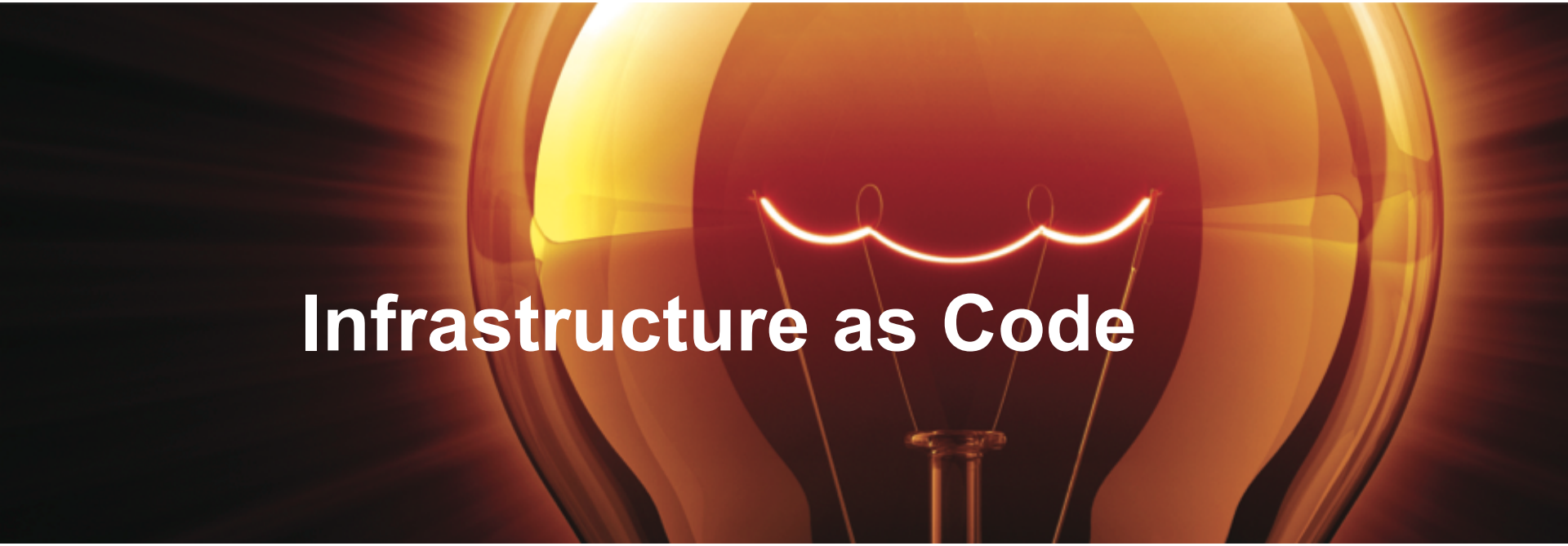
AWS Intensive

WELCOME!

Develop Intelligence

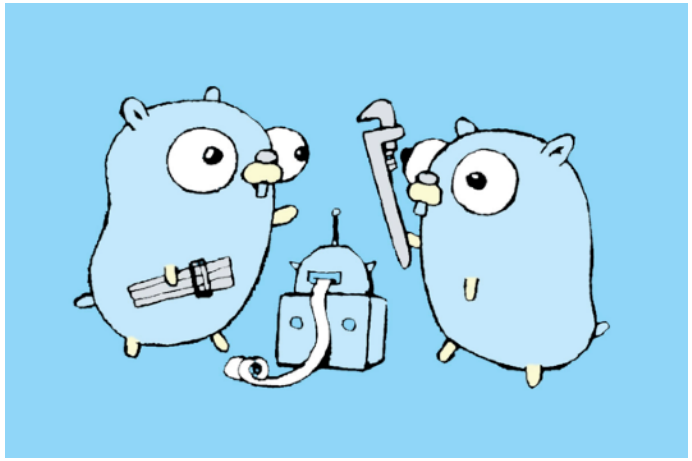
John Kidd



Infrastructure as Code

Cloudformation

- So the foundation of all AWS applications is CLOUDFORMATION templates.
- These are JSON OR YAML representations of everything running on your AWS platform.
- The next slide will show a very basic format around how we create a single EC2 instance with a cloudformation template.
- It will be in 3 parts



