We are still creating infrastructure manually .

We are manually creating Ec2 instance before we are deploying our docker image .

Or manually creation EKS cluster with all the roles , manually configure network .

Before we are able to deploy our application inside our cluster .

And once the infrastructure is created we need to manage that infrastructure manually .
Eg: updates ..

Automating this process just like we automate our build pipeline .

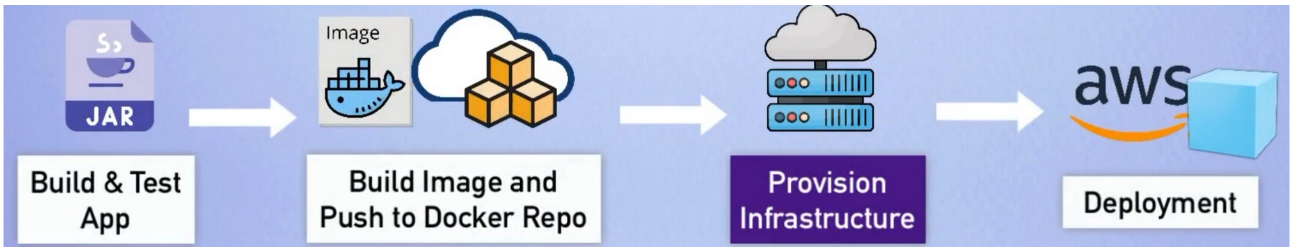
Use code to create and manage infrastructure .

Scripting it just like we script the pipeline using Jenkins file .

So that we can easier collaboration , version control , automation .

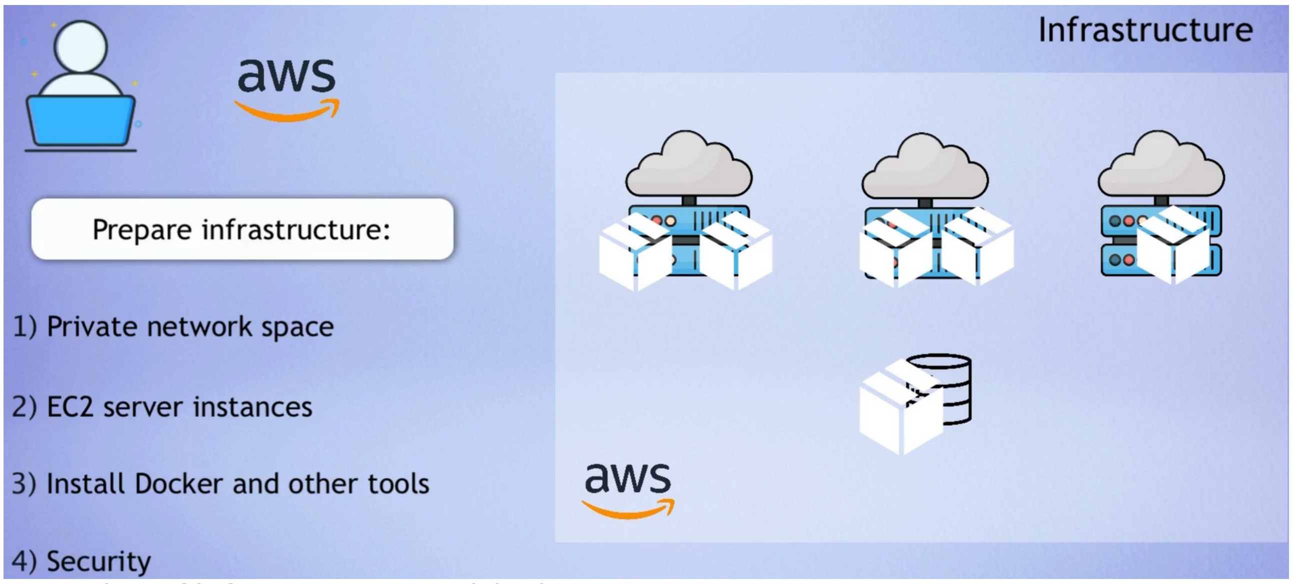
And that is InfrastructureAsCode . ===> **terraform**

What we will learn :
What is terraform
Providers
Resource and data storage
Terraform state
Output
Variables
Environment variables
Terraform commands
Automate eC2 server creation => creating VPC , subnet, route table , Internet gateway , firewall security group , deploy nginx docker container .
Using terraform modules
Modularising projects



What is terraform ?

Automate your infrastructure
Your platform
And services that run on that platform
Open source
Use declarative language means define what end result u want
Vs imperative style = define exact steps = how ?



Prepare infrastructure:

1) Private network space

2) EC2 server instances

3) Install Docker and other tools

4) Security

Meaning of infrastructure provisioning =>

Eg: Deploying multiple micro services in the different servers as a docker container .

Once the infrastructure is prepared u can deploy your docker container on those servers.

Two main task :
1. **Provisioning infrastructure (devOps)**
2. Deploying that application on that infrastructure (software developers)

Terraform comes in the provisioning the infrastructure

Eg: creation Ec2 , create VPC , create AWS user and permissions , install docker . Etc.

Difference btw Ansible and terraform .

Both are infrastructure as a code means both used for automate => provisioning ,configure , managing the infrastructure .

However terraform is mainly infrastructure provision tool , can deploy app on that infrastructure

Ansible is a configuration tool so once the infrastructure is provisioned now ansible can use to configure it and deploy application ,install and update software on it.

Ansible is more mature and terraform is relatively new and terraform is changing dynamically and its more advance in orchestration .

Note : terraform is better tool for provisioning the infrastructure and ansible is better tool for configuration infrastructure

Once u created your infrastructure u will be continuously managing of scaling up down your infrastructure .

So u need a automation tool to continuous changes to your infrastructure .

Replication infrastructure

Eg: we have a production env and a dev env or staging env where we copy all the infrastructure specific to that env use case .

As terraform makes that tasks more easy .
Terraform Architecture .

How does terraform connect to the platform providers eg: how terraform connect to AWS create a virtual space start Ec2 instance , configure networking

So in order to do this terraform has two main component :

1. **Terraform core** = core uses two input sources in order to do its job
A: terraform config   = u as a user will write this what needs to be created and config
B: terraform state = correct setup of infrastructure

So what core does is it takes a input and figure out a plan what needs to be done .
Eg: what need to be create / update / destroy etc.

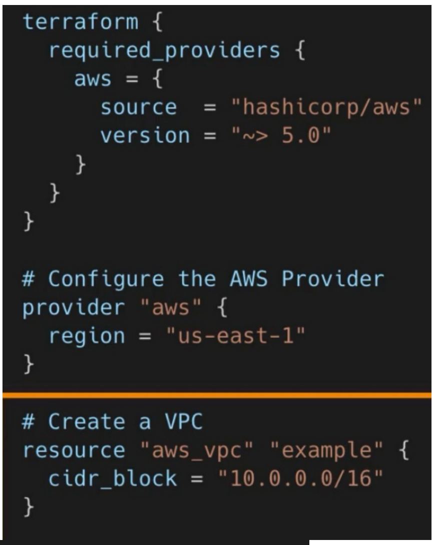
What is the correct state and what state u desire from the config file .

2. **Provides like AWS (iaas)or kubernetes (paaS)or software(saas) :**
Eg: create a aws server -> deploy or create k8s on top of it -> then create services or components inside that k8s cluster .

It does that through providers so terraform has over 100 provides or these technologies to over 1000 resources.

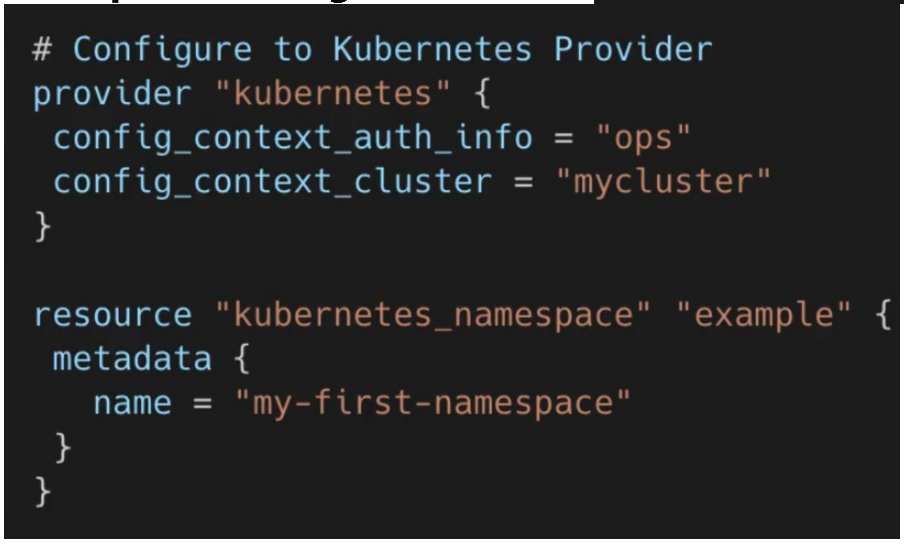
And each providers gives user access to its resources .
Eg: from AWS provider u have access 100 of AWS resources like EC2 , s3 , users
From k8s provider u have access to namespaces , services , deployment

Terraform expertise in in AWS (Iaas) provisioning and for other pass and saas configuration u can use Ansible .



```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 5.0"
    }
  }
}

# Configure the AWS Provider
provider "aws" {
  region = "us-east-1"
}

# Create a VPC
resource "aws_vpc" "example" {
  cidr_block = "10.0.0.0/16"
}
```

Example of configuration file :



```
# Configure to Kubernetes Provider
provider "kubernetes" {
 config_context_auth_info = "ops"
 config_context_cluster = "mycluster"
}

resource "kubernetes_namespace" "example" {
 metadata {
   name = "my-first-namespace"
 }
}
```

Defining provides and then deicing resources and its attributes .

Declarative vs imperative :

What does declarative mean >?

You define the **end state** in your config file : not the steps to create a EC2 or VPC etc..
Eg: u define 5 serves with following network config and AWS users with following permissions

Terraform go to it form me !!

While imperative means = > defining each steps = HOW .

Benefits when updating infrastructure :

With imperative config file :
-> remove 2 serves
-> add firewall config
-> add permission to AWS users
You are giving the instruction

With declarative approach :
I need my new desired state to a of 7 servers , this firewall config ,user with following permissions ,figure out what ever needs to be done !
And convert the correct state to my new desired state .

Adjust old config file and re-execute , clean and small config files , always know the correct setup just by looking the config file .

Commands :
Refresh = query infrastructure provide to get the correct state
Plan = create a execution plan => core is taking your config file and state and decide what needs to be one than a plan . / determine what actions are necessary to achieve desired state .
Plan is just a preview ,no changes to real resource .

Apply = execute the plan
Destroy = destroy the whole setup and removing elements one by one / infrastructure .

Key points : terraform is for configuring infrastructure and not for installing application on that .
Terraform is a universal infrastructure as a code tool
If u have a hybrid cloud in your company instead of having different tools for automation infrastructure creation u can have terraform .
Terraform can easily integrate with different tools like AWS , docker , k8s , GitHub, gitlab, Jenkins, docker , vm ware etc..
Just one tools to talk to all these technologies and there apis .

