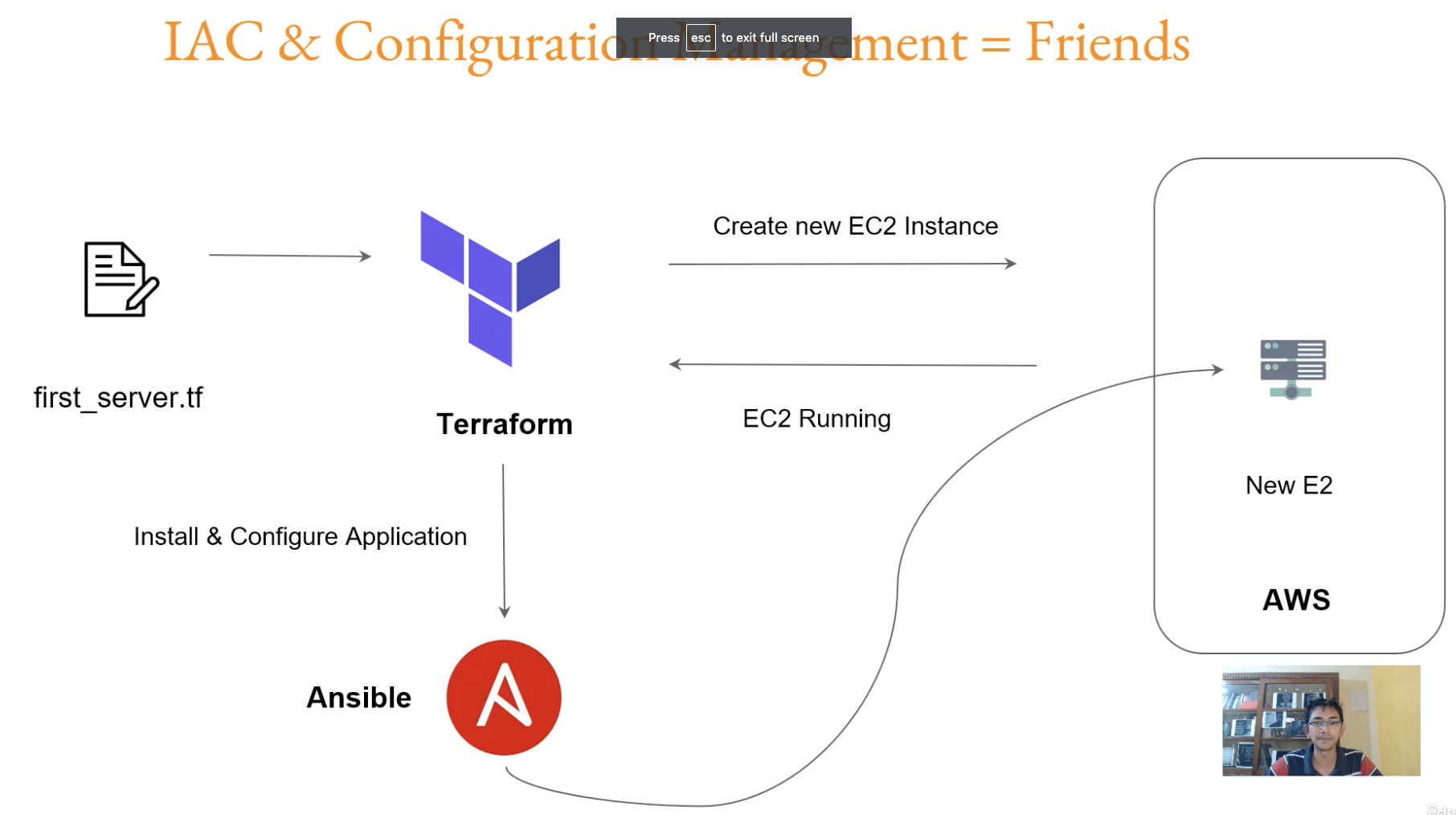
Configuration Management vs Infrastructure Orchestration:

-Ansible, Chef, Puppet – configuration management tools – primarily designed to **install and manage software** on existing servers

-Terraform, CloudFormation – infrastructure orchestration tools – can **provision the servers and infrastructure** by themselves

-configuration management tools can do some degree of infrastructure provisioning



Editor (preferred):

Visual Studio Code

Extension for VS Code:

Hashicorp Terraform

(makes life lot easier)

Launch first Virtual Machine through Terraform:

First authentication needs to be configured (In this example, it is access key and secret key)

provider "aws" {

access\_key = ""

secret\_key = ""

region = "us-west-1"

}

resource "aws\_instance" "myec2" {

ami = "ami-051ed863837a0b1b6"

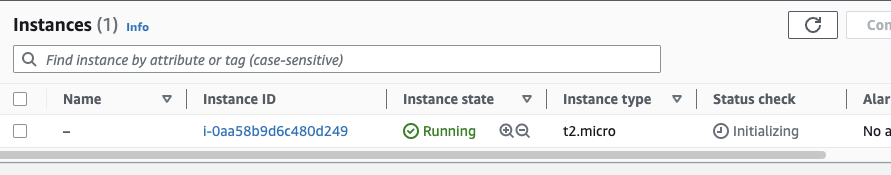
instance\_type = "t2.micro"

}

terraform init //will download the appropriate plugin associated with the provider (aws)

terraform plan //for our checking, will list the changes that will be made

terraform apply //will make the changes. Even here, changes that will be done will be showed



-----------------------

Say, we want to add a name to our EC2 instance created above

Update tags as below:

provider "aws" {

access\_key = ""

secret\_key = ""

region = "us-west-1"

}

resource "aws\_instance" "myec2" {

ami = "ami-051ed863837a0b1b6"

instance\_type = "t2.micro"

tags = {

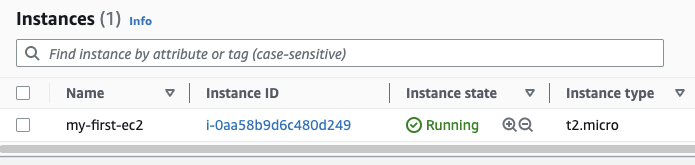
Name = "my-first-ec2"

}

}

Run   
terraform plan

terraform apply



--------------

Say, we want to remove the name

Remove the tags

provider "aws" {

access\_key = ""

secret\_key = ""

region = "us-west-1"

}

resource "aws\_instance" "myec2" {

ami = "ami-051ed863837a0b1b6"

instance\_type = "t2.micro"

}

Run

terraform plan  
terraform apply //this itself will be enough ie plan may not be needed

---------------

provider configuration at the top will remain to be the same, irrespective of the resources,…that you create

Tutor github which has examples:

<https://github.com/zealvora/terraform-beginner-to-advanced-resource/blob/master/Section%201%20-%20Deploying%20Infrastructure%20with%20Terraform/first-ec2.md>

Resource and Providers

Provider Plugins:

-**when** the **provider block is present, and** when **terraform init is run**, the corresponding **plugin is downloaded** and stored in .terraform folder

It is **not necessary that credentials need to be present** for downloading the plugin

E: For aws:

provider "aws" {

}

For Azure:

provider "azurerm" {

}

Above empty block is just for illustration

---------------

Resource:

Describes one or more infrastructure objects

Ex: taken from documentation

resource aws\_instance //to create EC2 instance

resource aws\_alb //to create aws ALB

resource iam\_user

resource digitalocean\_droplet

To create 2 EC2 instances:

resource "aws\_instance" "myec2" {

ami = "ami-051ed863837a0b1b6"

instance\_type = "t2.micro"

}

resource "aws\_instance" "web" {

ami = "ami-123"

instance\_type = "t3.micro"

}

----------------  
Do I have to learn Terraform for each provider?