Cloudformation Template

```
{
"AWSTemplateFormatVersion" : "2010-09-09",
"Description" : "A sample template",
"Resources" : {
   "MyEC2Instance" : {
      "Type" : "AWS::EC2::Instance",
      "Properties" : {
         "ImageId" : "ami-0ff8a91507f77f867",
         "InstanceType" : "t2.micro",
         "KeyName" : "testkey",
         "BlockDeviceMappings" : [
            { "DeviceName" : "/dev/sdm",
              "Ebs" : {
                 "VolumeType" : "io1",
                 "Iops" : "200",
                 "DeleteOnTermination" : "false",
"VolumeSize" : "20" }
              } ]
         }
      }
  } }
```

- **Description:** Self explanatory: string that contains a description of what the CloudFormation document is intended for
- **Resources:** The names of resources- TYPE is where you put in the AWS namespace separated by double colons
- **Properties:** Details like the image that we'll be using
- **BlockDeviceMappings:** Further details on the image

Cloudformation Template stacks

- The Cloudformation template is not set up on a one to one relationship with each asset that you create… in other words it's not like if I needed an EC2 instance and a DynamoDB table I would create TWO templates.
- Instead the Cloudformation template is set up in **stacks**- which are a collection of aws resources used for an application.
- Ideally it should be **one** cloudFormation template per **application.**

Cloudformation stack examples

- Let's go through a few examples here of Cloudformation templates that are used to create both STACKS and individual resources:
  - LAMP Stack
  - Ruby On Rails
  - Scalable LAMP with RDS
  - Sample EC2 Instance
  - IAM creation

CloudFormation Summary

- Obviously Cloudformation contains a **lot** of details about resources, permissions, etc. Some of those JSON documents go on for about a thousand lines for a single resource.
- The **stacks** add levels of complexity to the documents as well- making them pretty unmanageable.
- Do we need to learn Cloudformation end-to-end? Not really…but understand this: ***whatever manager we use for infrastructure as code for AWS (serverless, terraform, etc)- they ultimately compile down to cloudformation templates***

Cloudformation Summary (cont'd)

- Essentially- it doesn't **hurt** to learn something about cloudformation (and there are a lot of resources out there to do so)…but it's a bit like learning binary in order to increase your understanding of python: the most important part here is that you understand that what your infrastructure manager is compiling down to is basically a cloudFormation template.
- Your AWS application then runs off of that template.

Terraform

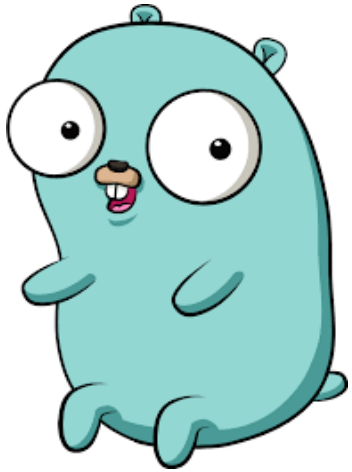
- As we briefly touched on before the break- terraform is an efficient way to store your infrastructure as code.
- The idea here is that all of the pieces of your application are ephemeral; you will be adding parts of your application around the code instead of putting the code INTO parts of your application.

Terraform

- There are three basic parts to every terraform deployment (and military operations, interestingly enough) :
  - **Create(Write)**: The part where you write the .tf file (JSON structured).
  - **Plan**: Preview the changes that you want to make to your stack before they are applied (also checks and lints your .tf file)
  - **Execute (Apply):** Obvious- this is where you make the changes/deletions/updates to your stack.

Terraform working in a team

- Terraform makes a big deal out of using enterprise when working in large teams to manage plans…but it's actually a lot simpler than that.
- My recommendation is to just publish plans to commits along with terraform (.tf) files and leave everything open for review as a PR.



Terraform files

- A couple of quick things to note before we dive into actual .tf syntax:
  - The command to **plan** or **apply** any terraform files will, when invoked, load all terraform files **in a directory in alphabetical order.** This can lead to some confusion so remember this rule.
  - The exception to the above rule is **override files** which we'll get to later.
  - Loaded files are **appended** to each other– as opposed to **merging** with each other.
  - **Overrides** merge instead of append.
  - Order of variables, resources, etc don't matter- configurations are declarative

Terraform Syntax

- The syntax used here is native to Hashicorp and is known as **Hashicorp Configuration Language (HCL)**.
- Terraform can also read JSON formatted syntax files but we won't be using those here (though definitely an option!)

