Terraform Modules Restructured

- ✓ Reusable
- ✓ Composable
- ✓ Automated Tests
- ✓ Confidence to Make Changes
- ✓ Implementation Abstraction

Credits

These slides are heavily influenced by the slides and talks from Yevgeniy Brikman from terragrunt.io

It goes hand in hand with the following talk:

https://blog.gruntwork.io/5-lessons-learned-from-writing-over-300-000-lines-of-infrastructure-code-36ba7fadeac1

Who Am I?

Ami Mahloof

Senior Cloud Architect at DoIT International.

LinkedIn Profile

Medium blog posts

At DoiT International, we tackle complex problems of scale which are sometimes unique to internet-scale customers while using our expertise in resolving problems, coding, algorithms, complexity analysis, and large-scale system design.

Outline

1. Introduction to Terraform
2. Module Anatomy
3. Modules Structure
4. Testing
5. Terraform Modules Best Practices
6. Migrating Existing Infrastructure Into New Code Structure

An Introduction To Terraform

Terraform is a tool for building, changing, and versioning infrastructure safely and efficiently

- Terraform can manage existing and new infrastructure
- Talk to multiple cloud/infrastructure providers
- Ensure creation and consistency
- Single DSL (Domain Specific Language) to express API agnostic calls
- Preview changes, destroy when needed
- Single source of truth infrastructure state
- Even order a pizza from Domino's

HashiCorp Configuration Language (HCL) Syntax

Just like in code a function has inputs (arguments) and outputs (attributes)

The following sudo code creates an EC2 instance from the given args

```
function create_ec2(name, type) {
   ec2 = aws.create_instance(name, type)
   print ec2.instance_ip
}
```

Mapping A Code To HCL Syntax

Inputs (arguments)

```
function create_ec2(name, type) {
   ec2 = aws.create_instance(name, type)
   print ec2.instance_ip
}
```

outputs (attributes)

Mapping A Code To HCL Syntax

name label

```
function create_ec2(name, type) {
   ec2 = aws.create_instance(name, type)
   print ec2.instance_ip
}

create_ec2("test", "t2.micro")
```

```
resource "aws_ec2_instance" "create_ec2" {
   name = "test"
   type = "t2.micro"
}

output "instance_ip" {
   value = aws_ec2_instance.create_ec2.ipv4_address
}
```

Mapping A Code To HCL Syntax

provider resource API

```
function create_ec2(name, type) {
   ec2 = aws.create_instance(name, type)
   print ec2.instance_ip
}

create_ec2("test", "t2.micro")
```

```
resource "aws_ec2_instance" "create_ec2" {
   name = "test"
   type = "t2.micro"
}

output "instance_ip" {
   value = aws_ec2_instance.create_ec2.ipv4_address
}
```

Mapping A Code To HCL Syntax

resource arguments

```
function create_ec2(name, type) {
   ec2 = aws.create_instance(name, type)
   print ec2.instance_ip
}

create_ec2("test", "t2.micro")
```

```
resource "aws_ec2_instance" "create_ec2" {
   name = "test"
   type = "t2.micro"
}

output "instance_ip" {
   value = aws_ec2_instance.create_ec2.ipv4_address
}
```

Mapping A Code To HCL Syntax

```
function create_ec2(name, type) {
   ec2 = aws.create_instance(name, type)
   print ec2.instance_ip
}

create_ec2("test", "t2.micro")
```

```
resource "aws_ec2_instance" "create_ec2" {
   name = "test"
   type = "t2.micro"
}

output "instance_ip" {
   value = aws_ec2_instance.create_ec2.ipv4_address
}
```

output values

Mapping A Code To HCL Syntax

resource blocks are for create API calls:

```
resource "aws_ec2_instance" "create_ec2" {...
```

data blocks are for get API calls:

```
data "aws_ec2_instance" "instance_data" {

   name = "test"

}


output "az" {

   value = data.aws_ec2_instance.instance_data.availbility_zone

}
```

Terraform State File

- JSON representation of known infrastrurcture state provisioned by terraform
- Stored in file or externally
- Locking (useful for team working on the same project or tasks)
- Source of truth for infrastructure

terraform.tfstate

```
{
 "version": 4,
 "terraform_version": "0.12.6",
 "serial": 18,
 "lineage": "b3bfb3fc-8417-cc89-d87f-6ab0008e2056",
 "outputs": {
   "group-id": {
     "value": "6521262259226325019",
     "type": "string"
   }
 },
 "resources": [
   {
     "mode": "data",
     "type": "template_file",
     "name": "filter_pattern",
     "provider": "provider.template",
     "instances":  ...
```

Monolithic Terraform

Issues with monolithic Terraform:

- *One/several huge files* - fear of making changes, no reusability, a mistake anywhere can break everything
- *Hard to find/debug* - variables and sections are harder to find when one needs to make a change
- *Guess work* - going back and forth between variables and resources just to understand what is required and what is the default
- *Slower development cycles* - increased time and effort needed to start working with it

10,000 ft View Approach

10,000 ft View Approach

Being able to quickly find what you’re looking for during development and debugging an issue is crucial when working with Terraform.

Since Terraform will combine all the files into a plan, we can use that to create smaller files with better visibility using the following simple module anatomy.

Module Anatomy

The module anatomy is a scaffold that provides better visibility and guidelines for developing, and working with Terraform modules.

Since Terraform compiles all resources in all files into an execution plan, we can use that to create better visibility and readability.

There are no hard-coded values, as each hard-coded value becomes a default variable, and every attribute is a variable.

Module Anatomy

Development of a module is done through the examples folder which holds a main.tf file with:

- Hard-coded values for all variables
- Lock down a specific version for a Terraform provider
- State location
- Terraform version

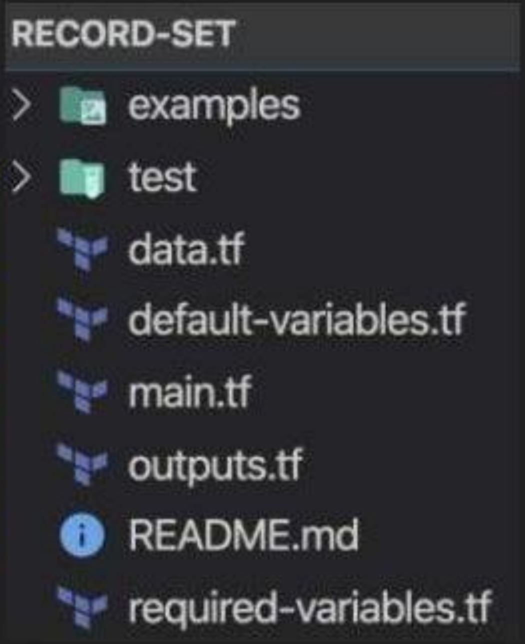
This will serve as a usage example when you finish development on the module

Module Anatomy

examples/main.tf

```
terraform {
 backend "s3" {
   region = "eu-west-3"
   bucket = "some-s3-bucket"
   key    = "dev/eu-west-3/infrastructure"
 }
 required_version = ">=12.6"
}
# This is where you setup the provider to use with the module
provider "aws" {
 version = "~> 2.0"
 region  = "us-east-1"
}

module "route53_record_name_cname_exmaple" {
 source      = "../"
 domain_name = "tf.domain.com"
 value       = "1.2.3.4"
}
```

Module Anatomy


RECORD-SET
- examples
- test
- data.tf
- default-variables.tf
- main.tf
- outputs.tf
- README.md
- required-variables.tf

- *examples* - a folder containing examples for usage
- *test* - Go Terratest folder
- *data.tf* - Terraform data sources
- *main.tf* - resources to be created
  if it's over 30 lines long, break it into files with names applied for resources i.e., `autoscaling.tf`, `ec2.tf`, etc.
- *outputs.tf* - outputs for the module
- *README.md* - clear inputs/outputs and description for the module as well as usage
- *default-variables* - variables with default values
- *required-variables* - variables with values that are required

