# 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 make the most out of your terraform code, specifically focusing on keeping your infrastructure code and configurations organized and easy to maintain; we will strive to implement [DRY](https://en.wikipedia.org/wiki/Don%27t_repeat_yourself) principles wherever possible. 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](https://www.terraform.io/language/settings). 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.](https://www.terraform.io/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 break it apart into child modules or components that encapsulate smaller pieces of your infrastructure; instead of having one module file that provisions all of your resources, break up your architecture into several components (i.e. appservice plan, appservice, storage account, redis cache) and reference them as dependencies 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 validation parameters for input variables.

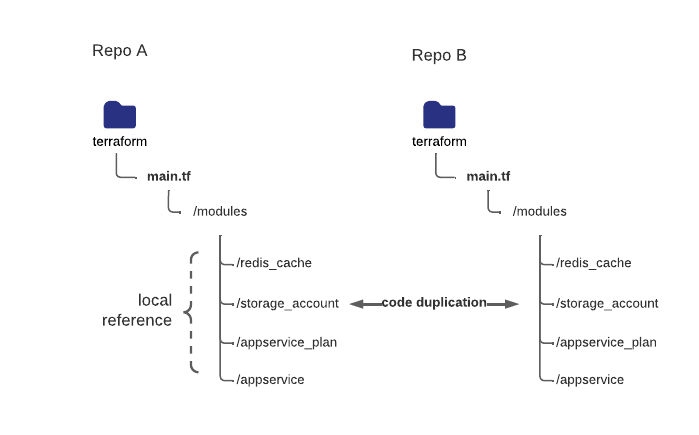
Now the question arises: where do I keep the code for my child modules so that they are easily distributable, re-usable, and maintainable? Creating these child modules and storing the code within the application code repository itself creates some disadvantages. What if you want to re-use these sub modules 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:

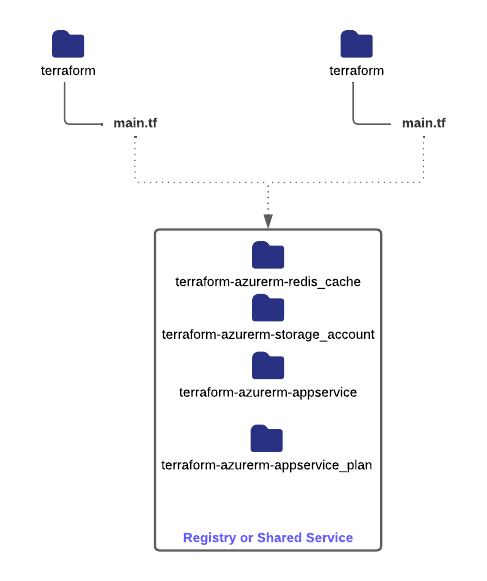
* GitHub
* Bitbucket
* Generic Git, Mercurial repositories
* HTTP URLs
* S3 buckets
* GCS buckets

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, you will frequently need to manage multiple development, testing, quality assurance, and production environments for several projects/components simultaneously; the number of configurations, variable files, cli arguments, and provider information will become untenable over time.

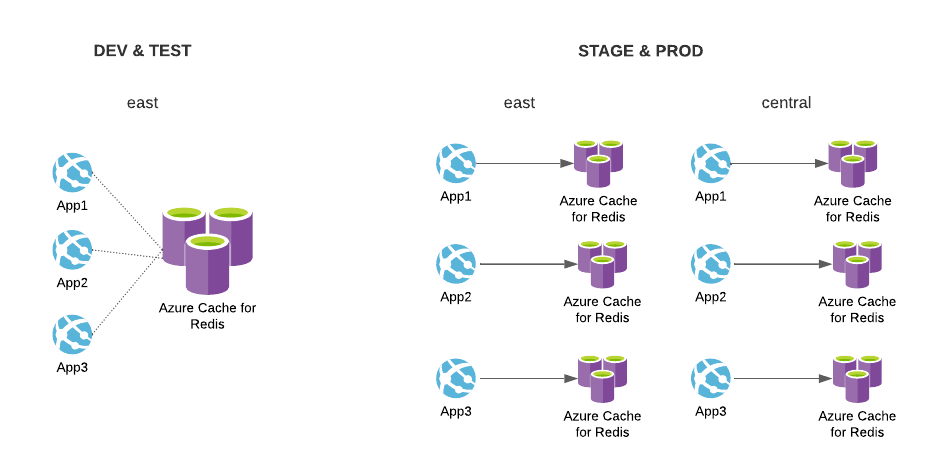
How do you maintain all this infrastructure reliably? Do you track all the applications for one tier in one state file? Or do you break them up and track them separately?

