Learning Terraform Notes

Contents

1. INTRODUCTION

What is Terraform; Infrastructure Orchestration tools vs Configuration Management tools:

- Terraform and CloudFormation are examples of Infrastructure as Code (IAC); they allow you to orchestrate the infrastructure and provision the servers by themselves.
- This is different from Configuration Management tools like Ansible, Chef and Puppet which are designed to install and manage software on existing servers that have their infrastructure provisioned already; they may be able to do some infrastructure provisioning but not as well as dedicated tools like Terraform.
    - These can be integrated with IAC (not always but Terraform does allow this natively) where the IAC provisions the infrastructure and the configuration tool installs and configures the application for the resources, like so:

Choosing the IAC tool for an organisation is done by answering the following questions:

- Is the infrastructure going to be vendor specific in the longer term
    - E.g., if they will only use AWS, then it may be better to use CloudFormation
- Will multi-cloud/hybrid-cloud based infrastructure be used?
    - E.g., if using both Azure and AWS, then consider a different IAC
- How well does the IAC integrate with configuration management tools?
- What is the price and support like?

Advantages of Terraform:

- Support multiple platforms & has hundreds of providers
- Simple configuration language with fast learning curves
- Easy integration with configuration management tools like Ansible
- Easily extensible with plugins
- Free

2. DEPLOYING INFTRASTRUCTURE WITH TERRAFORM IN AWS

Deploying a resource; Creating an EC2 instance

Creating your AWS account user and setting up a directory for the terraform manifests

Create a user in your AWS account and note the access key and the security key. Create a directory wherein you will keep the terraform code and create a new file called first_ec2.tf. This will be the file where we will write out configuration into. Also download either Visual Studio Code (I will be using this) or Atom, and add this folder as a project.

Creating your terraform manifest

Considerations:

- Ensuring authentication to AWS
- Specifying a region for the resource
- Specifying the resource

Navigate to registry.terraform.io to look at the requirements to launch a resource. Navigate to Documentation found in the AWS provider. Here they provide example usage of the provider and resource blocks (the first two scrn shots on the left) and an example for how you can authenticate withing the provider block (the scrn shot on the right):

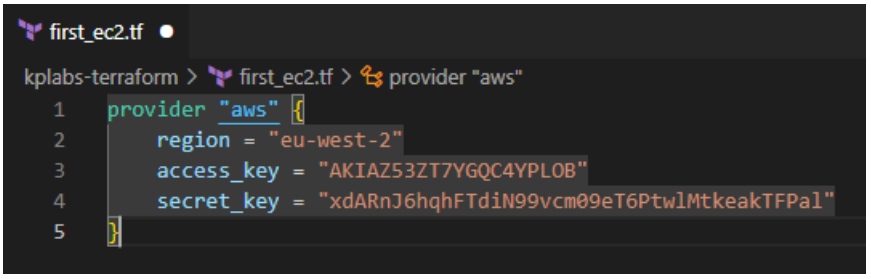
- Above the provider is aws as shown under 'Configure the AWS Provider'
    - If this was azure, then this would need to be azure

- Notice that 0.12 and earlier version hasn't got the extra writing at the top (these are referred to as blocks)
  - If you don't add this section things will still work but this block is recommended. Not adding it now is okay.

- Provider Block: Begin building the code for the provider block based of the example and signify the region:

```
provider "aws" {
    region = "eu-west-2"
}
```

  - Use the same structure shown in the authentication srn shot to include how to include your AWS users access-key and secret-key (acquired from IAM users):



Navigating further in the registry.terraform.io you can find information on the resources for the services that AWS provides; here we are interested in the aws_instance found under the EC2 banner. (https://registry.terraform.io/providers/hashicorp/aws/latest/docs/resources/instance). A simple description of what the aws_instance is that it provides an EC2 instance resource allowing it to be created, updated and deleted. Within the example usage, we can see how to write the code for this resource (i.e., the EC2_instance):

```
resource "aws_instance" "web" {
  ami           = data.aws_ami.ubuntu.id
  instance_type = "t3.micro"

  tags = {
    Name = "HelloWorld"
  }
}
```

- Resource Block: Begin building the code for the resource block, this requires an ami and the instance type that you are interested in. You can choose the t2.micro free instance type here on the Amazon Linux 2 operating system. The ami must be selected according to the region, in this case eu-west-2; check the console for this, you can see it if you zoom in the image here  . Also note how I have used the # at the start of a line to create notes here too:

- The writing in blue within the blocks are the arguments. In this aws_instance resource, the only arguments that were needed were the instance_type and the ami, this is known to us since it can be found in the registry.

Preparing your terraform directory and executing your manifest with the terraform core commands: terraform init, terraform validate terraform plan, terraform apply. terraform destroy

- terraform init
  - Navigate to your directory (in my case the kplabs-terraform) using the cd command in the terminal and perform the 'terraform init' command:

```
DELL@DESKTOP-9M3OOGL MINGW64 ~/Documents/Projects/kplabs-terraform
$ terraform init

Initializing the backend...

Initializing provider plugins...
- Finding latest version of hashicorp/aws...
- Installing hashicorp/aws v4.22.0...
- Installed hashicorp/aws v4.22.0 (signed by HashiCorp)

Terraform has created a lock file .terraform.lock.hcl to record the provider
selections it made above. Include this file in your version control repository
so that Terraform can guarantee to make the same selections by default when
you run "terraform init" in the future.

Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan" to see
any changes that are required for your infrastructure. All Terraform commands
should now work.

If you ever set or change modules or backend configuration for Terraform,
rerun this command to reinitialize your working directory. If you forget, other
commands will detect it and remind you to do so if necessary.
```

  - The terraform init command read the configuration file that we used and found the provider to be aws and downloaded the terraform plugins needed for this. These can be found in the path .terraform\providers\registry.terraform.io\hashicorp

    - Each time you configure our provider, ensure to run terraform init
- terraform plan
    - Perform this command in this directory to see what the configuration file (the manifest) will do and review it
- terraform apply
    - Perform this to execute the manifest that you have
    - Enter yes to execute it after it shows a review of what will happen
    - You can confirm this in the console
    -

```
aws_instance.myec2: Creating...
aws_instance.myec2: Still creating... [10s elapsed]
aws_instance.myec2: Still creating... [20s elapsed]
aws_instance.myec2: Still creating... [30s elapsed]
aws_instance.myec2: Creation complete after 31s [id=i-0ade6012000c8765a]

Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
```

- Now if you run terraform plan, you get the following:

```
DELL@DESKTOP-9M30OGL MINGW64 ~/Documents/Projects/kplabs-terraform
$ terraform plan
aws_instance.myec2: Refreshing state... [id=i-0ade6012000c8765a]

No changes. Your infrastructure matches the configuration.

Terraform has compared your real infrastructure against your configuration
and found no differences, so no changes are needed.
```

Note regarding the terraform block for 0.13 versions of terraform and onwards

- If a provider is not directly maintained by hashicorp, you must ensure to provide the block as this gives the path for the provider in the website. For maintained providers, this isn't necessary. Take the example:

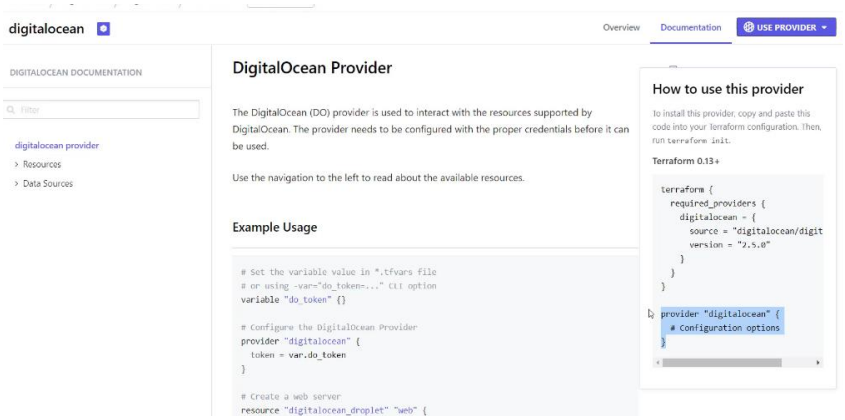
```
provider "aws" {
  region     = "us-west-2"
  access_key = "PUT-YOUR-ACCESS-KEY-HERE"
  secret_key = "PUT-YOUR-SECRET-KEY-HERE"
}
```

HashiCorp Maintained

```
terraform {
  required_providers {
    digitalocean = {
      source = "digitalocean/digitalocean"
    }
  }
}

provider "digitalocean" {
  token = "PUT-YOUR-TOKEN-HERE"
}
```

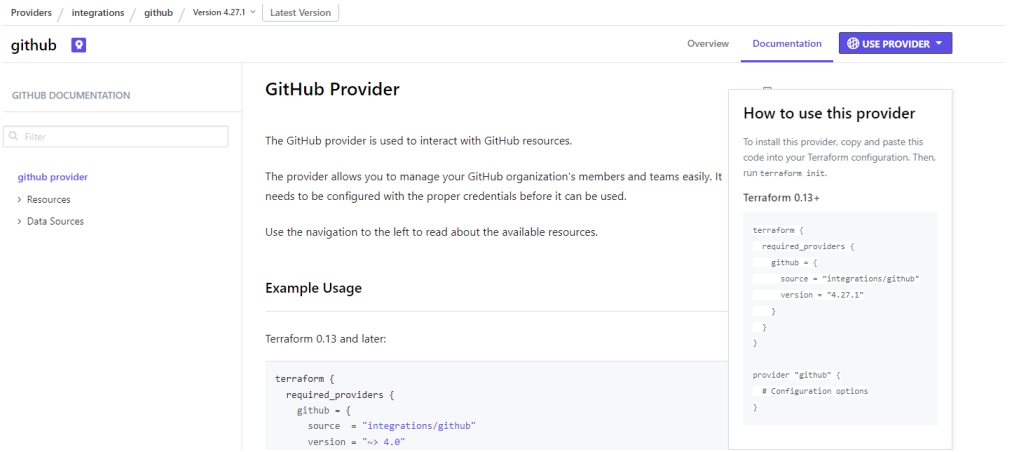
Non-HashiCorp Maintained

- You could find this information in the website under ‘USE PROVIDER’:



Demo showing the ease of using other providers: Creating a repository resource in the Github Provider

- Copy the use provider code to create your first block:

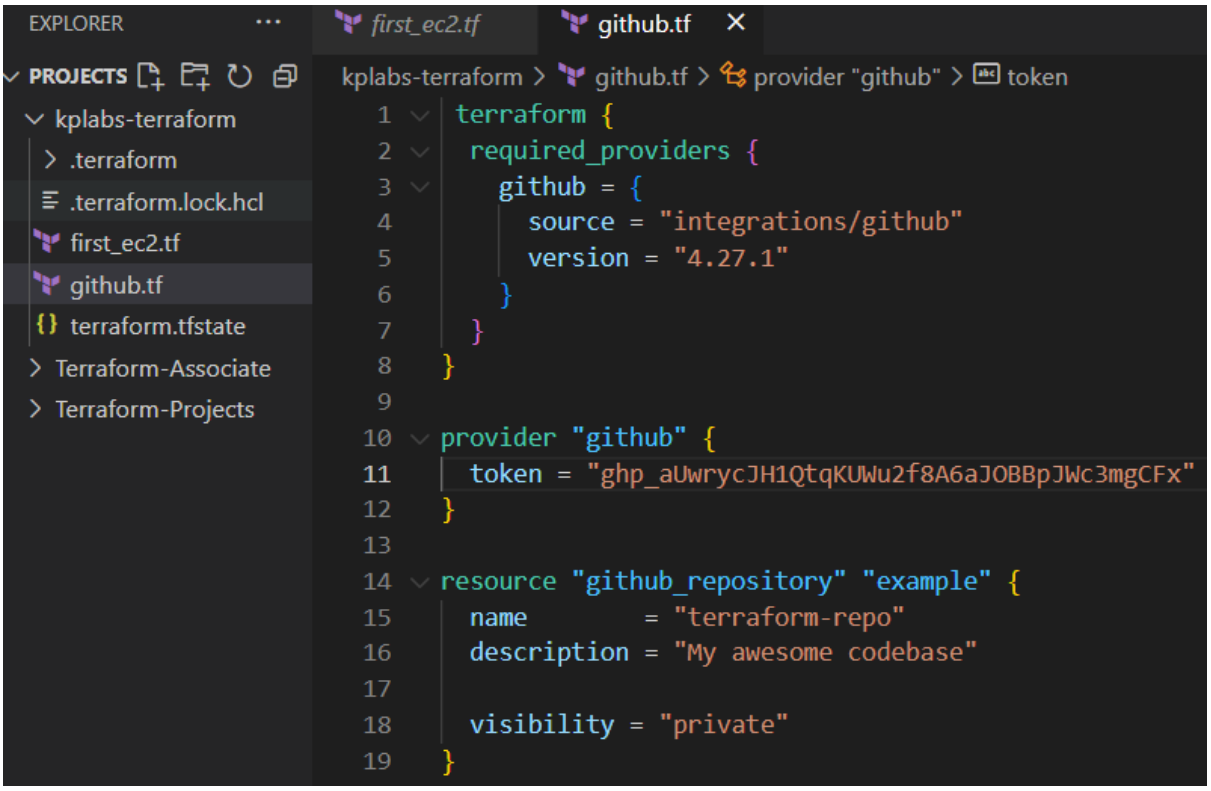


- Create a new manifest called github.tf in the directory, and paste that into there:

- Under authenticate in https://registry.terraform.io/providers/integrations/github/latest/docs, you see that a token is required. Generate a token in your github accoundt under the developer

settings within settings.When creating the token, select the operations for repo and delete_repo.

- Navigate to github_repository in the documentation and copy the example usage into your VS Code, You can remove the template and description part since they are optional, change the visibility to private and name it as terraform repo:



- Perform the terraform init (since a new provider has been added) and then check with terraform plan and then apply all from the CLI

Terminating the resources we created; terraform destroy

Task: Destroy EC2 instance to avoid being charged, without affecting the github repository you created. To achieve this, specifically terraform destroy -target option to hone in on the EC2 instances manifest. This is known as using a target flag, it is a combination of two things (recall the resource block takes two values) separated with a . :

1. Resource Type: aws_instance
2. Local Resource Name: myec2

Thus the command to run is: terraform destroy -target aws_instance.myec2:

```
DELL@DESKTOP-9M30OGL MINGW64 ~/Documents/Projects/Terraform-Associate/kplabs-ter
raform (main)
$ terraform destroy -target aws_instance.myec2
aws_instance.myec2: Refreshing state... [id=i-0ade6012000c8765a]

Terraform used the selected providers to generate the following execution
plan. Resource actions are indicated with the following symbols:
  - destroy

Terraform will perform the following actions:

  # aws_instance.myec2 will be destroyed
  - resource "aws_instance" "myec2" {
```

Note, if you run terraform plan now, you will see that it wants you to re-create this deleted aws_instance resource since it is still in the manifest. However, if you really don't need it anymore, you can comment it out or even delete it in the manifest for the instance using /* */:

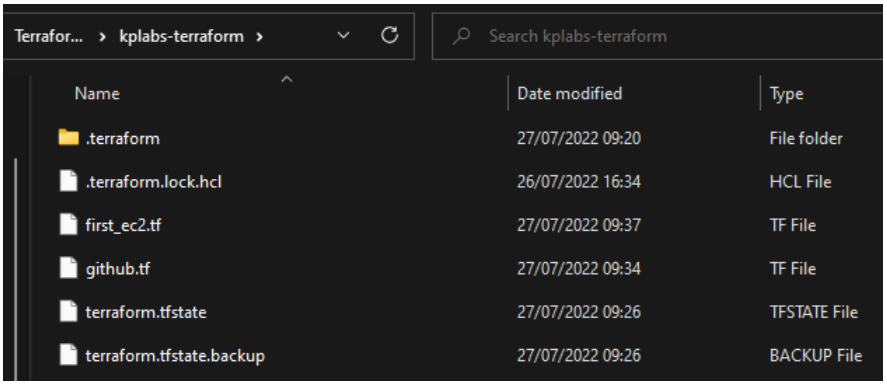
Terraform-Associate > kplabs-terraform > first_ec2.tf > ...

```
...
#Proider Block
...
provider "aws" {
    region = "eu-west-2"
    access_key = "AKIAZ53ZT7YGQC4YPLOB"
    secret_key = "xdARnJ6hqhFTdiN99vcm09eT6PtwlMtkeakT
}

/*
#Resource Block
resource "aws_instance" "myec2" {
    ami =  "ami-078a289ddf4b09ae0"
    instance_type = "t2.micro"
}
 */
```

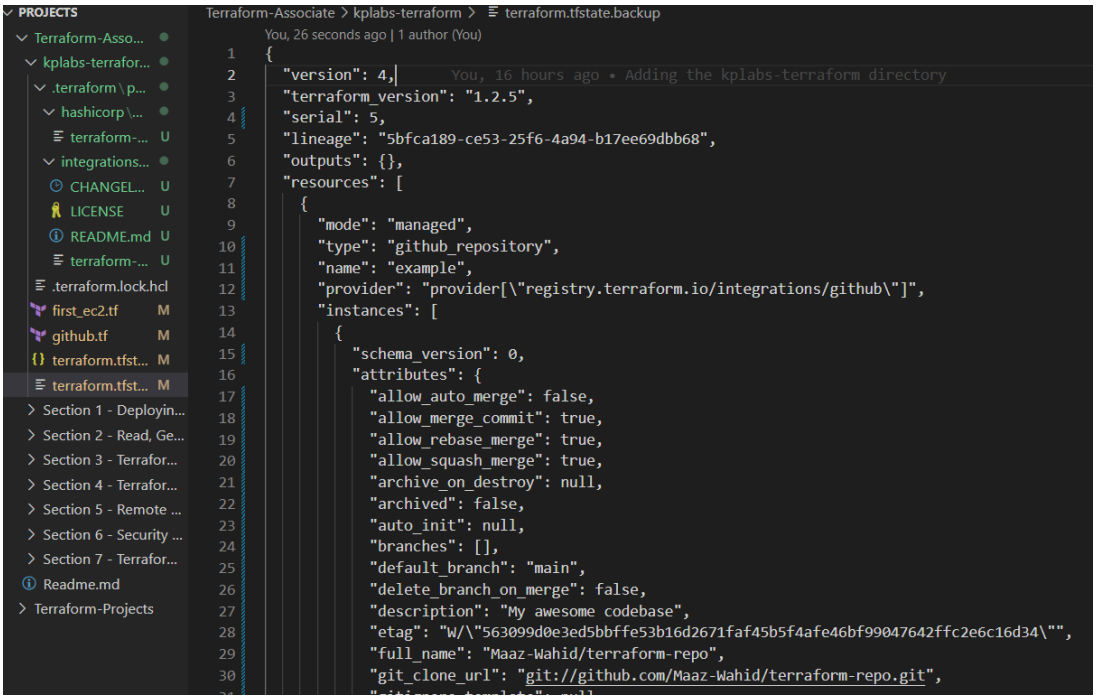
Understanding Terraform State Files

About State files

- Terraform stores the state of the infrastructure that is being created from the TF files allowing terraform to map real world resource to your existing configuration. These are stored in a state file.
- This is how terraform is aware of the deleted resources or the resource that need to be provisioned when you run terraform plan. You can see the state file in your terraform initialised directory:



Currently in the state file you can see the github repository resource with information regarding it:



However, I actually deleted this already on Github directly, so if I run terraform destroy, terraform initially thought it was there due to the state files, thus showing this:

```
DELL@DESKTOP-9M3OOGL MINGW64 ~/Documents/Projects/Terraform-Associate/kplabs-ter
raform (main)
$ terraform destroy
github_repository.example: Refreshing state... [id=terraform-repo]

No changes. No objects need to be destroyed.

Either you have not created any objects yet or the existing objects were
already deleted outside of Terraform.

Destroy complete! Resources: 0 destroyed.
```

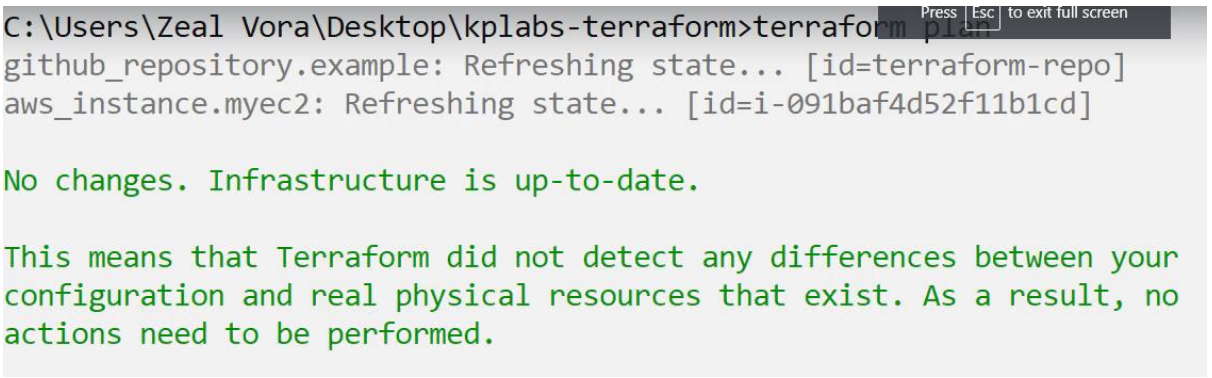
However, going into the state files, now we find the github resource is removed:

If you have done a change outside of terraform and want the state files to be aware of this, use the command:
terraform refresh

Desired state and Current State

The desired state is described in the Terraform Configuration files (the Manifests) and the function of terraform is to create modify and destroy infrastructure to match this state. The current state is the live state of the infrastructure which may not be the same as the desired state.

If you however, have optional arguments that you did not specify e.g., a security group for an ec2, then if the security group is changed (and you ensure terraform is aware of the change by refreshing the state files), terraform plan will show that all is okay:



This is why it is important to specify all the important things such as the IAM roles and Security groups ect.