Module Anatomy - Quick scaffolding with bash:

```
➜ export module_name="sample"

➜ mkdir -p $module_name/examples $module_name/test

➜ cd $module_name && touch \
     main.tf \
     versions.tf \
     default-variables.tf \
     required-variables.tf \
     outputs.tf \
     data.tf
```

3-Tier Modules Structure

The 3-Tier Module-Based Hierarchy Structure

Create and extend your own library of primitives building blocks (resources)

The 3-Tier Module-Based Hierarchy Structure

Build services from these primitives building blocks

The 3-Tier Module-Based Hierarchy Structure

Deploy end-to-end environments from services (live deployments)

The 3-Tier Module-Based Hierarchy Structure

What are the 3-Tier modules structure major benefits:

- Hide all lower level details to allow the end user to focus on building the infrastructure
- Each tier is tested providing a quicker development/debugging cycle
- Provides the confidence needed to make changes

Restructuring Existing Infrastructure

The goal is to isolate each (live) environment (dev, staging, production), then take each component in that environment and break it up into a generic service module, and for each generic service module break it into resource modules.

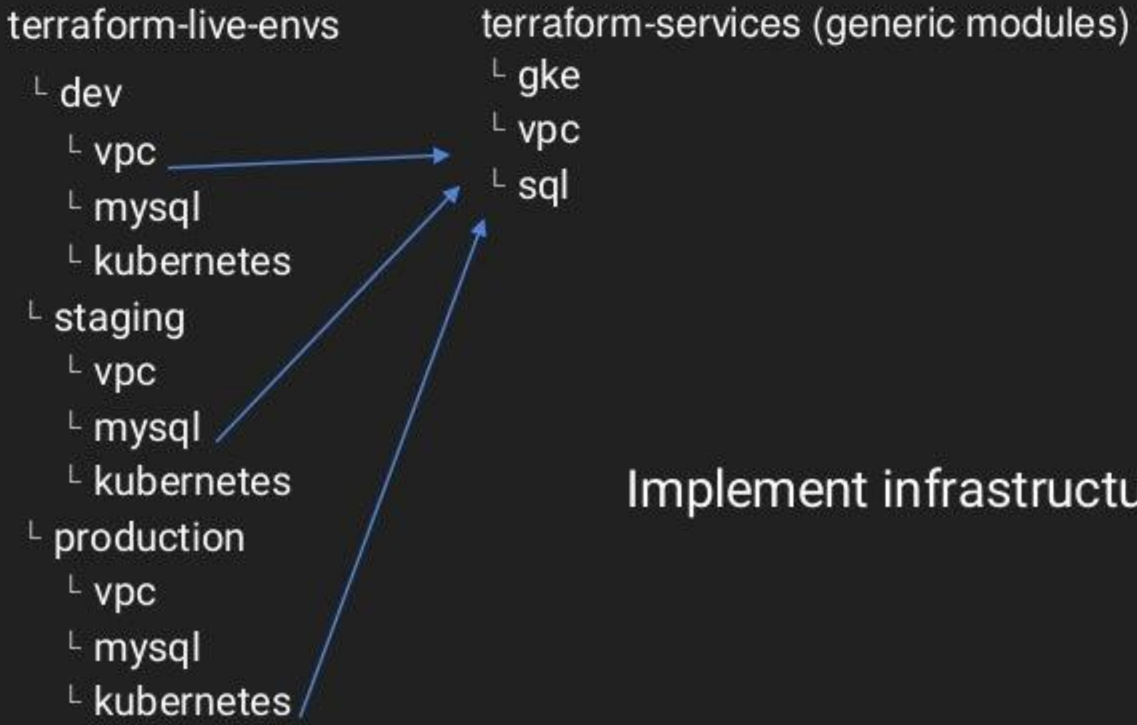
3-Tier Modules Structure

Break your architecture code down by live environment

```
terraform-live-envs
 └ dev
   └ vpc
   └ mysql
   └ kubernetes
 └ staging
   └ vpc
   └ mysql
   └ kubernetes
 └ production
   └ vpc
   └ mysql
   └ kubernetes
```

3-Tier Modules Structure

Then by service (infrastructure type)

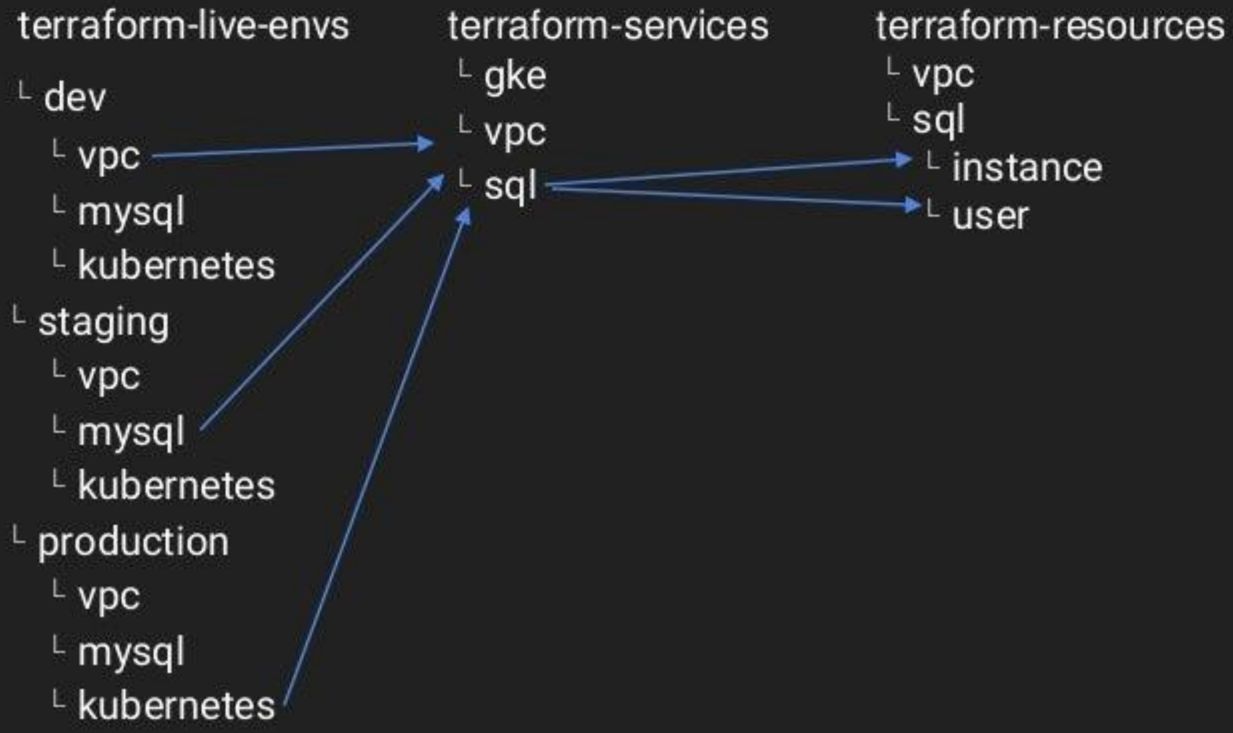


```
terraforms-live-envs                 terraform-services (generic modules)
 └ dev                                └ gke
   └ vpc                              └ vpc
   └ mysql                            └ sql
   └ kubernetes
 └ staging
   └ vpc
   └ mysql
   └ kubernetes
 └ production
   └ vpc
   └ mysql
   └ kubernetes
```

Implement infrastructure in modules

3-Tier Modules Structure

Build complex modules from smaller, simpler modules



Tier-1 Terraform Resources Modules

This is the lowest tier

`terraform-resources` is a folder containing modules with a single resource to be created

These resource modules are creating only one thing

These modules should have an `output.tf` file with outputs values providing information on the resource created

This information can be used to create hard dependencies between modules (required by the 2nd tier)

Tier-2 Terraform Services Modules

This is the middle tier

`terraform-services` is a folder containing modules combining resources modules together from the `terraform-resources` folder

Each service module is a generic service that can create multiple versions based on the variables passed in

Example; an SQL instance module can create postgreSQL or mySQL instance

Tier-3 Terraform-live-envs Modules

This is the top tier

The `terraform-live-envs` is a folder containing modules implementing the infrastructure that is deployed

These modules are usually built from the services modules but can also have resources modules mixed in

Every module attribute is a hard-coded value representing the value that is deployed

Tier-3 terraform-live-envs Modules

Each module should have one single file called `main.tf` that will contain:

- Terraform state block
- Modules with hard-coded values
- Locals block (shared variables between modules in this file)
- Outputs

This makes for a readable easy-to-use and maintainable deployment file

Terraform Remote State

By default, Terraform stores state locally in a file named `terraform.tfstate`

This does not scale because there's no locking or central location to work with Terraform in a team.