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
* Applications in development and test tier both share a redis cache
* Applications in Staging and production environments have individual redis caches
* Stage and production environments are deployed to two regions, east and central



Stage and Production tiers have up to 48 resources respectively; the diagram above only represents 3 applications and excludes some services. 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 presented 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 needing to declare it once. We also wanted to declare other variables and optionally override them on specific environments. With the Terraform CLI, there is no way to merge inputs, and share variables across environments. In addition, you cannot use expressions, functions, or variables in the terraform remote state configuration blocks, forcing you to either hardcode your configuration, or apply it dynamically through the cli; see this [issue](https://github.com/hashicorp/terraform/issues/13022).

We began 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 provision our terraform projects, what if we had a declarative way of defining how our terraform project was configured?

Terragrunt is a minimal wrapper around Terraform that allows you to dynamically assign and inherit remote state configurations, provider information, local variables, and module inputs for your terraform projects through a hierarchal folder structure with declarative configuration files. 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 do these important things:

* Define a configuration file to tell what remote state file to save based on application/tier (using folder structure)
* 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
* 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
* Allowed us to merge common input variables with tier specific input variables with desired precedence
* Allowed us to consistently name and create state files

Let’s consider the situation where we want to maintain the infrastructure for a system with two major components: an API, and a database solution. You also need to deploy dev, test, stage, and production environments for this system. Dev and Test environments are deployed to one region while stage and production environments are deployed to two regions.

To demonstrate how we might handle something like this with Terragrunt, we’ve created a preconfigured sample [repository](https://github.com/davoodharun/terragrunt-template). Now, although the requirements and scenario described above may not pertain to you, the preconfigured sample repository should give you a good idea of what you can accomplish with Terragrunt and the benefits it provides in the context of keeping your Terraform code organized. Also keep in mind that Terragrunt is unopinionated and allows you to configure it in several ways to accomplish similar results; we will only cover a few of the benefits Terragrunt provides, but be sure to check out their [documentation site](https://terragrunt.gruntwork.io/docs) for more information.

To get the most out of the code sample, you should have the following:

* [Terraform CLI](https://www.terraform.io/downloads)
* [Terragrunt CLI](https://terragrunt.gruntwork.io/docs/getting-started/install/)
* [An Azure Subscription](https://azure.microsoft.com/en-us/)
* [AZ CLI](https://docs.microsoft.com/en-us/cli/azure/)

Run through the setup steps if you need to. This will involve running a mini terraform project to provision a few resource groups in addition to a storage account to store your terraform state files.

The sample repo contains several top-level directories:

* /\_base\_modules
* /bootstrap
* /dev
* /test
* /stage
* /prod
* \_base\_modules - folder contains the top level terraform modules that your application will use. There are subfolders for each application type, the api and storage solution (/api and /sql). For example, there is a subfolder for the api which contains the terraform code for your api application, and one for sql which will contain the terraform code for your storage/database solution; take note of the main.tf, variables.tf, and outputs.tf files in each sub folder. Each application type folder will also contain a .hcl file that contains global configuration values for all environments that consume that respective application type
* [dev/test/stage/prod] – environment folders that contain sub folders for each application type. Each sub folder for each application type will contain Terragrunt configuration files that contain variables and inputs specific to that environment
* bootstrap – a small isolated terraform project that will spin up placeholder resource groups in addition to a storage account that can be used to maintain remote terraform state files

As mentioned above, there are several .hcl files in a few different places within this folder structure. These are Terragrunt configuration files. You will see one within each sub folder inside of the \_base\_modules directory, and you will see one in every sub folder within each environment folder as well. These files are how Terragrunt knows what terraform commands to use, where to store each applications’ remote state, and what variable files and input values to use for your terraform modules defined in the \_base\_modules directory. With this sample repository, global configurations are maintained in the /\_base\_modules folder and consumed by configurations in the environment folders.