Provider versioning

Provider Architecture

Here is the provider architecture:

- The provider architecture involves a terraform file which uses the terraform platform

- This platform communicates with your provider (e.g. AWS or Azure etc)
- The provider then provisions the services and infrastructure that was requested for it
- They then communicate this back with Terraform so that Terraform is aware of the state of infrastructure and keep note of this in a file called terraform.tfstate

Specifying the Version

- Providers may update, e.g., from Digital Ocean Version 1 to Version 2. As such, a new provider may have changes that cause your code to not work any longer. Thus when creating your manifest under terraform block, you need to specify the acceptable versions:

```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 3.0"
    }
  }
}

provider "aws" {
  region = "us-east-1"
```

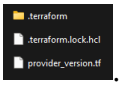
Version Number Arguments	Description
>=1.0	Greater than equal to the version
<=1.0	Less than equal to the version
~>2.0	Any version in the 2.X range.
>=2.10,<=2.30	Any version between 2.10 and 2.30

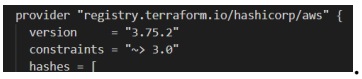
- So in the above case, this is anything in the version range of 3.X. However even this may be dangerous so it is better to just specify the version

```
terraform {
  required_providers {
    aws = {
      source = "hashicorp/aws"
      version = "4.23.0"
```

as shown in the use provider section in the registry.

Understanding Lock Files

- So when doing terraform init on this file, this also produces a .terraform.lock.hcl file including the provider plugins: 

. Within this file you can find the constraints involved: 

```
provider "registry.terraform.io/hashicorp/aws" {
  version     = "3.75.2"
  constraints = "~> 3.0"
  hashes = [
```

. If you change the version in the manifest and the version isnt within the constraint, then the lockfile wont allow terraform init to download such plugins. To override this (you may intentionally want to change to a newer version for e.g.), you need to delete the lock file.

3. READ, GENERATE AND MODIFY CONFIGURATIONS

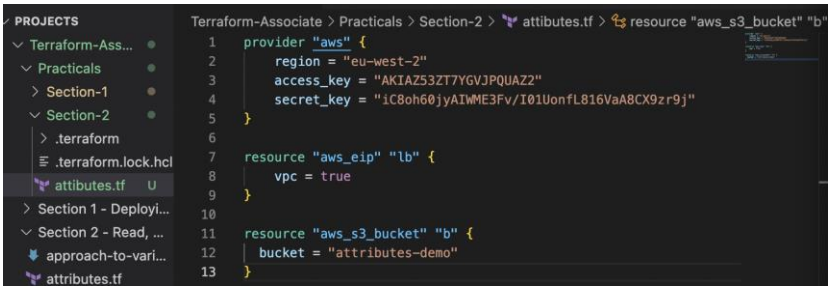
Attributes and Output Values

Understanding attributes and output values

- Terraform can output the attributes of a resource with output values.
    - If a resource like an ec2 and an s3 bucket is created and you wish to fetch the IP address of the instance and the domain name associated with the s3 bucket, you can do it by going to the console itself. However, you can use terraform to output these values while you do terraform apply.
- These outputted values can be used as an input for other resources, a little like piping in the command line.
    - For example, a security group needs you to inform it of 'allowed' (i.e. whitelisted) IP addresses. This allows the resource to work. Thus, when creating EIP (elastic IP) for example, you can output the IP address so that it gets whitelisted.

Applying output values in an example

Let's try the above blue use case (firs without the output values) with the following configuration code:

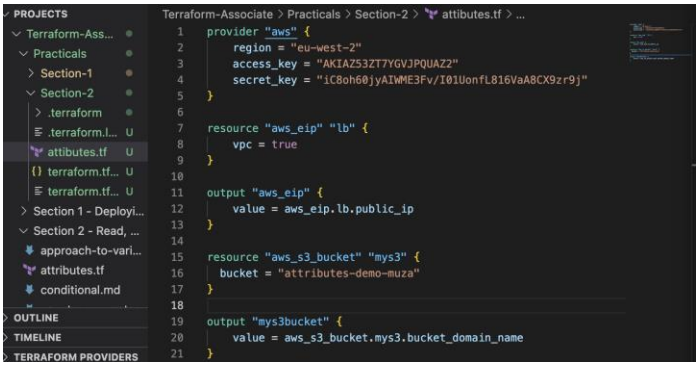


Perform terraform apply form terraform apply and then check your AWS console for the s3 bucket and the Elastic IP that you created to confirm this. Also make sure you use an s3 bucket name that has not been already used by someone else.

The output block takes the following configuration:

```
output = "<Name of the output>" {
    value = <resource_block>.<resource_block's_given_name>.<attribute_to_output>
}
```

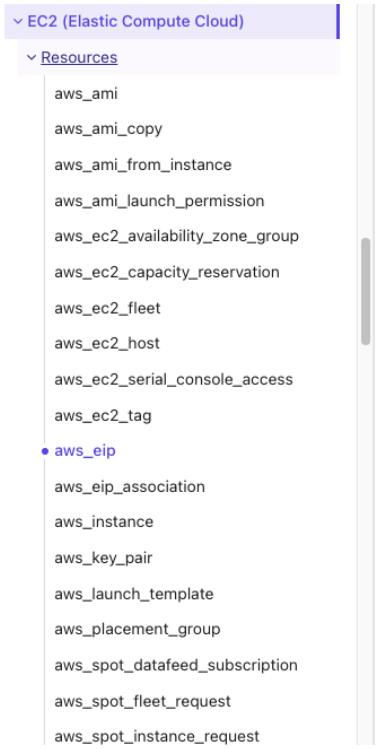
Let's now include the output values code into the configuration so that we can achieve the above blue writing use case:



Destroy the infrastructure you created and then perform terraform apply again. When doing this, you should see the following output values in your command line, thus saving you from needing to go to the console to get this information:

```
Apply complete! Resources: 2 added, 0 changed, 0 destroyed.

Outputs:

aws_eip = "3.11.63.143"
mys3bucket = "attributes-demo-muza.s3.amazonaws.com"
Maaz@MacBook-Air Section-2 %
```

You can find out what attributes you can reference in the page of the resource you are dealing with under the attributes reference section in the registry for that resource.



Attributes Reference

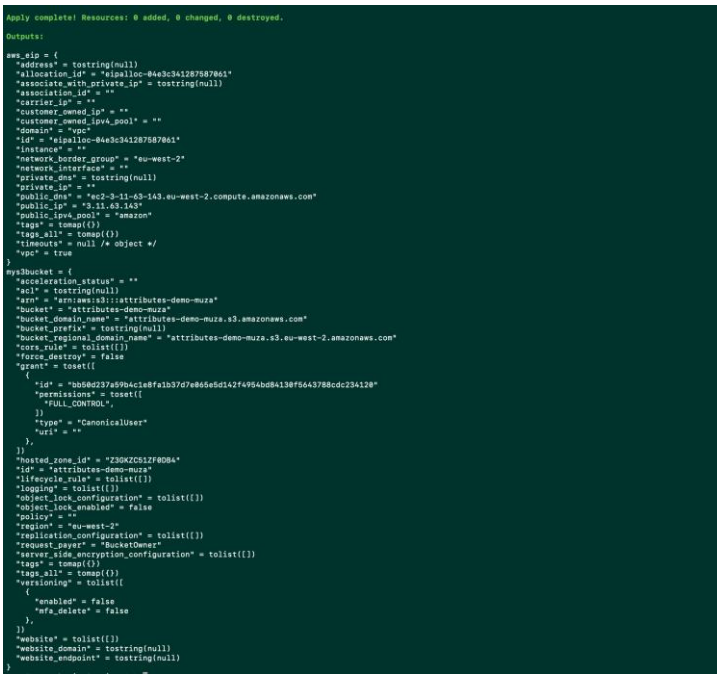
In addition to all arguments above, the following attributes are exported:

- `allocation_id` - ID that AWS assigns to represent the allocation of the Elastic IP address for use with instances in a VPC.
- `association_id` - ID representing the association of the address with an instance in a VPC.
- `carrier_ip` - Carrier IP address.
- `customer_owned_ip` - Customer owned IP.
- `domain` - Indicates if this EIP is for use in VPC (`vpc`) or EC2 Classic (`standard`).
- `id` - Contains the EIP allocation ID.
- `private_dns` - The Private DNS associated with the Elastic IP address (if in VPC).
- `private_ip` - Contains the private IP address (if in VPC).
- `public_dns` - Public DNS associated with the Elastic IP address.
- `public_ip` - Contains the public IP address.
- `tags_all` - A map of tags assigned to the resource, including those inherited from the provider `default_tags` configuration block.

If you don’t specify the attribute you want to output, and leave it blank like so:

```
provider "aws" {
    region = "eu-west-2"
    access_key = "AKIAZ53ZT7YGVJPQUAZ2"
    secret_key = "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"
}

resource "aws_eip" "lb" {
    vpc = true
}

output "aws_eip" {
    value = aws_eip.lb
}

resource "aws_s3_bucket" "mys3" {
  bucket = "attributes-demo-muza"
}

output "mys3bucket" {
    value = aws_s3_bucket.mys3
}
```

This results in the output of all the attributes possible into the command line:



The attributes can also be found in the terraform state files:

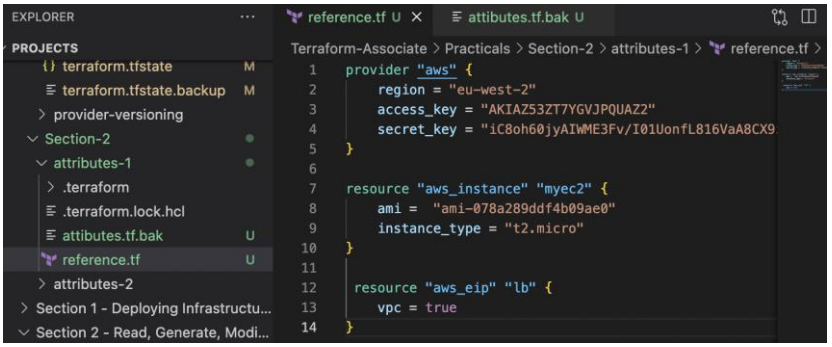
Referencing Cross-Account Resource Attributes

Examples to take to do this:

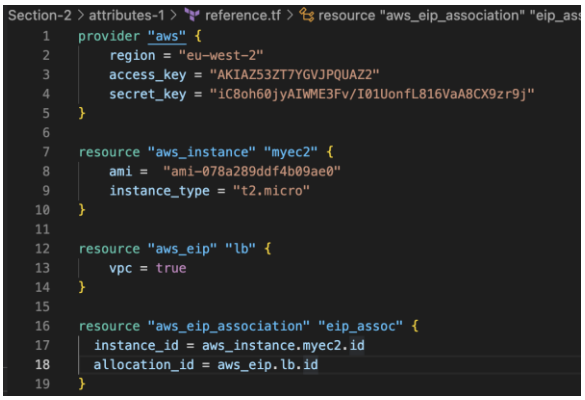
1. Create an elastic IP and an ec2 instance. The eip will automatically be associated with the ec2 instance.
2. Create an eip and a security group. The eip will automatically be added to the security groups inbound rules.

First case:

Create a new reference.tf file in the same folder and use the resource and provider block created in the first_ec2.tf example and add it into the manifest. Also add the resource for the eip file from the attributes.tf file. Also, since you will now have two configuration files in one folder, add .bak to the attirbutes.tf.bak to make it not functional here resulting in only one provider block. The reference.tf file should look like so:



Terraform plan shows two resources will be created. But we want the eip to automatically associate with the security group. There is a resource called aws_eip_association that allows this; the first argument is the instance_id which identifies which instance it should be linked with as well as the attribute .id (under aws_instance attributes section you can find that this is for the instance ID) and the second one is the allocation_id which is where you specify which eip should be attached to this instance. Your code will look like this:



Now if you do a terraform apply, the ec2 instance is created and the eip is automatically associated with it as you will find in the console.

Second case:

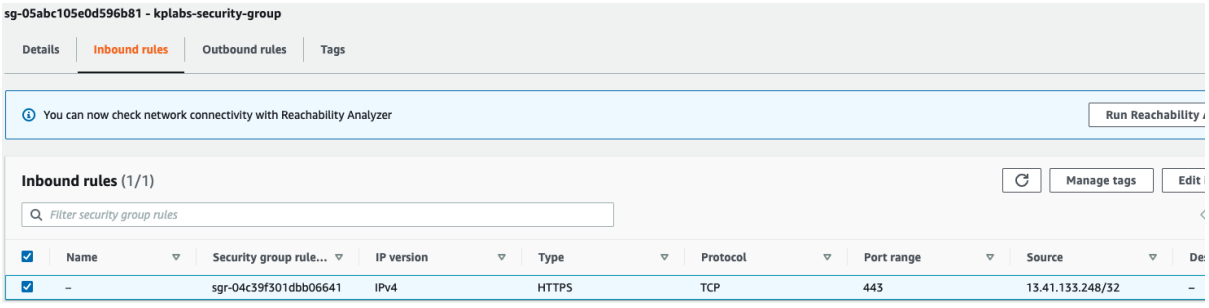
Add the aws_security_group resource using the basic usage in the registry and adjust as follows:

- Ingress is inbound (when creating one in the console you see this), when adding a rule you must specify a few things like the port (such as 22) and the IP address (ensure you specify the whole subnet string as well e.g. 10.20.50.30/32). We won't be adding any outbound rules.
- Cidr_blocks (allocating IP addresses for IP routing) allows you to reference the eip so insert: ["${eip_aws.lb.public_ip}/32"] vs ["eip_aws.lb.public_ip"]
    - Notice the extra attribute public_ip which is found from the registry from the eip registration and thus allows you to reference the public_ip
    - Notice the /32 – even though the IP address is given, you need to specify this in AWS
    - Notice the "${}/32" which wrapped around the IP address of the eip and subnet string, this is needed due to the /32 – don't worry, this is just a block style that used to be used before the .11 version of Terraform
- For the protocol, "-1" is used for ICMP but here we specified a port (being 443), so you must use tcp protocol

The resulting code which you will have added into your reference.tf file:

```
resource "aws_security_group" "allow_tls" {
  name        = "kplabs-security-group"

  ingress {
    from_port        = 443
    to_port          = 443
    protocol         = "tcp"
    cidr_blocks      = ["${aws_eip.lb.public_ip}/32"]
  }
}
```

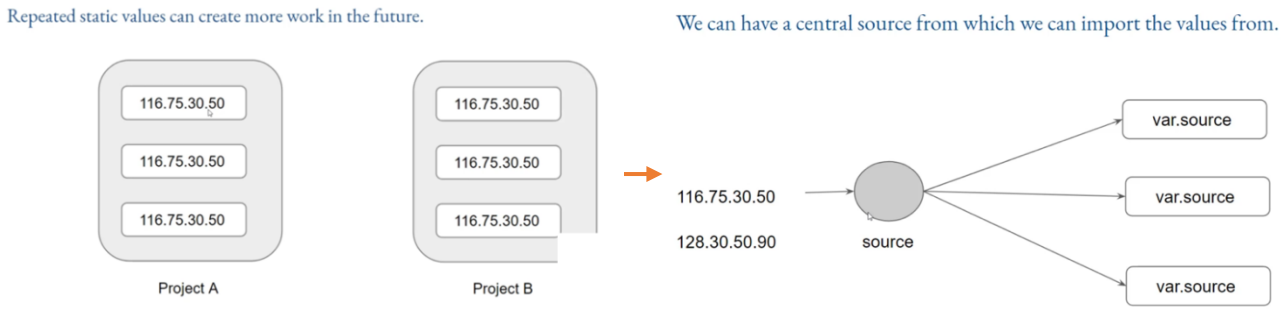
Executing terraform apply results in the creation of a security group with the eip's IP being whitelisted on port 443, look at the source section below:



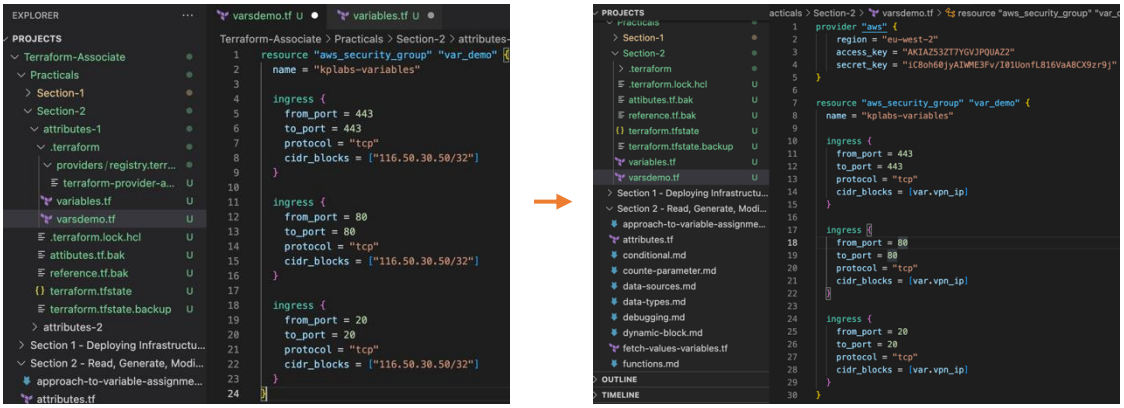
Terraform Variables

Advantages of a variable and demo of use case

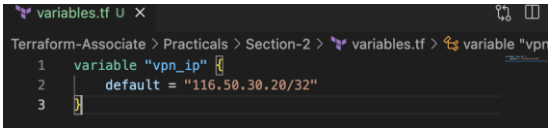
Having repeated static values results in more work in the code. For example, if you have a project A where you allow a specific IP address on 3 ports (22, 80, 443 for e.g), thus having 3 IP addresses that you have white listed. And in project B you do the same whitelisting for these IP addresses. This is okay, but if this IP address changes after some time, you have to update your code in 6 locations. Thus if you create a central source where you supply the static values to, you can reference that source with var.source which has the value. Thus



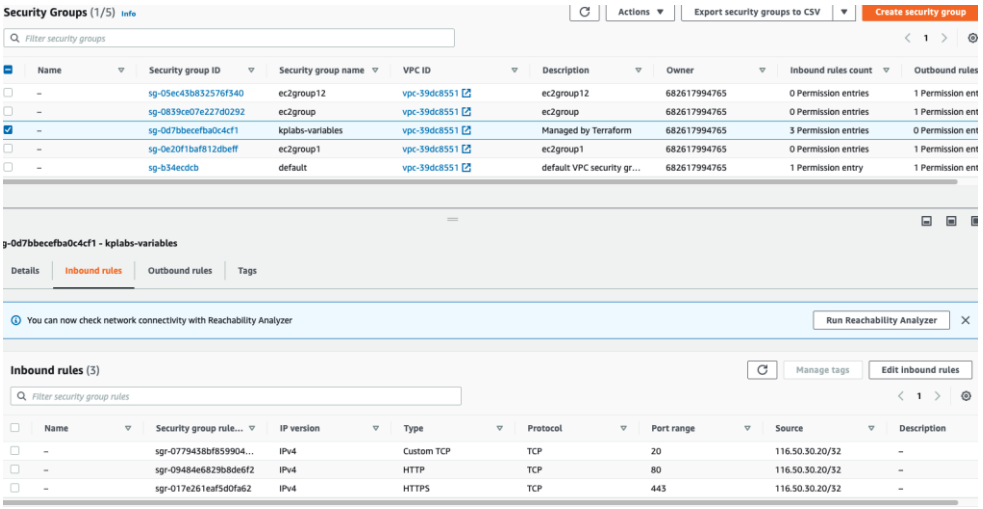
To do this in a demo, perform terraform destroy, then make sure to add .bak to your reference.tf since we will work in the same directory. Write the following code to whitelist the IP mentioned on the three ports as shown in the left image above and now below, then re write the code like the one shown on the right:



So now the source in this case the source is vpn_ip and the syntax to reference it is var.vpn_ip (I forgot to add my authentication to the configuration on the left). You must create a variable.tf file so that there is actually a place where this IP address is actually stored and referenced. Create a file in the directory called variables.tf, and write the following code within:



Now you can execute terraform apply to create the security group with the referenced IP address that lies in ‘vpn_ip’ – thus it will be whitelisted in the inbound rules:



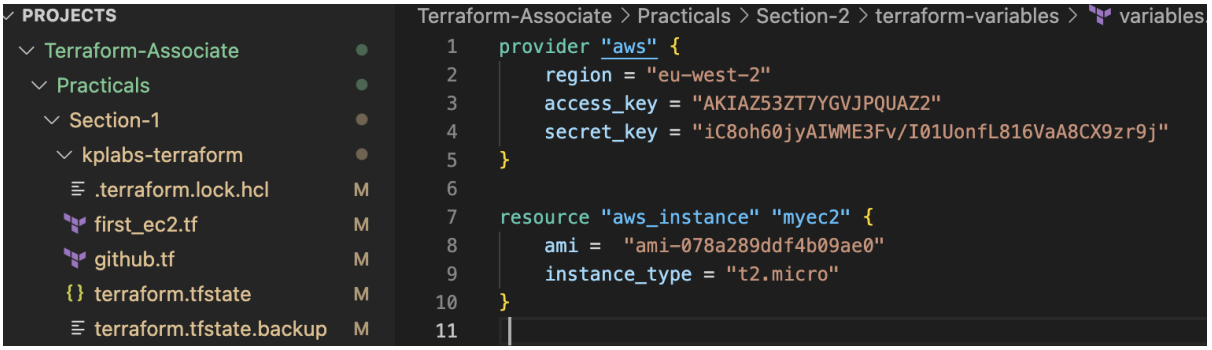
Now all you need to do to update the IP is adjust the default argument in the vpn_ip variable block that lies in the variable.tf manifest.