With remote state, Terraform writes the state data to a remote data store, which can then be shared between all members of a team. Terraform supports storing state in Terraform Cloud, HashiCorp Consul, Amazon S3, Alibaba Cloud OSS, and more.

Base Remote State

Often you need to create a base infrastructure for other deployments to use/read

Example:

You might create a VPC in one region only once, but you can deploy multiple services on that VPC.

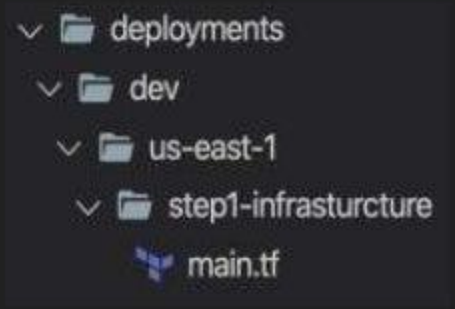
To do that, break your deployment into 2-steps:

- *step-1-infrastructure*
  Creates and outputs the VPC information (vpc_id, subnets etc..)

- *step-2-some-service*
  Accepts the remote state location (defined in step-1) as an input that will be used to read the output information for step-1, and creates the service on that VPC

Step 1 - Infrastructure

Base Remote State

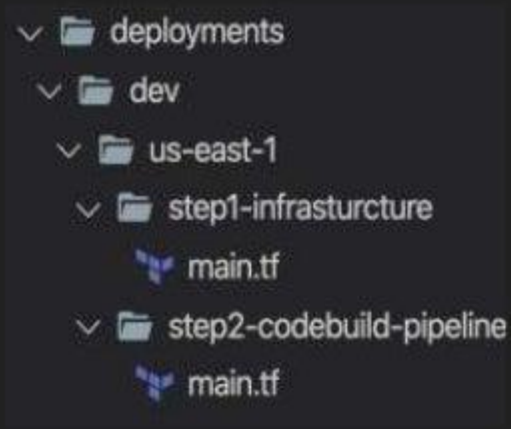


```
deployments
  dev
    us-east-1
      step1-infrasturcture
        main.tf
```

```
terraform {
 backend "s3" {
   region = "eu-west-3"
   bucket = "my-terraform-bucket"
   key    = "dev/eu-west-3/infrastructure"
  }
}

module "vpc" {
 vpc_name                   = "vpc-dev"
 vpc_availability_zones = ["eu-west-3a", "eu-west-3b"]
 vpc_private_subnets    = ["10.10.1.0/24", "10.10.2.0/24"]
 vpc_public_subnets     = ["10.10.11.0/24", "10.10.12.0/24"]
}

output "infra" {
  value  =  module.vpc
}
```

Step 2 - Services on Infrastructure

Base Remote State



```
deployments
  dev
    us-east-1
      step1-infrasturcture
        main.tf
      step2-codebuild-pipeline
        main.tf
```

```
terraform {
 backend "s3" {
   region = "eu-west-3"
   bucket = "my-terraform-bucket"
   # separate the state file location from the infrastructure state location
   key    = "dev/eu-west-3/deployment/code-pipeline/nodejs-app"
 }
}

module "code_pipeline" {
  source = "../codebuild-pipeline"

  # these are taken from step-1 terraform backend block
  terraform_state_store_region             = "eu-west-3"
  terraform_state_store_bucket             = "my-terraform-bucket"
  infrastructure_terraform_state_store_key = "dev/eu-west-3/infrastructure"
}


...
```

Base Remote State

deployment/step2-codepipeline/main.tf

```
module "code_pipeline" {
  source = "../codebuild-pipeline"
  ...
}
```

Codebuild-pipeline module

```
# Read the information from the remote state file of step 1 infrastructure
data "terraform_remote_state" "infra" {
 backend = "s3"
 config = {
   region = var.terraform_state_store_region
   bucket = var.terraform_state_store_bucket
   key    = var.infrastructure_terraform_state_store_key
 }
}

# Assign data to locals to read the data only once
locals {
 vpc_id = data.terraform_remote_state.infra.outputs.infra.vpc.vpc_id
 subnets = data.terraform_remote_state.infra.outputs.infra.vpc.subnets
}

# Use locals in the modules to get to the infrastructure data
module "pipeline" {
 source = "../../modules/pipeline"

 vpc_id = local.vpc_id
 ...
}
```

Refactoring existing terraform code

- Create a new bucket for the new terraform state to be stored at.
- Rewrite your new code into the 3-tiers modules (as illustrated above and detailed in the slides).
- Import each of the resources into your live-envs terraform code.
- terraform will show you the execution plan for the import operation:
  - values that exist in the deployed version but not in your code will be marked with a (-) minus sign for removal.
  - values that do not exist in the deployed version but do exists in your code will be marked with a (+) plus sign for addition.

The goal is to get a no change plan.

Code Versioning

Modules Git Repo

Create a separate git repository for each of the tiers, and an additional to hold the shared Go code for testing the modules:

- terraform-resources
- terraform-services
- terraform-live-envs
- terratest-common

Managing Module Versions

For the development process, it is recommended to use a relative path when working with the `source` attribute of a module

```
source = "../gcp/sql_instance"
```

You should change the `source` attribute value to a git repo when the module is ready for release

When a module is released, it should be tagged and added to the `source` attribute value using the `ref` argument

```
source = "git@github.com:unicorn/terraform-resources//gcp/sql_instance?ref=...v1.0.0"
```

Modules in Subdirectories

Since we are using modules in a repo, the module itself is in a subdirectory relative to the root of the repo.

A special double-forward-slash syntax is interpreted by Terraform to indicate that the remaining path after that point is a subdirectory.

```
source = "git@github.com:unicorn/terraform-resources//gcp/sql_instance?ref=v1.0.0"
```

The `ref` argument can be either a tag or a branch name

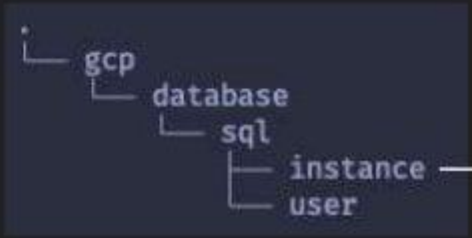
Modules Tagging Convention

Here is a recommended tagging convention for a module in the same repo:

```
<module-name>-v<semantic_versioning>
```

The module name should follow the directory structure you have in place.

Example:



```
.
└── gcp
    └── database
        └── sql
            ├── instance
            └── user
```

$\longrightarrow$ gcp-sql-instance-v1.0.0

Feel free to come up with your own tagging convention.

Terraform Modules Best Practices

Lock Down Terraform Version

Lock down the Terraform version that was used to create the module.