Terraform syntax

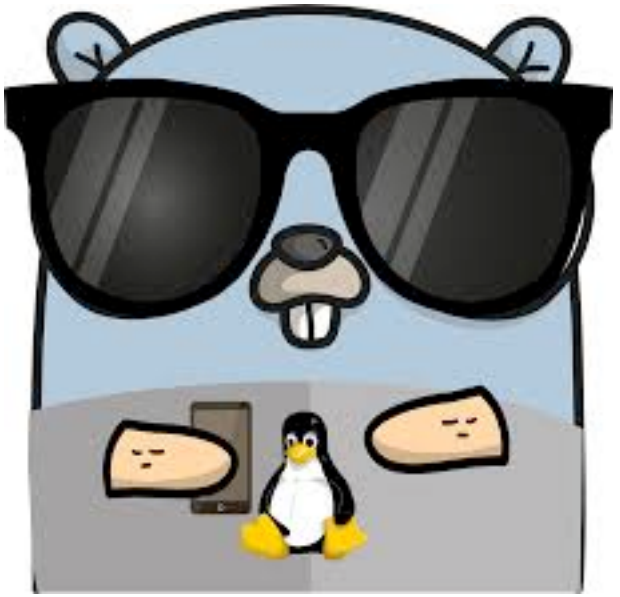
- Single line comments start with #
- Multi-line comments are wrapped with /* and */
- Values are assigned with the syntax of key = value (whitespace doesn't matter). The value can be any primitive (string, number, boolean), a list, or a map.
- Strings are in double-quotes.
- Strings can interpolate other values using syntax wrapped in ${}, such as ${var.foo}. The full syntax for interpolation is documented here.
- Multiline strings can use shell-style "here doc" syntax, with the string starting with a marker like <<EOF and then the string ending with EOF on a line of its own. The lines of the string and the end marker must *not* be indented.

Terraform Syntax (cont'd)

- Numbers are assumed to be base 10. If you prefix a number with 0x, it is treated as a hexadecimal number.
- **Boolean values**: true, false.
- Lists of primitive types can be made with square brackets ([]). Example: ["foo", "bar", "baz"].
- **Maps can be made with braces ({}) and colons (:):**
  - { "foo": "bar", "bar": "baz" }. Quotes may be omitted on keys, unless the key starts with a number, in which case quotes are required. Commas are required between key/value pairs for single line maps. A newline between key/value pairs is sufficient in multi-line maps.

Terraform Syntax Interpolations

- **Interpolation:** Similar to command line interpolation: *${var.hereIsMyInterpolation}*
- Double $$ will make interpolation literal- so *$$ {var.thing}* is *${var.thing}*
- For variable interpolation use *$ {**var**.hereIsMyVariable}*
- Note the var in front there.
- This is how you do **string interpolation**



Terraform Syntax: Variables

- Variable declarations have three arguments:
  - **Type:** (Optional) The data type of the values of the variable. Possible entries are “string”, “list”, “map”
  - **Default:** (Optional) So if you don’t instantiate the variable with a value (i.e: hereismyvariable=“totallyavariable”) then this will be the default.
  - **Description:** (Optional) So this is a human readable description of what the variable IS. Please use this.

Terraform: Instantiate String

- SO to create a string (single line:

```
variable "key" {
        type = "string"
        default = "value"
        }
```

- Multi-line:

```
variable "long_key" {
        type = "string"
        default = <<EOF This is a long key. Running over several lines. EOF }
```

Terraform: Instantiate List

- To create a list you use a similar setup to strings:

```
variable "users" {
        type = "list"
        default = ["admin", "ubuntu"]
}
```

And to instantiate a part of a list- similar to code:

`${var.users}` or `${var.users[1]}` for “ubuntu”

Terraform instantiate MAPs

- To create a MAP you use a similar setup:

```
variable "images" {
        type = "map"
        default = {
                "us-east-1" = "image-1234"
                "us-west-2" = "image-4567" }
}
```

And to instantiate a map - similar to code:

${var.images} or ${var.images[”us-east-1”]} for “ubuntu”

Terraform Environment Variables

- You can also pass environment variables in to terraform by appending **TF_VAR** to the front of the variable and then defining it.
- So, for example:
  - **export TF_VAR_stringvar=somestring**
  - **export TF_VAR_alist='["here", "is", "a", "list"]'**
  - **export TF_VAR_amap='{"best":"movie", "is":"theroom"}'**

  And then you get those as variables in the .tf file (and they will override default values)

Variable file

- The other option we have is to store all of our variables in a separate file- which is the preferred method for this class (makes it easier to find/change them).
- There is a sample of this usage in ./sample_files/variables.tf. The setup is the same and there are no special characters/titles that you have to add in to get everything working.