Installing terraform locally > :

On Mac : brew tap hashicorp/tap
: brew install hashicorp/tap/terraform

Providers in terraform :

First we need to connect to our AWS account
Interact with these technology with there APIs
Provider in terraform is a code or program that knows who to talk to that specific technology

Go to the terraform website -> providers .

AWS is one of the main providers as its widely used and manually working with AWS is very hard so automating AWS with terraform is preferred .

Officially owned and maintained by hashicorp

Partner providers -> owned by third party technology provides , actively maintained , hashicorp verify the authenticity .

Community providers ->

This means when every u need terraform for different technology integration and we need terraform to connect to AWS and create resources inside u can use providers

Terraform is well documented =>
And these are well search optimised => eg : eC2 terraform u will get the link to the provides documentation for Ec2 .

Install and connect to providers :
Don't hard code the credential In the config files .
As config file will be hosted in the git repo .

```
Provider "aws" {
Region = " "
access_key = ""
Secret_key = ""
}
```

How to find out what are the attribute to pass on all are documented in the terraform .

Providers needs to be installed in terraform in order to use them .

Use the select provider option to get the template:

Create a separate file provider.tf

open terminal inside the terraform = > : **terraform init** //
// it will initialise it as a working directory and install providers defined in the terraform config file .

For providers that are not part of hashicorp or not in the terraform official directory there we need to define that template by our self only .

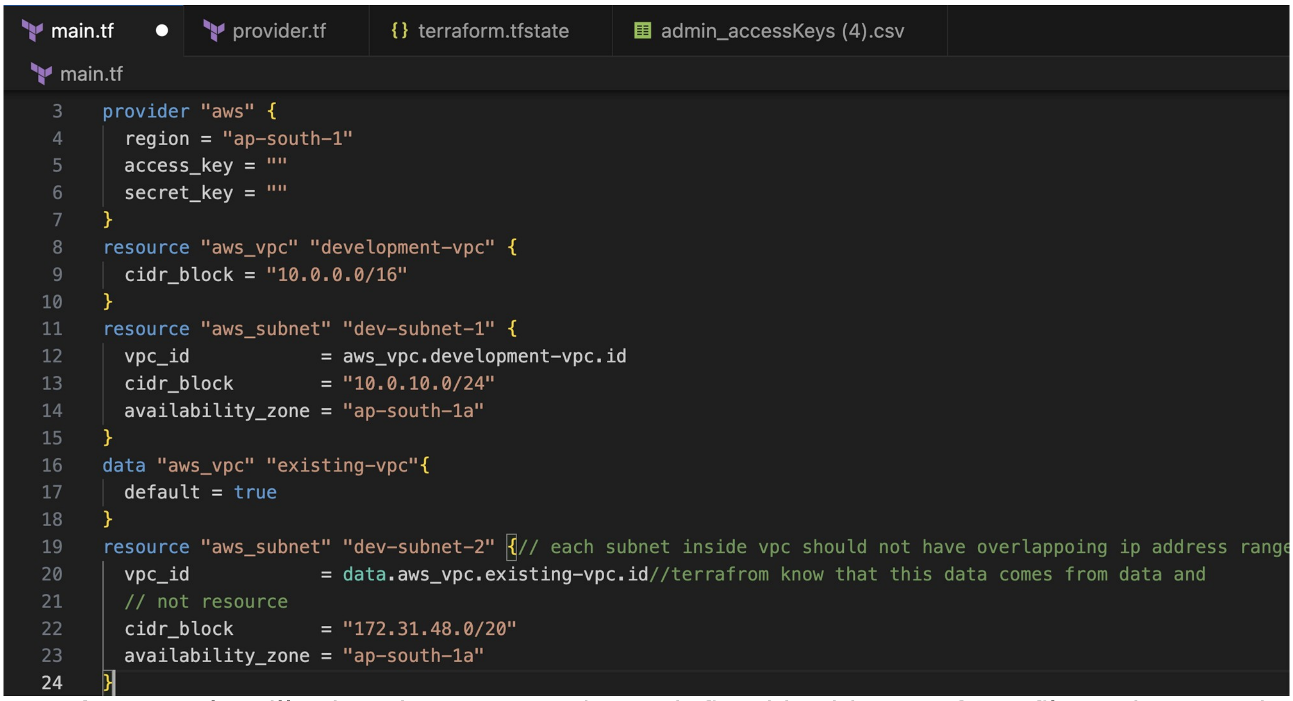
Eg: instead of AWS use linode

Providers exposing resources :

After we have defined the providers who can we interact with these providers.
Complete API is available now .
Now through provides now we have access to all the services of AWS and its resources.

Each resource block describes one or more infrastructure object s :

Terraform apply :



```
provider "aws" {
  region = "ap-south-1"
  access_key = ""
  secret_key = ""
}
resource "aws_vpc" "development-vpc" {
  cidr_block = "10.0.0.0/16"
}
resource "aws_subnet" "dev-subnet-1" {
  vpc_id            = aws_vpc.development-vpc.id
  cidr_block        = "10.0.10.0/24"
  availability_zone = "ap-south-1a"
}
data "aws_vpc" "existing-vpc"{
  default = true
}
resource "aws_subnet" "dev-subnet-2" {// each subnet inside vpc should not have overlappoing ip address range
  vpc_id            = data.aws_vpc.existing-vpc.id//terrafrom know that this data comes from data and
  // not resource
  cidr_block        = "172.31.48.0/20"
  availability_zone = "ap-south-1a"
}
```

Terraform apply will take what ever we have defined in this terraform file . When we do apply terraform will create the second vpc

Terraform determines what actions are necessary to achieve our desired state .

Data sources :
Allow data to be fetched for use in TF configuration .

Data allow us to query existing resources data from the aws while resources allow u to create new resources in aws .

Two type of components u get from provider -> 1. Resource 2. Data .

Go to the terraform AWS registry to see all of these data providers.

Name must be unique for each resource type :

```
data "aws-vpc" "existing-vpc"{
 default = true
}

resource "aws_subnet" "dev-subnet-1" {
 vpc_id            = data.aws-vpc.existing-vpc.id
 cidr_block        = "10.0.0.0/24"
 availability_zone = "ap-southeast-2"
}
```

using aws-vpc // hyphen in the resource naming will give u error

Note : each subnet inside the VPC has to have different set of ip addresses.

Configuration syntax id consistence for all the providers

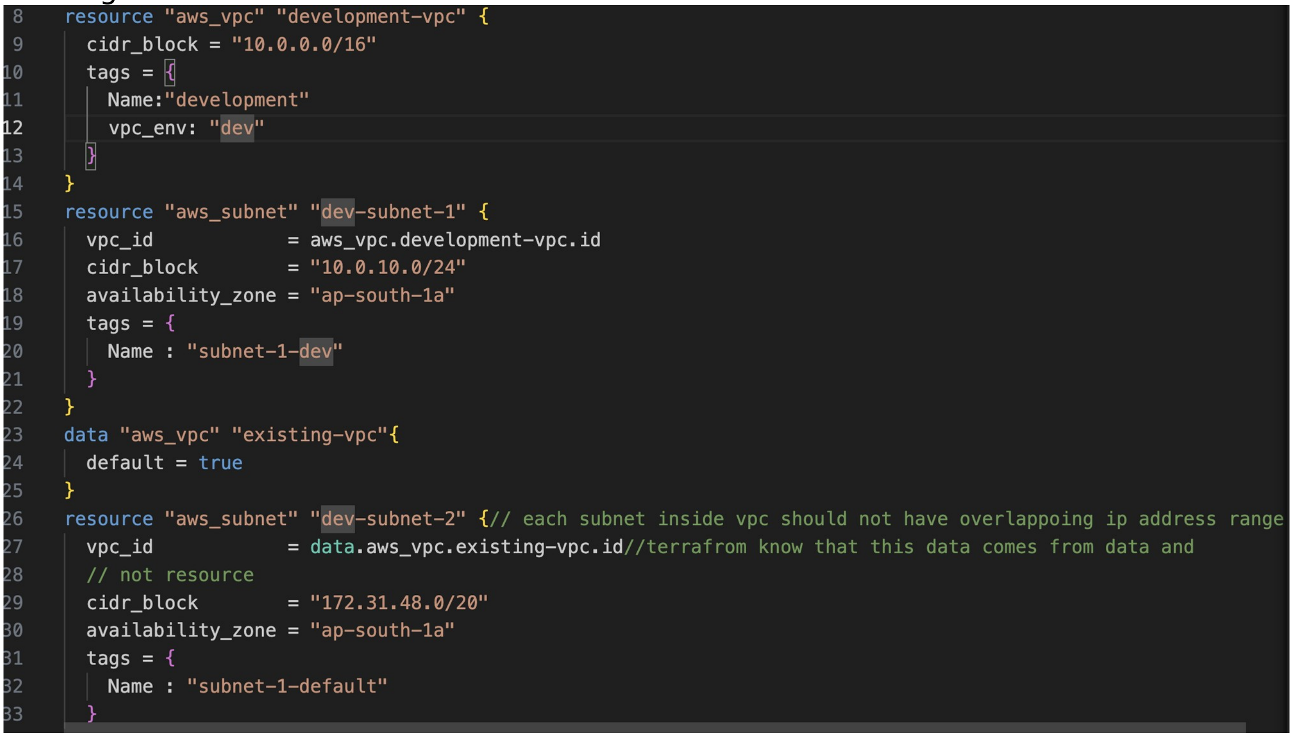
Same as programming -> provider as importing library
Defining Resources and data is like calling a function .
For that provider library that u just imported
By passing in some arguments = parameters to that function .
Resource is a function that creates something
Data is a function that returns something that already exists .
Access keys or secret keys are the user credentials we need to access any thing in AWS .

Note: if we hit apply again without changing any thing it will just refresh the states resources we have created .
This is because terraform is declarative we focus on end result .
Terraform will just check the current state and the desired state .

Note: terraform advantage => **idempodent** means when every u apply same exact configuration u alway get the same result . No need to remember the current state .