Place the following content in a file called `versions.tf` in the module:

```
terraform {
  required_version = ">= 0.12"
}
```

Using Provider in Module

```
provider "aws" {
 region  = var.region
 version = "~> 2.24"
}
module "this_module" {
 source "../"
}
 name = "unicorn"
}
```

Terraform provider is inherited in modules.

This means that a provider will be inherited by the modules your main module is calling.

Use an inline provider block inside your examples folder.

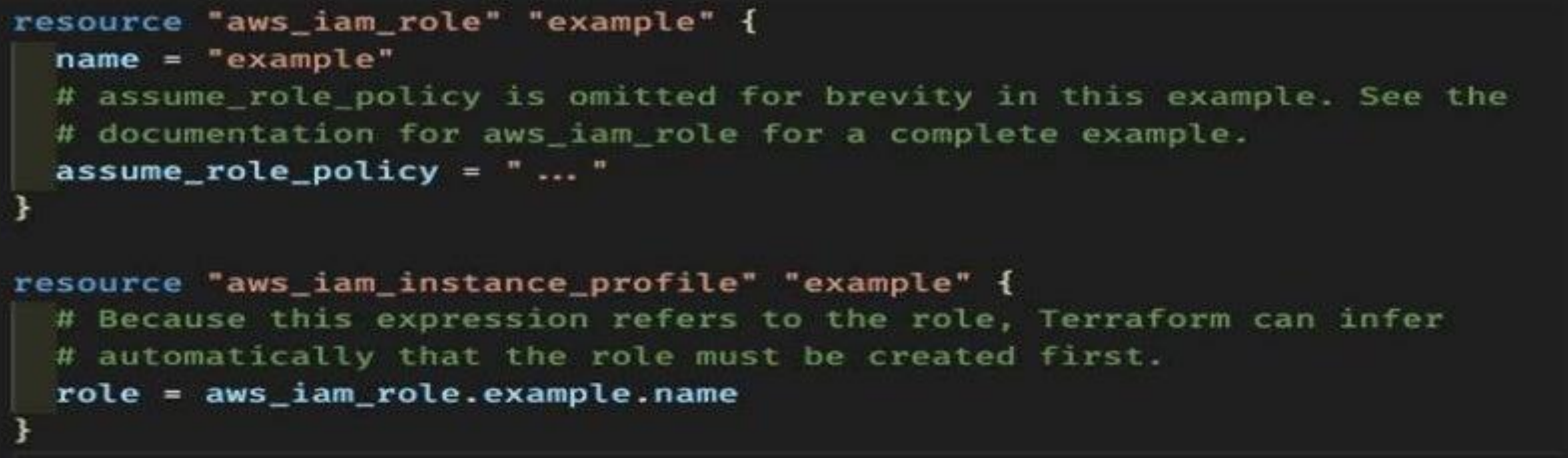
Only use the examples folder to test/develop your module.

Prefer Hard Dependencies Over depends_on

- depends_on doesn't work with modules (currently on 0.12.6)
- depends_on doesn't work with data sources
- There are some cases where depends_on would fail if the resource is it depends_on is conditionally created
- It's better to be consistent across all the code that needs dependencies

Prefer Hard Dependencies Over depends_on

Instead of using depends_on (i.e., resources), create a hard dependency in Terraform between resources:



```
resource "aws_iam_role" "example" {
  name = "example"
  # assume_role_policy is omitted for brevity in this example. See the
  # documentation for aws_iam_role for a complete example.
  assume_role_policy = " ... "
}

resource "aws_iam_instance_profile" "example" {
  # Because this expression refers to the role, Terraform can infer
  # automatically that the role must be created first.
  role = aws_iam_role.example.name
}
```

Prefer Hard Dependencies Over depends_on

Or between modules:

```
module "iam_role" {
  source = "../iam/role"

  # assume this module has an output variable called name
  name = "eample"
}

module "iam_policy" {
  source = "../iam/policy"

  # Because this expression refers to a module ouput, Terraform will
  # only create the outputs for a module after all resources have been created.
  role = module.iam_role.name
}
```

No Hard-coded Values

Each module should have the following files:

- ◎required-variables.tf
- ◎default-variables.tf

All of the resource attributes should be variables. If an existing module is hard-coded, you should move it into the default-variables.tf file.

You don’t have to use all attributes as documented in Terraform docs, you can add them as you go.

No tfvar Files

tfvar files are key=value lines of variables passed into a module.

The main problem with using this feature is that you can’t tell which variable belongs to which module.

This makes code usability hard to maintain and understand quickly.

Plugins Cache

Instead of having to download the same provider plugin to each module over and over again, you should set your plugin cache folder via an environment variable like so:

```
export TF_PLUGIN_CACHE_DIR="$HOME/.terraform.d/plugin-cache"
```

This will ensure that the provider plugin is linked to this folder and speed up running Terraform init on new modules.

Terraform State Management

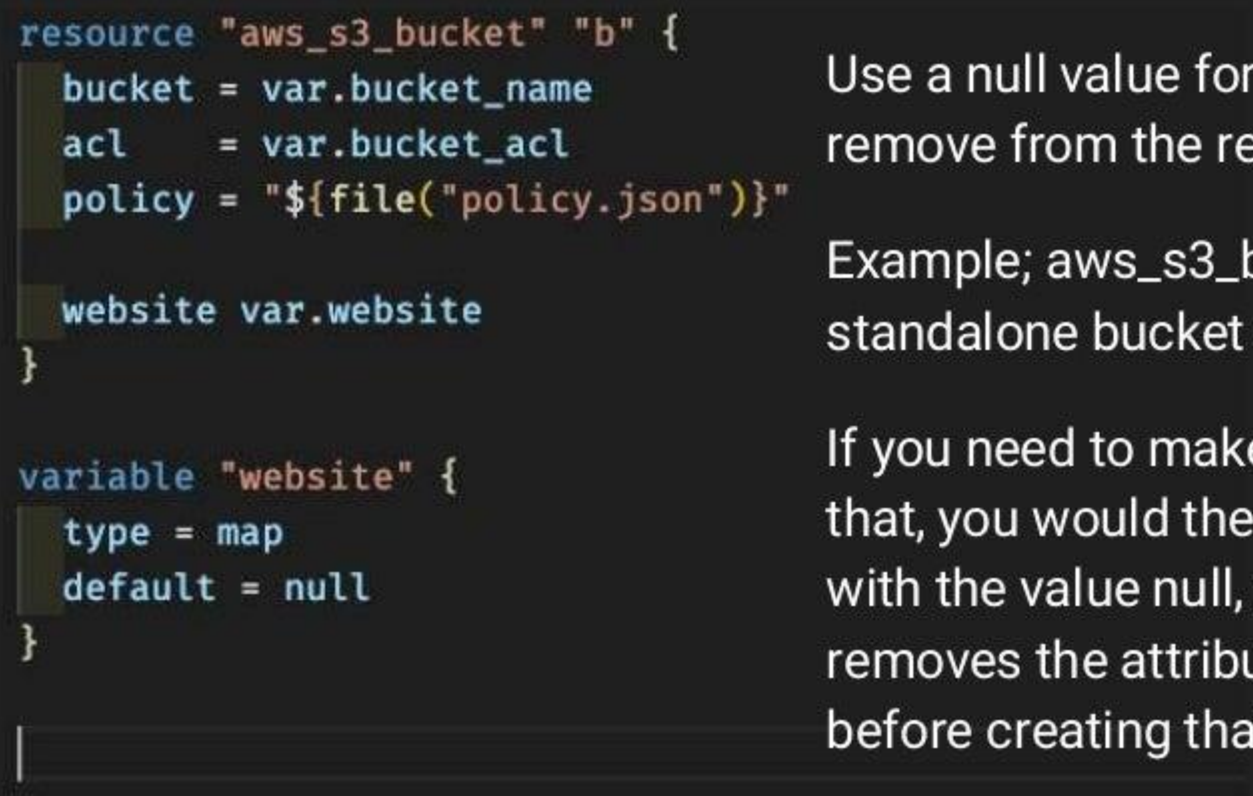
- Create a storage bucket (S3/GCS) per environment
  Do not use the same bucket for multiple envs