Different approaches to assigning Variables

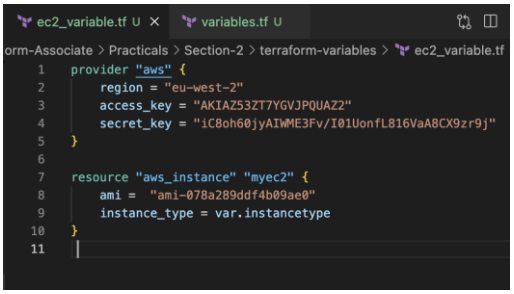
Some ways:

- Variable Defaults
- Command line Flags
- File based variables
- Environment variables

Variable Defaults

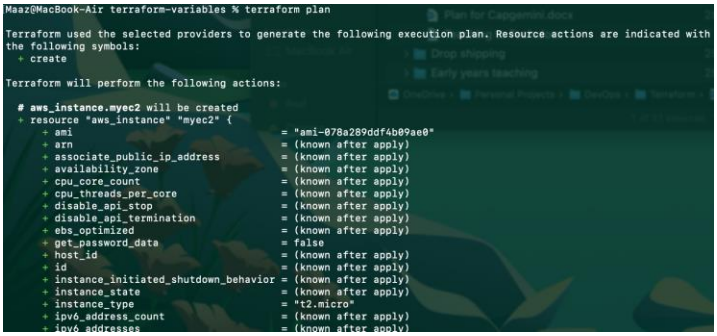


Here we find everything hardcoded, so lets add a variable into a file called variables.tf, and add the following variable whilst adjusting your code in ec2_variables.tf manifest:



AND

Running terraform plan, you find that the instance type is:

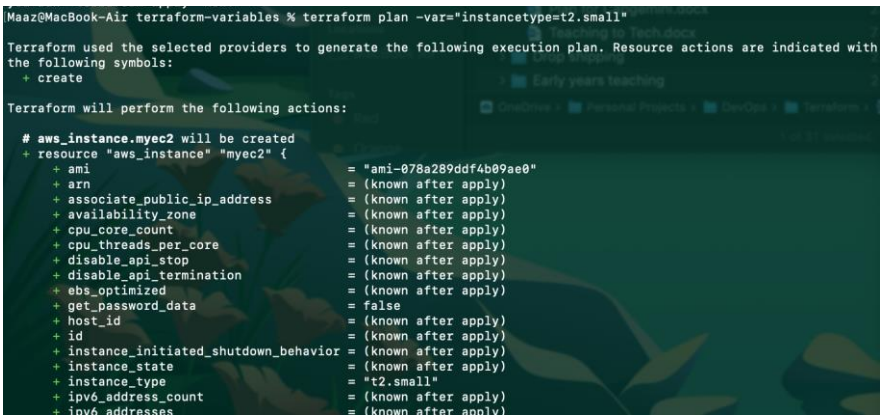


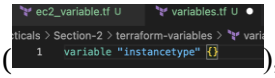
We can see here that the instance type is t2.micro so this is the Variable Default, you didn't specify a specific value for the variable so the default t2.micro was assigned to it.

Command line flags

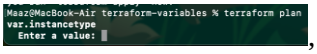
But now (although this isn't the best way), in the CLI if you use the following command is applied, you actually don't end up using the default value of the variable from the variables.tf (which in this case is t2.micro) and you end up specifying a variable:

- terraform plan -var="instancetype=t2.small"

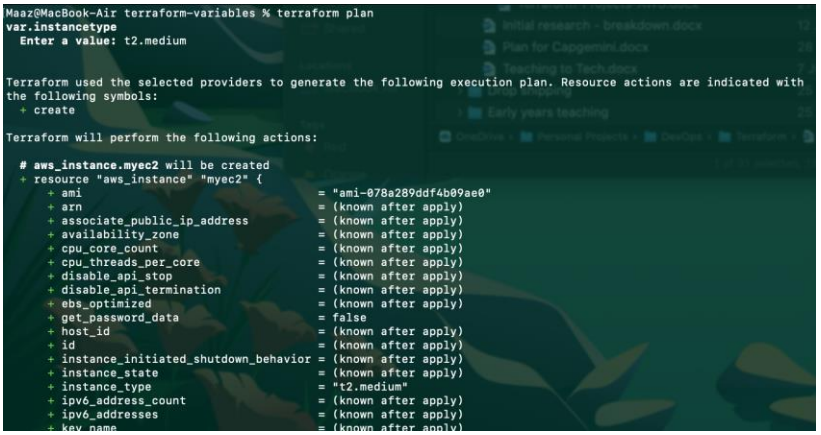


If in your variables.tf file, you don't specify the default value (

), and you don't use the -var option to specify when doing terraform plan, you end up with the following prompt

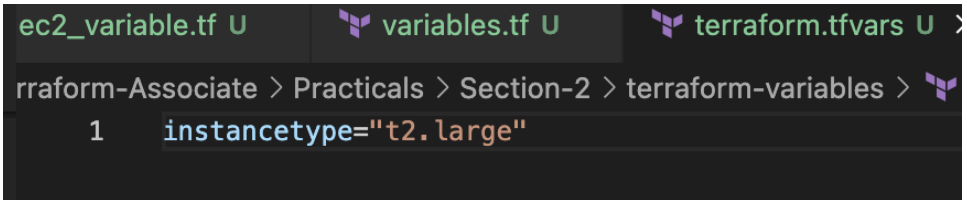


, and you can populate it with a desired instancetype, which you will find come out in the plan, like so:

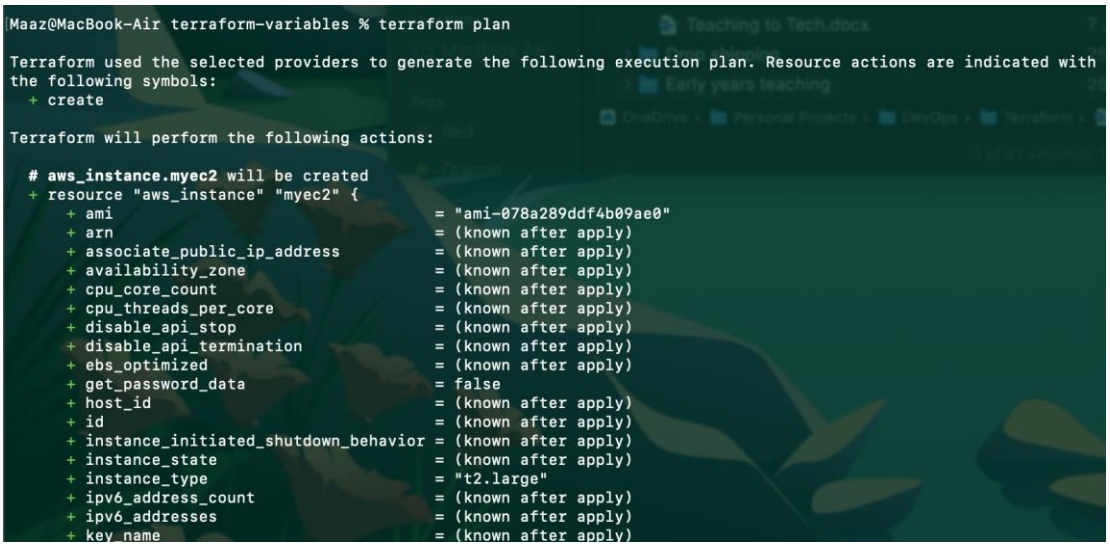


File based variables

As we know, using the -var=”” argument isn’t the best approach, rather you can create a file with the format .tfvars. This file will act as the -var=”” command, so if that doesn’t exist, the default variable (e.g. t2.micro in our case) will be used. However, if it does exist, all the values associated with the variable will override based of what’s in this file. So create a file called **terraform.tfvars** (it must literally be terraform.tfvars and not anything else like custom.tfvar; terraform only recognises the former) and add the following code in the directory, using the syntax shown:

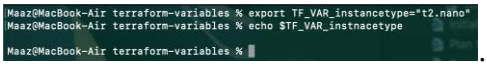


Now running ‘terraform apply’ results in:



Now it is clear that the var.instance type uses terraform.large. You can make the default instance type (in our case t2.micro as we defined in our variables.tf file) by adding .bak like so to the file: terraform.tfvars.bak

Environment Variables

For MacOS/Linux, the export command is used to ensure the environment variables and functions are passed to child (the next) processes. Also recall that $ means you're in the system shell. Run the following command in the commands in the command line: 

. But closing and opening the command line results in:

It didn't work alongside the Udemy Demonstration, need to figure this one out still…

Specifying Data Types for Variables

Understanding the type concept with demo

It is best practice that when you are creating large modules, you define the Type of value that will be accepted as the value for your variables in your variable.tf files. A Type is used to restrict a variables data type when someone defines the value of a variable in terraform.tfvars. For instance, if a variable needs to be a number input, you can define this using the type argument, which stops someone from being able to do use other than a number when they attempt terraform apply. Some common data types available to restrict the variables are:

- number
- list
- string
- map

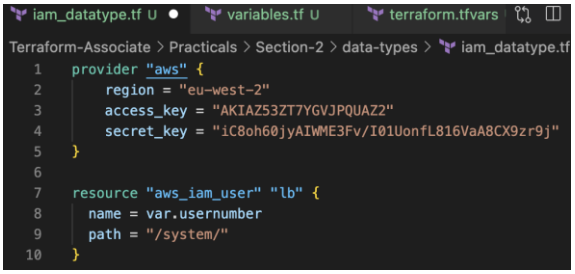
Say every employee in a company is assigned an identification number and whenever an ec2 instance is made, they need to make sure they call the instance_name their identification number. You can create a variable called instance_name and ensure that the type is a number; this way if someone then goes into the terraform.tfvars file and inserts other than a number, it will be flagged up in terraform plan. Take a look at how you would do this with the code here, John here will be flagged up since he did not put a number:

variables.tf	terraform.tfvars

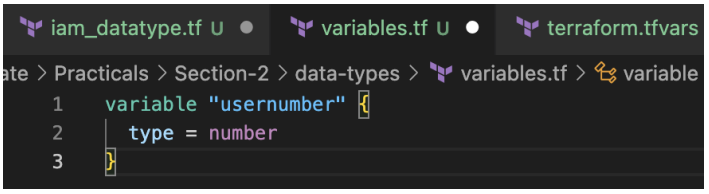
variable "instance_name" {
type=number
}
	instance_name="john-123"

Demo: Create an aws_iam_user resource and restrict the name type of the iam user to a number by using a variable (in this case call the variable usernumber). Thus create a variables.tf file to ensure this restriction is in place and the variable is created. Then create a terraform.tfvars file to test this out:

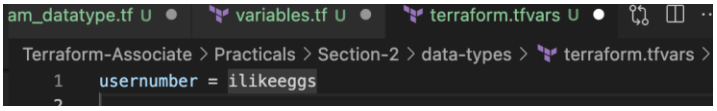
iam_datatype.tf:



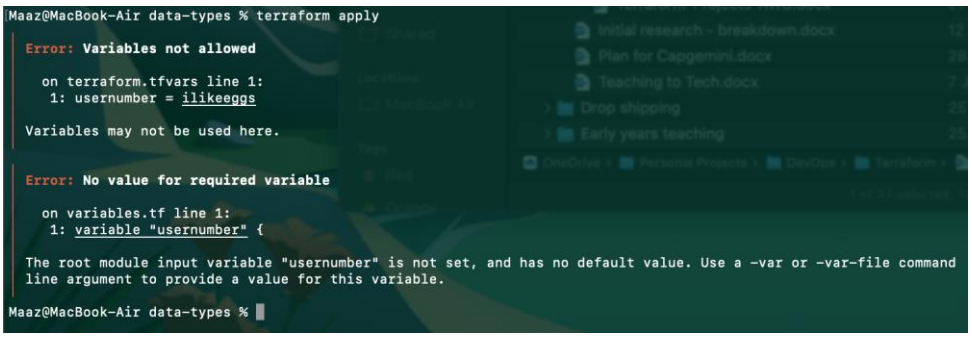
variables.tf:



terraform.tfvars:

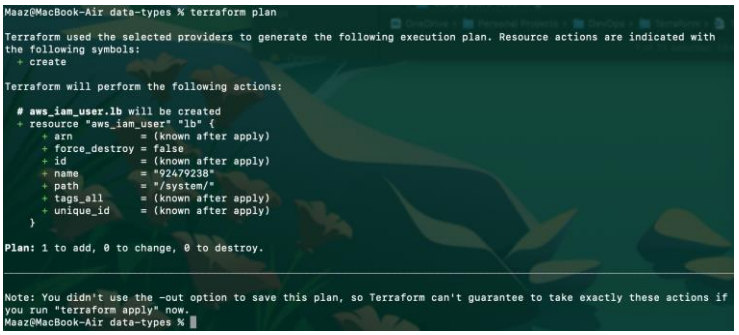


terraform plan:



New terraform.tfvars:

terraform apply:



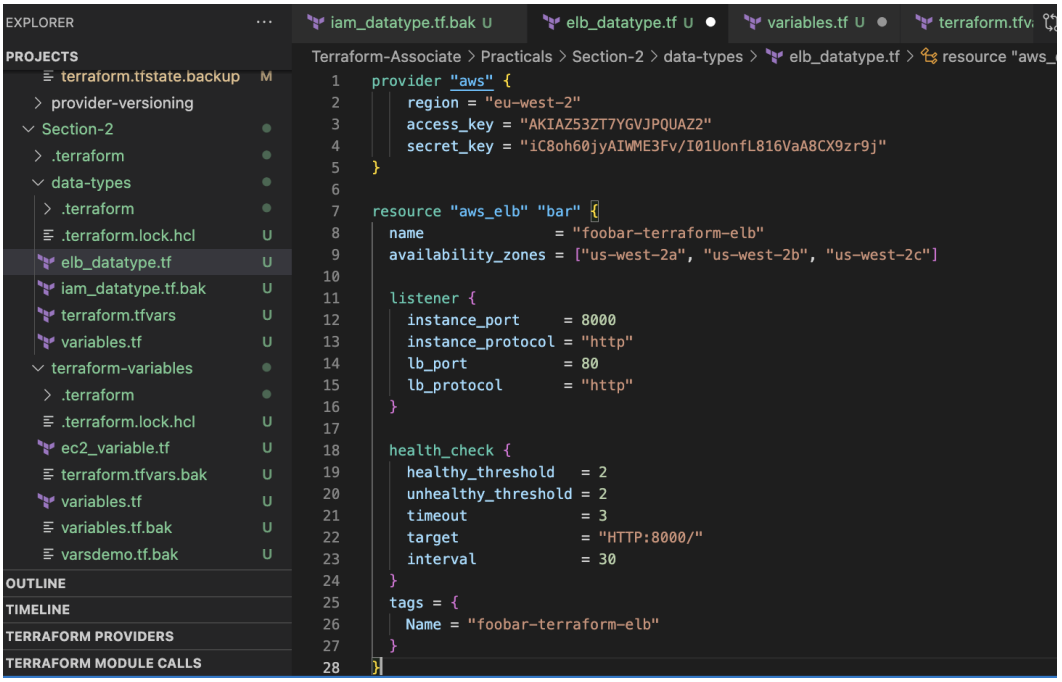
Common type restrictions for variables

Type:

- string = Sequence of Unicode characters like “terraform”
- list = Sequence of values which can be identified by their position. Their position starts with 0. They used to be identified like: list(“cat”, “dog”, “mouse”), but now they are identified as [“cat”, “dog”, “mouse”]
- map = These are values that have associated names. These names are the ways that they can be identified, for example {name = “Maaz”, age = 999}
- number = Will only accept numbers like ‘999’

Using the list type with a demo on availability zones within an elb

You have the following code for creating an elb:



You want to make a variables for the name (called elb_name, restricted to string), availability zones (called az, restricted to list) the idle_timeout (called timeout, restricted to number) and the draining timeout (called timeout as well, thus same variable), thus you now have (note that I commented out the previous demos variable, and added a few more lines from the elc registry example code):

terraform.tfvars U | elb_datatype.tf U | variables.tf U

Terraform-Associate > Practicals > Section-2 > data-types > elb_datatype.tf

```
provider "aws" {
    region = "eu-west-2"
    access_key = "AKIAZ53ZT7YGVJPQUAZ2"
    secret_key = "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"
}

resource "aws_elb" "bar" {
  name               = var.elb_name
  availability_zones = var.az

  listener {
    instance_port     = 8000
    instance_protocol = "http"
    lb_port           = 80
    lb_protocol       = "http"
  }

  health_check {
    healthy_threshold   = 2
    unhealthy_threshold = 2
    timeout             = 3
    target              = "HTTP:8000/"
    interval            = 30
  }

  cross_zone_load_balancing   = true
  idle_timeout                = var.timeout
  connection_draining         = true
  connection_draining_timeout = var.timeout

  tags = {
    Name = "foobar-terraform-elb"
  }
}
```

&

elb_datatype.tf U | variables.tf U

Terraform-Associate > Practicals > Section-2 > d

```
/*variable "usernumber" {
  type = number
}*/

variable "elb_name" {
    type = string
}

variable "az" {
  type = list
}

variable "timeout" {
    type = number
}
```

&

terraform.tfvars U | elb_c

Terraform-Associate > Practicals >

```
#usernumber = 92479238

elb_name="myelb"

az="eu-west-2a"

timeout="400"
```

Running terraform plan results in:

```
Maaz@MacBook-Air data-types % terraform plan

Error: Invalid value for input variable

  on terraform.tfvars line 5:
   5: az="eu-west-2a"

The given value is not suitable for var.az declared at variables.tf:9,1-14: list of any single type required.
```

This is since in the terraform.tfvars file, I did not use a list to specify the az variable which is the type restriction in the variables.tf file. Thus, if I adjust the terraform.tfvars to:

terraform.tfvars U | elb_datatype.t

Terraform-Associate > Practicals > Section-2

```
#usernumber = 92479238

elb_name="myelb"

az=["eu-west-2a","eu-west-2b"]

timeout="400"
```

Now running terraform plan results in success:

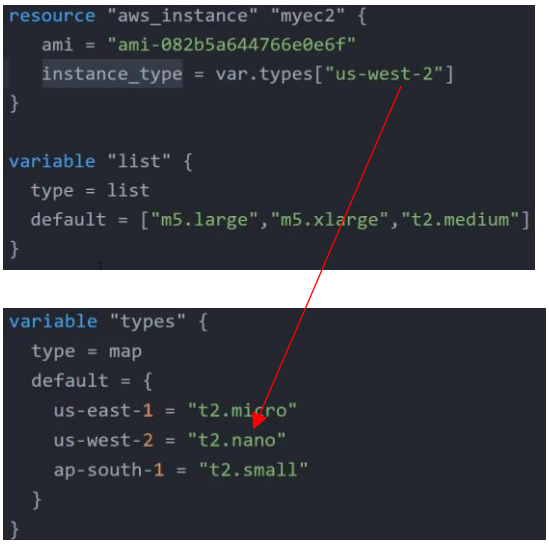
```
Maaz@MacBook-Air data-types % terraform plan

Terraform used the selected providers to generate the following execution plan. Resource actions are indicated with
the following symbols:
  + create

Terraform will perform the following actions:

  # aws_elb.bar will be created
  + resource "aws_elb" "bar" {
      + arn                         = (known after apply)
      + availability_zones          = [
          + "eu-west-2a",
          + "eu-west-2b",
        ]
      + connection_draining         = true
      + connection_draining_timeout = 400
      + cross_zone_load_balancing   = true
      + desync_mitigation_mode      = "defensive"
      + dns_name                    = (known after apply)
      + id                          = (known after apply)
      + idle_timeout                = 400
      + instances                   = (known after apply)
      + internal                    = (known after apply)
      + name                        = "myelb"
      + security_groups             = (known after apply)
      + source_security_group       = (known after apply)
      + source_security_group_id    = (known after apply)
      + subnets                     = (known after apply)
      + tags                        = {
          + "Name" = "foobar-terraform-elb"
        }
      + tags_all                    = {
          + "Name" = "foobar-terraform-elb"
        }
      + zone_id                     = (known after apply)

      + health_check {
          + healthy_threshold   = 2
          + interval            = 30
          + target              = "HTTP:8000/"
          + timeout             = 3
          + unhealthy_threshold = 2
        }

      + listener {
          + instance_port     = 8000
          + instance_protocol = "http"
          + lb_port           = 80
          + lb_protocol       = "http"
        }
    }

Plan: 1 to add, 0 to change, 0 to destroy.

Note: You didn't use the -out option to save this plan, so Terraform can't guarantee to take exactly these actions if
you run "terraform apply" now.
Maaz@MacBook-Air data-types %
```

It is best practice to restrict variables with types like this, as it allows individuals to refer to the variables.tf file to find out the desired data input for the variable that they are using.

Fetching data from a variable's defaults if restricted to type = list

Using the position number of the instance type within the list allows you to reference a specific instance type, so running terraform plan results in provisioning m5.large if 0 was used since it's in the first position.

Fetching data from a variable's defaults if restricted to type = map



Using the name of the name of the default given allows you to reference a specific instance type, so running terraform plan results in provisioning t2.nano if ["us-west-2"] is used.

Count and Count Index Parameters/Arguments

Definition of the Count parameter = The count parameter on resources can simplify configurations and let you scale resources by simply incrementing a number.

Let's assume, you need to create two EC2 instances. One of the common approach is to define two separate resource blocks for aws_instance.

But you can achieve this easily with the count parameter like so (in this case 5):

```
resource "aws_instance" "instance-1" {
   ami = "ami-082b5a644766e0e6f"
   instance_type = "t2.micro"
   count = 5
}
```

Use of count.index = Whenever count is used in resource blocks, an additional count object is available in expressions which allows you to modify the configuration of each instance with incremental numbers to tell them apart. The attribute .${count.index} can be added and the distinct index number beginning with 0 starts with this resource.