Variable (another level up)

- So…considering the fact that, if you remember, there is no “loading order” for file resources in terraform we can now use multi-level variable files to create templates that read data from a variables file.
- What I mean here is that we can have a basic template file (released by the dev/ops team) that uses a variable file kept locally by developers to fill in specifics on resources. There is an example of this in **./sample_files/resource.tf**

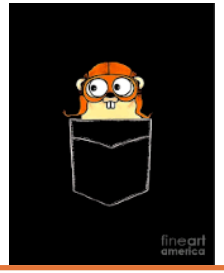
Managing resource.tf files

- So the idea here is that the devops team can:
  - Use this **resource** file as a template
  - Put the template into the version control system (git)
  - Have the developers download the template and **only edit the variables.tf resource**
  - Keep the local variables.tf file locally **only** (or in a separate, closed repo).

- This means that developers can (if you trust them) deploy resources using *only* variables files.

Output variable files

- Finally we have output variables. These are basically variables that contain information **about** our deployed resources that we can then output.
- Some examples of these include, for example:
  - The **public ip** of a deployed EC2 instance.
  - The **hostname** of a deployed server
  - The **private ip** of a deployed EC2 instance
- These output variables will be shown **after** terraform has applied and created the resources.



Output variables (cont'd)

- So along with showing at the end of a deployment you can always get the golang output variables with the command **terraform output public_ip** and the display will show that output variable for your deployed resource.
- An example of an output.tf file can be found in **./sample_files/output.tf**

Functions that work in Terraform

- Terraform documents also allow you to use functions within documents to do things like basic mathematical functions and lookups.
- I have included a **complete** list of these (as of terraform 0.11 anyway) in the **./sample_files/functions_in_terraform.txt**.
- I'm going to go over a few of the more commonly used ones here.

Terraform Common functions

- contains(list, element) - Returns *true* if a list contains the given element and returns *false* otherwise. Examples: contains(var.list_of_strings, "an_element")

- join(delim, list) - Joins the list with the delimiter for a resultant string. This function works only on flat lists. Examples:
  - join(",", aws_instance.foo.*.id)
  - join(",", var.ami_list)

Terraform common functions

- rsadecrypt(string, key) - Decrypts string using RSA. The padding scheme PKCS #1 v1.5 is used. The stringmust be base64-encoded. key must be an RSA private key in PEM format. You may use file() to load it from a file.
- timestamp() - Returns a UTC timestamp string in RFC 3339 format. This string will change with every invocation of the function, so in order to prevent diffs on every plan & apply, it must be used with the ignore_changes lifecycle attribute.

Terraform common functions

- lookup(map, key, [default]) - Performs a dynamic lookup into a map variable. The map parameter should be another variable, such as var.amis.
  If key does not exist in map, the interpolation will fail unless you specify a third argument, default, which should be a string value to return if no key is found in map. This function only works on flat maps and will return an error for maps that include nested lists or maps.

Terraform ternary operators

- Needless to say there are **loads** more of these functions and I encourage you to use the .txt file in sample_files as a guide (or google) when writing tf files. It's like the seeds of a programming language in there.
- Finaly there are ternary operators: so things like
  - count = "${var.private_zone == "true" ? 1 : 0}" will work

Terraform Overrides

- So if you recall we talked about terraform files using **append** instead of **merge** with one exception: **overrides**.
- Basically an override file is **merged** after all other terraform files are appended and can therefore be used to overwrite original variable values.
- To create an override file just name the file **override.tf** or end it with **_override.tf** and terraform will take it from there.

Terraform override example

So for the original tf file you might have:

```
resource "aws_instance" "web" {
        ami = "ami-408c7f28"
}
```

And in the override file:

```
resource "aws_instance" "web" {
        ami = ”ami-123v8fnd"
}
```

Terraform Providers

- So the beginning of any terraform document will start with a **provider**. The **provider** manages the CRUD operations of a **resource**.
- A terraform document **can** have multiple providers for things like different **regions** for different **resources- SO** if you had a dynamodb table in **eu-west-1** and you wanted to provision a lambda in **eu-west-2** you would use multiple **providers** to do this.

```
provider "aws" {
        access_key = "foo"
        secret_key = "bar"
        region = ”ew-west-1" }
```

Terraform Providers (cont'd)

- What you list as your terraform provider is the hint for terraform to know what plugin it needs to download in order to operate.
- In this class we will be dealing with **aws** as our provider (though multiple providers **can** be created if you have assets in different regions- so **aws:eu-west-1** and **aws:eu-west-2** can be considered different providers…but just one plugin).