- Enable versioning on the bucket - this will serve as a backup if state is corrupted or can be used to compare concurrent executions

- Use prefix with the same folder structure you set in terraform-live-envs folder

- Use a separate prefix for infrastructure
  i.e., vpc-network should be put into *infrastructure/us-west2/blog-network*

Terraform Null Attribute



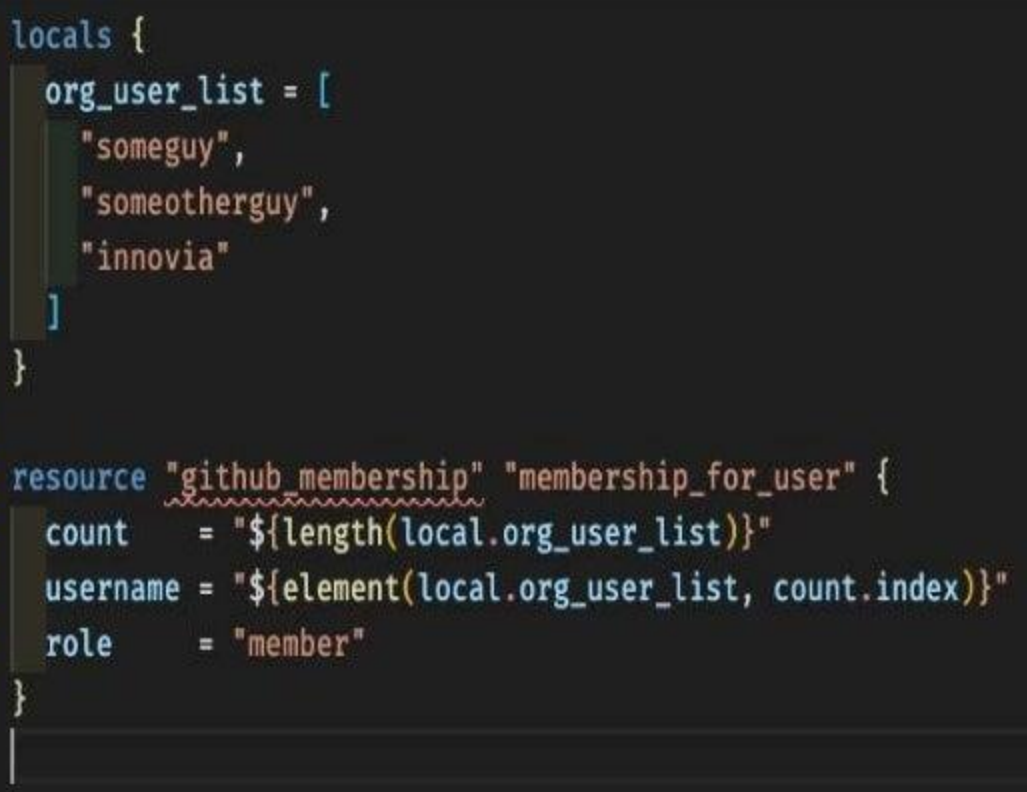
```
resource "aws_s3_bucket" "b" {
  bucket = var.bucket_name
  acl    = var.bucket_acl
  policy = "${file("policy.json")}"

  website var.website
}

variable "website" {
  type = map
  default = null
}
```

Use a null value for an attribute you want to remove from the resource.

Example; aws_s3_bucket can either be a standalone bucket or a website.

If you need to make a single resource for that, you would then make a default variable with the value null, which effectively removes the attribute from the resource before creating that resource.

Terraform and Lists



```
locals {
  org_user_list = [
    "someguy",
    "someotherguy",
    "innovia"
  ]
}

resource "github_membership" "membership_for_user" {
  count    = "${length(local.org_user_list)}"
  username = "${element(local.org_user_list, count.index)}"
  role     = "member"
}
```

Terraform will create a membership resource per user, but behind the scene, the count is also saved as an index in the state file.

If you remove someone from the middle of the list, the rest of the index will shift up, causing the rest to be added/recreated

In github this means delete user with its forks!

Cascade Variables and Outputs

Always cascade (copy over) the default and required variables along with the outputs to the next module tier, so the variable applied goes through all the modules.

Terraform Testing Using Terratest

Terratest is a Go library by terragrunt.io

It automates the creation of the IaC (Infrastructure as code), and then tests that the actual result is what you are expecting to get.

Once the tests are completed, Terratest will tear down and cleanup the resources it has created using Terraform destroy command.

Tips:

- You can use Terreatest with Docker, Packer and even helm charts!
- Use vscode with its Go extension for a quick coding with Go
- Learn Go interactively https://tour.golang.org

Typical Test Structure

```go
// TestVPCCreatedWithDefaults - test VPC is created without overriding any of the default variables
func TestVPCCreatedWithDefaults(t *testing.T) {
   terraformOptions := &terraform.Options{
       // The path to where our Terraform code is located
       TerraformDir: "../step1-infrastructure",

       // Variables to pass to our Terraform code using -var options
       Vars: map[string]interface{}{
           "region":  "us-east-1",
       },
   }
}
```

terraformOptions is a Golang struct defining the location of the code, as well as terraform variables for the execution.

Typical Test Structure

```go
func TestVPCCreatedWithDefaults(t *testing.T) {
   terraformOptions := &terraform.Options{
     ...
   }
   // At the end of the test, run `terraform destroy` to clean up any resources that were created
   defer terraform.Destroy(t, terraformOptions)
 ...
}
```

defer will run at the end of the test and call `terraform destroy` to clean up the resources created by this test.

Typical Test Structure

```
func TestVPCCreatedWithDefaults(t *testing.T) {
   terraformOptions := &terraform.Options{
     ...
   }
   // At the end of the test, run `terraform destroy` to clean up any resources that were created
   defer terraform.Destroy(t, terraformOptions)

   // Run `terraform init` and `terraform apply` and fail the test if there are any errors
   terraform.InitAndApply(t, terraformOptions)

   ...
}
```

Run `terraform init` followed by `terraform apply`

Typical Test Structure

```
func TestVPCCreatedWithDefaults(t *testing.T) {
   terraformOptions := &terraform.Options{
     ...
   }
   …
   vpc_id := terraform.Output(t, terraformOptions, "vpc_id"),
   validateVPC(t, vpcID)
}

func ValidateVPC(t *testing.T, vpcID string) {...}
```

Validate it works as expected

Terratest Built-in Functions

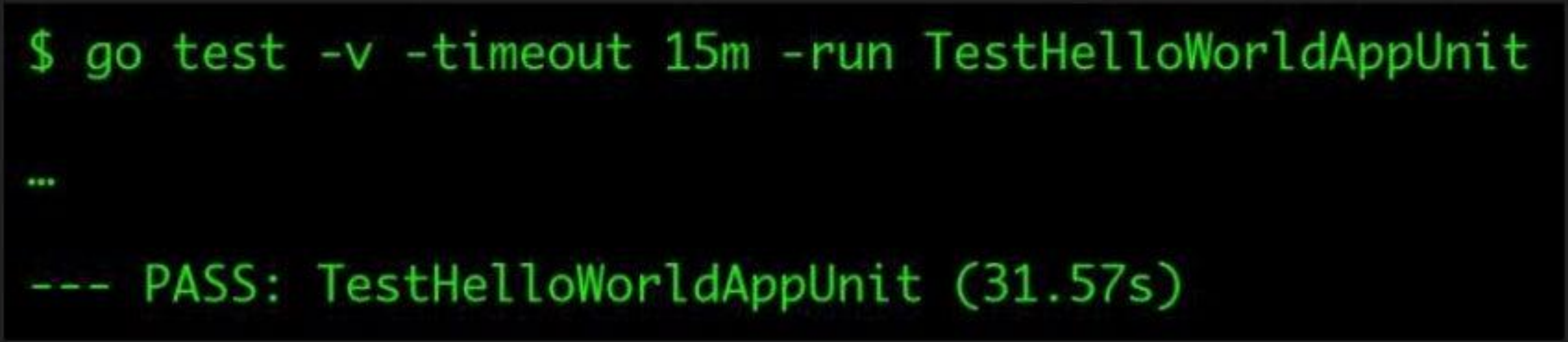
```
// Get IPs of servers
aws.GetPublicIpsOfEc2Instances(t, ids, region)

// Make HTTP requests in a retry loop
http.GetWithRetry(t, url, 200, expected, retries, sleep)

// Run command over SSH
ssh.CheckSshCommand(t, host, "vault operator init")
```

Terratest has many built-in functions to check your infrastructure - but it’s relatively easy to extend and write your own.