No. Core concepts are same irrespective of provider

Once you learn for one provider, you can easily write for other providers

----------------  
Issues and Bugs with Providers:

There are some cases, when the output of Terraform will be different from that of Provider documentation

In such cases, you can raise the issue at Provider page

Google search term: say terraform azure issues

<https://github.com/hashicorp/terraform-provider-azurerm/issues>

Provider Tiers:

Provider Tiers:

1) Official: owned and maintained by Hashicorp

2) Partner: owned and maintained by Technology company that maintains direct partnership with Hashicorp

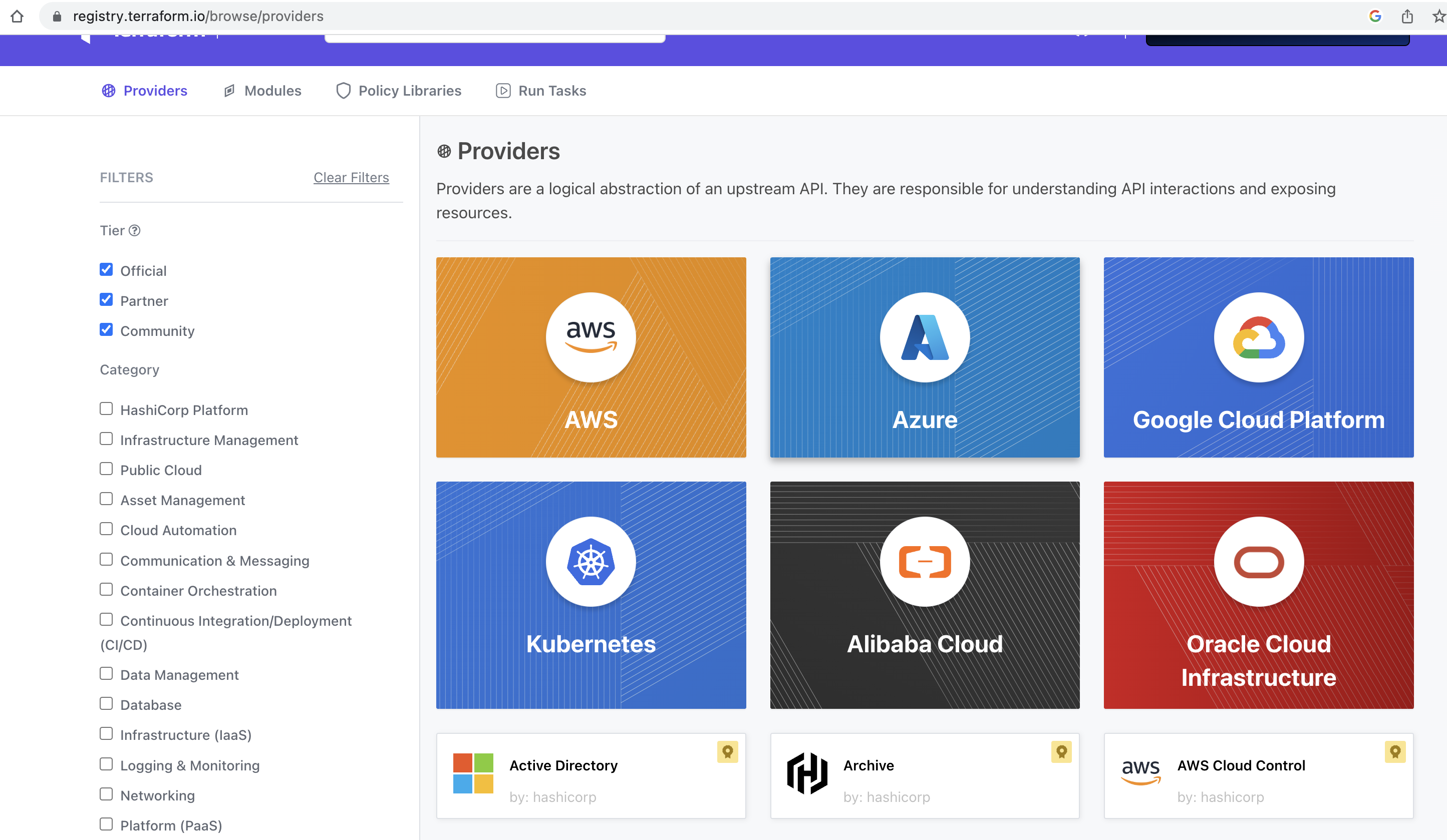
3) Community: owned and maintained by individual contributors

Your first preference would be to use Official ones

You can also use Partner

But, avoid Community

Read, Generate, Modify Configurations



Create Github repository through Terraform

Terraform Destroy

terraform destroy

When run inside a folder, will destroy all the resources present in .tf files

terraform destroy – target aws\_instance .myec2

Destroys specific resource

Also, don’t forget to remove that code from the file

You can also comment that code in the file. In that case also, the resource will be destroyed

(so, terraform destroy is not the only command to destroy?)

Understanding Terraform State files

-file ending with .tfstate

-where info like details of the resource(ec2, …) are stored

-don’t mess around with it manually

Understanding Desired and Current States:

Desired state: what is defined in the terraform file

Say to create an EC2 instance of t2.micro

Current state: Current state of the resource

May be the same as desired state or may not be

after terraform created a resource, it may have been modified directly in the console.

Ex: changing EC2 instance type from t2.micro to t2.medium

When terraform apply is run once again, terraform will apply the desired state

Ie will change t2.medium to t2.micro

Challenges with the current state on computed values:

resource "aws\_instance" "myec2" {

ami = "ami-051ed863837a0b1b6"

instance\_type = "t2.micro"

}

Say, we have above terraform configuration

Ie above is the desired state

With the above configuration, the instance created will have default security group

Now, in the console, security group has been changed to custom one

When the terraform is run again, it won’t change the security group to default

Why?

Security group is not mentioned in the terraform configuration

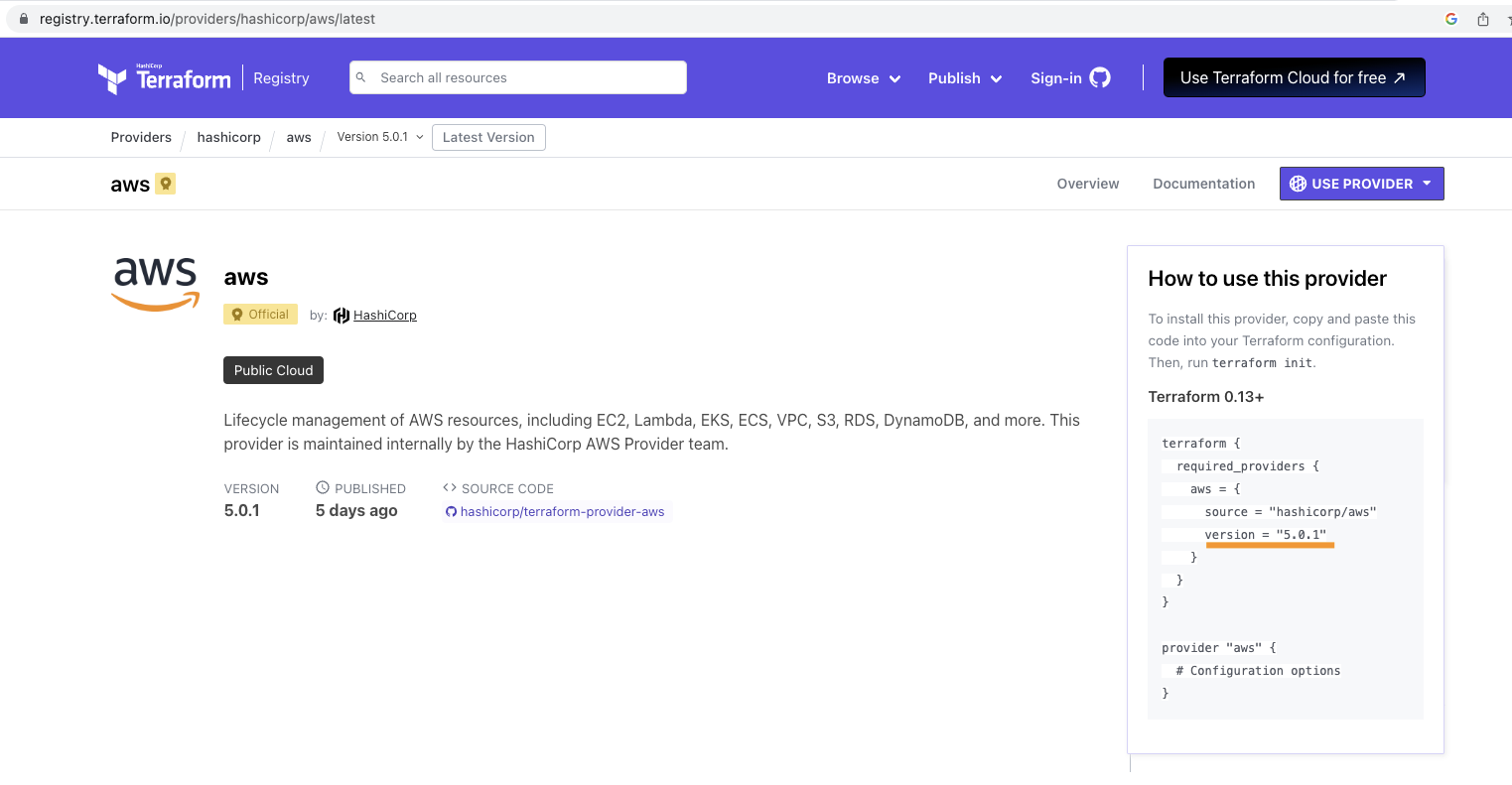
If you want security group to be default, then, mention it explicitly in terraform