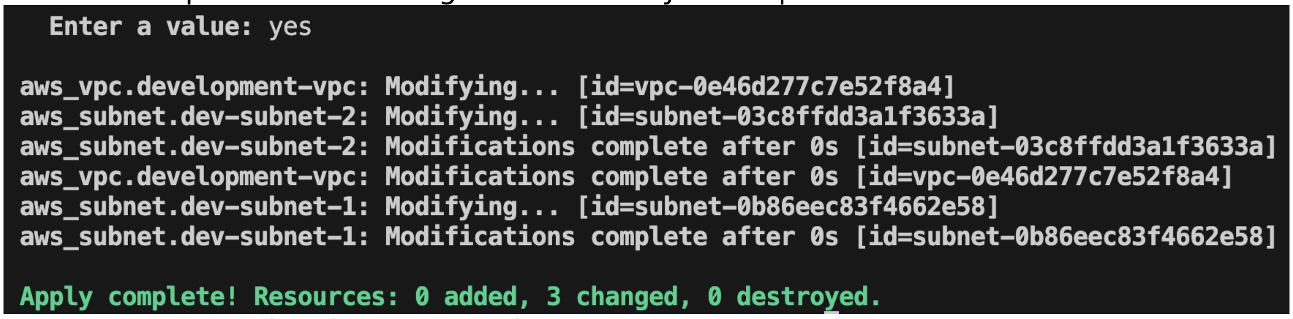
Change and destroy terraform resources :

Naming our subnet and VPC .



```
resource "aws_vpc" "development-vpc" {
  cidr_block = "10.0.0.0/16"
  tags = {
    Name:"development"
    vpc_env: "dev"
  }
}
resource "aws_subnet" "dev-subnet-1" {
  vpc_id            = aws_vpc.development-vpc.id
  cidr_block        = "10.0.10.0/24"
  availability_zone = "ap-south-1a"
  tags = {
    Name : "subnet-1-dev"
  }
}
data "aws_vpc" "existing-vpc"{
  default = true
}
resource "aws_subnet" "dev-subnet-2" {// each subnet inside vpc should not have overlappoing ip address range
  vpc_id            = data.aws_vpc.existing-vpc.id//terrafrom know that this data comes from data and
  // not resource
  cidr_block        = "172.31.48.0/20"
  availability_zone = "ap-south-1a"
  tags = {
    Name : "subnet-1-default"
  }
```

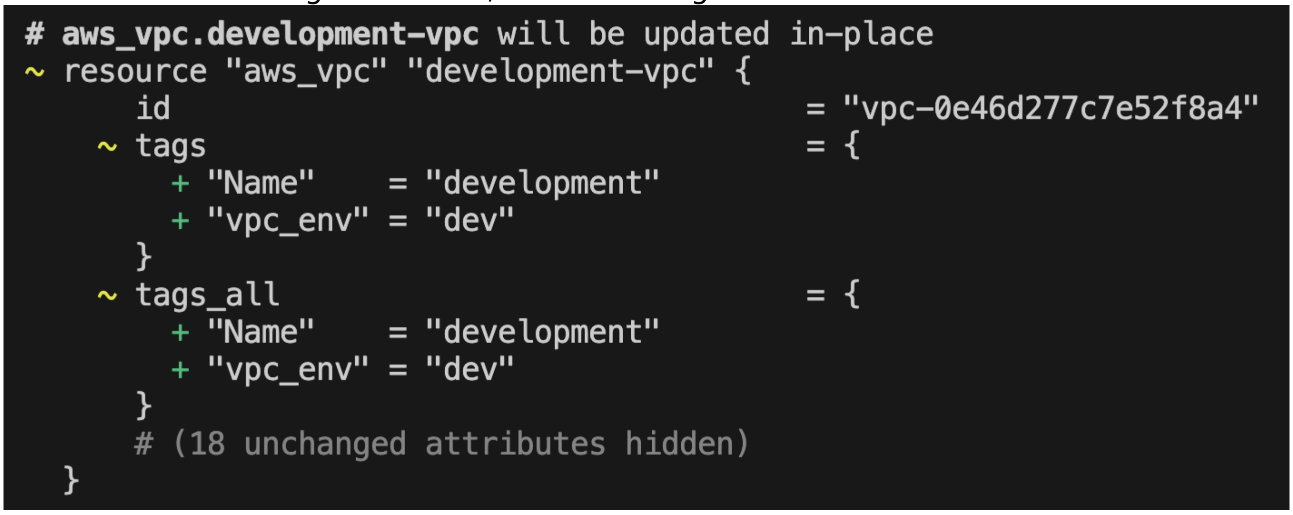
We can add parameter call tags : which is key value pair .



```
  Enter a value: yes

aws_vpc.development-vpc: Modifying... [id=vpc-0e46d277c7e52f8a4]
aws_subnet.dev-subnet-2: Modifying... [id=subnet-03c8ffdd3a1f3633a]
aws_subnet.dev-subnet-2: Modifications complete after 0s [id=subnet-03c8ffdd3a1f3633a]
aws_vpc.development-vpc: Modifications complete after 0s [id=vpc-0e46d277c7e52f8a4]
aws_subnet.dev-subnet-1: Modifying... [id=subnet-0b86eec83f4662e58]
aws_subnet.dev-subnet-1: Modifications complete after 0s [id=subnet-0b86eec83f4662e58]

Apply complete! Resources: 0 added, 3 changed, 0 destroyed.
```

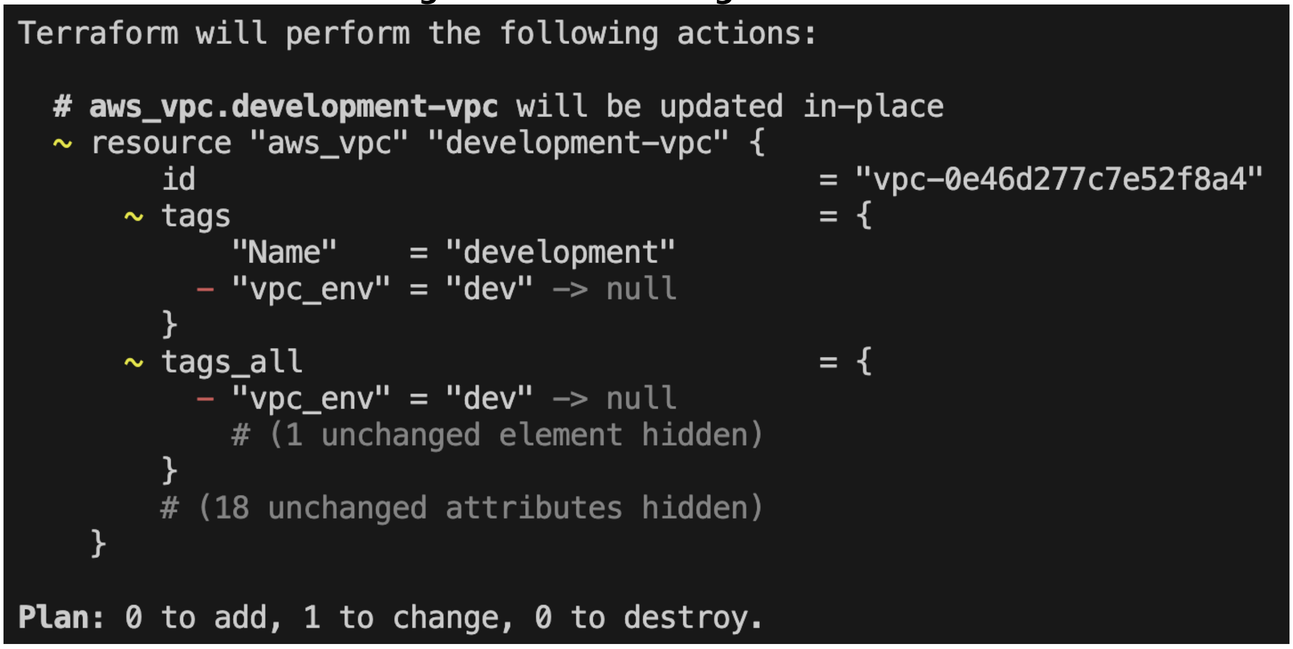
Here we can see the changes are done .

~ tild is for the change resource , + for creating a resource



```
 # aws_vpc.development-vpc will be updated in-place
 ~ resource "aws_vpc" "development-vpc" {
        id                                   = "vpc-0e46d277c7e52f8a4"
      ~ tags                                 = {
          + "Name"    = "development"
          + "vpc_env" = "dev"
        }
      ~ tags_all                             = {
          + "Name"    = "development"
          + "vpc_env" = "dev"
        }
        # (18 unchanged attributes hidden)
    }
```

How to remove something from the existing resource :



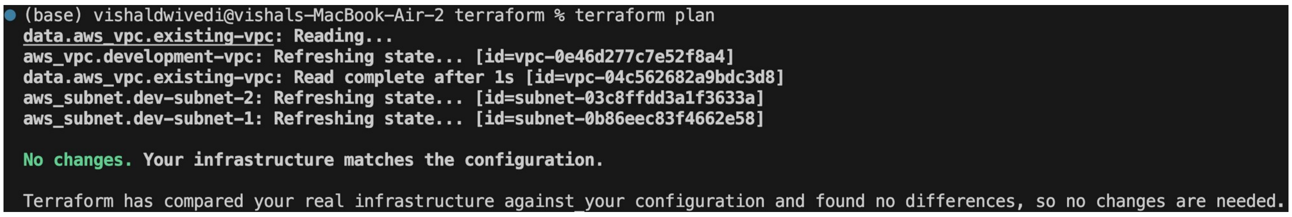
```
Terraform will perform the following actions:

  # aws_vpc.development-vpc will be updated in-place
  ~ resource "aws_vpc" "development-vpc" {
        id                                   = "vpc-0e46d277c7e52f8a4"
      ~ tags                                 = {
            "Name"    = "development"
          - "vpc_env" = "dev" -> null
        }
      ~ tags_all                             = {
          - "vpc_env" = "dev" -> null
            # (1 unchanged element hidden)
        }
        # (18 unchanged attributes hidden)
    }

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

REMOVING AND DESTROYING THE RESOURCES :

There are two ways to remove any resource .
1. Delete the resource block in the main.tf file.
2. Terraform destroy -target aws_subnet.dev-subnet-2

Always apply changes through terraform config file . Especially working in a team .

More terraform commands :
1. To check the difference btw the current state vs desired state .

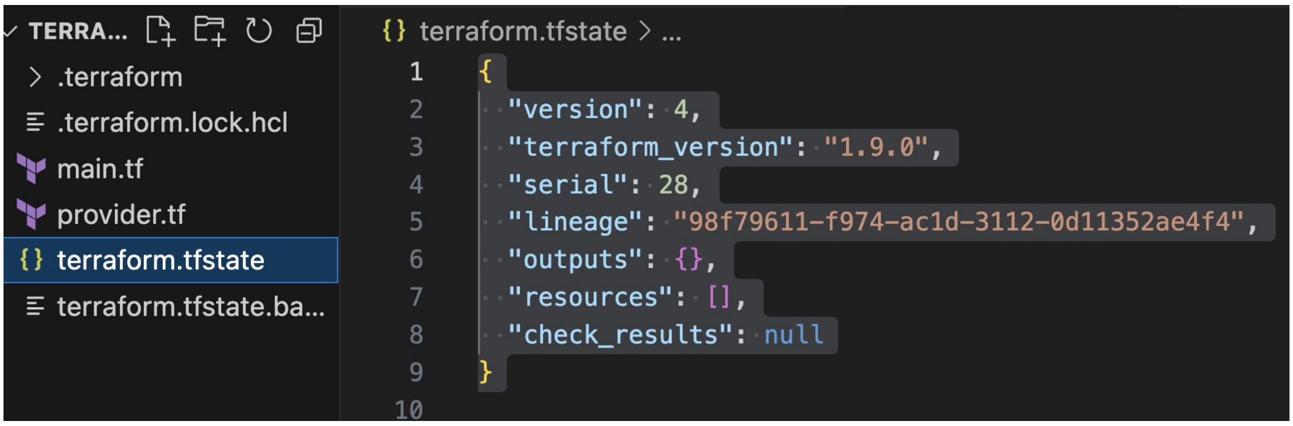


```
(base) vishaldwivedi@vishals-MacBook-Air-2 terraform % terraform plan
data.aws_vpc.existing-vpc: Reading...
aws_vpc.development-vpc: Refreshing state... [id=vpc-0e46d277c7e52f8a4]
data.aws_vpc.existing-vpc: Read complete after 1s [id=vpc-04c562682a9bdc3d8]
aws_subnet.dev-subnet-2: Refreshing state... [id=subnet-03c8ffdd3a1f3633a]
aws_subnet.dev-subnet-1: Refreshing state... [id=subnet-0b86eec83f4662e58]