Validate Function Example

```go
func validate(t *testing.T, opts *terraform.Options) {
  url := terraform.Output(t, opts, "url")
  http_helper.HttpGetWithRetry(t,
    url,              // URL to test
    200,              // Expected status code
    "Hello, World!", // Expected body
    10,               // Max retries
    3 * time.Second  // Time between retries
  )
}
```

Running the Test



```
$ go test -v -timeout 15m -run TestHelloWorldAppUnit

…

--- PASS: TestHelloWorldAppUnit (31.57s)
```

Run Go test.
You now have a unit test you can run after every commit!

Note:
Go tests timeout after 10 minutes, so make sure you set a greater timeout to allow infrastructure to be created

Terratest Testing Techniques

Reproducible Test Results:

Running tests does not take remote state configuration, thus the local module will end up having a local state file, which can lead to a stale test results.

Don't put the hard-coded path to the module in the test like so:

```
terraformOptions := &terraform.Options{
        // The path to where our Terraform code is located
        TerraformDir: "../step1-infrastructure",
        ...
```

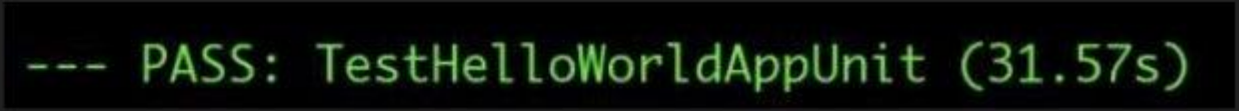
Use a built-in function in Terratest that will copy the module to a temporary folder and return the path to that folder:

Terratest Testing Techniques

```
import (
   test_structure "github.com/gruntwork-io/terratest/modules/test-structure"
)

func TestRoute53CNAMERecord(t *testing.T) {
   rootFolder := "../../.."
  modulePathRelativeToRootFolder := "terraform-aws-route53-resource/record-set"
  terraformTempDir := test_structure.CopyTerraformFolderToTemp(
        t,
        rootFolder,
        modulePathRelativeToRootFolder,
   )
  terraformOptions := &terraform.Options{
     TerraformDir: tempTestFolder,
  …
}
```

Multiple Tests Within a Test (subtests)

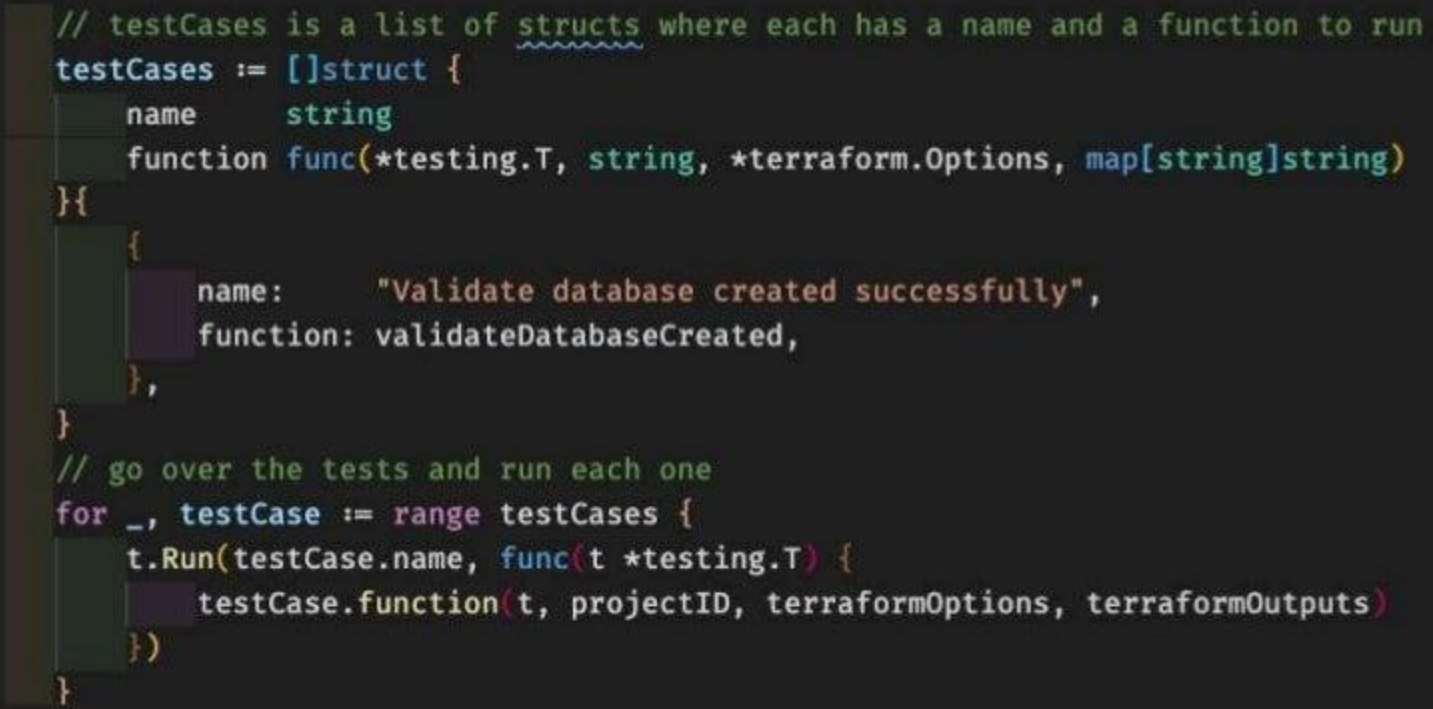
In Go, a test will only report PASS or FAILED on the function name:



```
--- PASS: TestHelloWorldAppUnit (31.57s)
```

Often you will need a few tests to check for the same function and reflect that in the test outputs. This concept is called subtests.

Multiple Tests Within a Test (subtests)



```
// testCases is a list of structs where each has a name and a function to run
testCases := []struct {
    name     string
    function func(*testing.T, string, *terraform.Options, map[string]string)
}{
    {
        name:     "Validate database created successfully",
        function: validateDatabaseCreated,
    },
}
// go over the tests and run each one
for _, testCase := range testCases {
    t.Run(testCase.name, func(t *testing.T) {
        testCase.function(t, projectID, terraformOptions, terraformOutputs)
    })
}
```

Use the following to run multiple validations for the same test:

Terratest Error Handling

Just about every method foo in Terratest comes in two versions: **foo** and **fooE** (e.g., `terraform.Apply` and `terraform.ApplyE`).

- foo: The base method takes a `t *testing.T` as an argument. If the method hits any errors, it calls `t.Fatal` to fail the test.
- fooE: Methods that end with the capital letter E always return an error as the last argument and never call `t.Fatal` themselves. This allows you to decide how to handle errors.