This applies to other resources also

So, **when you want some configuration to be definitely made, mention it in terraform**

**(When you mention it in Terraform, even when the resource has been changed in the console, when terraform is run again, it gets current AWS through its refresh, and asks you, if you want to override the AWS actual, as there is a discrepancy between terraform configuration and actual in AWS)**

Terraform Provider Versioning:



Terraform version and provider plugin version are different

**Above suggested approach for version, using = is the correct way for Production**

Else, if you use things like ~ versioning or so, then your things may break due to compatibility issues (ie when the higher version of plugin is automatically downloaded, your config may break)

In newer Terraform versions, terraform uses a lock file to maintain version of the plugin used

If you need to modify the plugin version, you need to modify in your terraform config file, and also use -upgrade CLI option

(Else, you can delete the lock file and run terraform init)

Terraform Refresh:

(terraform apply **-auto-approve**

does not ask for yes/no)

terraform refresh

**-is not to be used manually**, as it can lead to dangerous situations like wiping the state data

-it is also **deprecated**

**-terraform plan and apply commands internally use refresh**

**-refresh-only option is there, but, even it also is not advisable to be used**

AWS Provider – Authentication Configuration

Giving access keys and secret in provider block is not for Prod (as it is not safe)

==========

*provider “aws” {*

*region = “us-east-1”*

*}*

*resource “aws\_eip” “lb” {*

*domain = “vpc”*

*}*

Provider block given above, means, TF will look for credentials in:

Mac/Linux:

$HOME/.aws/config and

$HOME/.aws/credentials

Windows:

%USERPROFILE%\.aws\config and

%USERPROFILE%\.aws\credentials

===========  
AWS CLI installation is recommended

AWS CLI allows customers to manage AWS resources directly from CLI

(when you install AWS CLI, it will prompt to enter the access and secret keys.

It will then store them in above location)

When you configure Access/Secret keys in AWS CLI, the location in which these credentials are stored is the same default location that TF searches the credentials from

Location is given above

========  
Credentials can be given as env variables also

Read, Generate, Modify Configurations

Understanding Attributes and Output values in Terraform  
To print the attribute of a created resource

public\_ip in below case

resource "**aws\_instance**" "**myec2**" {

ami = "ami-051ed863837a0b1b6"

instance\_type = "t2.micro"

}

output "ec2\_address" {

value = **aws\_instance.myec2**.public\_ip

}

Output:

ec2\_address = "54.183.10.124"  
  
If you want to print all the attributes of the resource, just have as

output "ec2\_address" {

value = **aws\_instance.myec2**

}

Ie don’t specify any attribute

----------

Another use case of this attribute, is to pass the attribute to input of other resource

Say, add the ip to the security group allowed list

Referencing Cross-Account Resource Attributes

To associate an Elastic IP to Sec Grp:

This will be the way for most of the other resources also

resource "aws\_security\_group" "allow\_tls" {

name = "allow\_tls"

description = "Allow TLS inbound traffic"

vpc\_id = "vpc-02fbfb14aae4c64c9"

ingress {

description = “TLS from VPC”

from\_port = 443

to\_port = 443

protocol = "tcp"

cidr\_blocks = [“${aws\_eip.example.public\_ip}/32”]

}

}

Where aws\_eip.example would have been created as:

resource "aws\_eip" "example" {

=========  
To associate an Elastic IP to an EC2 instance, resource aws\_eip\_association to be used

This seems like an exception case

resource "aws\_instance" "myec2" {

ami = "ami-051ed863837a0b1b6"

instance\_type = "t2.micro"

}

resource "aws\_eip" "example" {

domain = "vpc"

}

resource "aws\_eip\_association" "eip\_assoc" {

instance\_id = aws\_instance.myec2.id

allocation\_id = aws\_eip.example.id

}

Approaches for Variable assignment:

-4 ways are there

**-below is the common way for Prod**

ec2\_variable.tf:

provider "aws" {

access\_key = "" //purushc user

secret\_key = ""

region = "ap-south-1"

}

resource "aws\_instance" "myec2" {

subnet\_id = "subnet-0466965d86d3e56d4"

ami = "ami-057752b3f1d6c4d6c"

**instance\_type = var.instancetype //value got from variable**

}

variables.tf:

variable "instancetype" {

default = "t2.micro"

}

terraform.tfvars:

//if developer wants to override, he can do it here

//Else, default entry in variables.tf will be considered

instancetype = "t2.medium"

**file name of terraform.tfvars should be given. Else, override will not work**

**If you want to use a different file name, then, you should explicitly specify that file as below:**

**terraform apply -var-file=”custom.tfvars”**

Data types for Variables

-can assign data types for variables

-variables.tf:

variable “instance\_name” {

type = number

}

terraform.tfvars:

instance\_name = “john-123”

-during plan/apply, error will be thrown

-----  
**if no data type is defined, the variable can accept any data type**

-----

**Best practice is to define data types:**

Why?

Ex:

elb.tf:

resource "aws\_elb" "bar" {

name = "foobar-terraform-elb"

availability\_zones = ["us-west-2a", "us-west-2b", "us-west-2c"]

…..

Say, we change above to use variables:

resource "aws\_elb" "bar" {

name = var.elb\_name

availability\_zones = var.az

variables.tf:

variable “elb\_name” {}

variable “az” {}

terraform.tfvars:

in this file, to provide values, how will the user know that he needs to give a list of values for “az”

So, we provide data type for az

variables.tf:

variable “elb\_name” {

type = string

}

variable “az” {

type = list

}

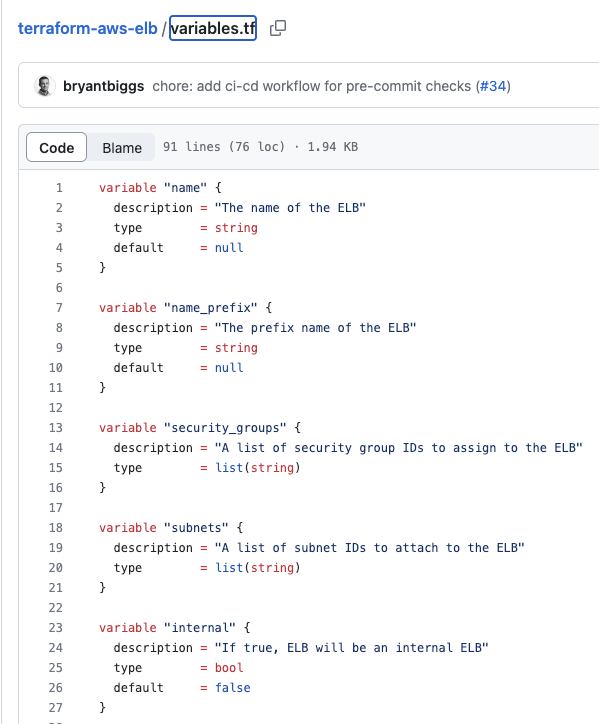
Now, the user knows and provides values:

terraform.tfvars:

elb\_name=”myelb”

az=[“us-west-1a”,”us-west-1b”]