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.
```

2. If u don’t want a conformation :
: terraform apply -auto-approve

3. Destroy complete infrastructure :
:terraform destroy

TERRAFORM STATE :



```json
{
  "version": 4,
  "terraform_version": "1.9.0",
  "serial": 28,
  "lineage": "98f79611-f974-ac1d-3112-0d11352ae4f4",
  "outputs": {},
  "resources": [],
  "check_results": null
}
```

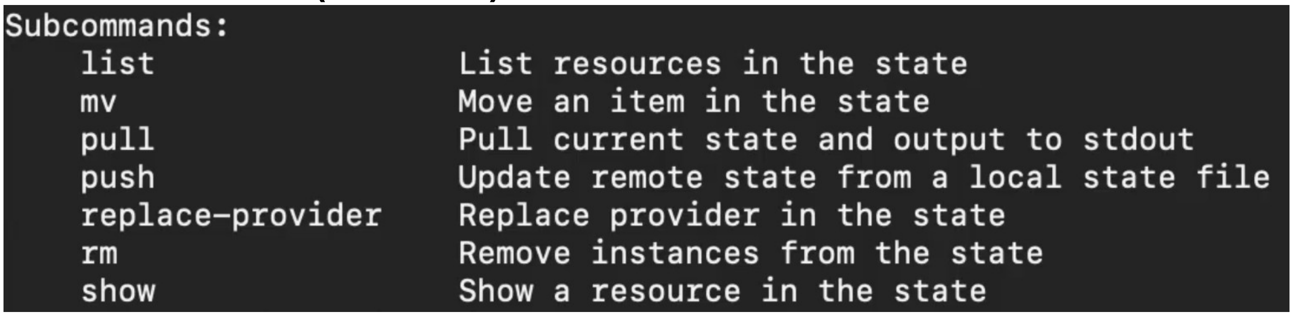
JSON file = : where terraform stores the state of your real world resources in your managed infrastructure .

From the first apply it record every thing and store the current state in that terraform file.

Refresh - update the state file with real world infrastructure.

Currently we can see that there is no recourse [] in the aws account .
Backup file is the previous state =>

NOTE : after applying , the terraform state file will be updated
: terraform state : (command) ->

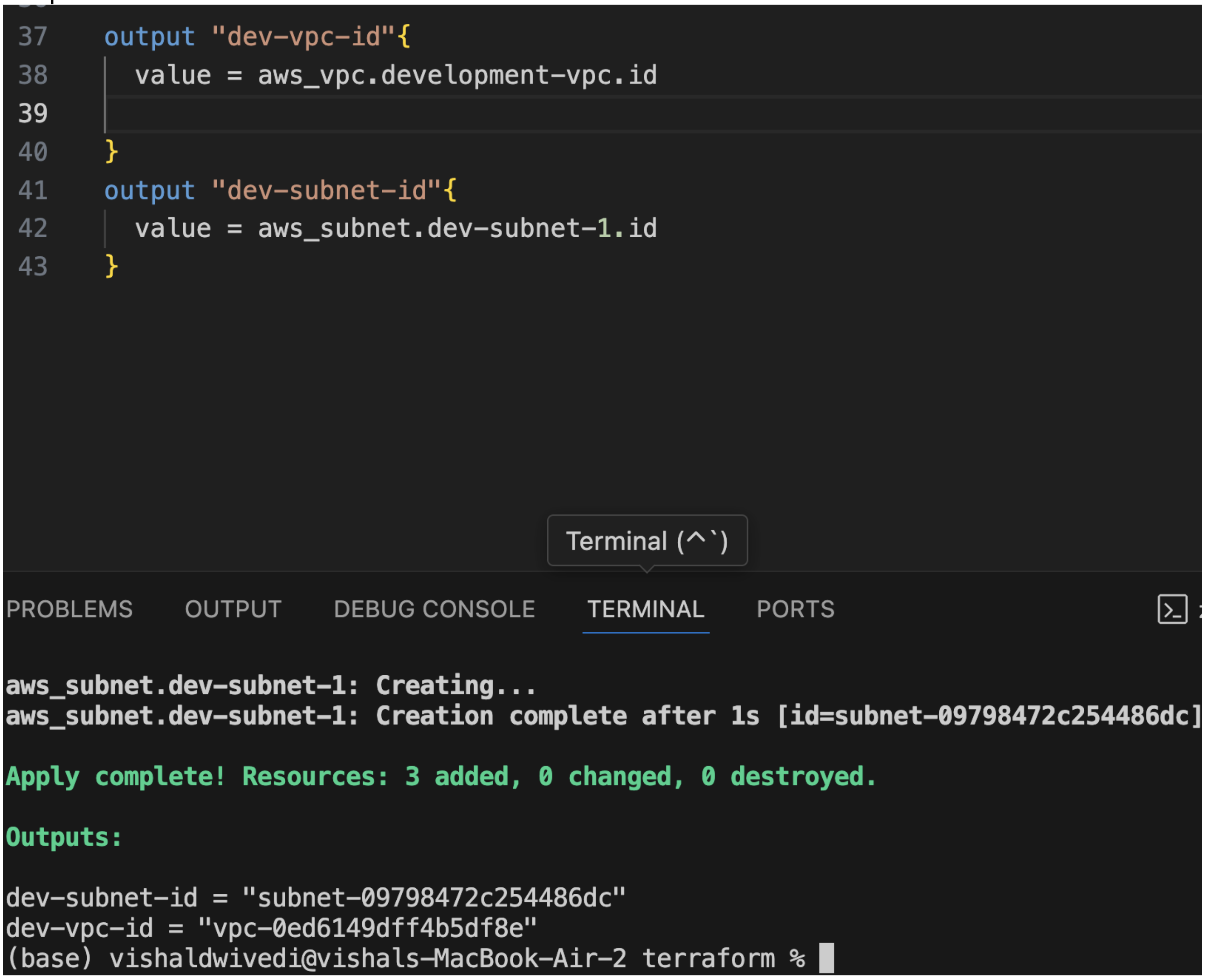


```
Subcommands:
    list                List resources in the state
    mv                  Move an item in the state
    pull                Pull current state and output to stdout
    push                Update remote state from a local state file
    replace-provider    Replace provider in the state
    rm                  Remove instances from the state
    show                Show a resource in the state
```

OUTPUT VALUES :
Like function return values : what type of value u need to output after applying our configuration u can tell that to terraform via output function .

Step1 : terraform destroy

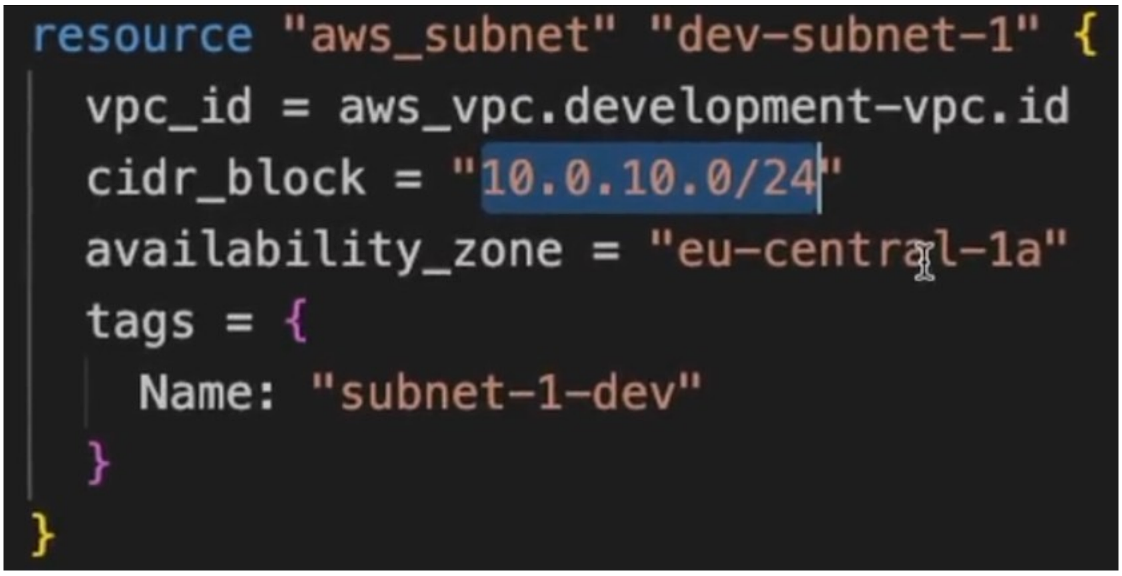
Step2 :



```
output "dev-vpc-id"{
  value = aws_vpc.development-vpc.id

}
output "dev-subnet-id"{
  value = aws_subnet.dev-subnet-1.id
}
```

```
aws_subnet.dev-subnet-1: Creating...
aws_subnet.dev-subnet-1: Creation complete after 1s [id=subnet-09798472c254486dc]

Apply complete! Resources: 3 added, 0 changed, 0 destroyed.

Outputs:

dev-subnet-id = "subnet-09798472c254486dc"
dev-vpc-id = "vpc-0ed6149dff4b5df8e"
(base) vishaldwivedi@vishals-MacBook-Air-2 terraform %
```

VARIABLES IN TERRAFORM :