Let’s go over some of the basic features that Terragrunt offers.

### Keeping your Remote State Configuration DRY

One thing that I immediately noticed about writing Terraform code, is that you can’t use variables, expressions, or functions within the [terraform configuration block](https://www.terraform.io/language/settings#terraform-block-syntax). You can override specific parts of this configuration through the command line, but there was no way to do this from code.

Terragrunt allows you to keep your backend and remote state configuration DRY by allowing you to share the code for backend configuration across multiple environments. Look at the /\_base\_modules/global.hcl file in addition to the /dev/Terragrunt.hcl file.

/\_base\_modules/global.hcl:

remote\_state {

  backend = "azurerm"

  generate = {

    path      = "backend.tf"

    if\_exists = "overwrite"

  }

  config = {

    resource\_group\_name  = "shared"

    storage\_account\_name = "4a16aa0287e60d48tf"

    container\_name       = "example"

    key            = "example/${path\_relative\_to\_include()}.tfstate"

  }

}

This file defines the remote state that will be used for all environments that utilize the api module. Take note of the ${path\_relative\_to\_include} expression. The remote state Terragrunt block that like this:

remote\_state {

backend = "s3"

config = {

bucket = "mybucket"

key = "path/for/my/key"

region = "us-east-1"

}

}

Is equivalent to terraform that looks like:

terraform {

backend "s3" {

bucket = "mybucket"

key = "path/to/my/key"

region = "us-east-1"

}

}

To inherit this configuration into a child sub folder or environment folder you can do this:

/dev/api/terragrunt.hcl

include "global" {

  path = "${get\_terragrunt\_dir()}/../../\_base\_modules/global.hcl"

  expose = true

  merge\_strategy = "deep"

}

The include statement above tells Terragrunt to merge the configuration file found at \_base\_modules/global.hcl with its own local configuration. The ${path\_relative\_to\_include} is a predefined variable that will return the relative path of the calling .hcl file, in this case /dev/api/terragrunt.hcl. Therefore, the resulting state file for this module would be in the example container at dev/api.tfstate. For the sql application in the dev environment, the resulting state file would be dev/sql.tfstate; look at the \_base\_modules/sql/sql.hcl file. For the api application in the test environment, the resulting state file would be test/api.tfstate.

Using Terragrunt, we only define the details of the remote state once, allowing you to cut down on code repetition. Read more about the [remote\_state](https://terragrunt.gruntwork.io/docs/reference/config-blocks-and-attributes/#remote_state) and [include](https://terragrunt.gruntwork.io/docs/reference/config-blocks-and-attributes/#include) blocks and how you can configure them by visiting the [Terragrunt documentation](https://terragrunt.gruntwork.io/docs). Pay special attention to merge strategy options, how you can override includes in child modules, and the specific limitations configuration inheritance in Terragrunt.

### Keeping your Terraform Configuration DRY

Merging of configuration files do not only apply to remote state configurations – you can also apply them to the sources and inputs of your modules.

In Terragrunt, you can define the source of your module (main.tf or top level terraform module) within the terraform block. Let’s consider the api application:

/\_base\_modules/api/api.hcl

terraform {

  source = "${get\_terragrunt\_dir()}/../../\_base\_modules/api"

  extra\_arguments "common\_vars" {

    commands = get\_terraform\_commands\_that\_need\_vars()

    required\_var\_files = [

    ]

  }

}

You’ll notice this is referencing a local path; alternatively, you can also set this to use a module from a [remote git repo or terraform registry](https://terragrunt.gruntwork.io/docs/reference/config-blocks-and-attributes/#terraform).