For instance, below the terraform code will create 5 IAM users but they will all have the same name (loadbalancer), thus resources where same name cannot be used, the terraform plan/apply will fail (although not in this case):

```
resource "aws_iam_user" "lb" {
  name = "loadbalancer"
  count = 5
  path = "/system/"
}
```

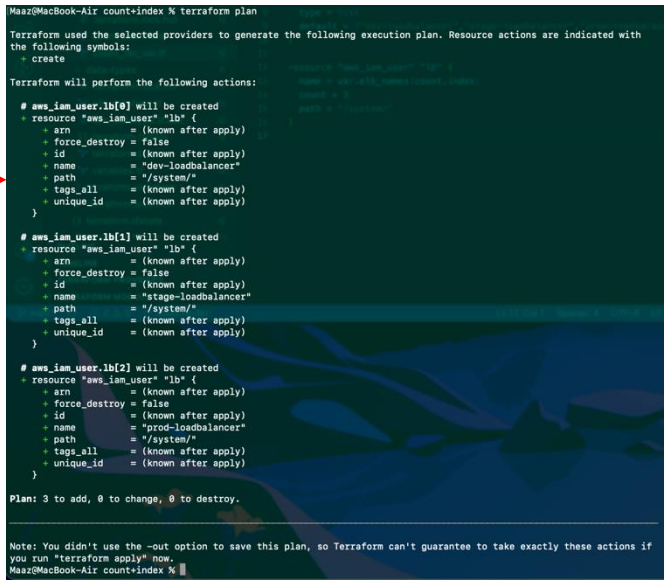
To solve this, apply the count index like so:

```
resource "aws_iam_user" "lb" {
  name = "loadbalancer.${count.index}"
  count = 5
  path = "/system/"
}
```

So now the result is under the name, each iam_user has an index starting from 0 till 4 (since 5 counts were specified and the index always starts at 0).

```
Maaz@MacBook-Air count+index % terraform plan

Terraform used the selected providers to generate the following execution plan. Resource actions are indicated with
the following symbols:
  + create

Terraform will perform the following actions:

  # aws_iam_user.lb[0] will be created
  + resource "aws_iam_user" "lb" {
      + arn           = (known after apply)
      + force_destroy = false
      + id            = (known after apply)
      + name          = "loadbalancer.0"
      + path          = "/system/"
      + tags_all      = (known after apply)
      + unique_id     = (known after apply)
    }

  # aws_iam_user.lb[1] will be created
  + resource "aws_iam_user" "lb" {
      + arn           = (known after apply)
      + force_destroy = false
      + id            = (known after apply)
      + name          = "loadbalancer.1"
      + path          = "/system/"
      + tags_all      = (known after apply)
      + unique_id     = (known after apply)
    }

  # aws_iam_user.lb[2] will be created
  + resource "aws_iam_user" "lb" {
      + arn           = (known after apply)
      + force_destroy = false
      + id            = (known after apply)
      + name          = "loadbalancer.2"
      + path          = "/system/"
      + tags_all      = (known after apply)
      + unique_id     = (known after apply)
    }

  # aws_iam_user.lb[3] will be created
  + resource "aws_iam_user" "lb" {
      + arn           = (known after apply)
      + force_destroy = false
      + id            = (known after apply)
      + name          = "loadbalancer.3"
      + path          = "/system/"
      + tags_all      = (known after apply)
      + unique_id     = (known after apply)
    }

  # aws_iam_user.lb[4] will be created
  + resource "aws_iam_user" "lb" {
      + arn           = (known after apply)
      + force_destroy = false
      + id            = (known after apply)
      + name          = "loadbalancer.4"
      + path          = "/system/"
      + tags_all      = (known after apply)
      + unique_id     = (known after apply)
    }

Plan: 5 to add, 0 to change, 0 to destroy.

Note: You didn't use the -out option to save this plan, so Terraform can't guarantee to take exactly these actions if
you run "terraform apply" now.
```

The above case is okay, but its better to be even more specific with the names. You can use a variable that has the names in a list format that you wish to apply to the first, second and third count for example, then you apply the [count.index] to the variable itself like so:



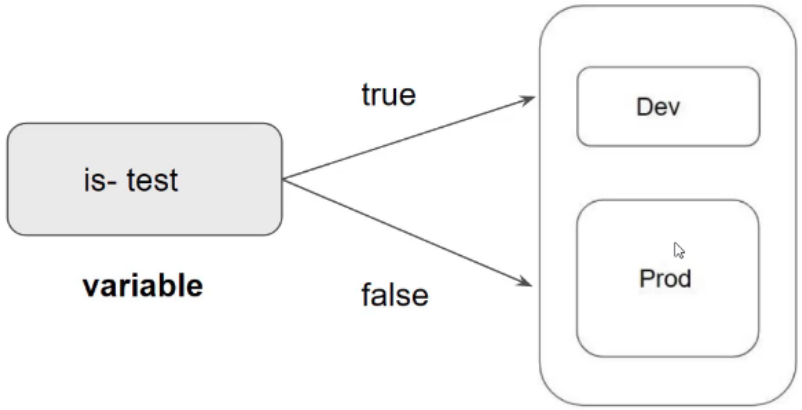
Conditional Expressions

A conditional expression uses the value of a bool expression to select one of two values. The syntax is as follows:

```
condition ? true_val : false_val
```

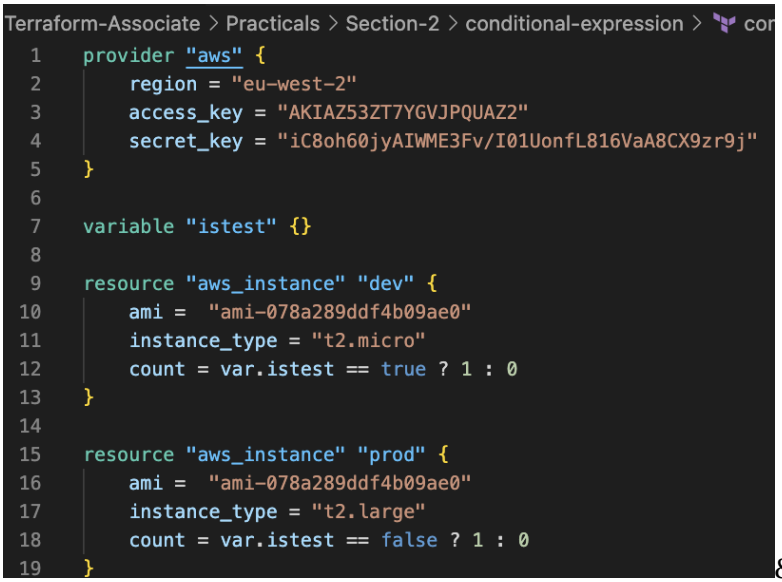
If the condition is true, then the result is true_val and vice versa.

Here is an example:

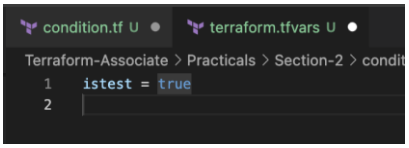


The following will create either both instances for dev and prod:

But from this vonfig file, they want to specify if they want a prod or dev instance. So to do this, create a variable called istest and then add as we shown in the diagram, if the is-test is true, then the resource should be created for the Dev environment and vice versa:



&

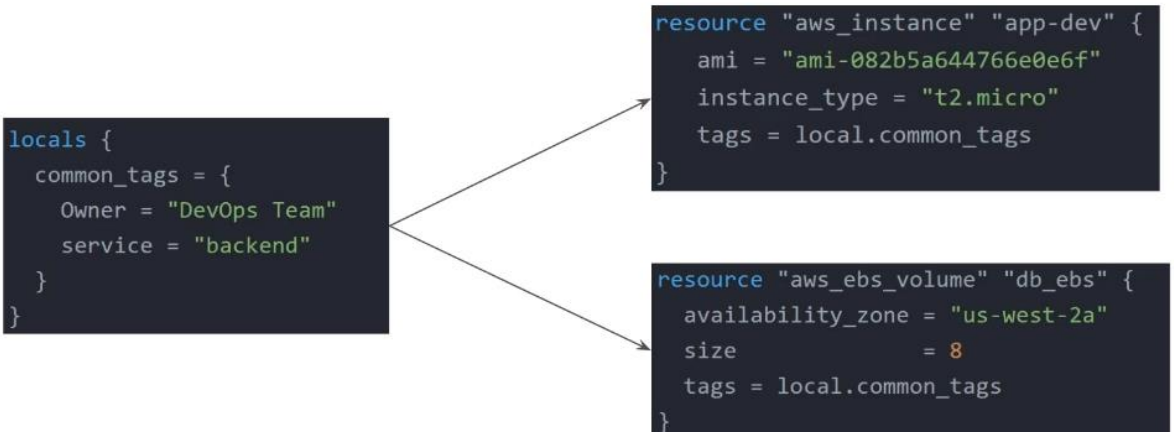


Breaking this down only for the dev reosurce:

- The ‘condition ?’ was: var.interest == true ?
- The ‘true_val’ = 1
- The ‘false_val’ = 0
    - Remember, if var.interest is true, then the count will be 1 and then the resource will be made 1 times; but if false the count is 0, then this resource will be made 0 times
- In this case since it is true as determined in the terraform.tfvars, the t2.micro will be provisioned under the name of dev

Local Values

A local value assigns a name to an expression which then allows it to be used as many times as it is desired within a module which proves to be a major benefit:



- Local values are helpful to avoid repeating the same values or expressions multiple times in a configuration.
- If overused, they can make a config hard to due to hiding the actual values
- Use them in moderation in situations where a single value/result is used in many places with the likelihood of being changed in the future

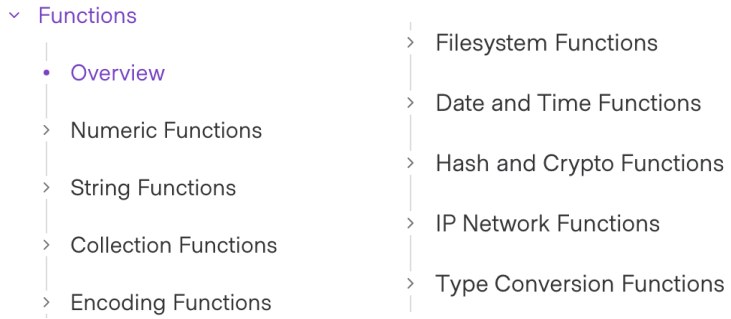
Terraform Functions

There are a number of built in functions which can be used to transform and combine values. The syntax is a function name followed by comma-separated arguments in parentheses:

function (argument1,argument2,etc) for example:

```
> max(5, 12, 9)
12
```

Function categories taken from the terraform website; these can be expanded, and the functions can be found within them:



You can find in the terraform docs the built-in functions: https://www.terraform.io/language/functions

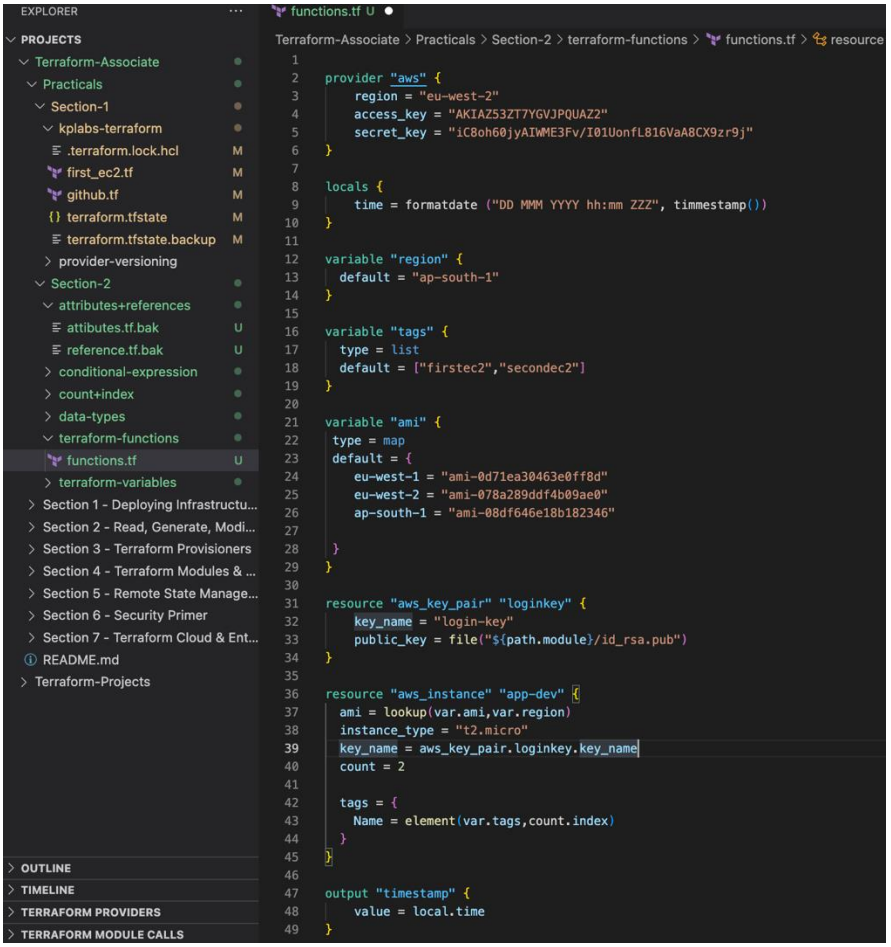
You can test the commands by running the ‘terraform console’ command in your CLI:

```
C:\Users\Zeal Vora>terraform console
[J> max(10,20,30)
30
[J> min(10,20,30)
10
[J>
```

In the terraform website for functions, there are example usage of each available function which helps to understand how you can use it in your configuration. Here, we take an example where we create an ec2 instance and a keypair. While doing so, we incorporate the following functions:

- file
- lookup
- element
- formatdate
- timestamp

Lookup function breakdown



- Syntax taken from the website:

`lookup` retrieves the value of a single element from a map, given its key. If the given key does not exist, the given default value is returned instead.

```
lookup(map, key, default)
```
Copy

- `ami = lookup(var.ami,var.region)`

References:

-
```
variable "region" {
  default = "ap-south-1"
}
```
&
```
variable "ami" {
 type = map
 default = {
    eu-west-1 = "ami-0d71ea30463e0ff8d"
    eu-west-2 = "ami-078a289ddf4b09ae0"
    ap-south-1 = "ami-08df646e18b182346"

 }
}
```

- So, in our case we didn't define the default, but it still works.
- Our region's variable defaults ap-south-1, thus the key would be this and this will be used in the map under the ami's variable. Thus, the ami of ap-south-1 will be used.

Element function breakdown

- Syntax and understanding taken from website: Example:
```
> element(["a", "b", "c"], 1)
b
```

`element` retrieves a single element from a list.

```
element(list, index)
```
Copy

-
```
resource "aws_instance" "app-dev" {
  ami = lookup(var.ami,var.region)
  instance_type = "t2.micro"
  key_name = aws_key_pair.loginkey.key_name
  count = 2

  tags = {
    Name = element(var.tags,count.index)
  }
}
```

- References:
```
variable "tags" {
  type = list
  default = ["firstec2","secondec2"]
}
```

- Since our resource is created twice, the first one will take a count.index number of 0 and the second of 1. This means that the first instance's tag will retrieve a tag of "firstec2" from the list and the second will retrieve "secondec2" from the list

File function breakdown

- Syntax and understanding taken from website:

`file` reads the contents of a file at the given path and returns them as a string.

```
file(path)
```
Example:
```
> file("${path.module}/hello.txt")
Hello World
```

-
```
resource "aws_key_pair" "loginkey" {
    key_name = "login-key"
    public_key = file("${path.module}/id_rsa.pub")
}
```
- So in our case, this will return the contents of a id_rsa.pub

Timestamp and formatdate function breakdown

- Syntax and understanding taken from website:

`timestamp` returns a UTC timestamp string in RFC 3339 format. Example:
```
> timestamp()
2018-05-13T07:44:12Z
```

`formatdate` converts a timestamp into a different time format.

```
formatdate(spec, timestamp)
```

Example:
```
> formatdate("DD MMM YYYY hh:mm ZZZ", "2018-01-02T23:12:01Z")
02 Jan 2018 23:12 UTC
```

-
```
output "timestamp" {
    value = local.time
}
```
- References:
```
locals {
    time = formatdate ("DD MMM YYYY hh:mm ZZZ", timmestamp())
}
```
- So the result is when the command is run there's an output of the timestamp in our desired format

Terraform Plan:

```
Maaz@MacBook-Air terraform-functions % terraform apply

Terraform used the selected providers to generate the following executio
symbols:
  + create

Terraform will perform the following actions:

  # aws_instance.app-dev[0] will be created
  + resource "aws_instance" "app-dev" {
      + ami                                  = "ami-08df646e18b182346"
      + arn                                  = (known after apply)
      + associate_public_ip_address          = (known after apply)
      + availability_zone                    = (known after apply)
      + cpu_core_count                       = (known after apply)
      + cpu_threads_per_core                 = (known after apply)
      + disable_api_stop                     = (known after apply)
      + disable_api_termination              = (known after apply)
      + ebs_optimized                        = (known after apply)
      + get_password_data                    = false
      + host_id                              = (known after apply)
      + id                                   = (known after apply)
      + instance_initiated_shutdown_behavior = (known after apply)
      + instance_state                       = (known after apply)
      + instance_type                        = "t2.micro"
      + ipv6_address_count                   = (known after apply)
      + ipv6_addresses                       = (known after apply)
      + key_name                             = "login-key"
      + monitoring                           = (known after apply)
      + outpost_arn                          = (known after apply)
      + password_data                        = (known after apply)
      + placement_group                      = (known after apply)
      + placement_partition_number           = (known after apply)
      + primary_network_interface_id         = (known after apply)
      + private_dns                          = (known after apply)
      + private_ip                           = (known after apply)
      + public_dns                           = (known after apply)
      + public_ip                            = (known after apply)
      + secondary_private_ips                = (known after apply)
      + security_groups                      = (known after apply)
      + source_dest_check                    = true
      + subnet_id                            = (known after apply)
      + tags                                 = {
          + "Name" = "firstec2"
        }
      + tags_all                             = {
          + "Name" = "firstec2"
        }
```

```
# aws_instance.app-dev[1] will be created
+ resource "aws_instance" "app-dev" {
    + ami                                  = "ami-08df646e18b182346"
    + arn                                  = (known after apply)
    + associate_public_ip_address          = (known after apply)
    + availability_zone                    = (known after apply)
    + cpu_core_count                       = (known after apply)
    + cpu_threads_per_core                 = (known after apply)
    + disable_api_stop                     = (known after apply)
    + disable_api_termination              = (known after apply)
    + ebs_optimized                        = (known after apply)
    + get_password_data                    = false
    + host_id                              = (known after apply)
    + id                                   = (known after apply)
    + instance_initiated_shutdown_behavior = (known after apply)
    + instance_state                       = (known after apply)
    + instance_type                        = "t2.micro"
    + ipv6_address_count                   = (known after apply)
    + ipv6_addresses                       = (known after apply)
    + key_name                             = "login-key"
    + monitoring                           = (known after apply)
    + outpost_arn                          = (known after apply)
    + password_data                        = (known after apply)
    + placement_group                      = (known after apply)
    + placement_partition_number           = (known after apply)
    + primary_network_interface_id         = (known after apply)
    + private_dns                          = (known after apply)
    + private_ip                           = (known after apply)
    + public_dns                           = (known after apply)
    + public_ip                            = (known after apply)
    + secondary_private_ips                = (known after apply)
    + security_groups                      = (known after apply)
    + source_dest_check                    = true
    + subnet_id                            = (known after apply)
    + tags                                 = {
        + "Name" = "secondec2"
      }
    + tags_all                             = {
        + "Name" = "secondec2"
      }
```

```
# aws_key_pair.loginkey will be created
+ resource "aws_key_pair" "loginkey" {
    + arn             = (known after apply)
    + fingerprint     = (known after apply)
    + id              = (known after apply)
    + key_name        = "login-key"
    + key_name_prefix = (known after apply)
    + key_pair_id     = (known after apply)
    + key_type        = (known after apply)
    + public_key      = "Sadly i dont have it lol, shame"
    + tags_all        = (known after apply)
  }
```

```
Changes to Outputs:
  + timestamp = (known after apply)

Do you want to perform these actions?
  Terraform will perform the actions described above.
  Only 'yes' will be accepted to approve.

  Enter a value: yes
```

Apply showing output:

```
timestamp = "31 Jul 2022 11:36 UTC"
```

Zipmap function breakdown

The zip map function constructs a map from a list of keys and a corresponding list of values like so:

Explanation, syntax and example taken from terraform website:

`zipmap` constructs a map from a list of keys and a corresponding list of values.

```
zipmap(keyslist, valueslist)
```

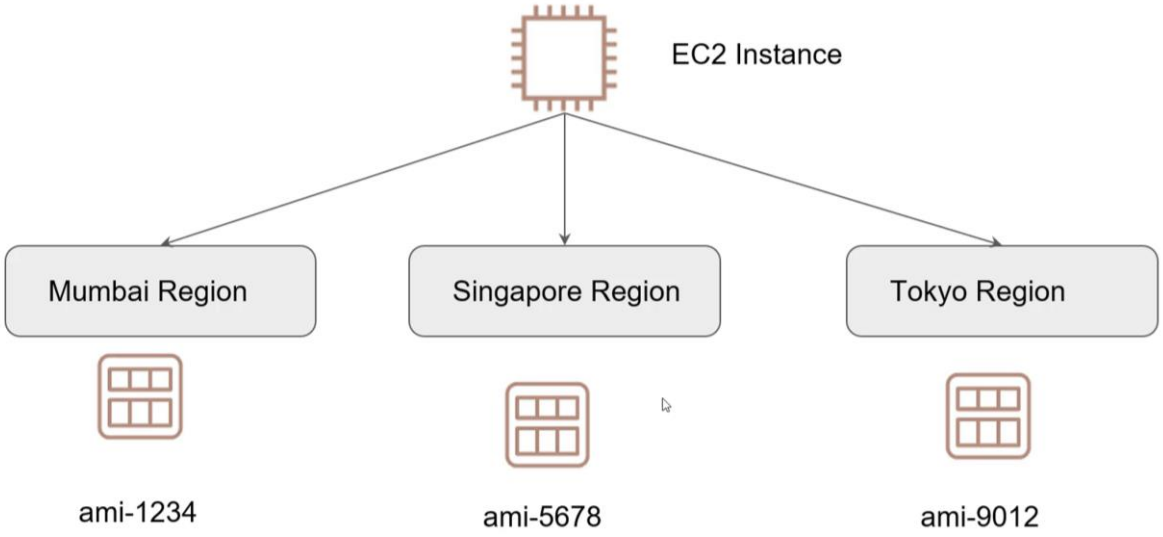
```
> zipmap(["a", "b"], [1, 2])
{
  "a" = 1
  "b" = 2
}
```

- Can be used when creating multiple IAM users and you want the output that maps the IAM users with their ARNs:

```
zipmap = {
  "demo-user.0" = "arn:aws:iam::018721151861:user/system/demo-user.0"
  "demo-user.1" = "arn:aws:iam::018721151861:user/system/demo-user.1"
  "demo-user.2" = "arn:aws:iam::018721151861:user/system/demo-user.2"
}
```

Data Sources

Data sources allow data to be fetched or computed for use elsewhere within your Terraform configuration:



- Look at the above ami ID's; we see that they are different for each region, and you need to specify this each time you create the resource. Moreover, this may update over time as well, so again you'll need to update this. You'd rather automate this.
- Until now, we've been manually hardcoding the ami like so, but this ami will not work in a different region:

```
provider "aws" {
    region = "eu-west-2"
    access_key = "AKIAZ53ZT7YGVJPQUAZ2"
    secret_key = "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"
}

resource "aws_instance" "myec2" {
    ami =  "ami-078a289ddf4b09ae0"
    instance_type = "t2.micro"
}
```