```
resource "aws_subnet" "dev-subnet-1" {
  vpc_id = aws_vpc.development-vpc.id
  cidr_block = "10.0.10.0/24"
  availability_zone = "eu-central-1a"
  tags = {
    Name: "subnet-1-dev"
  }
}
```

We don’t want value of cider block hardCoded here but we want to pass it as a parameter .

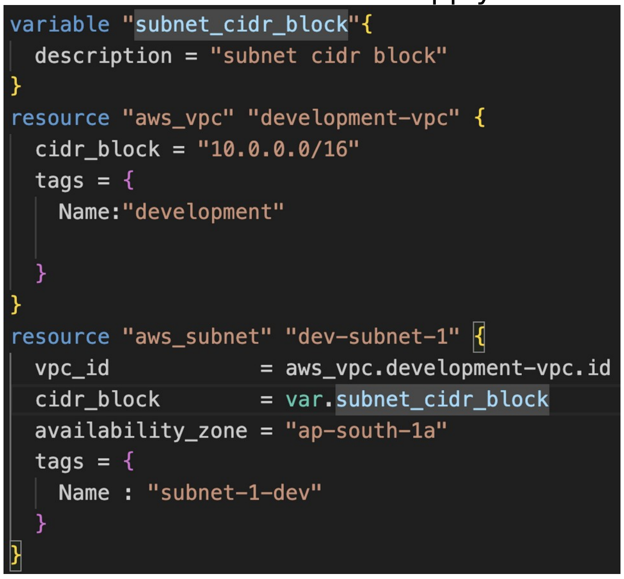
Input variable are like function arguments , reusability .

U want to u use same template parameters for different use cases .eg: DEV env , and production evn having same main.tf file .

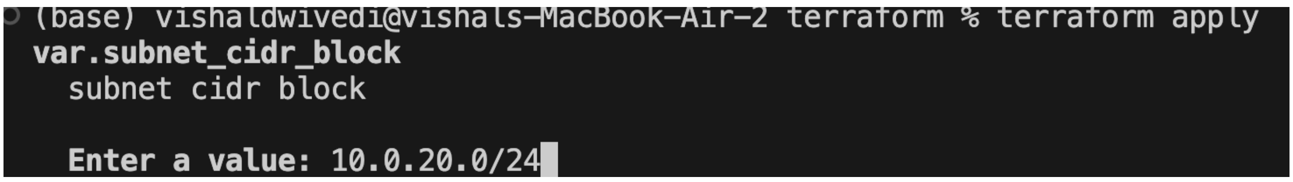
Variable in terraform is defined in variable keyWord = >

3 ways to pass values in the input variable =>

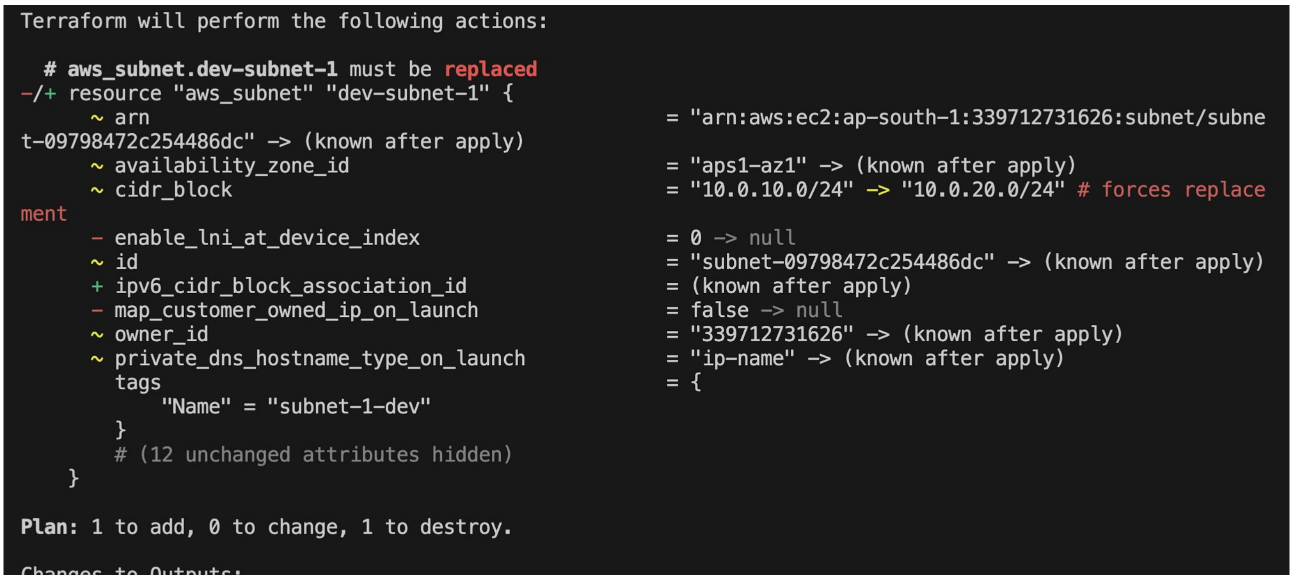
1. When we => : terraform apply : we will get the prompt to enter the cidr block



```
variable "subnet_cidr_block"{
  description = "subnet cidr block"
}
resource "aws_vpc" "development-vpc" {
  cidr_block = "10.0.0.0/16"
  tags = {
    Name:"development"


  }
}
resource "aws_subnet" "dev-subnet-1" {
  vpc_id            = aws_vpc.development-vpc.id
  cidr_block        = var.subnet_cidr_block
  availability_zone = "ap-south-1a"
  tags = {
    Name : "subnet-1-dev"
  }
}
```



```
(base) vishaldwivedi@vishals-MacBook-Air-2 terraform % terraform apply
var.subnet_cidr_block
  subnet cidr block

  Enter a value: 10.0.20.0/24
```



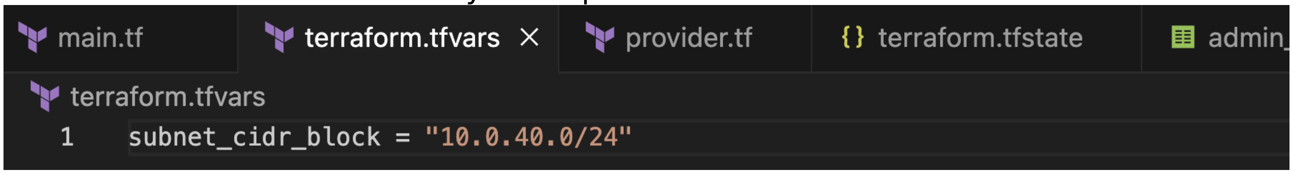
```
Terraform will perform the following actions:

  # aws_subnet.dev-subnet-1 must be replaced
-/+ resource "aws_subnet" "dev-subnet-1" {
      ~ arn                                            = "arn:aws:ec2:ap-south-1:339712731626:subnet/subne
t-09798472c254486dc" -> (known after apply)
      ~ availability_zone_id                           = "aps1-az1" -> (known after apply)
      ~ cidr_block                                     = "10.0.10.0/24" -> "10.0.20.0/24" # forces replace
ment
      - enable_lni_at_device_index                     = 0 -> null
      ~ id                                             = "subnet-09798472c254486dc" -> (known after apply)
      + ipv6_cidr_block_association_id                 = (known after apply)
      - map_customer_owned_ip_on_launch                = false -> null
      ~ owner_id                                       = "339712731626" -> (known after apply)
      ~ private_dns_hostname_type_on_launch            = "ip-name" -> (known after apply)
        tags                                           = {
            "Name" = "subnet-1-dev"
        }
        # (12 unchanged attributes hidden)
    }

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

So the only subnet 10.0.10.0/24 get removed and new subnet added 10.0.20.0/24

This way is not efficient way but for testing it is good .for automation we need without a user prompt .

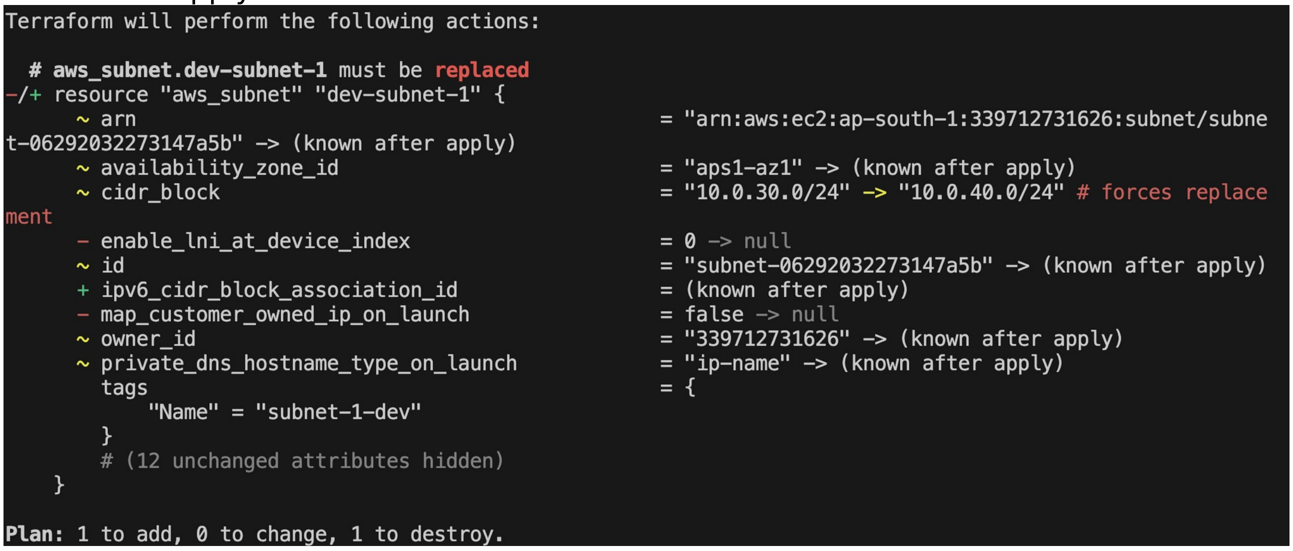
2. In the command line = > : terraform apply -var “subnet_cidr_block=10.0.30.0/24”
3. Defining variable file and assigning all the value in the TF configurations on that file =>

Default name which terraform will recognise as variable file is => terraform.tfvars
Inside this file we can define key value pairs for our variables .

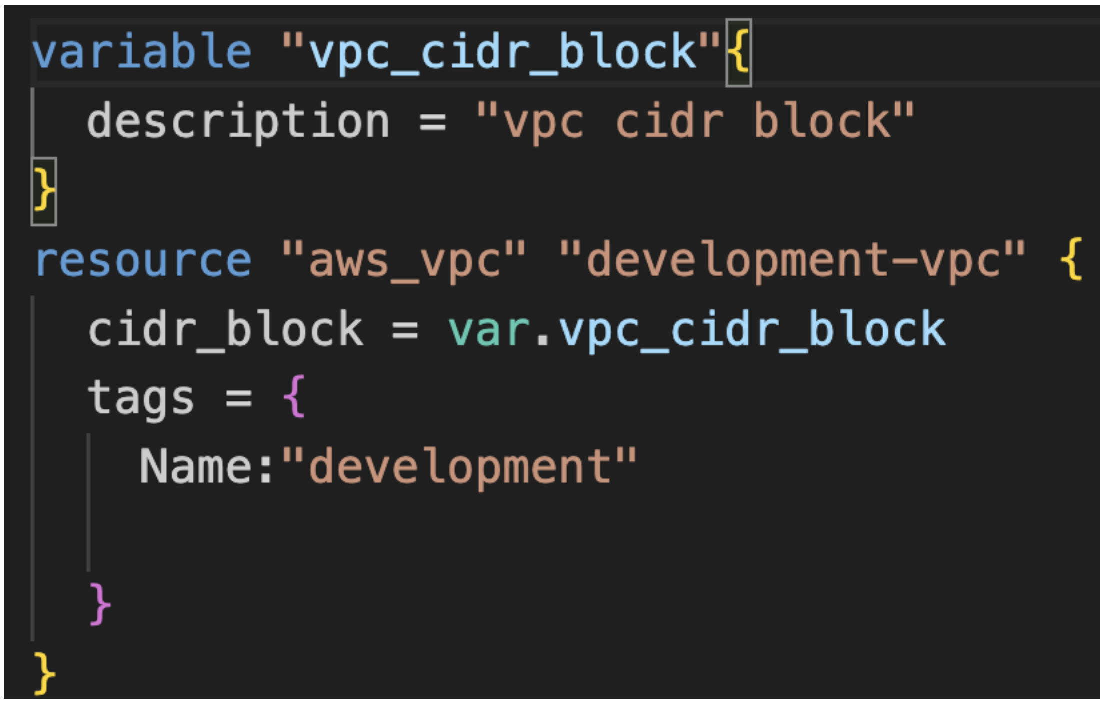


```
subnet_cidr_block = "10.0.40.0/24"
```

