we wanted to share this variable that across environments, only having to declare it once. We also wanted to declare specific variables, and override them on specific environments. With the Terraform CLI, there is no way to merge inputs, and share variables across environments.

So, instead of trying to conform our deployment pipelines to fit all the terraform projects and environments we had to maintain, we started to wonder if there was a better way to organize ourselves. This is where Terragrunt comes into play.

Instead of having to keep track of the various terraform commands, remote state configurations, variable files, and input parameters we needed to consolidate to run our project within our deployment pipeline (in addition to local runs), what if we had a declarative way of telling our pipeline how our terraform project is configured? What if we only needed to run the same command in all these different projects? Our deployment pipeline would be much simpler.

Terragrunt is a minimal wrapper around Terraform that enables you to create a hierarchal folder structure with declarative configuration files that allow you to dynamically assign and inherit remote state configurations, provider information, local variables, and module inputs for your terraform projects. It also gives you a flexible and unopinionated way of consolidating everything Terraform does before it runs a command; every option you pass to terraform can be specifically configured through a configuration file that inherits, merges, or overrides components from other higher-level configuration files.

Terragrunt allowed us to primarily do these important things:

* Define a configuration file that tells terraform what application setting template file to apply to an application; we used .tpl files to apply application settings to our Azure compute resources
* Define a configuration file that tells terraform what variable files to include in your terraform commands
* Define a configuration file to tell what remote state file to save based on application/tier (using folder structure)
* Allowed us to merge common input variables with tier specific input variables with desired precedence; this cut down
* Allowed us to run multiple terraform projects at once with a single command
* Pass outputs from one terraform project to another using a dependency chain

Application settings

This is a little tangent, but a cool feature terraform offers. You can define template files that can dynamically populate specific variables with information from terraform tied to your cloud resources.

**## Features**

**### Remote State Configurations**

**### Basic Terraform Example**

**### Architecture**

**# Working with Terraform**