SOLUTION (data source code):

- The data source is defined under a block known as 'data' (similar to how we have resource blocks)
- This block will read from a specific data source (aws_ami in this case) and the it will export what it reads under what you have defined ("app_ami" in this case)

```
data "aws_ami" "app_ami" {
  most_recent = true
  owners = ["amazon"]


  filter {
    name   = "name"
    values = ["amzn2-ami-hvm*"]
  }
}
```



```
resource "aws_instance" "instance-1" {
    ami = data.aws_ami.app_ami.id
   instance_type = "t2.micro"
}
```

- So you must specify the owner (as it can be Google/Amazon etc)
- Specify under filters, and in our case that you want the amazon linux2 based ami:

  ```
  "amzn2-ami-hvm*"
  ```

- Specify that you want the most recent ami of a specific region

Demo:

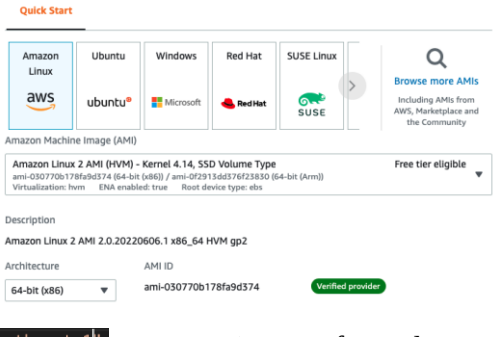
data-source.tf U

Terraform-Associate > Practicals > Section-2 > data-sources > data-source.tf > ...

```
1
2   provider "aws" {
3       region = "eu-west-2"
4       access_key = "AKIAZ53ZT7YGVJPQUAZ2"
5       secret_key = "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"
6   }
7
8   data "aws_ami" "app_ami" {
9     most_recent = "true"
10    owners = ["amazon"]
11
12        filter {
13          name =  "name"
14          values = ["amzn2-ami-hvm*"]
15      }
16  }
17
18  resource "aws_instance" "myec2" {
19      ami =  data.aws_ami.app_ami.id
20      instance_type = "t2.micro"
21  }
```

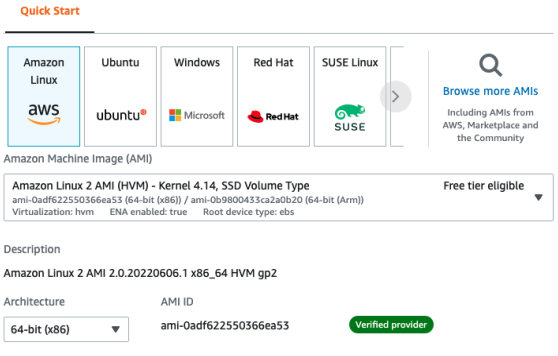
- Terraform plan and ami found in the console for eu-west-2:

```
Maaz@MacBook-Air data-sources % terraform plan
data.aws_ami.app_ami: Reading...
data.aws_ami.app_ami: Read complete after 0s [id=ami-030770b178fa9d374]

Terraform used the selected providers to generate the following execution pla
symbols:
  + create

Terraform will perform the following actions:

  # aws_instance.myec2 will be created
  + resource "aws_instance" "myec2" {
      + ami                                  = "ami-030770b178fa9d374"
      + arn                                  = (known after apply)
```



- Just changing the region in the code to `region = "ap-southeast-1"`, now running terraform plan:

```
Maaz@MacBook-Air data-sources % terraform plan
data.aws_ami.app_ami: Reading...
data.aws_ami.app_ami: Read complete after 1s [id=ami-0adf622550366ea53]

Terraform used the selected providers to generate the following execution pla
symbols:
  + create

Terraform will perform the following actions:

  # aws_instance.myec2 will be created
  + resource "aws_instance" "myec2" {
      + ami                                  = "ami-0adf622550366ea53"
```

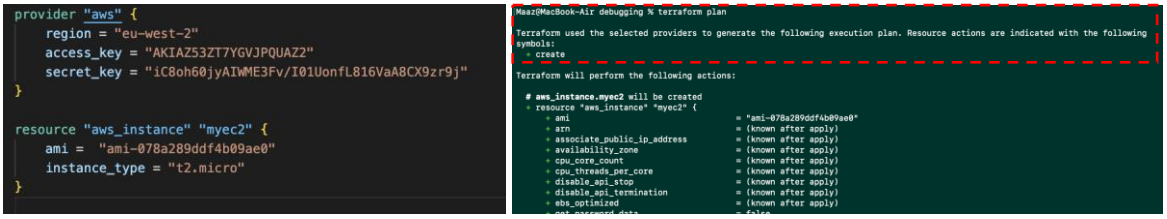


- You can even use the ami's you may have created by setting the owners to self like so:

```
data "aws_ami" "example" {
  most_recent = true

  owners = ["self"]
  tags = {
    Name   = "app-server"
    Tested = "true"
  }
}
```

Debugging Terraform

Terraform has detailed logs that can be enabled by setting the TF_LOG environment variable to any value. You can set TF_LOG to one of the log levels TRACE, DEBUG, INFO, WARN or ERROR to change the verbosity of logs. Say you have simple manifest below and you run terraform plan:



```
provider "aws" {
    region = "eu-west-2"
    access_key = "AKIAZ53ZT7YGVJPQUAZ2"
    secret_key = "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"
}

resource "aws_instance" "myec2" {
    ami =  "ami-078a289ddf4b09ae0"
    instance_type = "t2.micro"
}
```

```
Maaz@MacBook-Air debugging % terraform plan

Terraform used the selected providers to generate the following execution plan. Resource actions are indicated with the following
symbols:
  + create

Terraform will perform the following actions:

  # aws_instance.myec2 will be created
  + resource "aws_instance" "myec2" {
      + ami                          = "ami-078a289ddf4b09ae0"
      + arn                          = (known after apply)
      + associate_public_ip_address  = (known after apply)
      + availability_zone            = (known after apply)
      + cpu_core_count               = (known after apply)
      + cpu_threads_per_core         = (known after apply)
      + disable_api_stop             = (known after apply)
      + disable_api_termination      = (known after apply)
      + ebs_optimized                = (known after apply)
```

But now if you set the TF_LOG environment variable for TRACE, and in the CLI you write:

Now running terraform plan, look at the amount of logs:

```
Maaz@MacBook-Air debugging % export TF_LOG=TRACE
Maaz@MacBook-Air debugging % terraform plan
2022-07-31T16:45:55.596+0100 [INFO]  Terraform version: 1.2.6
2022-07-31T16:45:55.596+0100 [DEBUG] using github.com/hashicorp/go-tfe v1.0.0
2022-07-31T16:45:55.596+0100 [DEBUG] using github.com/hashicorp/hcl/v2 v2.12.0
2022-07-31T16:45:55.596+0100 [DEBUG] using github.com/hashicorp/terraform-config-inspect v0.0.0-20210209133302-4fd17a0faac2
2022-07-31T16:45:55.596+0100 [DEBUG] using github.com/hashicorp/terraform-svchost v0.0.0-20200729002733-f050f53b9734
2022-07-31T16:45:55.596+0100 [DEBUG] using github.com/zclconf/go-cty v1.10.0
2022-07-31T16:45:55.596+0100 [INFO]  Go runtime version: go1.18.1
2022-07-31T16:45:55.596+0100 [INFO]  CLI args: []string{"terraform", "plan"}
2022-07-31T16:45:55.596+0100 [TRACE] Stdout is a terminal of width 136
2022-07-31T16:45:55.596+0100 [TRACE] Stderr is a terminal of width 136
2022-07-31T16:45:55.596+0100 [TRACE] Stdin is a terminal
2022-07-31T16:45:55.596+0100 [DEBUG] Attempting to open CLI config file: /Users/Maaz/.terraformrc
2022-07-31T16:45:55.596+0100 [DEBUG] File doesn't exist, but doesn't need to. Ignoring.
2022-07-31T16:45:55.596+0100 [DEBUG] ignoring non-existing provider search directory terraform.d/plugins
2022-07-31T16:45:55.596+0100 [DEBUG] ignoring non-existing provider search directory /Users/Maaz/.terraform.d/plugins
2022-07-31T16:45:55.596+0100 [DEBUG] ignoring non-existing provider search directory /Users/Maaz/Library/Application Support/io.terrafor
m/plugins
2022-07-31T16:45:55.596+0100 [DEBUG] ignoring non-existing provider search directory /Library/Application Support/io.terraform/plugins
2022-07-31T16:45:55.596+0100 [INFO]  CLI command args: []string{"plan"}
2022-07-31T16:45:55.597+0100 [TRACE] Meta.Backend: no config given or present on disk, so returning nil config
2022-07-31T16:45:55.597+0100 [TRACE] Meta.Backend: backend has not previously been initialized in this working directory
2022-07-31T16:45:55.597+0100 [DEBUG] New state was assigned lineage "130e696a-e2ce-e41b-7698-11540a6b2634"
2022-07-31T16:45:55.597+0100 [TRACE] Meta.Backend: using default local state only (no backend configuration, and no existing initialized
 backend)
2022-07-31T16:45:55.597+0100 [TRACE] Meta.Backend: instantiated backend of type <nil>
2022-07-31T16:45:55.597+0100 [TRACE] providercache.fillMetaCache: scanning directory .terraform/providers
2022-07-31T16:45:55.597+0100 [TRACE] getproviders.SearchLocalDirectory: found registry.terraform.io/hashicorp/aws v4.24.0 for darwin_arm
64 at .terraform/providers/registry.terraform.io/hashicorp/aws/4.24.0/darwin_arm64
2022-07-31T16:45:55.597+0100 [TRACE] providercache.fillMetaCache: including .terraform/providers/registry.terraform.io/hashicorp/aws/4.2
```

Note, there is too much here to actually show on the screen. This is what you would share if you were trying to DEBUG. You can also export these logs using a path as shown below; then running terraform plan results in those extralogs in that /tmp/terraform-crash.log which you specified:

```
Maaz@MacBook-Air debugging % export TF_LOG_PATH=/tmp/terraform-crash.log
```

Teraform Format (fmt) and Validate

terraform fmt

The terraform fmt command is used to rewrite Terraform configuration files to take care of the overall formatting so that it is readable:

Before fmt

```
provider "aws" {
      region     = "us-west-2"
  access_key = "AKIAQIW66DN2W7WOYRGY"
  secret_key    = "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"
  version    = ">=2.10,<=2.30"
}
```

After fmt

```
provider "aws" {
  region     = "us-west-2"
  access_key = "AKIAQIW66DN2W7WOYRGY"
  secret_key = "KOy9/Qwsy4aTltQliONu1TN4o9vX9t5UVwpKauIM"
  version    = ">=2.10,<=2.30"
}
```

In the CLI, just run the terraform fmt command in your directory:

```
Maaz@MacBook-Air debugging % terraform fmt
```

terraform Validate

Terraform Validate checks whether a configuration is syntactically valid. It can check things like unsupported arguments, undeclared variables and others:

```
resource "aws_instance" "myec2" {
  ami           = "ami-082b5a644766e0e6f"
  instance_type = "t2.micro"
  sky = "blue"
}
```

→

```
bash-4.2# terraform validate

Error: Unsupported argument

  on validate.tf line 10, in resource "aws_instance" "myec2":
  10:   sky = "blue"

An argument named "sky" is not expected here.
```

Terraform plan will show an error as well just like this if it is there.

Load order and Semantics

When you write terraform files, you don't need to write them all into one manifest in the directory. Recall how we had a separate file within our directory called variables.tf before.

Terraform generally loads all the configuration files (also known as manifests) within the directory specified in alphabetical order.

These files need to have either of the following file extensions:

- .tf
- .tf.json

When writing production code its not recommended to write everything into a single file since it wont be clean. Rather, you can split the code into multiple manifests (means configuration files), like provider.tf, variables.tf, for resources {ec2.tf, iam_user.tf, e.t.c.}. Also you need to be careful when you're creating resources, that the same name is not used for a resource, otherwise terraform plan will fail.

Dynamic Blocks

Under many scenarios there are repeated blocks within a block, these are known as nested blocks.

The issue with them is the result of long code being created which makes management of the code difficult, for instance needing multiple ingress blocks within the security group resource like so:

```
ingress {
  from_port   = 9200
  to_port     = 9200
  protocol    = "tcp"
  cidr_blocks = ["0.0.0.0/0"]
}
```

```
ingress {
  from_port   = 8300
  to_port     = 8300
  protocol    = "tcp"
  cidr_blocks = ["0.0.0.0/0"]
}
```

Now, imagine you needed to add 40 ports to a security group resource, then you would need to create 40 ingress blocks.

Dynamic Block allows you to construct repeatable nested blocks which is supported inside resource, data, provider, and provisioner blocks. Take this example here where a dynamic block is created for this ingress block. We also need to create a variable with all the code that defines the var.ingress_ports, so you can have the ports you wish to add there instead of being separate ingress blocks.

```
dynamic "ingress" {
  for_each = var.ingress_ports
  content {
    from_port   = ingress.value
    to_port     = ingress.value
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }
}
```

Demo

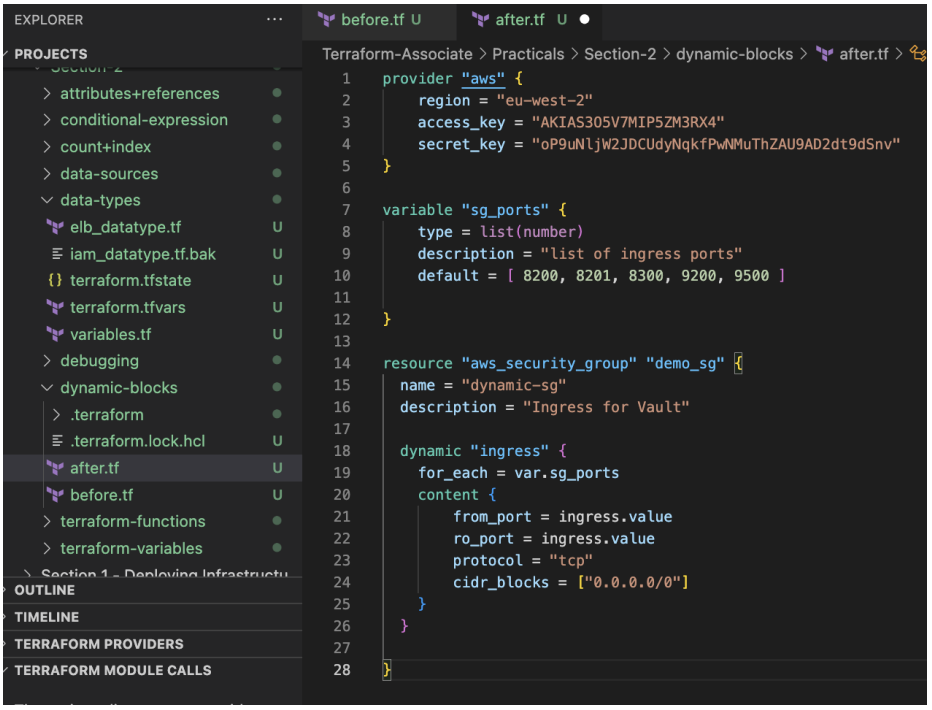
Say you wanted to achieve this:

dynamic_ingress.tf U

Terraform-Associate > Practicals > Section-2 > dynamic-blocks > dynamic_ingress.tf

```
provider "aws" {
    region = "eu-west-2"
    access_key = "AKIAS3O5V7MIP5ZM3RX4"
    secret_key = "oP9uNljW2JDCUdyNqkfPwNMuThZAU9AD2dt9dSnv"
}

resource "aws_security_group" "demo_sg" {
  name = "sample-sg"

  ingress {
    from_port = 8200
    to_port   = 8200
    protocol  = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }

    ingress {
    from_port = 8201
    to_port   = 8201
    protocol  = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }

    ingress {
    from_port = 8300
    to_port   = 8300
    protocol  = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }

    ingress {
    from_port = 8200
    to_port   = 8200
    protocol  = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }

    ingress {
    from_port = 9500
    to_port   = 9500
    protocol  = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }
}
```

So now we have this:



Breakdown:

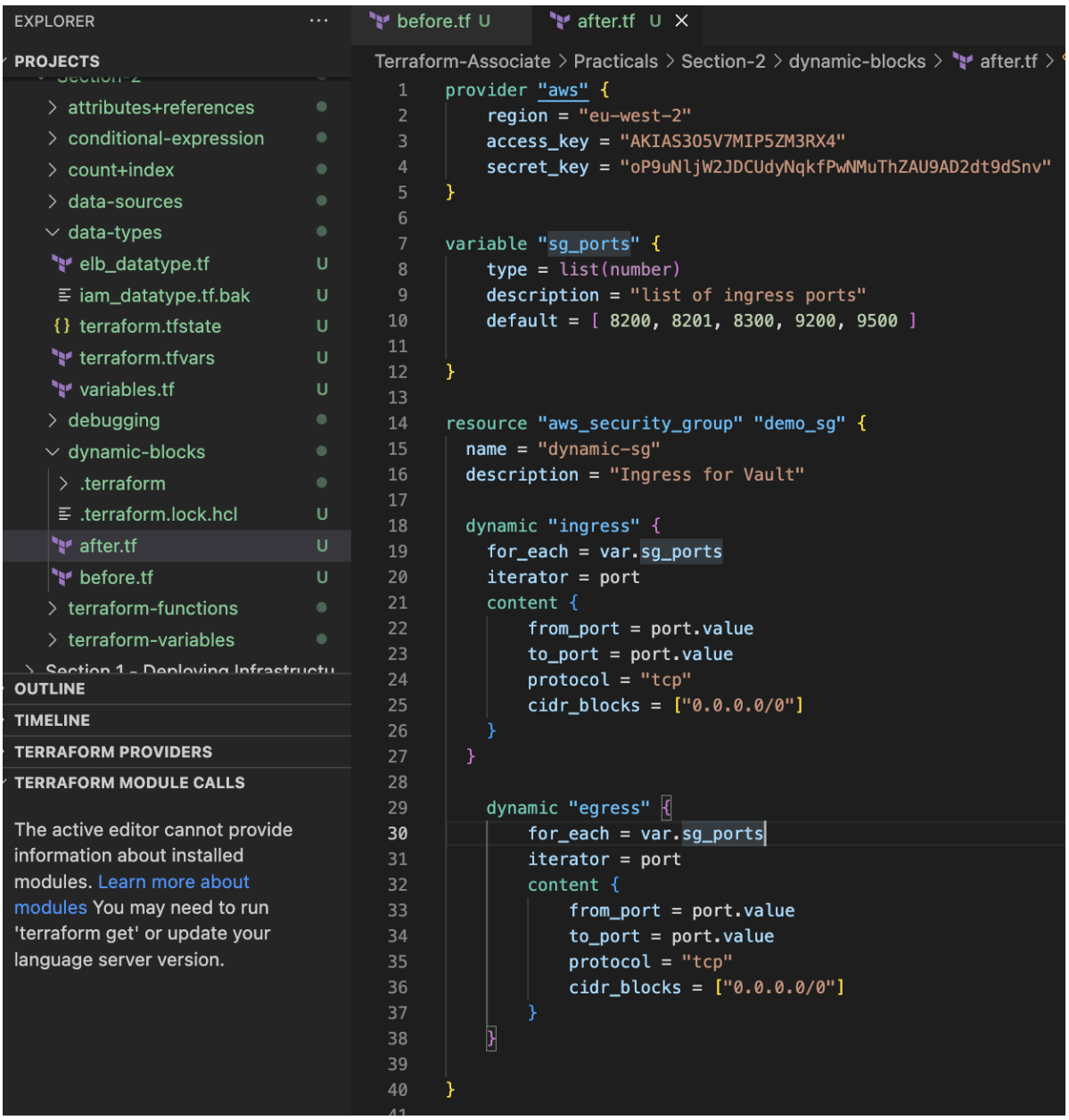
- for_each = var.sg_ports : This means that for each value in that variable list (named sg_ports), the dynamic block that you have defined will be created. As in this section of the block:

```
from_port = ingress.value
ro_port = ingress.value
protocol = "tcp"
cidr_blocks = ["0.0.0.0/0"]
```

- ingress.value : This is where the list is referenced from the variables. So this will take the value of 8200, then 8201 for the next block etc.
- If you wanted to rename ingress.value you can re-write the code using the iterator argument like so:

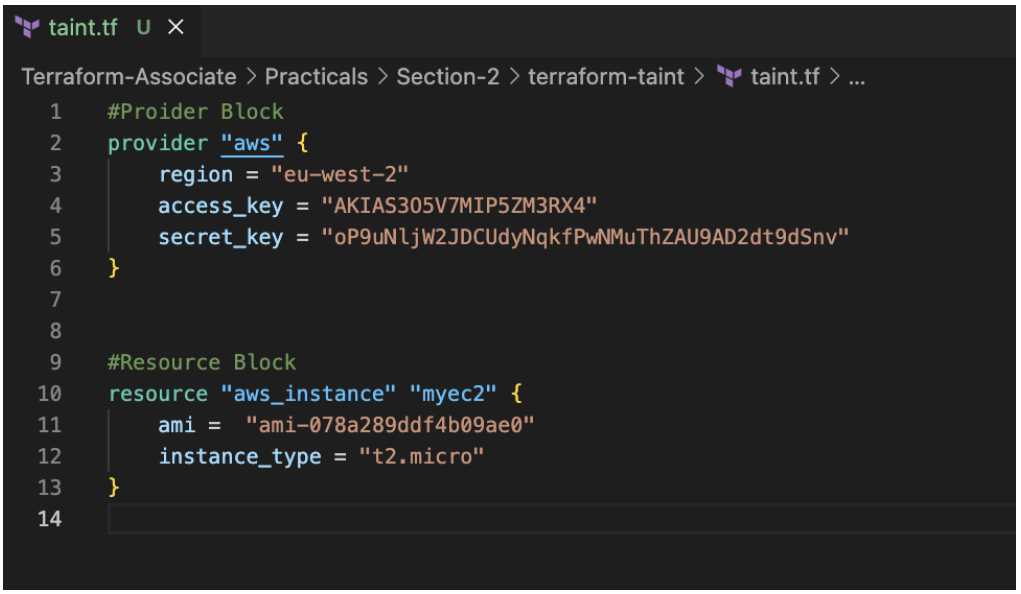
```
resource "aws_security_group" "demo_sg" {
  name = "dynamic-sg"
  description = "Ingress for Vault"

  dynamic "ingress" {
    for_each = var.sg_ports
    iterator = port
    content {
        from_port = port.value
        to_port = port.value
        protocol = "tcp"
        cidr_blocks = ["0.0.0.0/0"]
    }
  }
```

You can also do all of this for the outbound rules by replacing ingress to egress like so:

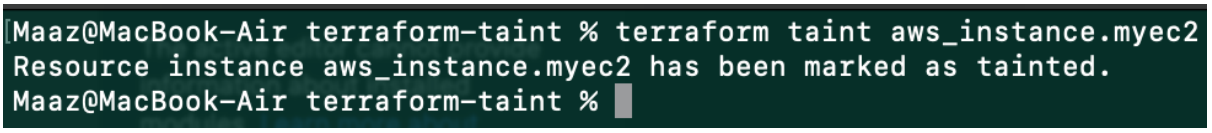


Terraform Taint

If a resource after it has been created had changes done to it and you wanted to fix it, you can either import the changes to Terraform or deleted it and recreate that resource. The second is what 'terraform taint' does. You can mark the resource as taint and then when you next do terraform apply, it will delete and re-create the resource. Say you create a new resource:

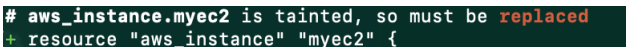


Now you terraform apply and then there were some changes done by someone to this resource in for example the console. Now if you wish to mark it as tainted, you run the following command in the you CLI:



Now running terraform plan:

&



Here are a few important points:

The command will not modify the infrastructure, rather it modifies the state file so that it can be marked as tainted.