: terraform apply :



```
Terraform will perform the following actions:

  # aws_subnet.dev-subnet-1 must be replaced
-/+ resource "aws_subnet" "dev-subnet-1" {
      ~ arn                                            = "arn:aws:ec2:ap-south-1:339712731626:subnet/subne
t-06292032273147a5b" -> (known after apply)
      ~ availability_zone_id                           = "aps1-az1" -> (known after apply)
      ~ cidr_block                                     = "10.0.30.0/24" -> "10.0.40.0/24" # forces replace
ment
      - enable_lni_at_device_index                     = 0 -> null
      ~ id                                             = "subnet-06292032273147a5b" -> (known after apply)
      + ipv6_cidr_block_association_id                 = (known after apply)
      - map_customer_owned_ip_on_launch                = false -> null
      ~ owner_id                                       = "339712731626" -> (known after apply)
      ~ private_dns_hostname_type_on_launch            = "ip-name" -> (known after apply)
        tags                                           = {
            "Name" = "subnet-1-dev"
        }
        # (12 unchanged attributes hidden)
    }

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

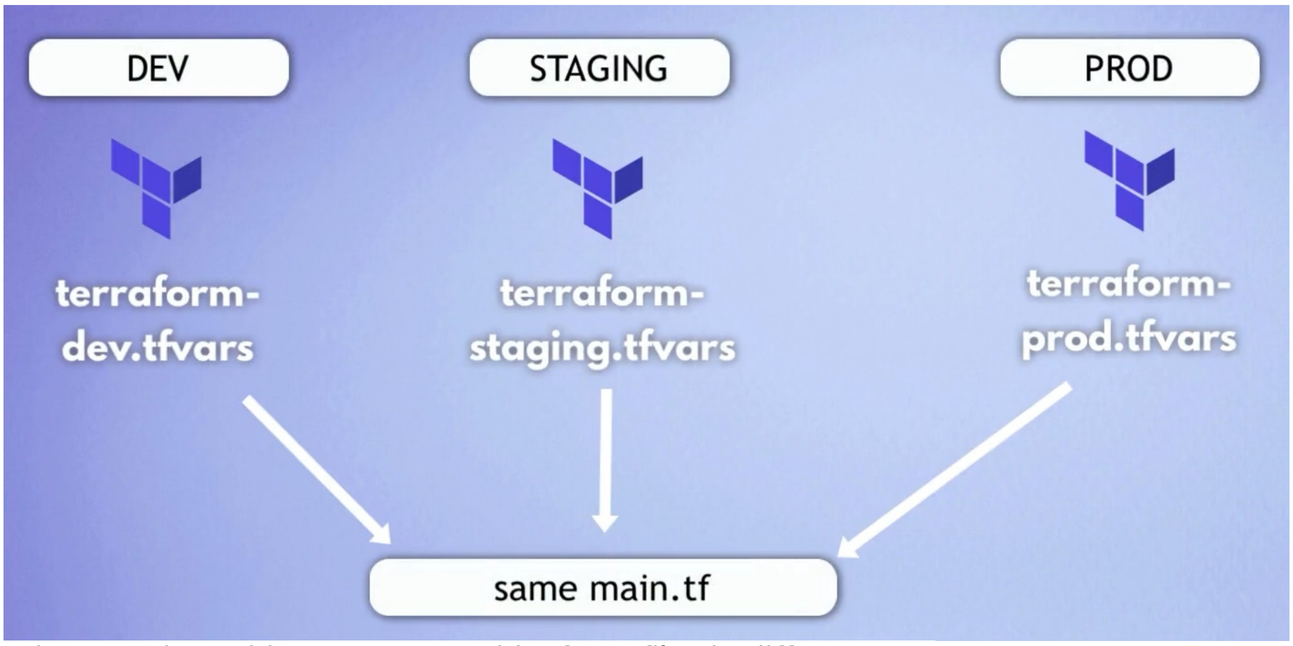


```
variable "vpc_cidr_block"{
  description = "vpc cidr block"
}
resource "aws_vpc" "development-vpc" {
  cidr_block = var.vpc_cidr_block
  tags = {
    Name:"development"

  }
}
```

Use cases of input variables :

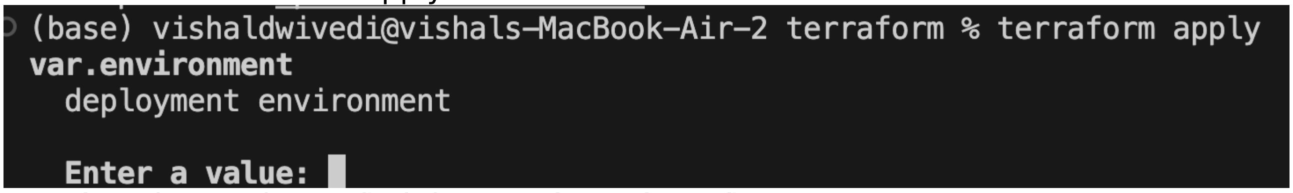
1.replicate same infrastructure for different env eg: in dev , staging , production etc.

By don’t this u will have =>



U just need to add new name to this tfvars files in different env

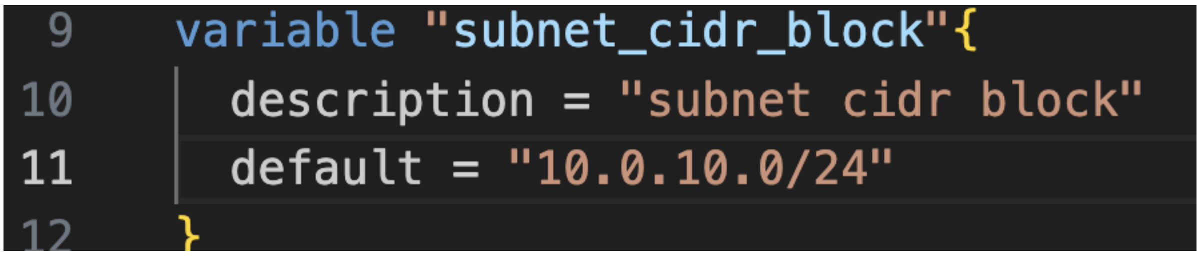
Note : we have changed the file name to : terraform-dev.tfvars
And if we do : terraform apply it will ask for



```
(base) vishaldwivedi@vishals-MacBook-Air-2 terraform % terraform apply
var.environment
  deployment environment

  Enter a value:
```

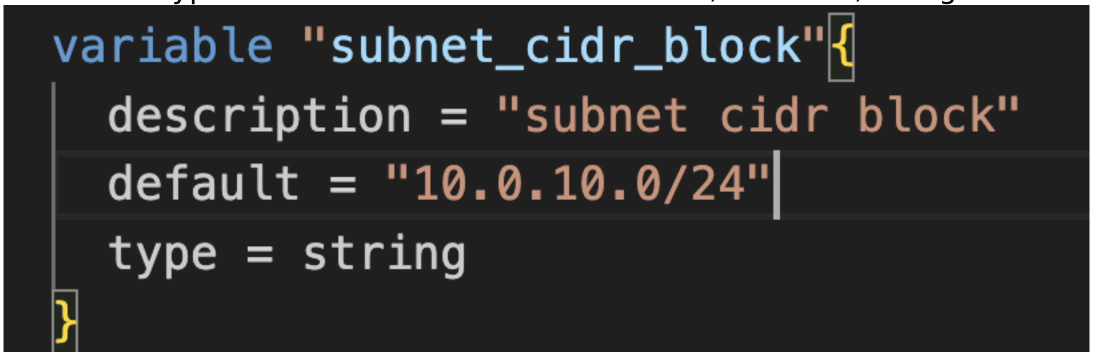
Terraform is not able to find the terraform.tfvars file .
Instead we do : terraform apply -var-file terraform-dev.tfvars

Default values :
We can assign default value to our variables .
So if we remove the variable declaration on the terraform.tfvars file it will take the default value .



```
variable "subnet_cidr_block"{
  description = "subnet cidr block"
  default = "10.0.10.0/24"
}
```

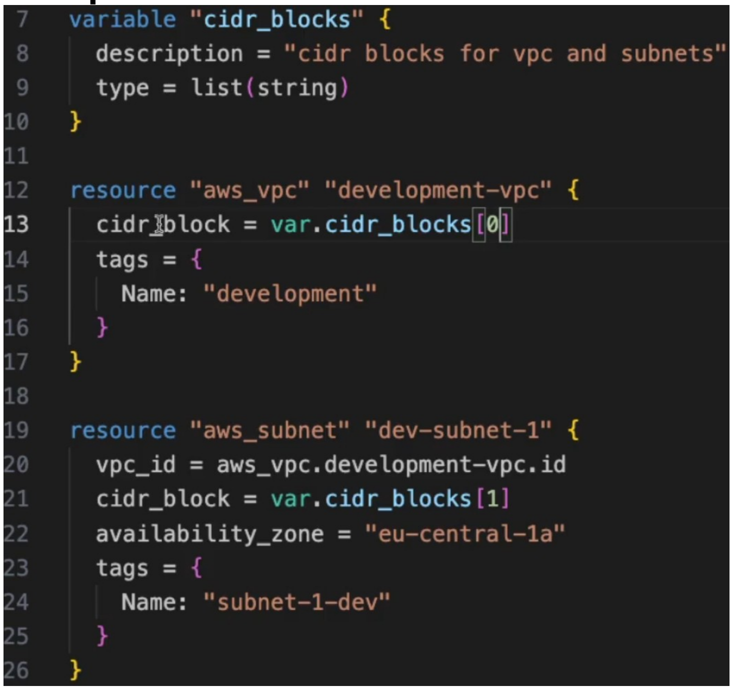
Type constraints :
U can set type attribute in variable it can be no, boolean , string



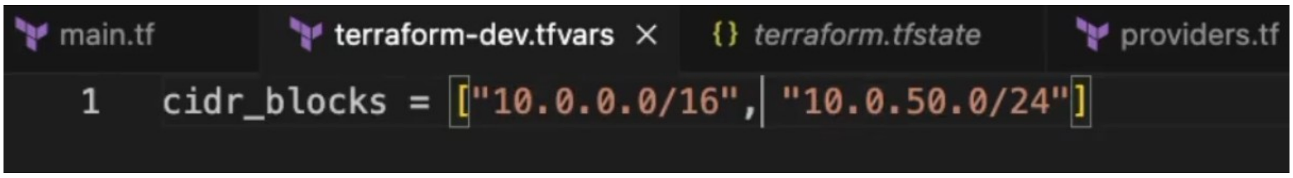
```
variable "subnet_cidr_block"{
  description = "subnet cidr block"
  default = "10.0.10.0/24"
  type = string
}
```

If u pass different type as defined in the type attribute it will throw an error . -> invalid value of input variable .

Multiple cidr-blocks =- >

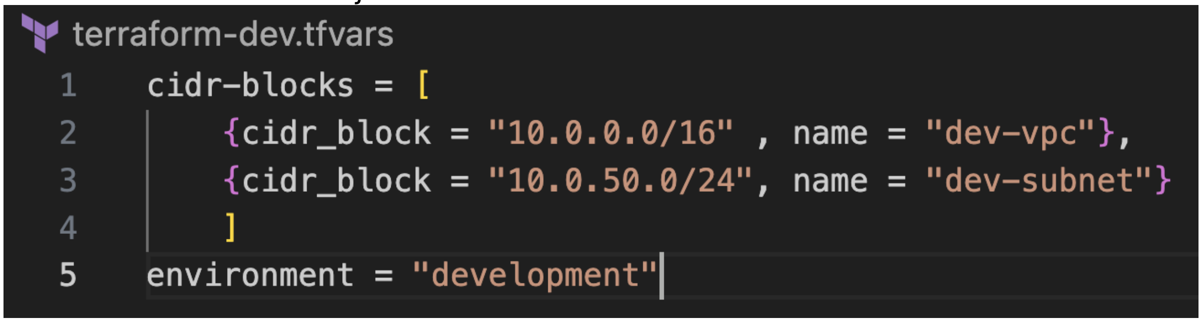