---------------

Ex:  
  
  
Fetching Data from Maps and List in Variable  
Fetching from List:

fetch\_from\_list.tf

resource “aws\_instance” “myec2” {

ami = ”ami-…..”

instance\_type = var.regionList[2]

}

variable “regionList” {

**type = list**

**default = [“t2.micro”, “t2.nano”, “t2.small”]**

}

-----------

Fetching from Map:

fetch\_from\_map.tf

resource “aws\_instance” “myec2” {

ami = ”ami-…..”

instance\_type = var. regionMap[“ap-south-1”]

}

variable “regionMap” {

**type = map**

**default = {**

**us-east-1 = “t2.micro”**

**us-west-2 = “t2.nano”**

**ap-south-1 = “t2.small”**

**}**

}

Count and Count Index

**count – specifies the number of instances to be created**

say, we need to create 3 LB, or 10 EC2 instances, or…

instead of having 3 blocks of LB/10 blocks of EC2 instance, we can use **for loop** kind of thing

ex:

count.tf:

provider “aws” {

…

}

variable “elb\_names” {

type = list

default = [“dev-loadbalancer”, “stage-loadbalancer”, “prod-loadbalancer”]

}

resource “aws\_elb” “myelb” {

….

name = var.elb\_names[**count.index**]

**count = 3**

…

}

Conditional Expressions  
**condition ? true\_val : false\_val**

Ex:

conditional.tf:

provider “aws” {

…

}

variable “istest” {}

resource “aws\_instance” “dev”{

ami = “ami-…..”

instance\_type = “t2.micro”

count = var.istest = true ? 5 : 0

}

terraform.tfvars:

istest = true

-in above example, if variable istest is true, then 5 instances will be created, else, none

Local Values

-similar to variable initialization

Ex:

local.tf

…

**locals** {

common\_tags = {

owner = “DevOps team”

service = “backend”

}

}

resource “aws\_instance” “app-dev” {

ami = “ami-……”

instance\_type = “t2.micro”

tags = **local.**common\_tags

}

resource “aws\_instance” “db-dev” {

ami = “ami-……”

instance\_type = “t2.small”

tags = **local.**common\_tags

}

-in above, tags defined in one place, is used in many places

--------

**Expressions can also be used inside locals**

--------

**variable:**

**-meant to be overridden (say, in terraform.tfvars)**

**local:**

**-not meant to be overridden**

**Hence the name “local”, will be visible only within that module**

**------------**As per tutor:

-if overused, they can also make a configuration hard to read by future maintainers by hiding the actual values used

-use local values only in moderation, in situations where a single value or result is used in many places and that value is likely to be changed in future

Terraform functions

**-supports inbuilt functions, does not support user defined functions**

**Functions can be tried out in terraform console**

>terraform console

>max(10,20,30)

>30

------------  
>terraform console

>file(“${path.module}/id\_rsa.pub”)

><contents of the file will be returned as string>

path.module – refers to current directory

Data Sources

**EC2 ami (ids) differ by each region. using this approach, we don’t need to define AMIs for each region specifically**

Ie for amazon linux 2 –

Singapore region – will have an ami id

Oregon region – will have an ami id

---------

**data** "aws\_ami" "app\_ami" {

most\_recent = true

owners = ["amazon"] //amazon owned AMI; can be Microsoft owned, ubuntu owned

filter {

name = "name"

values = [“amzn2-ami-hvm\*”]

}

}

provider "aws" {

access\_key = "" //purushc user

secret\_key = ""

region = "ap-south-1"

}

resource “aws\_instance” “instance-1” {

**ami = data.aws\_ami.app\_ami.id //ami not hardcoded**

instance\_type = “t2.micro”

}

---------

owners = ["self"]

-organizations would have created their own AMI, say base would have been Amazon linux, plus security controls added on top of it

-then, they publish their custom AMI in their account

-self refers that kind of custom ami

Debugging in Terraform

-**to enable terraform logging** (not sysout, for which output to be used), set **TF\_LOG** environment variable to one of the below:

TRACE, DEBUG, INFO, WARN or ERROR

Ex:

For Linux

export TF\_LOG=TRACE

-when any name other than the above log level is set for TF\_LOG, TRACE is the default that will be used

Also, TRACE is the most verbose (ie most detailed)

-to output the logs to a file, set TF\_LOG\_PATH env variable

Ex:   
For Linux

export **TF\_LOG\_PATH**=/tmp/terraform-crash.log

Now, when you terraform plan or apply, logs will be sent to that file

Terraform Code formatting

**-terraform fmt**

Does code formatting for the terraform file and variable file present in the directory

Validating Terraform Configuration files

-ex:

resource "aws\_instance" "myec2" {

subnet\_id = "subnet-0466965d86d3e56d4"

ami = "ami-057752b3f1d6c4d6c"

instance\_type = var.instancetype

sky="blue"

}

> terraform validate

╷

│ Error: Unsupported argument

│

│ on ec2\_variable.tf line 11, in resource "aws\_instance" "myec2":

│ 11: sky="blue"

│

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

-**even when you run, terraform plan, same error message will be reported**

**So, I think, we don’t need to run terraform validate explicitly**

Load order and Semantics

-Terraform generally loads all the configuration files within the directory specified in alphabetical order

-the files loaded must end in either .tf or .tf.json to specify the format that is in use

**-directory structure for prod:**

test\_directory

-provider.tf (file where provider block will be placed)

-ec2.tf (example filename. To place all ec2 creation resources in one file)

-iam\_user.tf (example filename. To place all iam user creation resources in one file)

-variables.tf (where variables block will be placed)

-terraform.tfvars (if needed, to overwrite entry in variables.tf)

provider.tf

provider "aws" {

access\_key = ""

secret\_key = ""

region = "us-west-1"

}

ec2.tf

resource "aws\_instance" "myec2\_1" {

ami = "ami-051ed863837a0b1b6"

instance\_type = var.instancetype

}

resource "aws\_instance" "myec2\_2" {//names must be unique

ami = "ami-051ed863837a0b1b6"

instance\_type = var.instancetype

}

variables.tf

variable "instancetype" {

type = string

default = "t2.micro"

}

Dynamic Blocks

-A dynamic block acts much like a **“for”** expression, **but produces nested blocks**

-It iterates over a given complex value, and **generates a nested block for each element** of that complex value

Ex:

Before using dynamic block, we need to create separate blocks for each ingress

(also note, protocol and cidr blocks are same for each ingress. In reality, not sure, if this usecase will come

But, maybe, dynamic blocks can be used for other usecases also)

provider "aws" {...}

resource "aws\_security\_group" "dynamic\_sg" {

name = "dynamic\_block\_test"

description = "To test dynamic block"

vpc\_id = "vpc-02fbfb14aae4c64c9"

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"]

}

}

After using dynamic block:

provider "aws" {. . .}

variable "sg\_ports" {

type = list(number)

description = "list of ingress ports"

default = [ 8200, 8201, 8300 ]

}

resource "aws\_security\_group" "dynamic\_sg" {

name = "dynamic\_block\_test"

description = "To test dynamic block"

vpc\_id = "vpc-02fbfb14aae4c64c9"

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"]

}

}

}

Iterator syntax is optional

If iterator syntax is not used, then, resource name of dynamic block to be used

dynamic "ingress" {

for\_each = var.sg\_ports

content {

from\_port = ingress.value

to\_port = ingress.value

protocol = "tcp"

cidr\_blocks = ["0.0.0.0/0"]