Terratest Error Handling

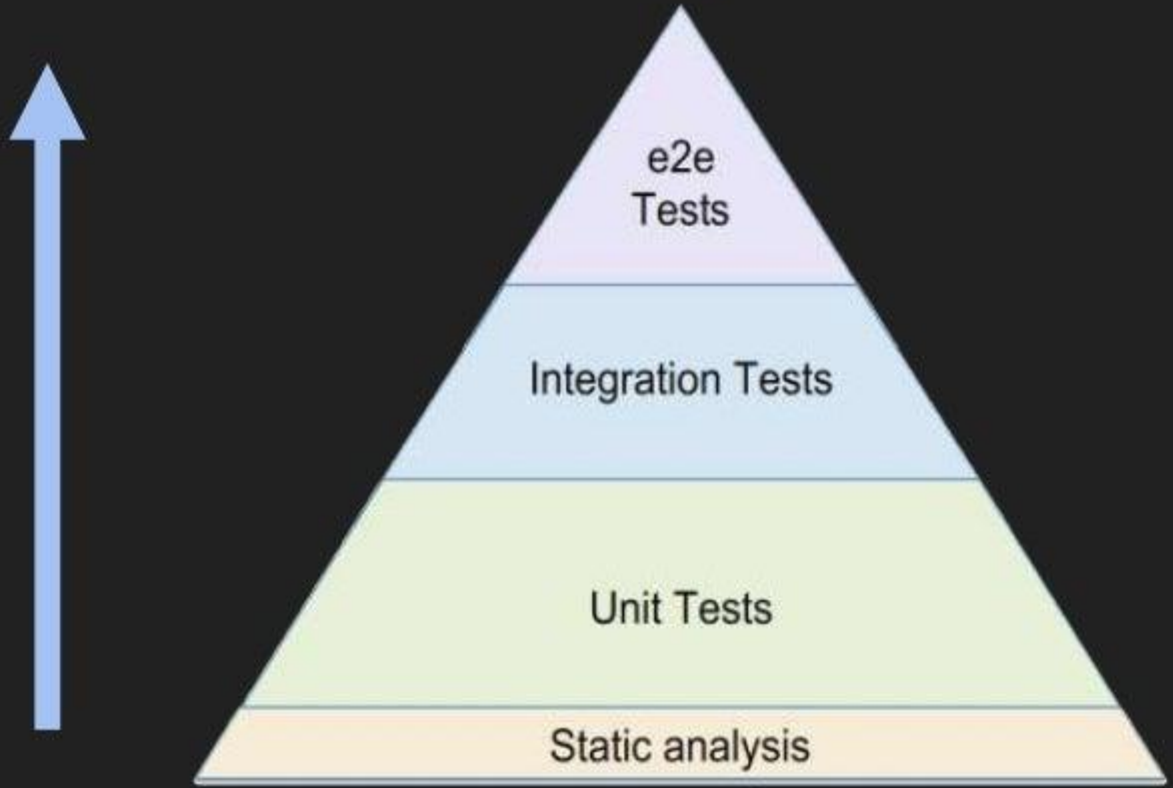
You will use the base method name most of the time, as it allows you to keep your code more concise by avoiding `if err != nil` checks all over the place:

```
terraform.Init(t, terraformOptions)
terraform.Apply(t, terraformOptions)
url := terraform.Output(t, terraformOptions, "url")
```

In the code above, if Init, Apply, or Output hits an error, the method will call `t.Fatal` and fail the test immediately, which is typically the behavior you want. However, if you are expecting an error and don't want it to cause a test failure, use the method name that ends with a capital E:

```
if _, err := terraform.InitE(t, terraformOptions); err != nil {
 // Do something with err
}
```