```
variable "cidr_blocks" {
  description = "cidr blocks for vpc and subnets"
  type = list(string)
}

resource "aws_vpc" "development-vpc" {
  cidr_block = var.cidr_blocks[0]
  tags = {
    Name: "development"
  }
}

resource "aws_subnet" "dev-subnet-1" {
  vpc_id = aws_vpc.development-vpc.id
  cidr_block = var.cidr_blocks[1]
  availability_zone = "eu-central-1a"
  tags = {
    Name: "subnet-1-dev"
  }
}
```



```
cidr_blocks = ["10.0.0.0/16", "10.0.50.0/24"]
```

We can have a list of objects as well =>



```
cidr-blocks = [
    {cidr_block = "10.0.0.0/16" , name = "dev-vpc"},
    {cidr_block = "10.0.50.0/24", name = "dev-subnet"}
    ]
environment = "development"
```

Environment variables :

writing credentials in the env variable files is not a good practice .

So do not hardCode the credentials .

1. Export the env variables :
Export AWS_SECRET_ACCESS_KEY= in terminal terraform directory .
Export AWS_ACESS_KEY_ID=

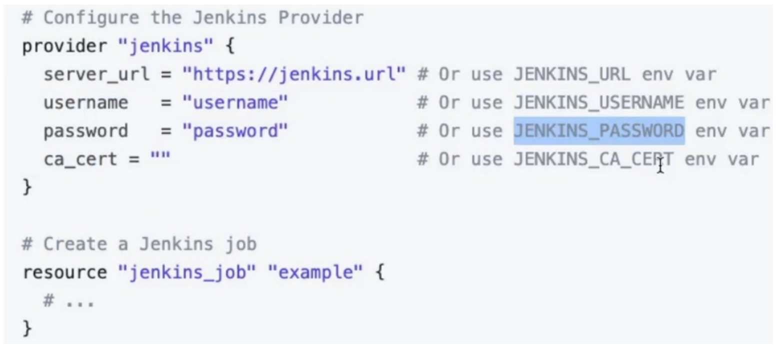
But if I open another terminal tab : I will not be able to access : env | grep AWS

: so if we want to configure the credential in the global scope -> we need to configure them in the

: in home directory => ls ~/.aws/credentails // this is the default location to store AWS credentials .

And if u have AWS CLI installed in your machine u can configure credentials using command : AWS configure

Terraform documentation will list u what are the necessary ENV name to setup ->



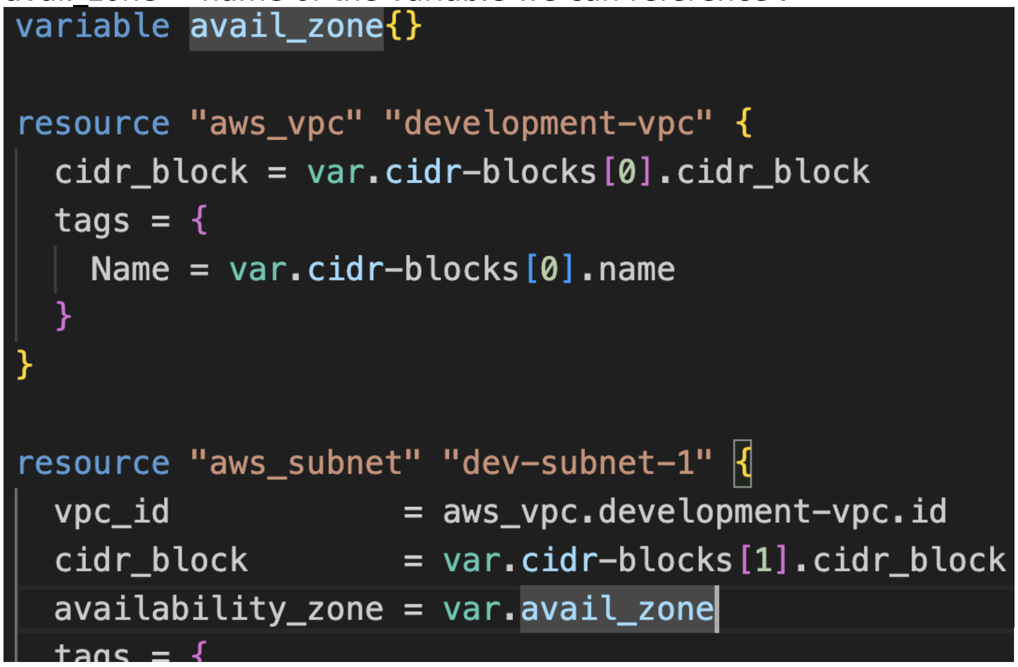
```
# Configure the Jenkins Provider
provider "jenkins" {
  server_url = "https://jenkins.url" # Or use JENKINS_URL env var
  username   = "username"           # Or use JENKINS_USERNAME env var
  password   = "password"           # Or use JENKINS_PASSWORD env var
  ca_cert = ""                      # Or use JENKINS_CA_CERT env var
}

# Create a Jenkins job
resource "jenkins_job" "example" {
  # ...
}
```

Defining your own costume ENV :
:export TF_VAR_avail_zone=“**ap-south-1a”**

There should no space around = sign when setting up env .

TF_VAR = tells the terraform that its a global variable .

avail_zone = name of the variable we can reference .



```
variable avail_zone{}

resource "aws_vpc" "development-vpc" {
  cidr_block = var.cidr-blocks[0].cidr_block
  tags = {
    Name = var.cidr-blocks[0].name
  }
}

resource "aws_subnet" "dev-subnet-1" {
  vpc_id            = aws_vpc.development-vpc.id
  cidr_block        = var.cidr-blocks[1].cidr_block
  availability_zone = var.avail_zone
  tags = {
```

(base) vishaldwivedi@vishals-MacBook-Air-2 terraform % export TF_VAR_avail_zone="ap-south-1b"

an reference .
Dev-subnet-1 will be moved from ap-south-1a to ap-south-1b

Creating a remote git repo :

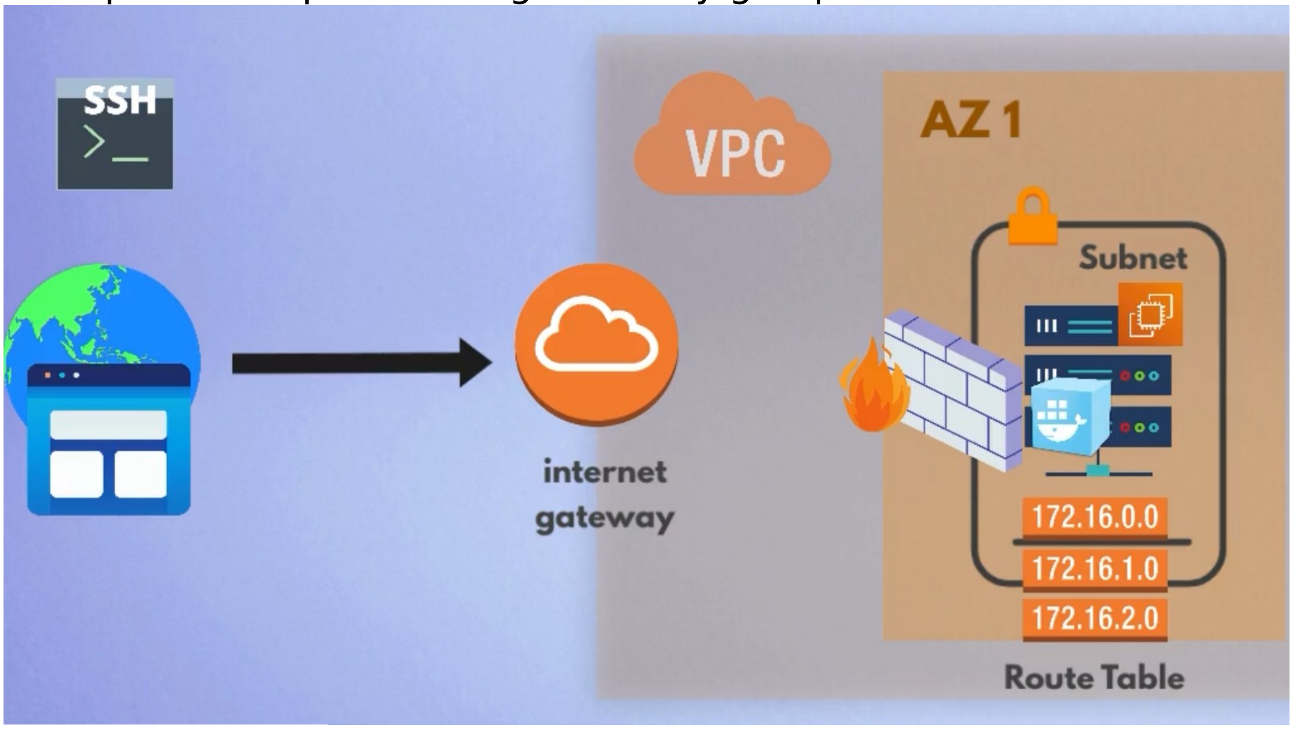
We don’t want .terraform file our git repo , also tfstate files , tfvars files.

Project Automating AWS infrastructure :

What we have leaned till now => Terraform configuration file , terrafrom syntax
Resources , data source , variables .

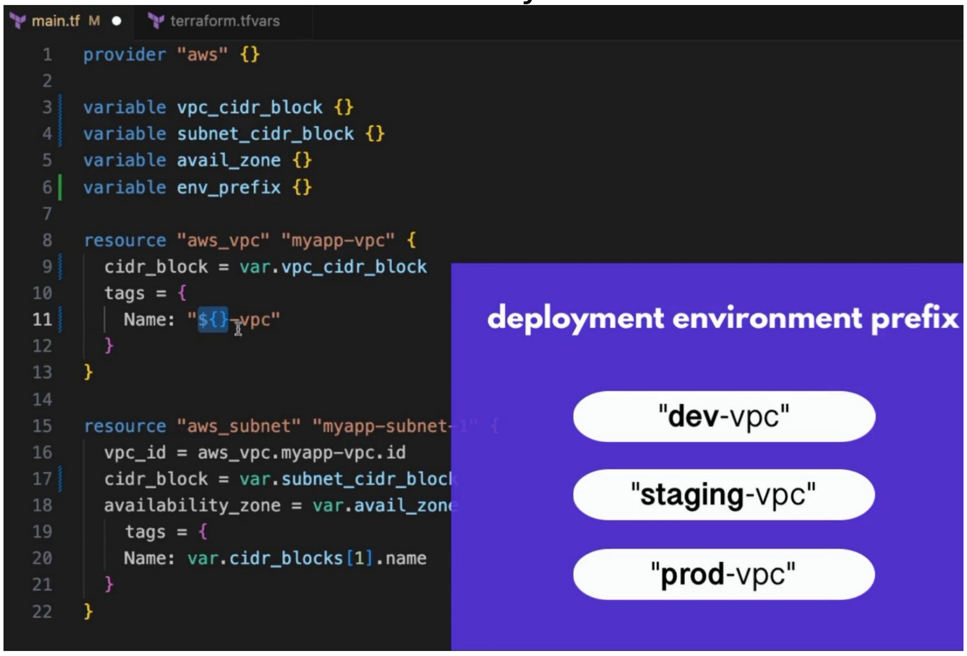
Task : provision an EC2 instance on AWS infrastructure .
Run nginx docker container on EC2 instance .

1. Creating a custom VPC
2. Create custom subnet in one of the Availability zone
3. Create a internet gateway and route table
4. We will allow traffic to and from the VPC with internet
5. And inside this VPC we will deploy our EC2 instance
6. This ec2 instance will run nginx docker container
7. Create a security group (firewall)
8. Also ssh to our server so we need to open a ssh port on the server as well .
9 . Open those ports through security groups



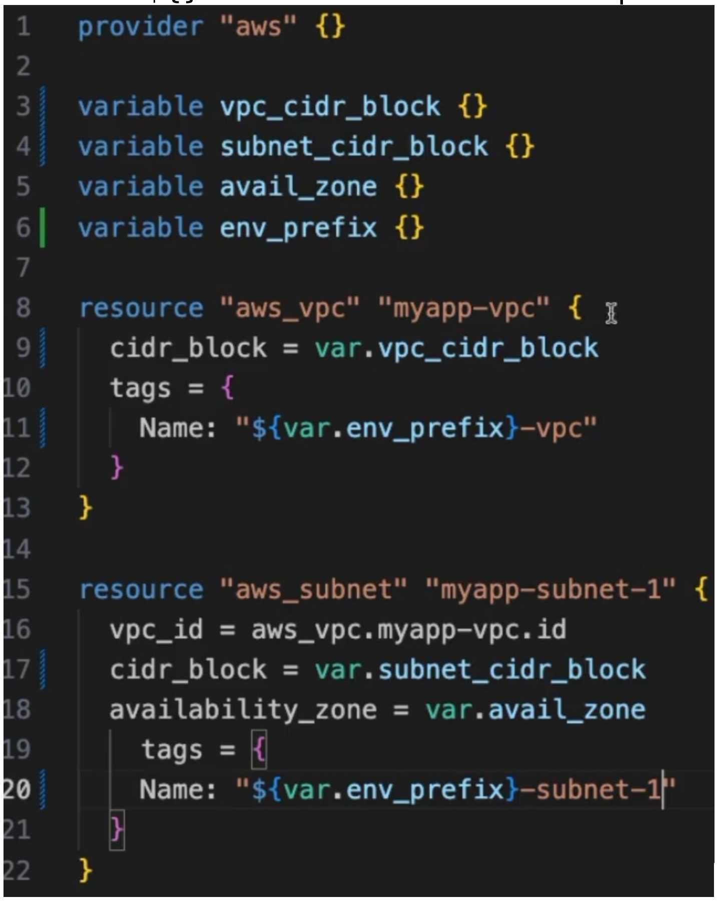
Best practice is :
To create a infrastructure from the scratch

Leave the default created by AWS

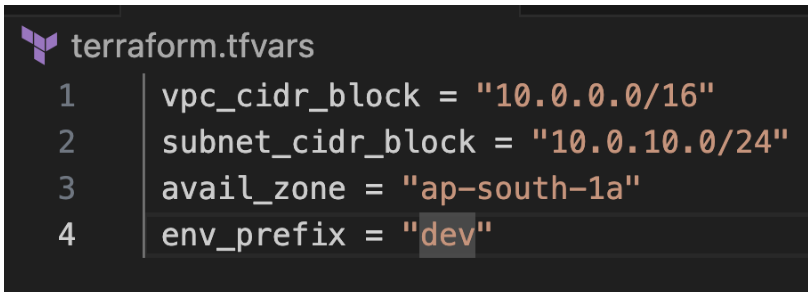