When a resource is marked as tainted, the next plan will show that the resource will be destroyed and re-created and thus the next apply will implement this.

Note that tainting a resource for re-creation may affect resources that depend on the newly tainted resource.

- For the last point, if you were dealing with an ec2 instance, the new resource that is created will probably have a new IP, so you need to modify your DNS to refer to the new IP
- Also you can know a file is tainted by looking into the tfstate files

Splat Expression

Definition: Allows you to get the list of all the attributes:

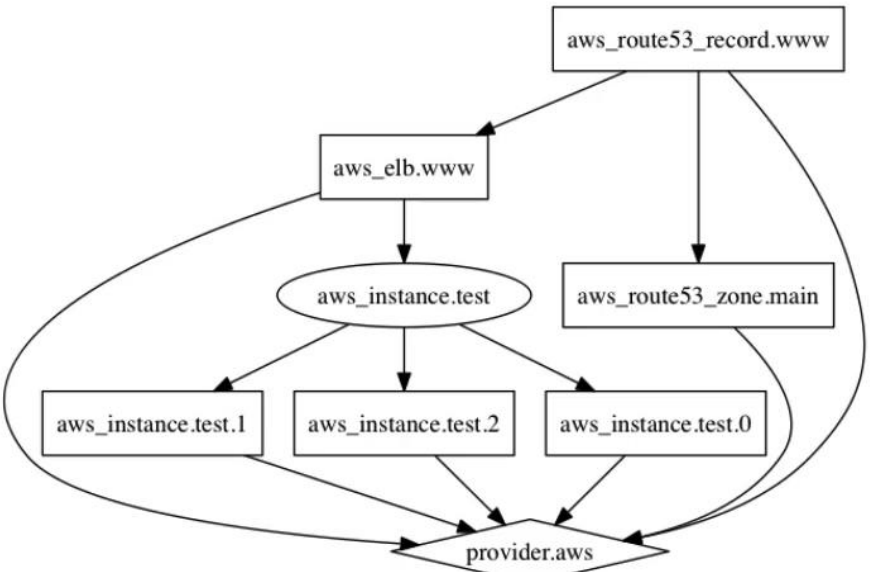
```
resource "aws_iam_user" "lb" {
  name = "iamuser.${count.index}"
  count = 3
  path = "/system/"
}

output "arns" {
  value = aws_iam_user.lb[*].arn
}
```

- Here, we create 3 iam users, as well as use the count.index to name them from 0 to 2
- If you look in value, there is [*].arn in the output
    - This star means you will get all the users, but if you specified [0] for example, then you'd get only the first users output

```
Outputs:

arns = [
  "arn:aws:iam::795574780076:user/system/iamuser.0",
  "arn:aws:iam::795574780076:user/system/iamuser.1",
  "arn:aws:iam::795574780076:user/system/iamuser.2",
]
```

Terraform Graph

Gives a visual representation of either a configuration or execution plan. The output of the terraform graph command is a file in DOT format which can be easily converted into an image like the one shown below:



- Say you had a manifest that creates an instance, an eip and a security group that depends on the eip

```
resource "aws_eip" "lb" {
  instance = aws_instance.myec2.id
  vpc      = true
}

resource "aws_security_group" "allow_tls" {
  name        = "allow_tls"
```

&

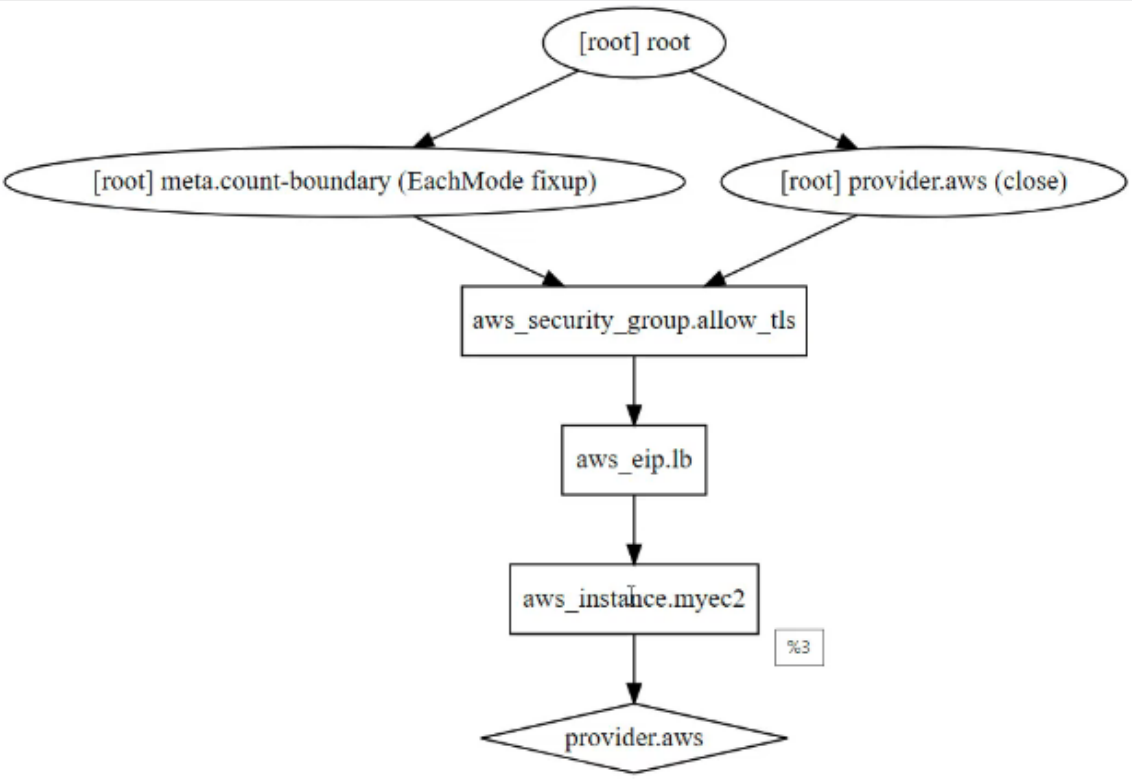
```
resource "aws_security_group" "allow_tls" {
  name        = "allow_tls"

  ingress {
    description = "TLS from VPC"
    from_port   = 443
    to_port     = 443
    protocol    = "tcp"
    cidr_blocks = ["${aws_eip.lb.private_ip}/32"]

  }
```

- Run in your CLI ‘terraform graph’
    - Here it is exported to graph.dot:

```
digraph {
  compound = "true"
  newrank = "true"
  subgraph "root" {
    "[root] aws_eip.lb" [label = "aws_eip.lb", shape = "box"]
    "[root] aws_instance.myec2" [label = "aws_instance.myec2", shape = "box"]
    "[root] aws_security_group.allow_tls" [label = "aws_security_group.allow_tls", s
    "[root] provider.aws" [label = "provider.aws", shape = "diamond"]
    "[root] aws_eip.lb" -> "[root] aws_instance.myec2"
    "[root] aws_instance.myec2" -> "[root] provider.aws"
    "[root] aws_security_group.allow_tls" -> "[root] aws_eip.lb"
    "[root] meta.count-boundary (EachMode fixup)" -> "[root] aws_security_group.allo
    "[root] provider.aws (close)" -> "[root] aws_security_group.allow_tls"
    "[root] root" -> "[root] meta.count-boundary (EachMode fixup)"
    "[root] root" -> "[root] provider.aws (close)"
```

- You can convert this into a nice diagram by downloading a suitable tool from, graphvic.gitlab.io/download

- `bash-4.2# cat graph.dot | dot -Tsvg > graph.svg`
- Now if you open this graph.svg file and you open this with google chrome, you see this:



Saving Terraform Plan

When terraform apply is run, it can be saved to a path that you specify.

This plan can then be used with terraform apply down the line, this makes certain that only the changes that are mentioned in this terraform occurs – someone may have changed the configuration in this time for instance. This is how you would run the command:

```
terraform plan -out=path
```

Terraform Output

The terraform output command is used to extract the value of an output variable from the state file. Here is an example usage from the command line:

```
Last login: Wed Aug  3 11:51:43 on ttys000
Maaz@MacBook-Air ~ % terraform output iam_names
[
 "iamuser.0"
 "iamuser.0"
 "iamuser.1"
 "iamuser.1"
]
```

Terraform Settings

There is a block you would have noticed in the use provider section of the website called ‘terraform’ – this is basically a settings block that configures Terraform (rather than the providers). For instance, here you can specify a minimum Terraform version that your configuration should use.

```
terraform {
  # ...
}
```

Required Version

The required_version setting accepts a version constraint string that specifies which versions of Terrafomr can be used with your configuration.

If the running version of Terrafrom doesn’t match the constraints specified, Terrafrom will produce an error and exit without any further action.

```
terraform {
  required_version = "> 0.12.0"
}
```

Required Provider

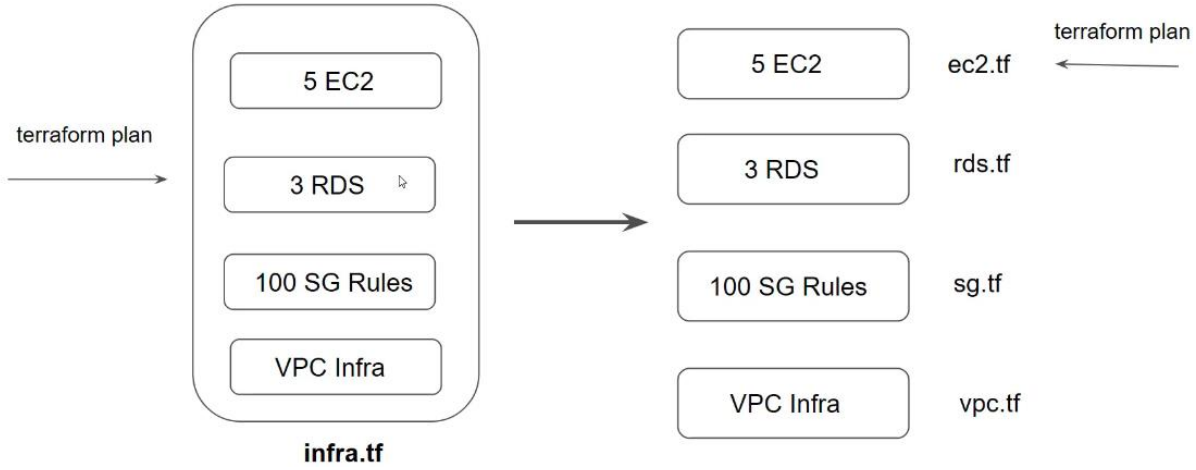
The required_providers block specifies all of the providers required by the current module, mapping each local provider name to a source address and a version container:

```
terraform {
  required_providers {
    mycloud = {
      source  = "mycorp/mycloud"
      version = "~> 1.0"
    }
  }
}
```

Dealing with Large Infrastructure

Having large infrastructure results in issues with the API limits for a provider.

The solution is to switch from a large manifest with all the resources and code there, into having separate manifests:



However, if a large manifest has been created, there are two solutions

- You can force stop refreshes whenever you perform terraform plan by stopping the querying the current state of the infrastructure during operations like terraform plan:

```
Maaz@MacBook-Air data-sources % terraform plan -refresh=false
data.aws_ami.app_ami: Reading...
data.aws_ami.app_ami: Read complete after 0s [id=ami-001c1821bbfc6dfd5]

Terraform used the selected providers to generate the following execution plan.
Resource actions are indicated with the following symbols:
  + create

Terraform will perform the following actions:

  # aws_instance.myec2 will be created
  + resource "aws_instance" "myec2" {
      + ami                                  = "ami-001c1821bbfc6dfd5"
```

- Notice the command that was used:

```
terraform plan -refresh=false
```

  - Now you will see there is no refresh of state for the resources:

Another solution is to operate on an isolated portion with -target-resource as your flag, for instance using the following command:

```
terraform plan -target=aws_instance.myec2
```

Note that this isn't recommended but is a solution to scenarios like this, as you will find from the output in the CLI:

```
Warning: Resource targeting is in effect

You are creating a plan with the -target option, which means that the result
of this plan may not represent all of the changes requested by the current
configuration.

The -target option is not for routine use, and is provided only for
exceptional situations such as recovering from errors or mistakes, or when
Terraform specifically suggests to use it as part of an error message.
```

Adding comments in Terraform

There are three different syntaxes that can be used to add comments when writing in the Terraform Language.

You will even see the comments getting either greyed out if using the Atom editor or becoming a dark green if using the Virtual Studio code as your editor. Here are the three types:

Type	Description
#	begins a single-line comment, ending at the end of the line.
//	also begins a single-line comment, as an alternative to #.
/* and */	are start and end delimiters for a comment that might span over multiple lines.

4. TERRAFORM PROVISIONERS

Introduction

Provisioners allow you to go beyond just creating and destroying resources. You can create a full end-to-end solution by use of them, such as installing software into an ec2 instance. So you can also configure applications into the instances/webservers:

They allow you to execute scripts on a local or remote machine as part of resource creation/destruction. For instance, when you create a web-server, you may want to install someone on it like so:

Here is the above example for demonstration:

```
provisioner "remote-exec" {
  inline = [
    "sudo amazon-linux-extras install -y nginx1.12",
    "sudo systemctl start nginx"
  ]

  connection {
    type        = "ssh"
    host        = self.public_ip
    user        = "ec2-user"
    private_key = "${file("./terraform.pem")}"
  }
 }
}
```

Types of Provisioners

Terraform has two main types of provisioners (although there are more as you will find on the terraform website):

1) local-exec
    - These allow you to run a command on the computer that you ran terraform apply from (locally) after it has been created:

```
resource "aws_instance" "web" {
  # ...

  provisioner "local-exec" {
    command = "echo ${aws_instance.web.private_ip} >> private_ips.txt"
  }
}
```

In this example the private ip of the ec2 instance will be stored into private_ips

- Other examples is when you want ansible playbook to run automatically on the resource

2) remote-exec
   - These allow you to run commands (invoke scripts) directly on the remote server <u>after</u> you created like the example shown in the introduction

```
resource "aws_instance" "web" {
  # ...

  provisioner "remote-exec" {
```

   - .....

Remote-exec Provisioner – Demo

Recall on this is directly above.

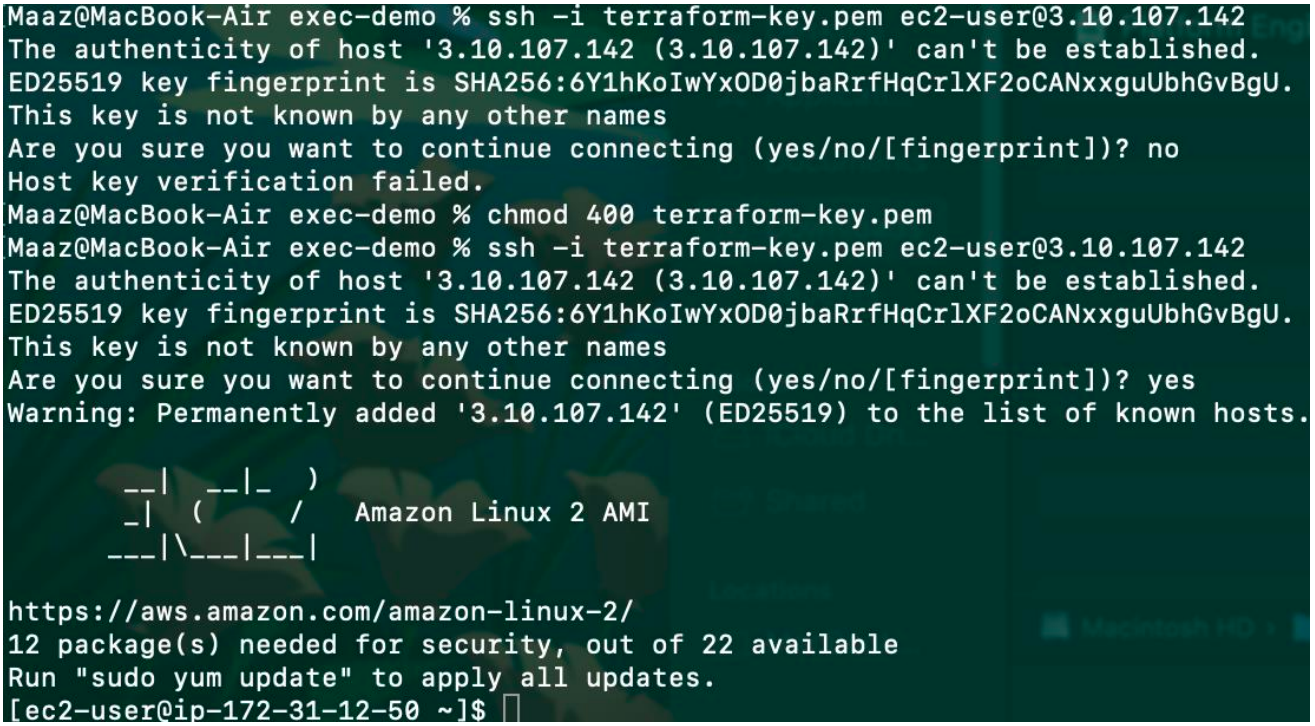
<u>Using the console</u>

Let's create an ec2 instance and then install Ngix onto it. In this process. If doing this in the console, we will have to:

- Create a key pair
  - We'll call it 'terraform-key' and use the .pem format
  - Notice that the key pair is downloaded; put this in a directory called exec-demo for later
- Create an ec2 instance called 'manual-ec2' under amazon linux 2 in your chosen region
  - You will only have to specify the key pair, so select the terraform-key which you created 

- Allow the ec2 to run on port 80 in the security group since Ngix is similar to apache
- Now that we have an ec2 instance, we want to install some software. So we go into the CLI, into the exec-demo folder which has the key stored in it and run the command:
  ssh -i terraform-key.pem ec2-user@3.10.107.142
  - Ssh is the protocol, i signifies install, terraform-key.pem is the keypair, ec2-user is the default username and the @3.10.107.142

- This command will likely give an error of authentication, so if on windows, go to properties of the terraform-key.pem file and go to security and click on advance and disable inheritance and then remove the other users. If on mad, run the command: chmod 400 terraform-key.pem



  - You can also directly connect through the browser by clicking connect but this is the way via the terminal
  - Now, switch to the roor user with
    sudo su –
  - Now run the command to install ngix:
    yum -y install ngix
  - As promped, run:
    sudo amazon-linux-extras install ngix1
  - Now you need to systemctl start command:
    systemctl start ngix

- Now you can copy the ip of the server and paste it into chrome and you have the ngix there ready to create a website from
- Do ctrl+d multiple times to un-connect from the server in your CLI

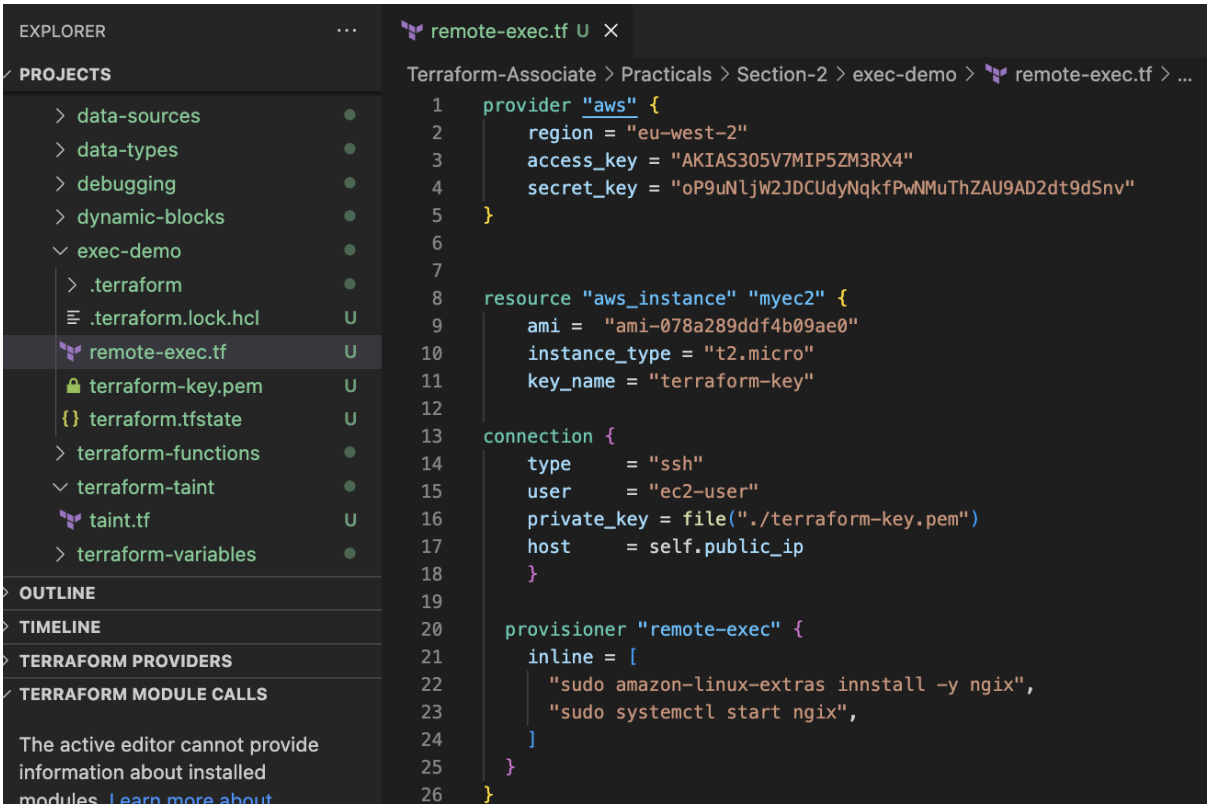
Using terraform

Lets now do the above with terraform code:

- First create a simple manifest like so (use the ami from the instance you created before above as best practice):

```
provider "aws" {
    region = "eu-west-2"
    access_key = "AKIAS3O5V7MIP5ZM3RX4"
    secret_key = "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"
}


resource "aws_instance" "myec2" {
    ami =  "ami-078a289ddf4b09ae0"
    instance_type = "t2.micro"


}
```

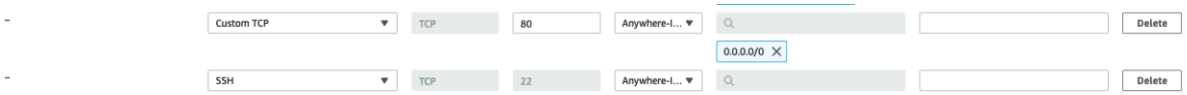
- Referring to https://www.terraform.io/language/resources/provisioners/remote-exec, copy what comes after the resource section and add it within your resource block and edit it like so:



- Breakdown:
    - We added the key_name in the resource block to specify the keypair used by the ec2 instance
    - We used type = ssh to ensure it is ssh protocol
    - We used user = “ec2_user” as this is the name of the user for an aws_instance resource

- We gave the private_key = file(PATH) since we wont be using a password for a connect but rather the keypair, this is found under connection providers in the registry
- Under the inline are the two codes we want to run, we can also add -y, because we don't want the confirm prompt to come since we will be stuck there as we want to automate this

- You also must ensure that the default security group in your console has an inbound rule of allowing port 80 since this is the security group that this ec2 will use since we did not specify one for it, as well as the SSH port since this is the protocol we are using:



- Now we can run terraform 'apply -auto-approve' (this means you don't have to say yes after)

Local-exec Provisioner – Demo

Recall:

local-exec provisioners allows us to invoke a local executable after the resource is created.