}

}

For egress dynamic block

dynamic "egress" {

for\_each = var.sg\_ports

content {

from\_port = egress.value

to\_port = egress.value

protocol = "tcp"

cidr\_blocks = ["0.0.0.0/0"]

}

}

Tainting resources

-Forcing Re-creation of resources

<https://developer.hashicorp.com/terraform/cli/state/taint>

Splat Expressions

\* is splat expression

Below prints arn of all users

provider "aws" {. . .}

resource "aws\_iam\_user" "users" {

name = "iamuser.${count.index}"

count = 3

path = "/system/"

}

output "arns" {

value = aws\_iam\_user.users[\*].arn

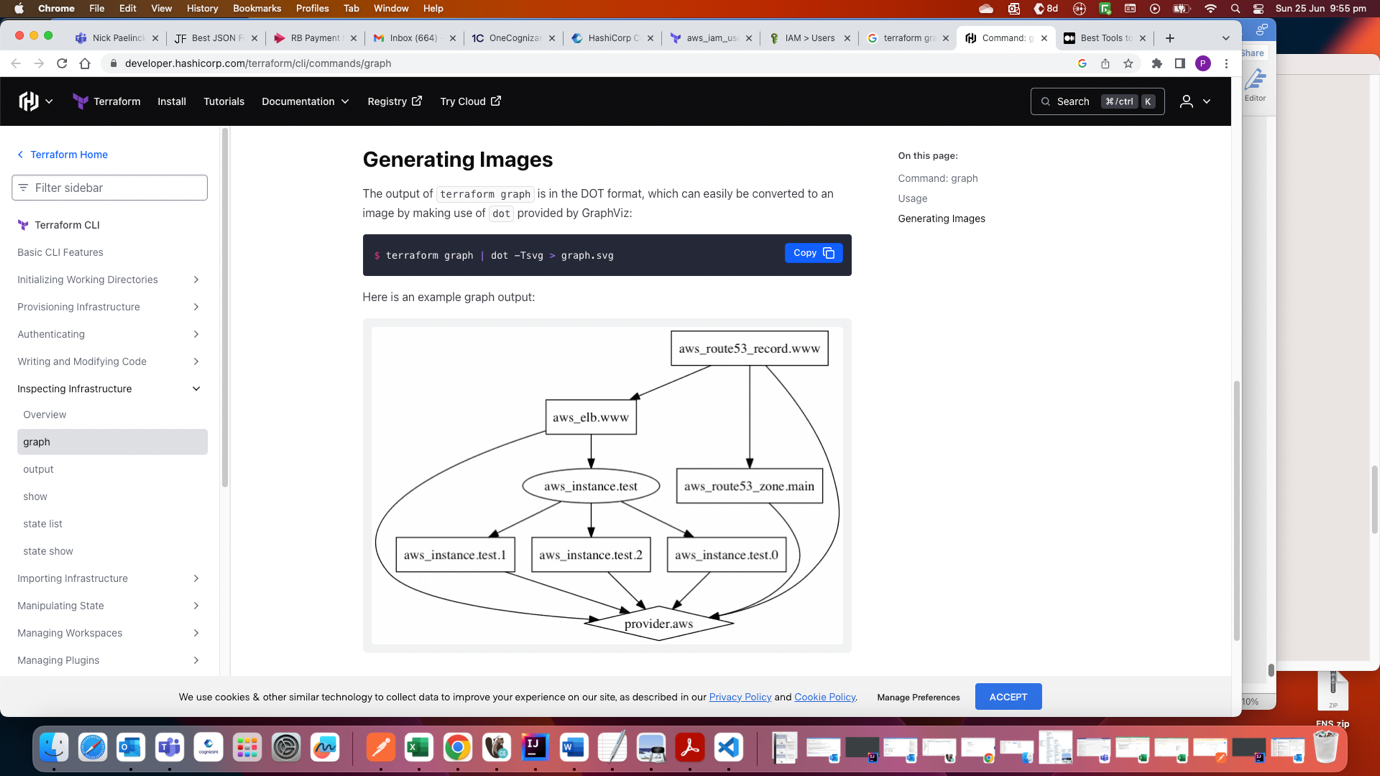
//value = aws\_iam\_user.users[0].arn

}

Terraform Graph

-terraform graph command used to generate a visual representation of either a configuration or execution plan

-output of terraform graph is in the DOT format, which can easily be converted to an image like below



Saving Terraform plan to file

In a team setup, terraform file can be modified by other developers

Say, you have made the below configuration ie t2.micro

provider "aws" {. . .}

resource "aws\_instance" "myec2" {

ami = "ami-051ed863837a0b1b6"

instance\_type = "t2.micro"

}

Other developer changes to t2.large

If you want to run exactly your configuration, then there are 2 ways:

1) you can take a copy of your files

2) you can save the terraform plan to a file, using the command

terraform plan -out=<filename>

ex: terraform plan -out=demopath

file created will be a binary file (not human readable)

you can then apply the plan using:

terraform apply <filename>

ex: terraform apply demopath

-this approach may not be widely used

Terraform output

Ways to see the output of your resources:

1)

provider "aws" {. . .}

resource "aws\_iam\_user" "users" {

name = "iamuser.${count.index}"

count = 3

path = "/system/"

}

output "arns" {

value = aws\_iam\_user.users[\*].arn

}

Say, you have run terraform apply and created the above resource

**When you run terraform apply again, it will say 0 added, 0 changed, 0 destroyed, but, output will still be printed**

*Apply complete! Resources: 0 added, 0 changed, 0 destroyed.*

*Outputs:*

*arns = [*

*"arn:aws:iam::758085500738:user/system/iamuser.0",*

*"arn:aws:iam::758085500738:user/system/iamuser.1",*

*"arn:aws:iam::758085500738:user/system/iamuser.2",*

*]*

2) **you can also the output in terraform state (.tfstate) file – inside outputs tag**

{

"version": 4,

"terraform\_version": "1.4.6",

"serial": 14,

"lineage": "6096b4f9-47d9-17ad-1f85-76a530d31b6c",

"outputs": {

"arns": {

"value": [

"arn:aws:iam::758085500738:user/system/iamuser.0",

"arn:aws:iam::758085500738:user/system/iamuser.1",

"arn:aws:iam::758085500738:user/system/iamuser.2"

],

3) below command can also be used:

**terraform output arns**

Terraform Settings

**terraform settings can be applied inside terraform block**

Few settings:

1) Terraform Version

terraform {

required\_version = “> 0.12.0”

}

If the running version of TF doesn’t match the constraints specified, TF will produce an error and exit

2) Provider Version

(ie, aws/azure/… plugin version)

terraform {

required\_providers {

aws = “~> 2.0”

}

}

Ex in a file:

terraform {

required\_providers {

aws = “~> 2.0”

}

}

provider "aws" {. . .}

resource "aws\_instance" "myec2" {

ami = "ami-051ed863837a0b1b6"

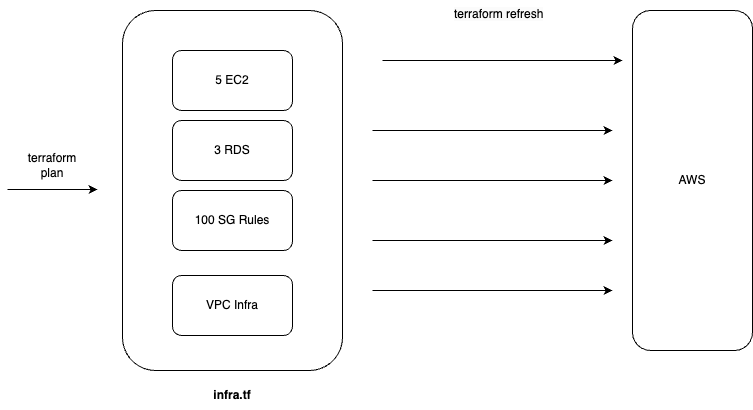
instance\_type = "t2.micro"

}

**Dealing with large infrastructure**

When you have a larger infrastructure, you will face issues related to API limits for a provider

ex:



Say, you have 5 EC2, 3 RDS, 100 SG rules, VPC infra in a single file, infra.tf

(I think, this applies to below case also:  
where even though resource per file (ie EC2 in file1, Sec Grp in file2,…) is used, but, all files are kept in same folder)

When you run terraform plan, the first thing that would happen is terraform refresh, where depending on the amount of resources, amount of calls that would be made to Cloud provider will also increase

(terraform refresh: to check if TF config and AWS actual are same/different)

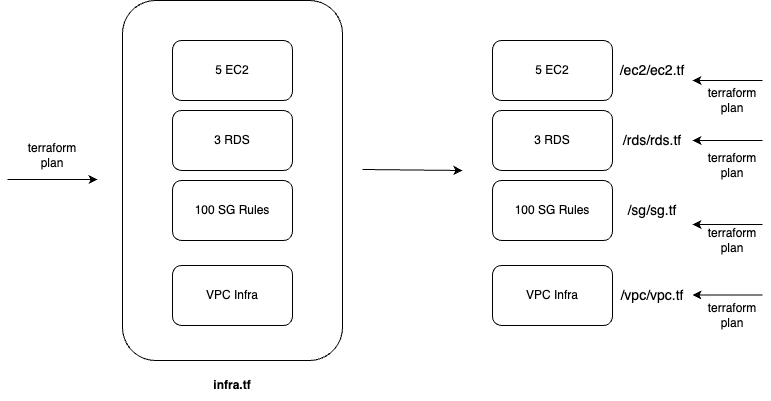
There are 2 disadvantages here:

1) Overall API calls can increase

2) Can slow down the operations

What can be done:

**Have files split based on resources and placed in separate folder**



Ex:

EC2 things in a separate folder, say, ec2/ec2.tf

RDS things in a separate folder, say, rds/rds.tf

SG rules things in a separate folder, sg.tf

VPC infra things in a separate folder, say, vpc/vpc.tf

Now, when you run terraform apply inside a folder, say, ec2

Terraform refresh will not be applied to other resources (rds, sg, vpc infra)

This saves a good amount of time, and also reduces the calls made to AWS

===============

In a project where, they already have put all the resources in single file

Ie 5 EC2, 3 RDS, 100 SG rules, VPC infra in a single file, infra.tf

What you can do:

1) (While doing terraform plan/apply) we can instruct TF to not do refresh at all for any of the resources using **-refresh=false**

2) **-target=resource** can be used to target a specific resource

target: generally used as a means to operate on isolated portions of very large configuration

ex: terraform plan -target=aws\_security\_group.resource1

Shows a warning when target is used

I think, 2 is better, not sure

**Both the above options are to be used only when definitely needed, and not to be used regularly**

**I think, we need to think before using target**

==========  
+ means resource to be added

- to be deleted

**~ means update in-place (**means to be modified)

========

Zipmap Function

**Constructs a map from a list of keys and a corresponding list of values**

provider "aws" {

access\_key = "" //purushc user

secret\_key = ""

region = "ap-south-1"

}

resource "aws\_iam\_user" "lb" {

name = "iamuser.${count.index}"

count = 3

path = "/system/"

}

output "name" {

value = aws\_iam\_user.lb[\*].name

}

output "arns" {

value = aws\_iam\_user.lb[\*].arn

}

output "combined" {

value = zipmap(aws\_iam\_user.lb[\*].name, aws\_iam\_user.lb[\*].arn)

}

Outputs:

arns = [

"arn:aws:iam::758085500738:user/system/iamuser.0",

"arn:aws:iam::758085500738:user/system/iamuser.1",

"arn:aws:iam::758085500738:user/system/iamuser.2",

]

**combined = {**

**"iamuser.0" = "arn:aws:iam::758085500738:user/system/iamuser.0"**

**"iamuser.1" = "arn:aws:iam::758085500738:user/system/iamuser.1"**

**"iamuser.2" = "arn:aws:iam::758085500738:user/system/iamuser.2"**

**}**

name = [

"iamuser.0",

"iamuser.1",

"iamuser.2",

]

Comments in Terraform

Single line comment:

#

//

Multi line comment:

/\*

\*/

**Challenges with Count Meta-Argument**

**count: not to be used for creating unique items. Instead for\_each to be used**

When count is used for creating unique items,

Say, below has been created

provider "aws" {…}

variable "iam\_names" {

type = list

default = ["user-01", "user-02", "user-03"]

}

resource "aws\_iam\_user" "iam" {

name = var.iam\_names[count.index]

count = 3

path = "/system/"

}

Then, you update user-0 by changing the index (count will be fine, if you add items at the end)

Ie index 0 was user-01. Now, you put user-0 in that

variable "iam\_names" {

type = list

default = ["user-0", "user-01", "user-02", "user-03"]

}

You will get below error:

Error: updating IAM User (user-03): EntityAlreadyExists: User with name user-02 already exists.

│ status code: 409, request id: 4a6bae1d-cdb6-4e46-89f1-d615526dac7c

│

│ with aws\_iam\_user.iam[2],

│ on count\_issue.tf line 12, in resource "aws\_iam\_user" "iam":

│ 12: resource "aws\_iam\_user" "iam" {

**So, for these cases, for\_each is recommended**

--------------

**count to be used when you don’t need unique values**

ex: below creates 3 similar EC2 instances

resource "aws\_instance" "server" {

count = 3

ami = "ami-a1b2wewew"

instance\_type = "t2.micro"

}

**In above, if 4 instances types need to be different, say t2.micro, t2.large,…then count shouldn’t be used, for\_each to be used**

Data type – SET

List allow duplicate values

Similar to List, but,

Set doesn’t allow duplicate values

No zero based index maintained

---------  
toset function converts list to set

Ex: toset([“a”, “b”, “c”, “a”])

Output: [“a”, “b”, “c”]

**for\_each in terraform**

Issue with count discussed in section “Challenges with Count Meta-Argument” is addressed here

Ran TF apply

provider "aws" {…}

resource "aws\_iam\_user" "iam" {

for\_each = toset(["user-01", "user-02", "user-03"])

name = each.key

}

Then, add user-0 at the start

provider "aws" {…}

resource "aws\_iam\_user" "iam" {

for\_each = toset(["user-0", "user-01", "user-02", "user-03"])

name = each.key

}

Things will still work fine

Reason:

for\_each uses set

set doesn’t use 0 based index, unlike list

In TF state file, you can see index\_key as below, whereas for count/list, it would have been 0,1,…

"instances": [

{

"index\_key": "user-0",

. . .

"attributes": {

. . .

"id": "user-0",

"name": "user-0",

----------  
for-each has each.key and each.value

|  |  |
| --- | --- |
| 4  5  6  7  8  9  10  11  12  13  14  15 | locals {  virtual\_machines = {     "vm1" = { vm\_size = "e2-small", zone = "us-central1-a" },     "vm2" = { vm\_size = "e2-medium", zone = "us-central1-b" },     "vm3" = { vm\_size = "f1-micro", zone = "us-central1-c" }  }  }    resource "google\_compute\_instance" "vm" {  for\_each = local.virtual\_machines  name = **each.key**  machine\_type = **each.value**.vm\_size  zone = **each.value**.zone  (...)  } |

-------

Terraform Provisioners

This topic is not for 003 exam

You have created EC2 instance using Terraform

After that, if you want to install Softwares on the EC2, Provisioners can be used

Ex: installing Nginx on the created EC2 instance

-------------

Few types:

Mentioning 2 here:

1) Local exec provisioners:

When you want to execute something in your local machine

Ex: when the private Ips of the created EC2 instances, need to be written to a local file

2) Remote exec provisioners

Ex: when you want to install nginx in EC2 and start the nginx server

Terraform Modules and Workspaces

Tutor example code is fine, but, tutor’s input on where the module needs to be used is incorrect. He is suggesting to use module even for EC2 instance creation, which is incorrect

Ex usecase:

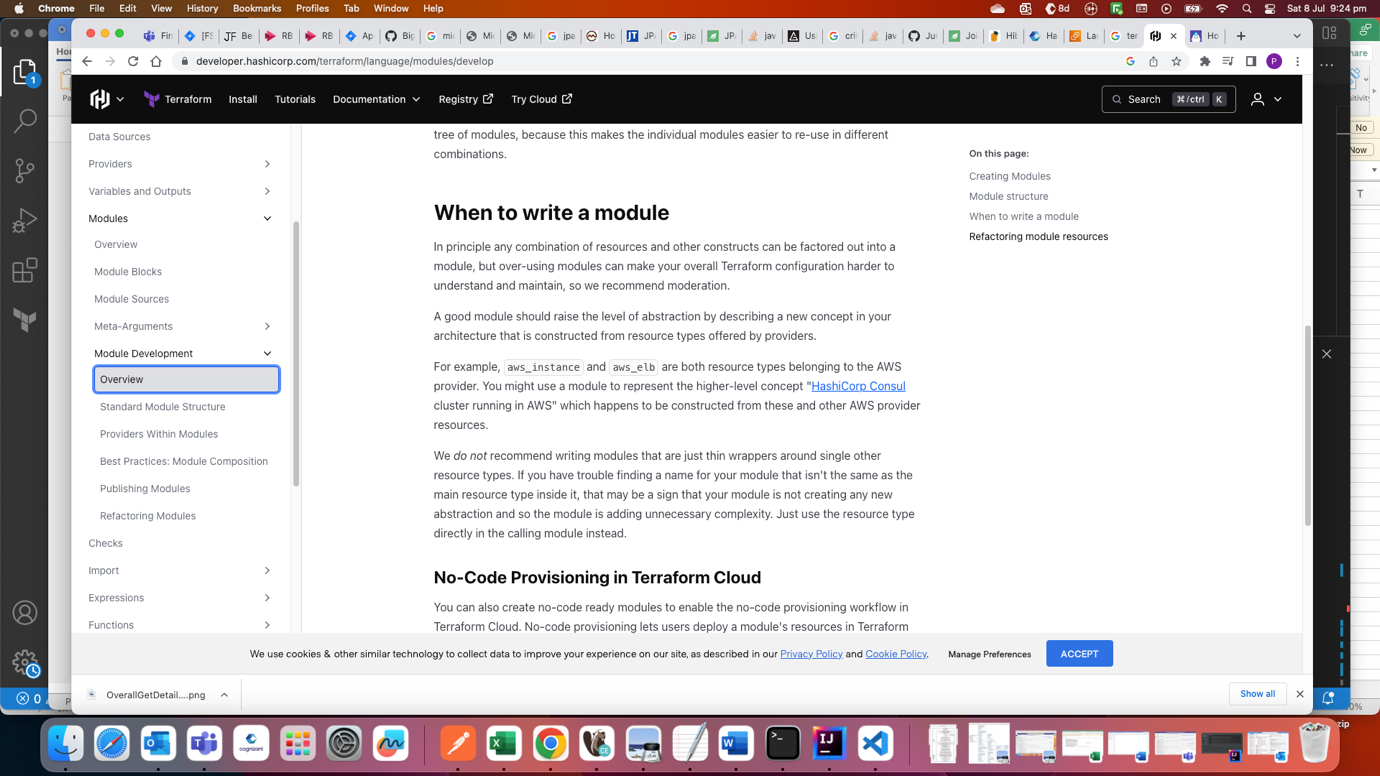
-you have staging and prod env, which have the same set of resources, but, number may be different. Say, staging has less number of EC2 instances

Your whole infra can be made as a module

But, the number of EC2 inst can be made as variables in the module

These variables can be overridden from staging project, and also from prod project

https://developer.hashicorp.com/terraform/language/modules/develop

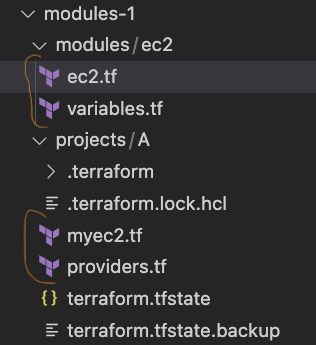


Below can be used as high level reference, which explains concepts like variables, locals, outputs…wrt modules:

<https://blog.gruntwork.io/how-to-create-reusable-infrastructure-with-terraform-modules-25526d65f73d>

Implementing EC2 module with Terraform

Variables and Terraform Modules



In the above,

modules-1/**modules**/ec2/ec2.tf

resource "aws\_instance" "myec2" {

subnet\_id = "subnet-0466965d86d3e56d4"

ami = "ami-0d13e3e640877b0b9"

instance\_type = var.instance\_type

}

-instance\_type can be overridden by the consuming project

modules-1/**modules**/ec2/variables.tf:

variable "instance\_type" {

default = "t2.micro"

}

-in case, instance\_type is not overridden by consuming client, above default value will be used

modules-1/**projects**/A/myec2.tf

module "ec2module" {

source = "../../modules/ec2"

instance\_type = "t2.large"

}

-note that the module is being given a name in the consuming side. Henceforth, in myec2.tf, the module should be referred only using this name (ec2module)

-instance\_type is being overridden

-when the module block is executed, code referred by that module is executed

Ie in this example, an EC2 instance is created

-consuming project is called as root module

-terraform init, plan, apply should be run inside A ie root module

modules-1/**projects**/A/providers.tf

provider "aws" {

access\_key = "" //purushc user

secret\_key = ""

region = "ap-south-1"

}

-provider details should always be declared at project level (not at module level)

Using Locals with Modules

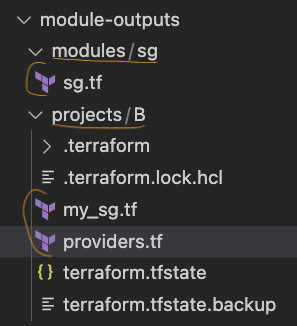
-locals can be used inside Modules as well

-example can be seen in next section

Referencing Module Outputs

-output of child module can be used as input to root module (consuming side)

-in module case, output will not print anything in console (in normal case, output will print to console)



module-outputs/modules/sg/sg.tf

**locals** {//locals being used

port = 443

}

resource "aws\_security\_group" "ec2-sg" {

name = "myec2-sg"

vpc\_id = "vpc-02fbfb14aae4c64c9"

ingress {

description = "Inbound"

from\_port = local.port

to\_port = local.port

protocol = "tcp"

cidr\_blocks = ["0.0.0.0/0"]

}

}

**output** "sg\_id" {//passed as input to root module

value = aws\_security\_group.ec2-sg.id

}

module-outputs/projects/B/my-sg.tf

module "sgmodule" {

source = "../../modules/sg"

}

resource "aws\_instance" "myec2" {

subnet\_id = "subnet-0466965d86d3e56d4"

ami = "ami-057752b3f1d6c4d6c"

instance\_type = "t2.micro"

vpc\_security\_group\_ids = [module.sgmodule.sg\_id]//uses output from child module

}

module-outputs/projects/B/providers.tf

provider "aws" {

access\_key = "" //purushc user

secret\_key = ""

region = "ap-south-1"

}

Terraform Registry

Modules present in Terraform Registry are remote modules

(modules that we write and not published to Terraform Registry ie which was discussed in previous section are local modules)

Local modules – not meant for say, creation of EC2 instance

Remote modules – there is module for EC2 instance creation ie even for minor things

----------

When to use Remote modules ?:

Reason for above question: Say, for EC2 instance creation, we can use “resource” of Terraform. We can also use AWS provided module

Answer:

Modules (Both local and remote)

https://developer.hashicorp.com/terraform/tutorials/modules/module

A screenshot of a document

Description automatically generated

**So, you should build configurations as local modules**

**In local modules, use remote modules as much as possible**

--------------  
example of using terraform remote module of AWS:

provider "aws" {

access\_key = "" //purushc user

secret\_key = ""

region = "ap-south-1"

}

module "ec2\_instance\_cluster" {

source = "terraform-aws-modules/ec2-instance/aws"

version = "5.2.1"

for\_each = toset(["one", "two", "three"])

name = "instance-${each.key}"

ami = "ami-057752b3f1d6c4d6c"

instance\_type = "t2.micro"

subnet\_id = "subnet-0466965d86d3e56d4"

tags = {

Terraform = "true"