The api.hcl configuration is then imported as a configuration into each environment folder for the api application type:

include "env" {

  path = "${get\_terragrunt\_dir()}/../../\_base\_modules/api/api.hcl"

  expose = true

  merge\_strategy = "deep"

}

Include statements with certain merge strategies can be overwritten by configurations in child modules, allowing you to configure each environment separately, if needed.

Merging inputs before they are applied to your terraform module is also extremely helpful if you need to share variables across environments. For example, all the names of your resources in your project might be prefixed with a certain character set. You can define any global inputs this in the inputs section of the \_base\_modules/global.hcl file. Because Terragrunt configuration files are written in the HCL language, you can also utilize all the expressions and functions you are used to using in Terraform to modify or restructure input values before they are applied. Take a look at how we are defining the identifier input variable found in both sql and api modules:

Here is the terraform variable:

/\_base\_modules/api/variables.tf or /\_base\_modules/sql/variables.tf

variable "identifier" {

  type = *object*({

      primary = *string*

      secondary = *string*

      type = *string*

  })

}

Here is the primary property being assigned from the global env:

/\_base\_modules/global.hcl

...

inputs = {

    identifier = {

        primary = "EXAMPLE"

    }

}

...

Here is the secondary property being assigned from the dev/dev.hcl file:

/dev/dev.hcl

inputs = {

 identifier = {

     secondary = "DEV"

 }

}

And here is the type property being applied in the module folders

/\_base\_modules/sql/sql.env.tf

...

inputs = {

    identifier = {

        type = "SQL"

    }

}

/\_base\_modules/api/api.hcl

...

inputs = {

    identifier = {

        type = "API"

    }

}

All configurations are included in the environment configuration files with:

include "global" {

  path = "${get\_terragrunt\_dir()}/../../\_base\_modules/global.hcl"

  expose = true

  merge\_strategy = "deep"

}

include "api" {

  path = "${get\_terragrunt\_dir()}/../../\_base\_modules/api/api.hcl"

  expose = true

  merge\_strategy = "deep"

}

include "dev" {

  path = "../dev.hcl"

  expose = true

  merge\_strategy = "deep"

}

dependency "sql" {

  config\_path = "../sql"

    mock\_outputs = {

    database\_id = "temporary-dummy-id"

  }

}

would result in something like:

inputs = {

    identifier = {

      primary = "EXAMPLE"

secondary = "DEV"

type = "API"

    }

}

### Running Multiple Modules at once

You can also run multiple terraform modules with one command using Terragrunt. For example, if you wanted to provision dev, test, stage, and prod with one command, you could run the following command in the root directory:

terragrunt run-all [init|plan|apply]

If you wanted to provision the infrastructure for a specific tier, you could run the same command inside an environment folder (dev, test, stage etc.).

This allows you to neatly organize your environments, instead of having to maintain everything in one state file or trying to remember what variable, backend, and provider configurations to pass in your CLI commands when you want to target a specific environment.

It is important to note that you can maintain dependencies between application types within an environment ( between the sql and api application) and pass outputs from one application to another. Take a look at the dev/api environment configuration file

/dev/api/terragrunt.hcl

dependency "sql" {

  config\_path = "../sql"

    mock\_outputs = {

    database\_id = "temporary-dummy-id"

  }

}

locals {

}

inputs = {

    database\_id = *dependency*.sql.outputs.database\_id

    ...

}

Notice that it references the dev/sql environment as a dependency. The dev/sql environment uses the \_base\_modules/sql application so look at that module, specifically the outputs.tf file.

/\_base\_modules/sql/outputs.tf

output "database\_id" {

  value = azurerm\_mssql\_database.test.id

}

Notice that this output is being referenced as a dependency of the dev/api environment.

The client requirements described earlier in this post proved to be especially difficult to maintain without the benefit of being able to configure separate modules that depend on one another. With the ability to isolate different components of each environment and the ability to share their code and dependencies across environments, we were able to share the terraform module code for our redis cache and our web servers to configure each tier specifically without having to repeat code.