One of the most used approach of local-exec is to run ansible-playbooks on the created server after the resource is created.

```
provisioner "local-exec" {
  command = "echo ${aws_instance.web.private_ip} >> private_ips.txt"
}
```

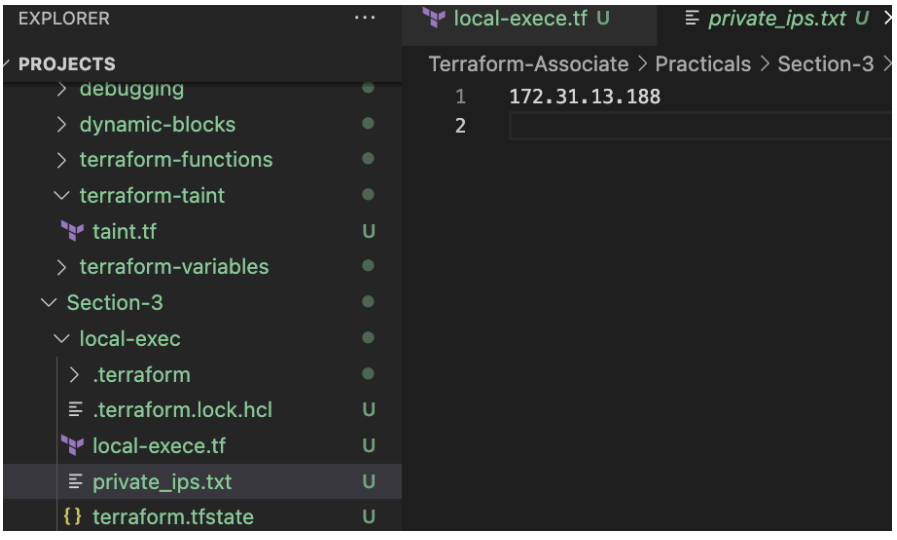
```
resource "aws_instance" "web" {
  # ...

  provisioner "local-exec" {
    command = "echo ${self.private_ip} >> private_ips.txt"
  }
}
```

This is the example usage taken from the terraform site. Now if you wanted to run the echo command which outputs the private IP of an ec2 that you created into a file called private_ips.txt within the working directory, you can use the following configuration (must run terraform apply to see the results):

```hcl
provider "aws" {
    region = "eu-west-2"
    access_key = "AKIAS3O5V7MIP5ZM3RX4"
    secret_key = "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"
}


resource "aws_instance" "myec2" {
    ami =  "ami-078a289ddf4b09ae0"
    instance_type = "t2.micro"
    key_name = "terraform-key"

    provisioner "local-exec" {
    command = "echo ${aws_instance.myec2.private_ip} >> private_ips.txt"
  }
}
```

Note, you don't need the connection block since this command is run on the cli that you are working on yourself (locally). As you can see after terraform apply:

```
aws_instance.myec2: Creating...
aws_instance.myec2: Still creating... [10s elapsed]
aws_instance.myec2: Still creating... [20s elapsed]
aws_instance.myec2: Provisioning with 'local-exec'...
aws_instance.myec2 (local-exec): Executing: ["/bin/sh" "-c" "echo 172.31.13.188 >> private_ips.txt"]
aws_instance.myec2: Creation complete after 22s [id=i-0b2557d92615c93aa]

Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
```



Creation-Time & Destroy-Time Provisioners

Types of Provisioners	Description
Creation-Time Provisioner	Creation-time provisioners are only run during creation, not during updating or any other lifecycle If a creation-time provisioner fails, the resource is marked as tainted.
Destroy-Time Provisioner	Destroy provisioners are run before the resource is destroyed.

Destroy-time Provisioner

https://www.terraform.io/language/resources/provisioners/syntax#destroy-time-provisioners

```
resource "aws_instance" "web" {
  # ...

  provisioner "local-exec" {
    when    = destroy
    command = "echo 'Destroy-time provisioner'"
  }
}
```

If when = destroy is specified, the provisioner will run when the resource within which it has been defined is destroyed:

```
resource "aws_instance" "web" {
  # ...

  provisioner "local-exec" {
    when    = destroy
    command = "echo 'Destroy-time provisioner'"
  }
}
```

- Note, this provisioner will only run at this point, and not before this point.
- You may want to use this if for example your ec2 has an antivirus agent that you are paying for but then you want to make sure to remove it from the instance before you destroy it.
- This way, the destroy doesn't occur until that provisioner is executed

Failure Behaviour for Provisioners

By default, provisioners that fail will also cause the Terraform apply itself to fail.

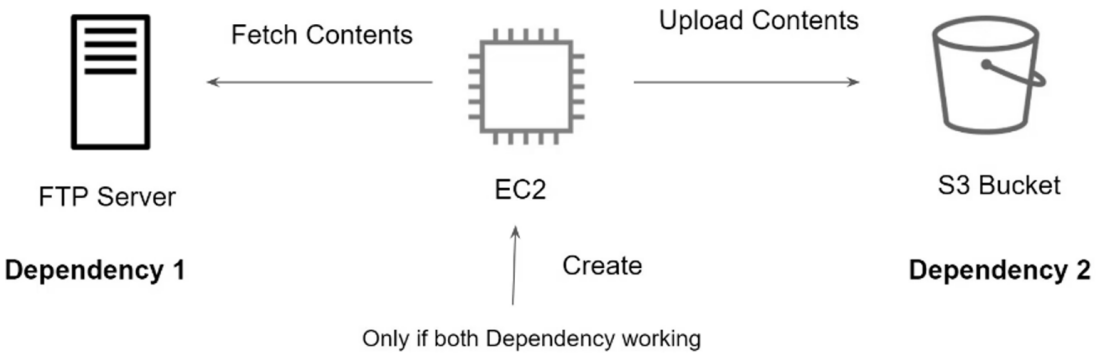
The on_failue setting can be used to change this. The allowed values are:

Allowed Values	Description
continue	Ignore the error and continue with creation or destruction.
fail	Raise an error and stop applying (the default behavior). If this is a creation provisioner, taint the resource.

```
resource "aws_instance" "web" {
  # ...

  provisioner "local-exec" {
    command    = "echo The server's IP address is ${self.private_ip}"
    on_failure = continue
  }
}
```

Null resource

The null_resource implements the standard lifecycle of a resource but will take no further action:



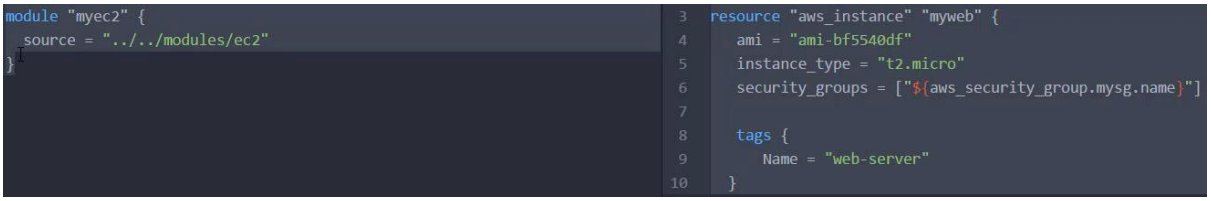
- So here the ec2 will only be created if both the dependencies are working
    - Since for example, there may be a firewall that is blocking the FTP Server
- This is quite rarely used

5. TERRAFORM MODULES & WORKSPACES

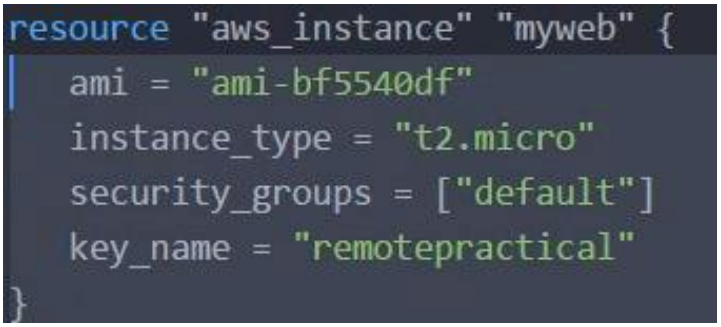
What is a module? Understanding DRY principles…

You can centralise the terraform resources and can then call them out from the terraform file which you wrote them in using modules; this makes repetitive resources easier to be called and makes the terraform file more tidy:

- ‘terraform modules’ similar to variables. But instead of referencing something that is a portion of a resource, you reference a whole resource that may be repeatedly used
- This is to adhere to DRY principles = ‘Don’t repeat yourself’
- Here is the difference between code when using it:

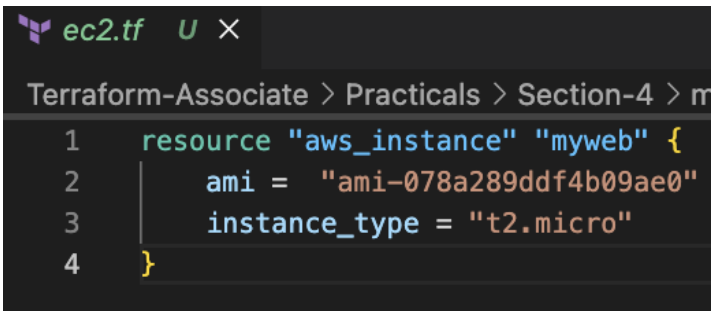


- This is since in a separate file called a module (in our case called mod_ec2) wherein there is a code for this resource block which can be reused:

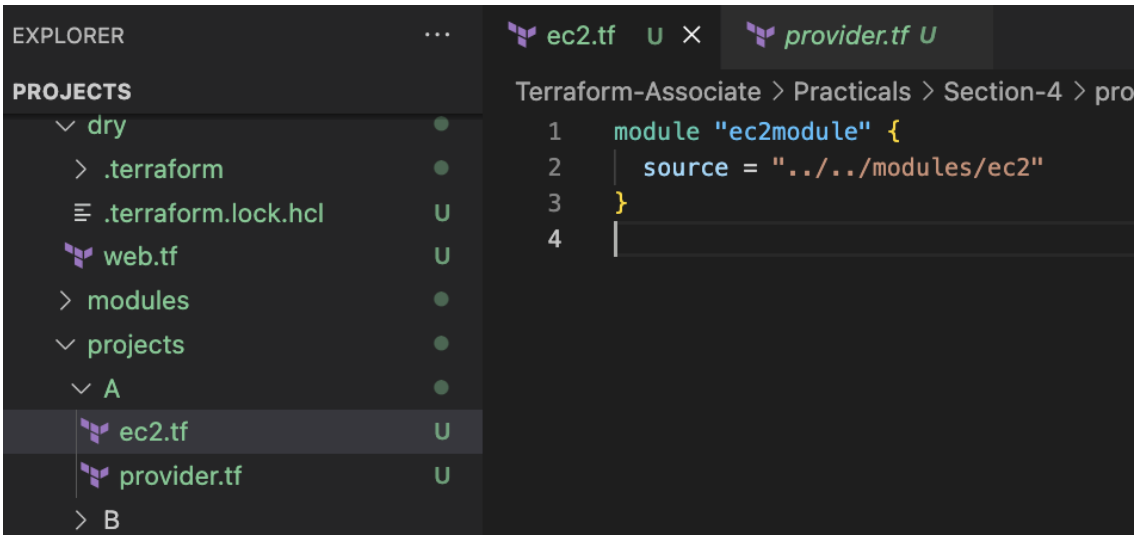


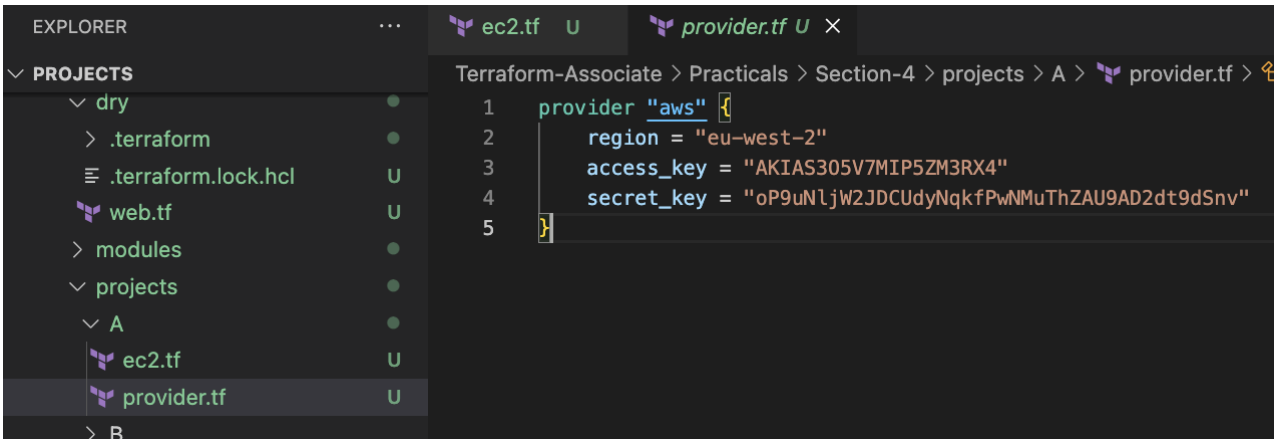
Implementing an EC2 Module

Create a new folder (I called mine section 4). Create two more folders within called modules and projects. In projects create a folder called A and B. In modules create a folder called ec2 and create the tf file within:



So now in project A and B, we'll reference this specific module. Create a new file called myec2.tf in project A with the code on the left. Also create another file called provider.tf with the provider config in project A as well as shown below the top one:





Following terraform plan, you will see that the resource will have been provisioned:

```
Plan: 1 to add, 0 to change, 0 to destroy.
```

Challenges with Modules

Using Variables to specify configurations

When you hardcode using a module, you can lose your ability to have the ability to have different variables like.

This is since when you are writing your module block that will connect to the resource you wrote, you can't override the code that you are referencing from within that module block. The left scrn shot will not work:

myec2.tf

```
module "ec2module" {
  source = "../../modules/ec2"
  instance_type = "t2.large"
}
```

ec2.tf

```
resource "aws_instance" "myec2" {
  ami = "ami-082b5a644766e0e6f"
  instance_type = "t2.micro"

}
```

Thus, you utilise variables in the source code, so that you can override the source code using an explicit value in your module. In the variables code you can give a default variable like t2.micro:

- Module folders' source ec2.tf:

ec2.tf .../A U | variables.tf U | ec2.tf .../ec2 U

Practicals > Section-4 > modules > ec2 > ec2.tf > resource "av

```
resource "aws_instance" "myweb" {
    ami =  "ami-078a289ddf4b09ae0"
    instance_type = var.instance_type
}
```

Module folders' variables.tf:

ec2.tf .../A U | variables.tf U | ec2.tf .../ec2 U

Terraform-Associate > Practicals > Section-4 > modules > ec2 >

```
variable "instance_type" {
    default = "t2.micro"
}
```

- Project A folders' myec2.tf:

myec2.tf U | variables.tf U | ec2.tf U

Terraform-Associate > Practicals > Section-4 > projects > A

```
module "ec2module" {
  source = "../../modules/ec2"
  instance_type = "t2.large"
}
```

Project A folders' provider.tf:

myec2.tf U | provider.tf U | variables.tf U | ec2.tf U

Terraform-Associate > Practicals > Section-4 > projects > A > provider.tf

```
provider "aws" {
    region = "eu-west-2"
    access_key = "AKIAS3O5V7MIP5ZM3RX4"
    secret_key = "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"
}
```

Using Locals to stop overriding variables in module block when referencing resource blocks

Say you have a variable for the ports in a security group which has a default of 8444, but you sometimes enter your variables.tf to change the port; then this is okay to do:

- In Modules folder:

```
variable "app_port" {
  default = "8444"
}
```

Security group (sg) resource in Modules folder

```
ingress {
  description      = "Allow Inbound from Secret Application"
  from_port        = var.app_port
  to_port          = var.app_port
  protocol         = "tcp"
  cidr_blocks      = ["0.0.0.0/0"]
}
```

But someone can come along when they are referencing the sg in a module block and do the following:

```
module "sgmodule" {
  source = "../../modules/sg"
  app_port = "22"
}
```

This overrides the port to 22 now, this is risky as it opens up attacks from the internet (SSH)

So the solution is that instead of variables, you can make use of locals to assign values.

You can then centralise them using variable.

```
resource "aws_security_group" "ec2-sg" {
  name        = "myec2-sg"

  ingress {
    description      = "Allow Inbound from Secret Application"
    from_port        = local.app_port
    to_port          = local.app_port
    protocol         = "tcp"
    cidr_blocks      = ["0.0.0.0/0"]
  }

locals {
  app_port = 8443
}
```

- Users can't override the locals at the bottom, as it lies in the source code itself, which they are referencing to. This allows you to still have the functionality of variables reducing the amount of code whilst not permitting anyone to override the source code.

How to reference Modul Outputs

Recall that output values make information about your infrastructure available on the command line. This can expose information for other Terraform configurations to use:

```
output "instance_ip_addr" {
  value = aws_instance.server.private_ip
}
```

In a parent module, outputs of child modules are available in expressions as module.<MODULE NAME>.<OUTPUT NAME>. Look here:

```
resource "aws_security_group" "ec2-sg" {
  name        = "myec2-sg"

  ingress {
    description      = "Allow Inbound from Secret Application"
    from_port        = 8433
    to_port          = 8433
    protocol         = "tcp"
    cidr_blocks      = ["0.0.0.0/0"]
  }
}

output "sg_id" {
  value = aws_security_group.ec2-sg.arn
}
```

Child module

```
module "sgmodule" {
  source = "../../modules/sg"
}

resource "aws_instance" "web" {
  ami           = "ami-0ca285d4c2cda3300"
  instance_type = "t3.micro"
  vpc_security_group_ids = [module.sgmodule.sg_id]
}
```

Parent module

- In the child module, we have an output of the arn (the id of the sg) attribute of the security group that has been created. In the parent module, we reference this output using the code:

module.sgmodule.sg_id

- module is always used
- sgmodule is the name of the module
- sg_id is the name of the output within that module (i.e. the sgmodule)

Terraform Registry Modules

The Terraform Registry is a repository of modules written by the Terraform community. The registry can help you get started with Terraform more quickly.

Within Terraform Registry, you can find verified modules that are maintained by various third-party vendors. These modules are available for various resources like AWS VPC, RDS, ELB and others.

Verified modules are reviewed by HashiCorp and actively maintained by contributors to stay up-to-date and compatible with both Terraform and their respective providers. The blue verification badge appears next to modules that are verified.

- Note: Module verification is currently a manual process restricted to a small group of trusted HashiCorp partners.

To use Terraform Registry module within the code, we can make use of the source argument that contains the module path.