Environment = "dev"

}

}

Module name “ec2\_instance\_cluster” can be changed

Source refers to the module being invoked

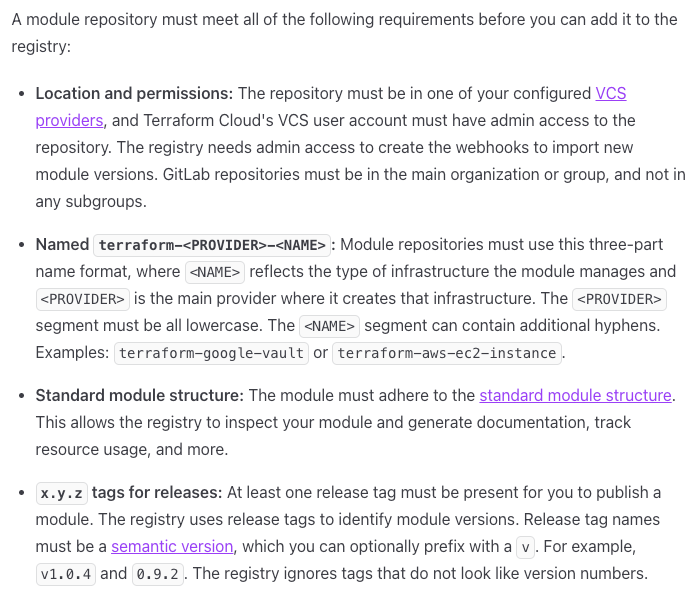
I think, all other fields like for\_each,…present within the module block are inputs as prescribed by the module

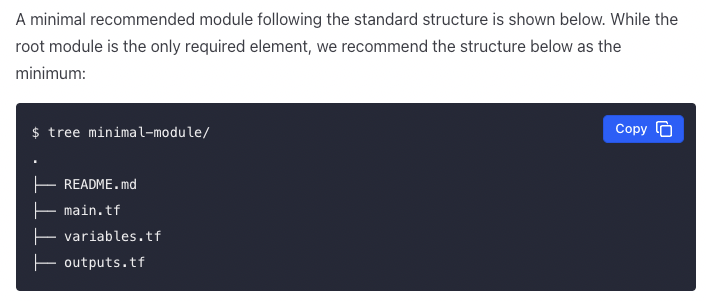
Module page for reference:

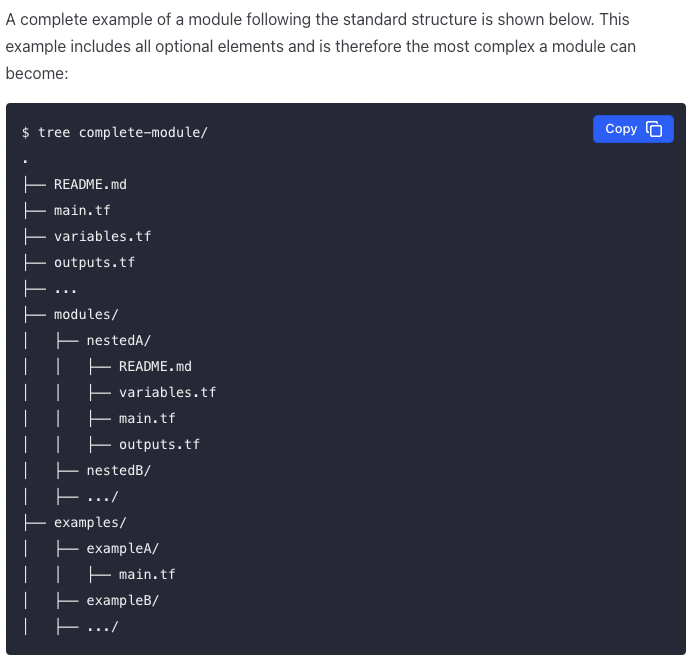
<https://registry.terraform.io/modules/terraform-aws-modules/ec2-instance/aws/latest>

Requirement for Publishing Modules in Terraform Registry  
**This topic is important for exam, as per tutor**

VCS below refers to Version Control System



----------------  


----------------  


Implementing Terraform Workspace

(<https://developer.hashicorp.com/terraform/cli/workspaces#use-cases>)

-----

-similar to some extent to switch db command of mongoDB

Ie you can switch to different workspaces, from the same directory

1 workspace = 1 environment

Ie say, dev workspace = dev env; test workspace = test env; default workspace = prod

-----

When a workspace is not explicitly created, it implies we are in default workspace

-----

Terraform code will be same, but, the variables per workspace will be different

-----

197002@AMBAU001115 modules\_registry1 % **terraform workspace show**

**default**

197002@AMBAU001115 modules\_registry1 % **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" from dev workspace, Terraform does not see state (ie details) of other workspaces

197002@AMBAU001115 modules\_registry1 % **terraform workspace list**

**default**

**\* dev**

Where \* denotes current workspace

----------

provider "aws" {

access\_key = "" //purushc user

secret\_key = ""

region = "ap-south-1"

}

resource "aws\_instance" "myec2" {

subnet\_id = "subnet-0466965d86d3e56d4"

ami = "ami-057752b3f1d6c4d6c"

// lookup(map, key) – will return the corresponding value

instance\_type = lookup(var.instance\_type, terraform.workspace)

}

variable "instance\_type" {

type = map

default = {

default = "t2.nano"

dev = "t2.micro"

prd = "t2.large"

}

}

In above, depending on the current workspace, instance\_type will be getting its value

If current workspace is dev, then instance\_type will be t2.micro

Remote State Management

Integrating with GIT for team management

This lecture, just shows, how to create a new repo and push the terraform code to it using git commands like git add, commit, push

Security Challenges in committing TFState file to Git

Say, you want to create RDS thro TF

*resource “aws\_db\_instance” “default”{*

*allocated\_storage = 5*

*…*

*username = “foo”*

*password = “${file(“../rds\_pass.txt”)}”*

*…*

*}*

Passwords should never be stored inside code

In some startups, passwords are referred from a file, say rds\_pass.txt in above case

rds\_pass.txt will not be committed to Git

There is a drawback of above approach as well:

When terraform apply is done, in generated TF state file, clear text passwords will be present

When we commit TF state file to Git, clear text passwords can be easily taken by hacker

Main idea of this lecture:

**TF state file should not be committed to Git**

Module Sources in Terraform

Child modules can be:

-local

-Terraform Registry

-Github/Bitbucket/…repo

-------------

Local:

-needs to be referenced using . or .. (same as node.js)

module “consul” {

source = “**..**/consul”

}

--------

Git Module source:

(can be github, bitbucket,…)

module “vpc” {

source = “**git::**https://example.com/vpc.git”

}

module “storage” {

source = “**git::**ssh://username@example.com/storage.git”

}

“Above refers to branch referenced by HEAD”

Can reference specific branch and tag names also, using ref parameter

module “storage” {

source = “**git::**ssh://username@example.com/storage.git?**ref**=dev\_branch”

}

Also, github and bitbucket has specific things like (git:: used above is generic way)

module “storage” {

source = “**bitbucket**.org/hashicorp/terraform-consul-aws”

}

Terraform and .gitignore

Tutor referred gitignore contents from below link:

<https://github.com/github/gitignore/blob/main/Terraform.gitignore>

search term: gitignore terraform

A screenshot of a computer program

Description automatically generated

Terraform Backends

**In real world**

**-Terraform files will be stored in Central Git Repo**

**-State files (terraform.tfstate) will be stored in Central Backend**

--State file will always stay in backend ie when a resource is created/destroyed, state will get updated in the central backend (and not in your local. Infact, there will be no state file in local)

--------

Some of the popular backends:

S3

Consul

Azurerm

Kubernetes

HTTP

ETCD

In S3,…, terraform.tfstate file is present, where the state info will be stored

--------

Actually,

local – the one that we use is also a backend, but, not for real world

Implementing S3 Backend