Terraform: Resources

- So finally we get to provisioning resources…which is the main purpose of terraform.
- The **resource** block of a terraform document creates an AWS resource of a given TYPE (first parameter) and NAME (second parameter)… so something like:

```
resource "aws_instance" ”examplenamething" {
        ami = "ami-408c7f28"
        instance_type = "t1.micro"
}
```

Terraform Meta-parameters

- **count**: How many instances do we want to make?
- **depends_on:** If you want to explicitly state dependencies- though this really isn't necessary.
- **lifecycle:** which is used to customize the lifecycle of the resource. It's a separate block where you can do things like:
  - **create_before_destroy (bool):** Create new resource before destroying current one
  - **prevent_destroy (bool):** Pretty obvious
  - **ignore_changes (list):** Customize changes to ignore on a resource

Terraform Resource Parameters

- **Timeouts**: You can also set timeouts to customize how long an operation is allowed to take before it is considered an error. This is useful for when you are using **provisioners** with your ec2 instance… speaking of which:
- **Provisioners:** We'll go over this more in the lab but you can use provisioners to run commands set up resources:

```
resource "aws_instance" ”helloImathing" {
        provisioner "local-exec" {
                command = "echo ${self.private_ip} > file.txt" }
}
```

Third party provisioners

- So with provisioners- you can either do a custom provisioner, in bash, and have it set run.
- You can write out provisioner commands DIRECTLY in terraform OR
- You can use third party provisioners (read **chef** or **ansible** to when you launch your instance.
- Provisioners can also be used in situations like “on create” and “on_destroy”.

Provisioners (cont'd)

Creation time provisioner is the default behavior BUT

```
resource "aws_instance" "fernisawesome" {
  "local-exec" {
     when = "destroy"
     command = "echo 'NOOOOOOOOO!'" }
}
```

```
resource "aws_instance" "fernisawesome" {
  provisioner "local-exec" {
     command = "echo How" }
   provisioner "local-exec" {
      command = "echo Awesome?" }
}
```

Provisioners (cont'd)

- Finally we need to go through **file provisioners** which copy FILES from the host machine (executing terraform) into the provisioned resource. The idea is that you can send .sh or Makefiles up there and run them using the provisioners

```
resource "aws_instance" ”fernisawesome”
{
    provisioner "file" {
        source = "conf/myapp.conf"
        destination = "/etc/myapp.conf"
}
```

Terraform Provisioners (cont'd)

- Here's how we would use a combination of the **file provisioner** and a new one called the **remote-exec** provisioner to run a configuration file that we put in the ec2 resource we created:

```
resource "aws_instance" ”fernisawesome" {
    provisioner "file" {
        source = ”setupsomethingawesome.sh"
        destination = "/tmp/setupsomethingawesome.sh"
      }
     provisioner "remote-exec" {
        inline = [ "chmod +x /tmp/setupsomethingawesome.sh",
                   "/tmp/setupsomethingawesome.sh args", ]
         }
}
```

Terraform: Data Sources

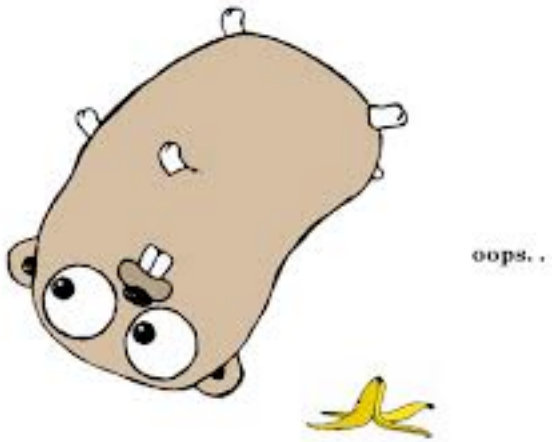
- So- with **resources** we are using terraform to create, update, read, and and delete assets in the aws infrastructure… but what if we just want data from a **pre-existing**, **non-terraform managed** (or managed by a completely *separate* terraform configuration) asset?