Below, the code references to the EC2 Instance module within terraform registry.

```
module "ec2-instance" {
  source  = "terraform-aws-modules/ec2-instance/aws"
  version = "2.13.0"
  # insert the 10 required variables here
}
```

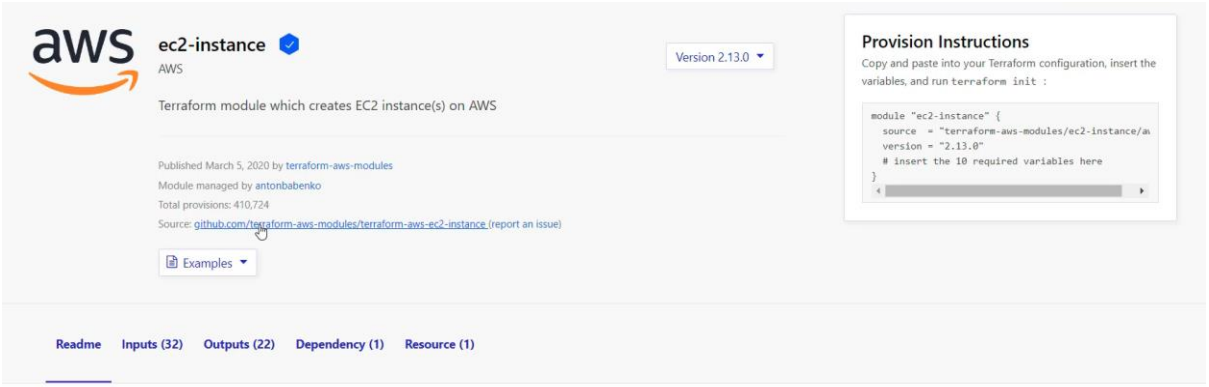
Say you were utilising the following module:

```
untitled                registry-module.tf
provider "aws" {
  region     = "us-west-2"
  access_key = "AKIAQIW66DN2W7WOYRGY"
  secret_key = "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"
}

module "ec2_cluster" {
  source                 = "terraform-aws-modules/ec2-instance/aws"
  version                = "~> 2.0"

  name                   = "my-cluster"
  instance_count         = 1

  ami                    = "ami-ebd02392"
  instance_type          = "t2.micro"
  subnet_id              = "ami-0d6621c01e8c2de2c"

  tags = {
    Terraform   = "true"
    Environment = "dev"
  }
```

Note, there is a slight typo in this code, the subnet)id was not fetched and the ami_id was placed into the subnet Id's place.

Now when you perform terraform init, you can go into the .terraform file that is created. This is where the child module is stored with the resource, it will be called modules which if you g into it you will find main.tf, outputs.tf and variables.tf. going into the main.tf, that is the module itself.

You can also read the source of code of the module by going into the following link in the module site:



AWS EC2 Instance Terraform module

Publishing Modules

Requirement	Description
GitHub	The module must be on GitHub and must be a public repo. This is only a requirement for the public registry.
Named	Module repositories must use this three-part name format terraform-<PROVIDER>-<NAME>
Repository description	The GitHub repository description is used to populate the short description of the module.
Standard module structure	The module must adhere to the standard module structure.
x.y.z tags for releases	The registry uses tags to identify module versions. Release tag names must be a semantic version, which can optionally be prefixed with a v. For example, v1.0.4 and 0.9.2

Terraform Workspaces

Overview

This is a little like having multiple desktops, they're completely different workspaces. With workspaces in terraform, the difference is with environment variables.

Terraform allows us to have multiple workspaces, with each of the workspace we can have different set of environment variables associated. Say you have the following workspaces, they may result in the following instances being created:

Staging → instance_type = t2.micro

Production → instance_type = t2.large

Workspace sub-commands

Running the following command in the CLI:

```
Maaz@MacBook-Air workspaces % terraform workspace -h
Usage: terraform [global options] workspace

  new, list, show, select and delete Terraform workspaces.

Subcommands:
    delete    Delete a workspace
    list      List Workspaces
    new       Create a new workspace
    select    Select a workspace
    show      Show the name of the current workspace
```

Showing a workspace then creating a new workspace:

```
Maaz@MacBook-Air workspaces % terraform workspace show
default
Maaz@MacBook-Air workspaces % terraform workspace new dev
Created and switched to workspace "dev"!

You're now on a new, empty workspace. Workspaces isolate their state,
so if you run "terraform plan" Terraform will not see any existing state
for this configuration.
```

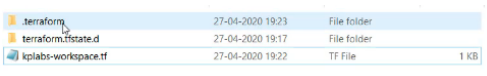
Also creating a prd workspace then list workspaces:

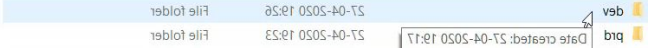
```
for this configuration.
Maaz@MacBook-Air workspaces % terraform workspace new prd
Created and switched to workspace "prd"!

You're now on a new, empty workspace. Workspaces isolate their state,
so if you run "terraform plan" Terraform will not see any existing state
for this configuration.
Maaz@MacBook-Air workspaces % terraform workspace list
  default
  dev
* prd
```

To switch between workspaces:

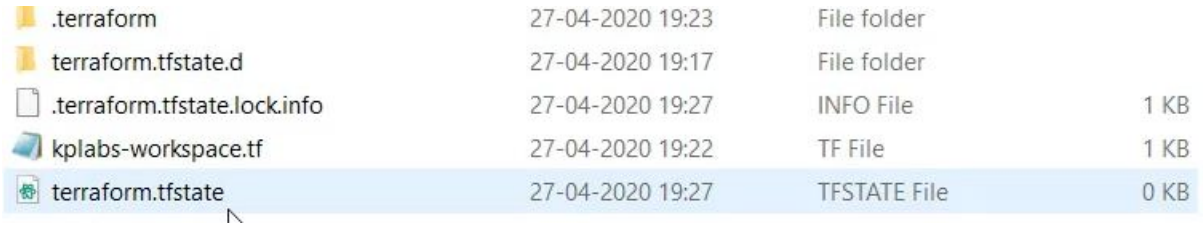
```
Maaz@MacBook-Air workspaces % terraform workspace select <workspacename>
```

Also terraform maintains the statefiles within workspaces separately:





Meanwhile, for the default workspace, the terraform state files maintains specifically the root directory:



- Note that this all occurs after terraform apply.

Implementing a workspace

Say you wanted 3 workspaces, one for default, one for developers, one for production, and for instances we wanted the following configuration for each:

```
default - t2.nano
dev - t2.micro
prd - t2.large
```

To do this we'll create a variable using the type = "map" argument, in the map place the requirements we want. Then under our aws_instance resource, we'll reference the instance_type to the map. Here we will use the lookup function so that the key is terraform.workspcae, which will specify which workspace you are in, recall that the lookup function needs a lookup(key,map) `lookup(map, key, default)` :

workspace.tf U

Terraform-Associate > Practicals > Section-4 > workspaces > workspace.tf > var

```
provider "aws" {
    region = "eu-west-2"
    access_key = "AKIAS3O5V7MIP5ZM3RX4"
    secret_key = "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"
}


resource "aws_instance" "myec2" {
    ami =  "ami-078a289ddf4b09ae0"
    instance_type = lookup(var.instance_type,terraform.workspace)
}

variable "instance_type" {
  type = map(string)

  default = {
    default = "t2.nano"
    dev     = "t2.micro"
    prd     =  "t2.large"
  }
}
```

While running terraform plan in the prd workspace, this is the result:

```
Maaz@MacBook-Air workspaces % terraform plan

Terraform used the selected providers to generate the following execution plan. Resource actions are indicated with
the following symbols:
  + create

Terraform will perform the following actions:

  # aws_instance.myec2 will be created
  + resource "aws_instance" "myec2" {
      + ami                                  = "ami-078a289ddf4b09ae0"
      + arn                                  = (known after apply)
      + associate_public_ip_address          = (known after apply)
      + availability_zone                    = (known after apply)
      + cpu_core_count                       = (known after apply)
      + cpu_threads_per_core                 = (known after apply)
      + disable_api_stop                     = (known after apply)
      + disable_api_termination              = (known after apply)
      + ebs_optimized                        = (known after apply)
      + get_password_data                    = false
      + host_id                              = (known after apply)
      + id                                   = (known after apply)
      + instance_initiated_shutdown_behavior = (known after apply)
      + instance_state                       = (known after apply)
      + instance_type                        = "t2.large"
```

6. REMOTE STATE MANAGEMENT

Recommendations and Security

- It is recommended to use git to store your code so that you can collaborate. You can use GitHub or any other repository storage (like bitbucket for example)
- However, you should be very careful not to commit and push any secret access keys or passwords into the repository
    - Some organisations make you refer to a file that stores your secret keys or access ketys etc,in a file that won't be pushed into git, but there is s separate issue being that the tfstate files contain the keys there
    - So DON'T commit the terraform.tfstate to the git repository

Module Sources in Terraform

The source argument in a module block tells Terraform where to find the source code for the desired child module.

- Local paths Terraform
- Registry GitHub
- Bitbucket
- Generic Git, Mercurial repositories
- HTTP URLs
- S3 buckets
- GCS buckets

A local patch must begin with either ./ or ../ to indicate that a local path is intended:

```
module "consul" {
  source = "../consul"
}
```

Arbitrary Git repositories can be used by prefixing the address with the special git:: prefix.

After this prefix, any valid Git URL can be specified to select one of the protocols supported by Git.

```
module "vpc" {
  source = "git::https://example.com/vpc.git"
}

module "storage" {
  source = "git::ssh://username@example.com/storage.git"
}
```

By default, Terraform will clone and use the default branch (referenced by HEAD) in the selected repository.

You can override this using the ref argument:

```
module "vpc" {
  source = "git::https://example.com/vpc.git?ref=v1.2.0"
}
```

The value of the ref argument can be any reference that would be accepted by the git checkout command, including branch and tag names.

So, to see the difference in an example, compare the top to the bottom scrn shot:

Terraform also directly supports GitHub (the above are for generic Git Repositories):

Terraform will recognize unprefixed `github.com` URLs and interpret them automatically as Git repository sources.

```
module "consul" {
  source = "github.com/hashicorp/example"
}
```

Terraform and .gitignore

Depending on the environments, it is recommended to avoid committing certain files to GIT:

Files to Ignore	Description
.terraform	This file will be recreated when terraform init is run.
terraform.tfvars	Likely to contain sensitive data like usernames/passwords and secrets.
terraform.tfstate	Should be stored in the remote side.
crash.log	If terraform crashes, the logs are stored to a file named crash.log

If you navigate to https://github.com/github/gitignore/ and find the terraform, you can get the recommended git.ignore files.

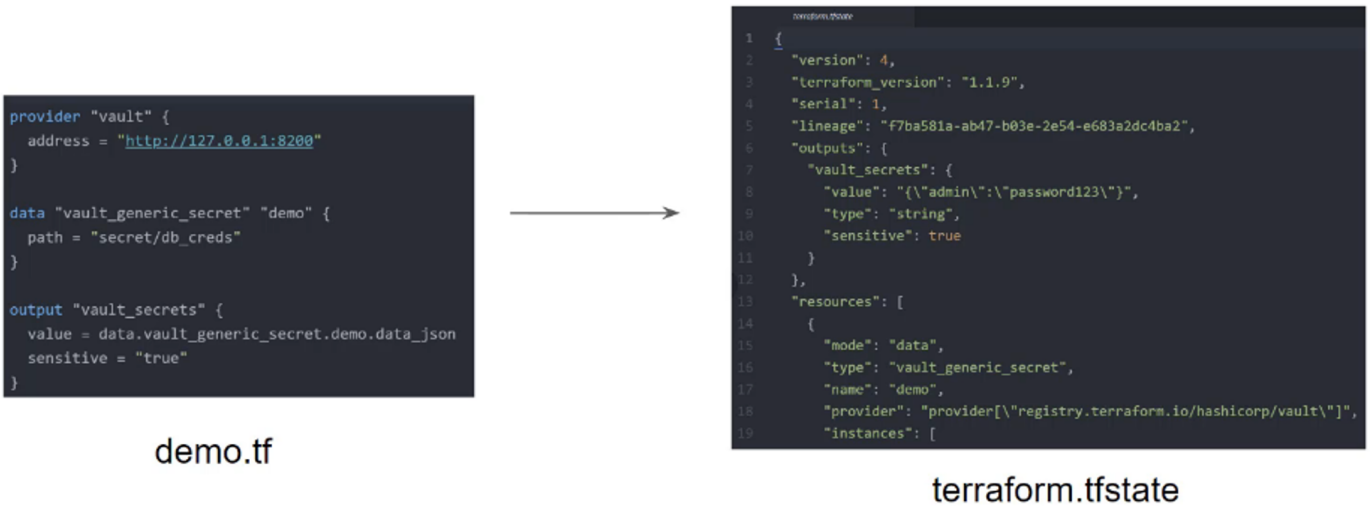
Terraform Backends

Terraform supports multiple backends that allows remote service-related operations. Some of the popular backends include:

- S3
- Consul
- Azurerm
- Kubernetes
- HTTP
- ETCD

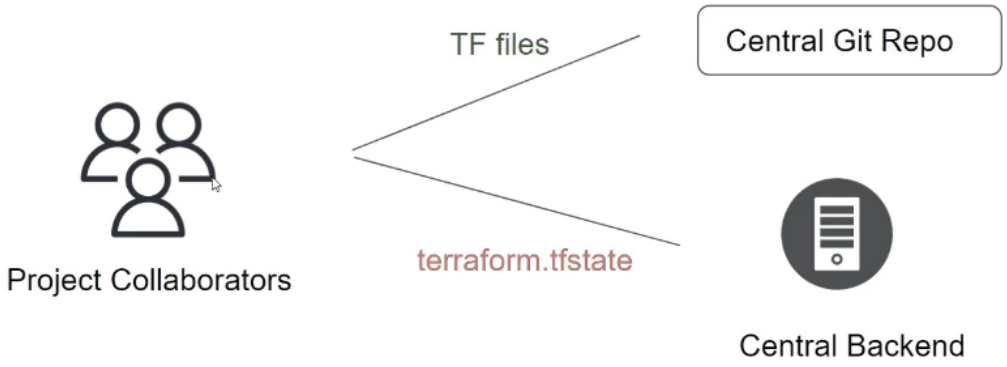
Backends primarily determine where Terraform stores its state.

By default, Terraform implicitly uses a backend called local to store state as a local file on disk



The image below describes one of the recommended architectures.

Here the Terraform Code is stored in Git Repository while the the State file is stored in a Central backend.



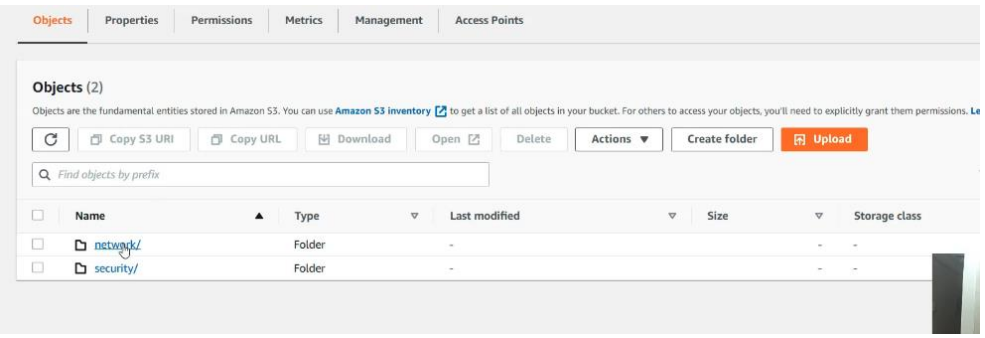
Accessing state in a remote service generally requires some kind of access credentials.

Some backends act like plain "remote disks" for state files; others support locking the state while operations are being performed, which helps prevent conflicts and inconsistencies:

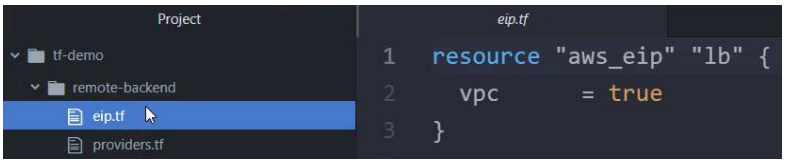
Implementing S3 Backends

https://www.terraform.io/language/settings/backends/s3

To create an s2 backend first create an S3 bucket with the following within it:

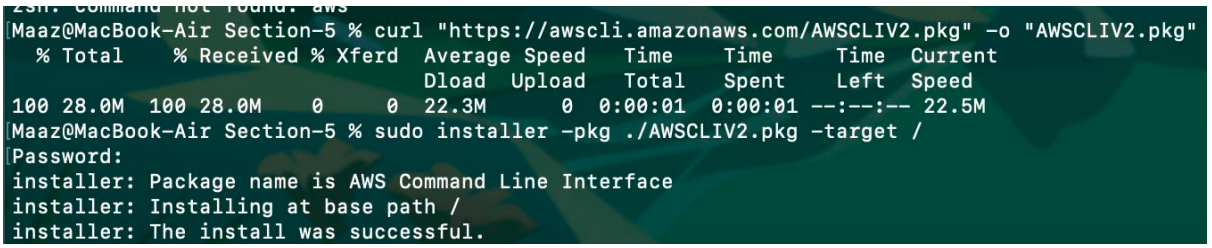


Now say you have the following resources:



Now create a file called backend.tf and copy into it the configuration from the terraform site and configure it as shown on the right:

We also must ensure the authentication credentials, you need to be able to access the s3 bucket. If you look into credentials and shared configuration below in the terraform site, specifically at secret_ky, you see that you can store this access by making it a part of AWS shared configuration file like ~/.aws/config. To do this, download the AWS CLI and then run the 'aws configure' command, on mac its like this (for windows refer to https://docs.amazonaws.cn/en_us/cli/latest/userguide/getting-started-install.html):



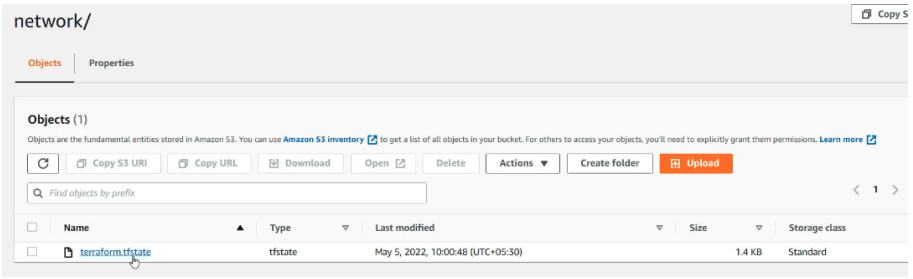
You then run your aws configure command and input your access and secret key:

```
AWS Access Key ID [****************SLMM]:
AWS Secret Access Key [****************sO8L]:
Default region name [us-east-1]:
Default output format [json]:
```

The config and credentials files would be created and so long as you have this you can now authenticate and access the s3 bucket:

```
ls s3://kplabs-terraform-backend
 PRE network/
 PRE security/
```

Now running terraform apply results in the terraform.tfstate file being stored in the network directory of your s3 bucket:



Ensuring State File Locking

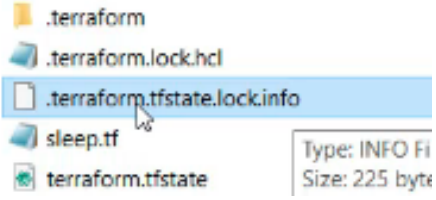
Whenever you are performing write operation, terraform would lock the state file.

This is very important as otherwise during your ongoing terraform apply operations, if others also try for the same, it can corrupt your state file.

```
Error: Error acquiring the state lock

Error message: Failed to read state file: The state file could not be read: read terraform.tfstate: The process
cannot access the file because another process has locked a portion of the file.

Terraform acquires a state lock to protect the state from being written
by multiple users at the same time. Please resolve the issue above and try
again. For most commands, you can disable locking with the "-lock=false"
flag, but this is not recommended.
```

- This error basically means that someone else is performing a terraform plan and that's why you can't do it

Terraform knows an operation is occurring through the following file:



This file is automatically removed once the operation is over.

State locking happens automatically on all operations that could write state. You won't see any message that it is happening.

If state locking fails, Terraform will not continue.

Not all backends support locking. The documentation for each backend includes details on whether it supports locking or not.

Terraform has a force-unlock command to manually unlock the state if unlocking failed.

If you unlock the state when someone else is holding the lock it could cause multiple writers. Force unlock should only be used to unlock your own lock in the situation where automatic unlocking failed.

State File Management – Modifying State Files

As your Terraform usage becomes more advanced, there are some cases where you may need to modify the Terraform state.

It is important to never modify the state file directly. Instead, make use of terraform state command.

There are multiple sub-commands that can be used with terraform state, these include:

State Sub Command	Description
list	List resources within terraform state file.
mv	Moves item with terraform state.
pull	Manually download and output the state from remote state.
push	Manually upload a local state file to remote state.
rm	Remove items from the Terraform state
show	Show the attributes of a single resource in the state.

The terraform state list command is used to list resources within a Terraform state, like so:

- Like so:

```
bash-4.2# terraform state list
aws_iam_user.lb
aws_instance.webapp
```

Terraform state mv

S

The terraform state my command is used to move items in a Terraform state.

This command is used in many cases in which you want to rename an existing resource without destroying and recreating it.

Due to the destructive nature of this command, this command will output a backup copy of the state prior to saving any changes.

Overall Syntax:

terraform state mv [options] SOURCE DESTINATION

Take this example: `bash-4.2# terraform state mv aws instance.webapp aws instance.myec2`

Terraform state pull

The terraform state pull command is used to manually download and output the state from remote state.

This is useful for reading values out of state (potentially pairing this command with something like jq).

Terraform state push

The terraform state push command is used to manually upload a local state file to remote state.

This command should rarely be used.

However, if it is used, it can result in quick fixes that are desired.

Terraform state remove

The terraform state rm command is used to remove items from the Terraform state.

Items removed from the Terraform state are not physically destroyed.

Moreover, removed from the Terraform state are only no longer managed by Terraform.

For example, if you remove an AWS instance from the state, the AWS instance will continue running, but terraform plan will no longer see that instance. Terraform state show:

Terraform state show

The terraform state show command is used to show the attributes of a single resource in the Terraform state.

Take the following example:

```
bash-4.2# terraform state show aws_instance.webapp
# aws_instance.webapp:
resource "aws_instance" "webapp" {
    ami                          = "ami-082b5a644766e0e6f"
    arn                          = "arn:aws:ec2:us-west-2:018721151861:instance/i-0107ea9ed06c467e0"
    associate_public_ip_address  = true
    availability_zone            = "us-west-2b"
    cpu_core_count               = 1
    cpu_threads_per_core         = 1
    disable_api_termination      = false
    ebs_optimized                = false
```

Terraform Import

It may occur that all the resources in an organisation are created manually, but you want to then import them Into Terraform (maybe after the company realises that that it's better to manage their infrastructure like this after they have grown).

To help with the migration of the resources and avoid the company having to stop and recreate their manually created resources from the console using Terraform, there is the option of Terraform import.

This command results in any existing infrastructure imported from the cloud into the Terraform state files.

HOWEVER, terraform will not create the configuration files of the infrastructure, these must be created manually so that they may be attached to the imported terraform code.