Terraform authentication methods:

1. Static method, access key/secret key on file
2. Using environmental variables
3. Shared credential file

Terraform backends---maintains state of resources, local and remote backend

Terraform modules---set of terraform configuration in a single directory like RDS Provisoning

Terraform registry---modules are linked with registry

Tainted resources---resources forced to be destroyed and create on next apply command

apply -replace

Remote backend: If multiple users working in the same resources then it will be hard to manage state so remote backend is required where state can be maintained and stored in the cloud like AWS S3.

Statefile locking: It’s a mechanism that blocks the operation on the state file to avoid the conflict between the multiple users performing the same operation

Once the lock is released then other user can work on the state file

Resource graph: it’s the visual representation of the resources and helps to create and modify the independent resources simultaneously

Terraform FMT command: it is used to rewrite the terraform configuration files to connonical format and style

Terraform providers command: it gives information of the providers working in the current configuration

uses HCL language

coding..

provider..

resource..

Init:  
The terraform init command is used to initialize a working directory containing Terraform configuration files.

It initilises plugins installation, child modules installation and backend init

Plan (The terraform plan command **creates an execution plan, which lets you preview the changes that Terraform plans to make to your infrastructure**

)

Apply

Destroy

Terraformf files:

Config file: terra.tf

State file: terraform.tfstate

Terraform var file: terraform.tfvars

provider "aws" {  
 region = "us-east-1"  
  
}  
  
  
resource "aws\_vpc" "myvpc" {  
 cidr\_block = "10.0.0.0/16"  
}  
resource "aws\_internet\_gateway" "igw" {  
 vpc\_id = aws\_vpc.myvpc.id  
  
 tags = {  
 name="gw"  
 }  
}  
  
resource "aws\_subnet" "mysubnet" {  
 vpc\_id = aws\_vpc.myvpc.id  
 cidr\_block = "10.0.1.0/24"  
 tags = {  
 name="sub"  
 }  
}  
  
resource "aws\_route\_table" "example" {  
 vpc\_id = aws\_vpc.myvpc.id  
  
 route = []  
 tags = {  
 Name = "example"  
 }  
}  
  
  
resource "aws\_route" "route" {  
 route\_table\_id = aws\_route\_table.example.id  
 destination\_cidr\_block = "0.0.0.0/0"  
 gateway\_id = aws\_internet\_gateway.igw.id  
 depends\_on = [aws\_route\_table.example]  
}  
  
resource "aws\_security\_group" "sg" {  
 name = "allow\_all\_tr"  
 description = "Allow all inbound traffic"  
 vpc\_id = aws\_vpc.myvpc.id  
  
 ingress {  
 description = "inbound"  
 from\_port = 0  
 to\_port = 0  
 protocol = "-1"  
 cidr\_blocks = ["0.0.0.0/0"]  
 ipv6\_cidr\_blocks = null  
 prefix\_list\_ids = null  
 security\_groups = null  
 self=null  
 }  
  
 egress {  
 from\_port = 0  
 to\_port = 0  
 protocol = "-1"  
 cidr\_blocks = ["0.0.0.0/0"]  
 ipv6\_cidr\_blocks = ["::/0"]  
 description = "outbound"  
 prefix\_list\_ids = null  
 security\_groups = null  
 self=null  
 }  
  
 tags = {  
 Name = "all traffic"  
 }  
}  
  
resource "aws\_route\_table\_association" "a" {  
 subnet\_id = aws\_subnet.mysubnet.id  
 route\_table\_id = aws\_route\_table.example.id  
}  
  
resource "aws\_network\_interface\_sg\_attachment" "sg\_attachment" {  
 security\_group\_id = aws\_security\_group.sg.id  
 network\_interface\_id = aws\_instance.myservv.primary\_network\_interface\_id  
}  
resource "aws\_instance" "myservv" {  
 ami = "ami-0e472ba40eb589f49"  
 instance\_type = "t2.micro"  
 subnet\_id = aws\_subnet.mysubnet.id  
  
 tags = {  
 name="Terraform"  
 }  
}

Terraform init:

It speaks to the provider and download the necessary dependencies, it creates the folder

.terraform where all the dependencies are.

Terraform plan: to tell, how many resources to add/destroy

Terraform apply: It will execute the plan, it will create the EC2 instance as an example

Variables:

4 ways:

1. Use variables inside the main.tf file
2. Use variables in the separate file called variables.tf
3. Use variables in the separate file called variables.tf and make another file stage.tfvars and prod.tfvars and give values here

Terraform plan -var-file=”stage.tfvars”

Terraform apply -var-file=”stage.tfvars”

1. Use variables in the separate file called variables.tf and give variables values at apply level like terraform apply -var=”instance\_type=t2.micro”

Terraform locals: to optimise the code, not to hard code value which are changing for staging and production

Terraform output: it is uses to show output in the console

Terraform count: for looping

Terraform for\_each loop

Terraform for loop

Terraform dynamic blocks

Terraform provioners: it is used to run any custom task on local or remote machine.

3 types:

File

Local-exec

Remote-exec

If two or more people working on the terraform code and applying at the same time then what will happen? Only one apply/process can work at one time as it will do the statefile lock.