Terraform DATA sources (cont'd)

- So let's say we wanted to get the ID of an **existing resource** with the tag "fernisawesome". This is what the *data source* would look like…

```
data "aws_ami" "fern" {
    filter {
        name = "state"
        values = ["available"]
        }
    filter {
        name = "tag:Component"
        values = ["fernisawesome"]
       }
     most_recent = true
}
```

```
resource "aws_instance" "web" {
    ami = "${data.aws_ami.fernisawesome.id}"
    instance_type = "t1.micro"
}
```



Organizing your terraform code

- So now we need to go into how to organize what can easily grow to be an incredibly large terraform setup.
- Modules, in terraform, are basically the root level setup for a **group** of resources that form an **app**.
- Usually a module is made up of a few files:
  - main.tf (just like the **main** file in GO, Java, etc)
  - variables.tf
  - outputs.tf
  - README.MD

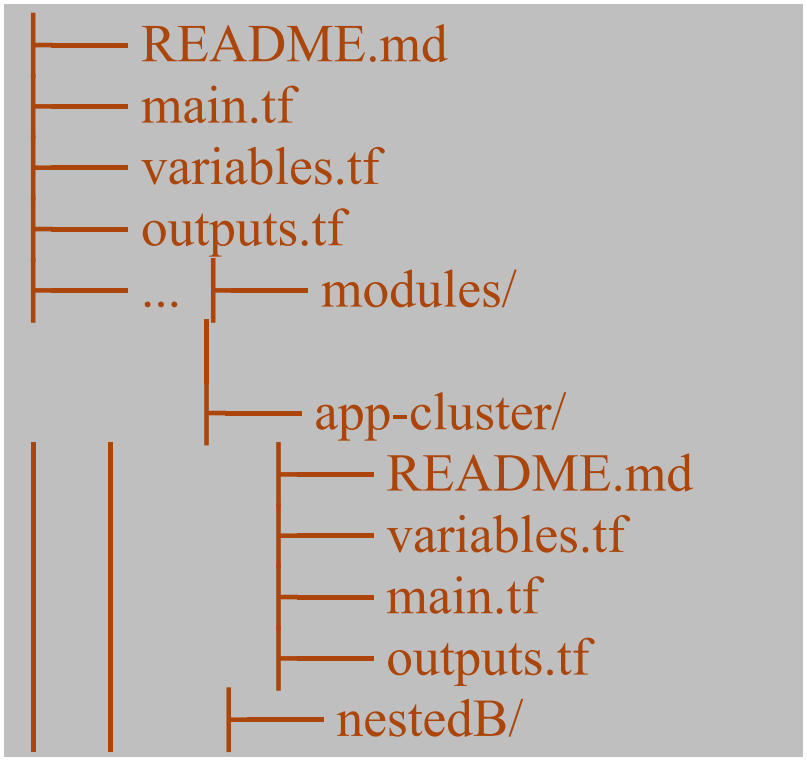
Importing and using terraform modules

- So nested modules can be used and imported by other modules (remember- **modules** are full applications- if you find yourself struggling to keep a module name different from resource names it means you are probably not abstract enough at your module level).

```
module "assets_bucket" {
    source = "./publish_bucket"
    name = "assets"
  }
module "media_bucket" {
    source = "./publish_bucket"
    name = "media"
}
```

Overall Directory Structure

So this is the external structure on the left with the imports on the right




```
module "servers" {
        source = "./app-cluster"
servers = 5
}
```

Terraform modules

- For this class we will primarily be using a single module as we're just making one application.
- IF you end up in a situation where you have multiple applications to tie together (this is frequent with complex data ingestion pipelines) then try to divide up your modules into logical structures.
- There are things like *inheritance* that we will not be getting in to as it's outside the scope of this class

Terraform steps

- So at this point we've got our terraform files written and we're ready to create and deploy our resources (and tie them together). There are four steps to every terraform deployments:
  - *terraform init:* initializes a terraform repo
  - *terraform plan*: shows you what's going to change
  - *terraform apply:*

Terraform: Creating a plan

- So the first level of the plan in terraform is to set up the high level **provider** data (remember- terraform works across **aws, gcloud, azure**).
- So our first entry in a terraform deployment file will be to list the provider we want to use.
- This lets terraform know which provider plugin it needs to download to start working effectively with your other files.

Confused????

- GOOD! Ask Questions!!