The Test Pyramid

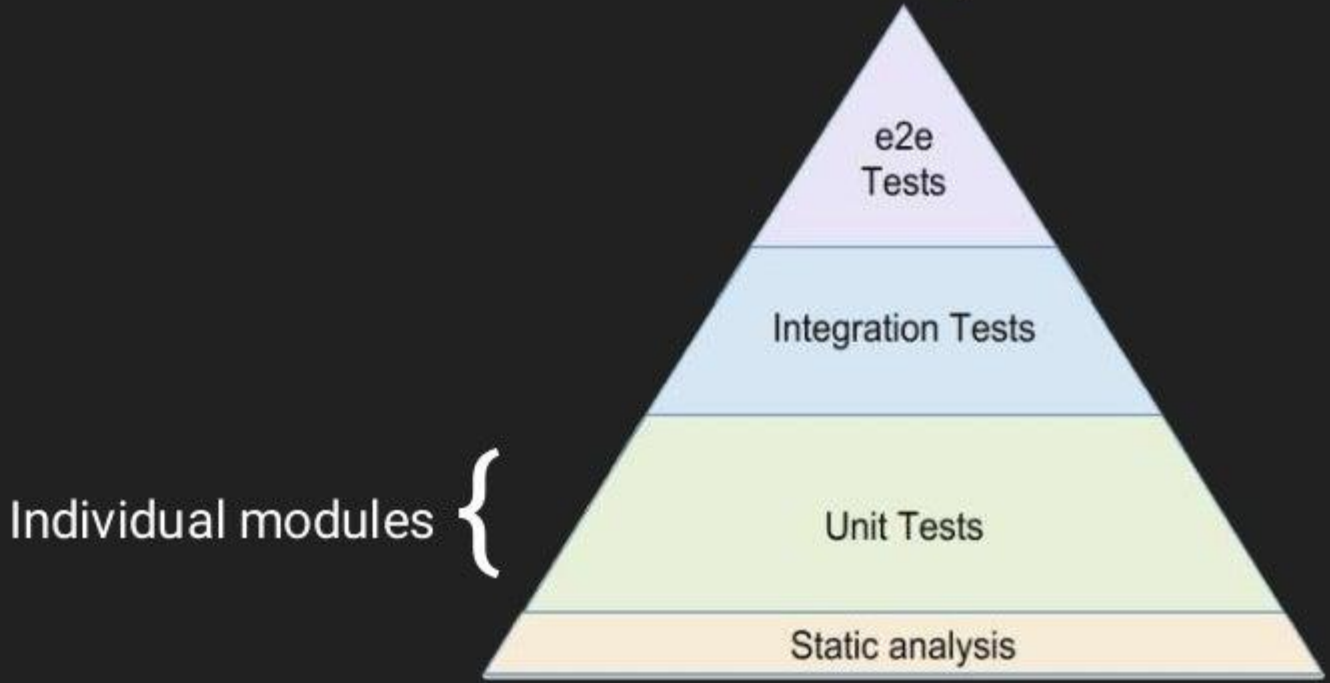


As you go up the pyramid, tests get more expensive, brittle and slower

The Test Pyramid

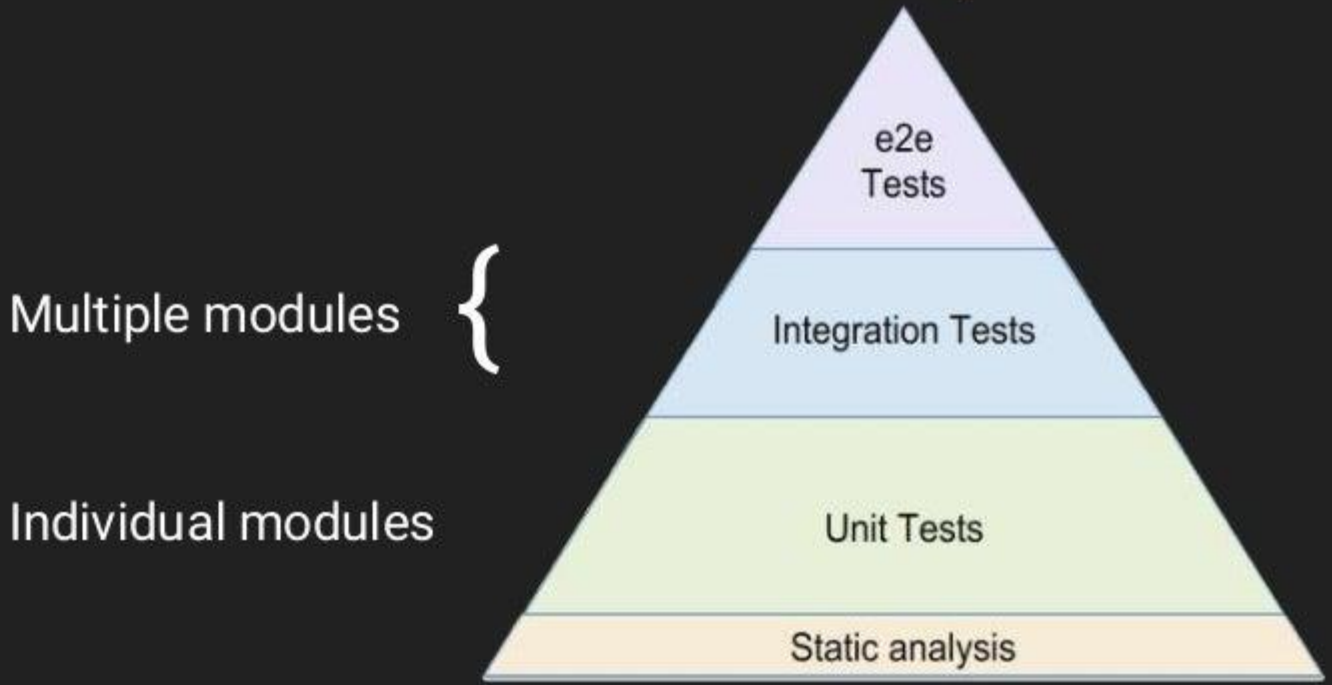
How the test pyramid works with infrastructure code:

The Test Pyramid



Lots of unit tests:
test individual sub-modules (keep them small!)
and static analysis (TFLint, Terraform validate)

The Test Pyramid



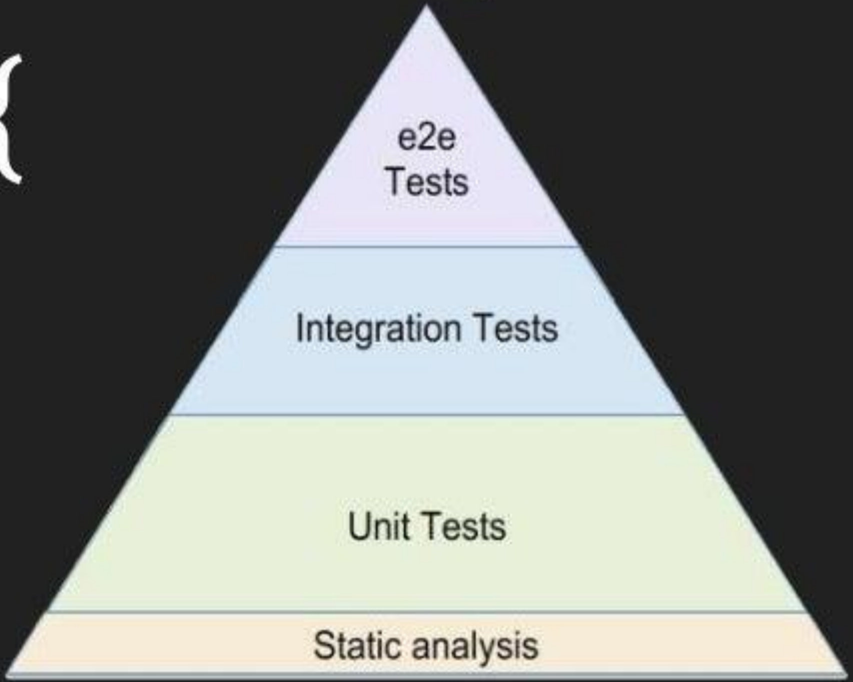
Fewer integration tests:
test multiple sub-modules together

The Test Pyramid

Entire stack {

multiple modules

Individual modules



A handful of high value E2E tests:
test entire environments (stage, prod)

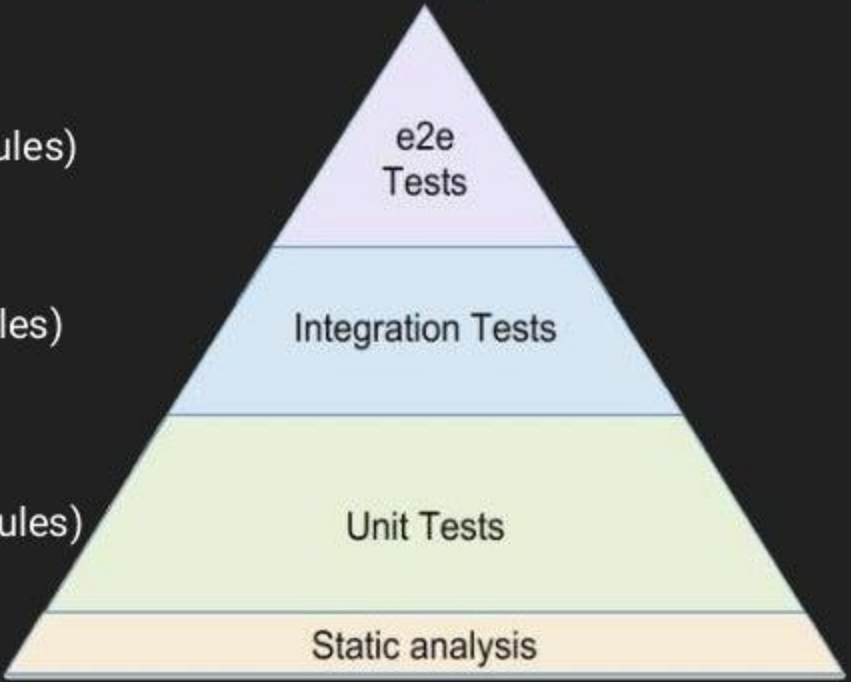
The Test Pyramid

60-240 minutes
(terraform-live-envs modules)

5-60 minutes
(terraform-services modules)

1-20 minutes
(terraform-resources modules)

1-60 seconds
(tflint/terraform validate)



Note the test times!
This is another reason to use small modules

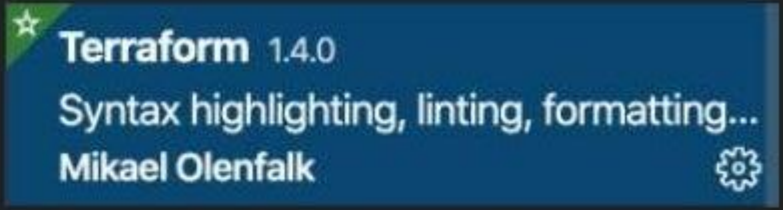
Terratest Best Practices

To minimize the downsides of testing infrastructure as a code on a real platform, follow the guidelines below:

1. Unit tests, integration tests, end-to-end tests
2. Testing environment
3. Namespacing
4. Cleanup
5. Timeouts and logging
6. Debugging interleaved test output
7. Avoid test caching
8. Error handling
9. Iterating locally using Docker
10. Iterating locally using test stages

Pro Tips

- Use VSCode for Terraform and Go integration
  (you can opt to use JetBrains intelliJ too too or Vim)
- VSCode extensions:



Terraform 1.4.0
Syntax highlighting, linting, formatting...
Mikael Olenfalk

Once installed you need to open the command palette (View -> Command Palette) and type: “Terraform: Enable Language Server” which will prompt you for installing the latest package for the terraform 0.12 support.

Questions?