```
provider "aws" {}

variable vpc_cidr_block {}
variable subnet_cidr_block {}
variable avail_zone {}
variable env_prefix {}

resource "aws_vpc" "myapp-vpc" {
  cidr_block = var.vpc_cidr_block
  tags = {
    Name: "${}-vpc"
  }
}

resource "aws_subnet" "myapp-subnet-1" {
  vpc_id = aws_vpc.myapp-vpc.id
  cidr_block = var.subnet_cidr_bloc
  availability_zone = var.avail_zon
    tags = {
    Name: var.cidr_blocks[1].name
  }
}
```

Use = > ${} so that we can use the env-prefix



```
provider "aws" {}

variable vpc_cidr_block {}
variable subnet_cidr_block {}
variable avail_zone {}
variable env_prefix {}

resource "aws_vpc" "myapp-vpc" {
  cidr_block = var.vpc_cidr_block
  tags = {
    Name: "${var.env_prefix}-vpc"
  }
}

resource "aws_subnet" "myapp-subnet-1" {
  vpc_id = aws_vpc.myapp-vpc.id
  cidr_block = var.subnet_cidr_block
  availability_zone = var.avail_zone
   tags = {
    Name: "${var.env_prefix}-subnet-1"
  }
}
```


terraform.tfvars

```
vpc_cidr_block = "10.0.0.0/16"
subnet_cidr_block = "10.0.10.0/24"
avail_zone = "ap-south-1a"
env_prefix = "dev"
```

: terraform plan to see what will terraform will be executing .

; terraform apply —auto-approve

Note : two resources have been created -> custom vpc and custom subnet .

Note : by default a VPC comes with 3 subnet 1 per Availability zone . 2 AZ in my region .

Next one : **is route table and internet gateway .**

By default in your VPC there is a route table created in your private network .
Virtual router in your VPC

Also by default a Network ACL is also there in VPC , which is a firewall configuration for subnets .

** just like fire wall configuration in server level there is security groups same is for the subnet level security network ACL .

NACL also has inbound and outbound rules .
NACL is open by default for all the traffic .
But the security group is closed by default .

Route table = routes the traffic inside the VPC inside the vpc there are multiple EC2 instance so route table is used to send traffic to the required EC2 instance .

The route table provided by our created VPC will have the same destination cidr_block as our VPC ie : 10.0.0.0/16
, local taget means which is handled within the VPC.

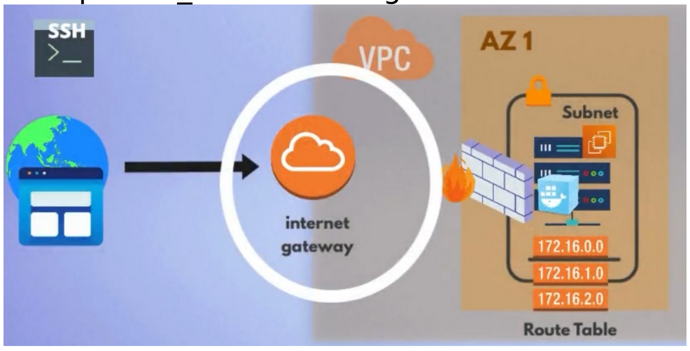
All the traffic is inside our VPC nothing comes inside or outside till now .

NOTE : but if we see the default VPC route table it has both destination , vpc cidr block as well as the whole global ip address ie: 0.0.0.0/0 all the ip addresses in the world .
And target is set as the internet gate way .

That means that this default vpc route table will handle all the traffic coming from the Internet and leaving the internet VPC .

That means that we want the same entry of route table inside our VPC and we are gonna do that using terraform .

->
Steps 1 : first we need to create a new route table , in the new route table by default the vpc cidr_block will be given as destination . So no need to define it .



We are creating igw in the vpc we are using that igw using our routable to tell the route table to handle the req of the vpc to go to and come in from the Internet .

NOTE: internet gateway resource needs to be created first then routetabel resource as we are using internet gateway in route-table .

Terraform is smart enough to know which resource needs to be created first .

So u can leave this like that only

Internet gateway is like a virtual model which will allow your Internet access or connect u to internet
Route table is like a virtual router inside your VPC .

: terraform plan // to see what will happen before terraform apply
:terraform apply —auto-approve // to apply the changes and confirm .

In AWS u will see the route table created is not main by default .

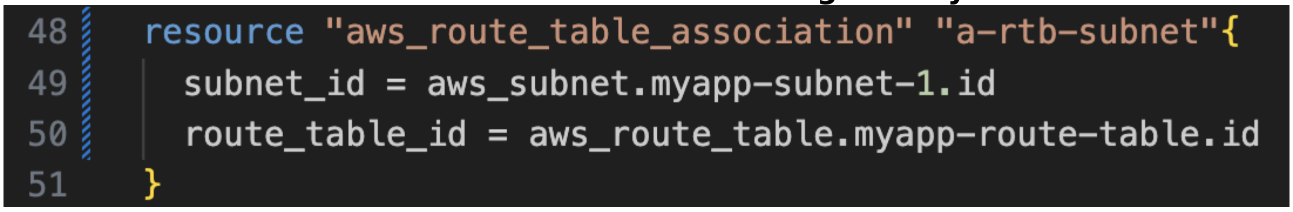
Now we have created route table and internet gateway .
Now are VPC is configured to allow internet traffic.

Subnet Association with Route table . =>

We have created a route table inside our VPC , however we need to associate subnets with our route table . So that the traffic within the subnet can also be handles by the route table.

Subnets get automatically assigned to the main route table .
We want the subnet to be associated with the route table that we have created .
Because this is the one that has Internet gate way configured .
And we want out subnet to be connected to the internet gate way as well .

NOTE: 2 route table created 1 default (main) and 1 custom in VPC . 1 custom route table has internet gateway and default vpc traffic configured . Now we want our subnet to be connected to the internet gate way as well .



```
resource "aws_route_table_association" "a-rtb-subnet"{
  subnet_id = aws_subnet.myapp-subnet-1.id
  route_table_id = aws_route_table.myapp-route-table.id
}
```

Now we have explicit subnet association with our route table .

EC2 resource that we are gonna deploy on our subnet of that request will be handled by the route table.

Use the main route table .

What if we use the default route table instead of are dev-rtb .=??

1. Go to the main.tf config file and remote the ->

```
resource "aws_route_table_association" "a-rtb-subnet"{
 subnet_id = aws_subnet.myapp-subnet-1.id
 route_table_id = aws_route_table.myapp-route-table.id
}
```

And comment the aws_route_table resource .->
And do : terraform apply

Now we will be having a default route table => now we need to configure the this default route table.

:terraform state show aws_vpc.myapp-vpc = it will show all the attribute that this resource has .
How ever that resource has to be listed on your aws account .

```
resource "aws_default_route_table" "main_rtb"{
 default_route_table_id = aws_vpc.myapp-vpc.default_route_table_id
}
```

```
default_route_table_id = aws_vpc.myapp-vpc.default_route_table_id
```

This is how to can access the route table in our VPC

: terraform apply

We don't need to apply subnet association here as by default the main route-table of default route table has associated the myapp-subnet-1 without explicit association .
So we don't need to explicitly association again because it will happen by default .

NOTE: this is how u create a default resource here default route table instead of creating a new one .

Next step is security groups :

So when we deploy our application to a subnet we need to ssh into it so for that we need to open the port no. 22 as well as we want to access ngnix on port 8080 as a container through the webbrowser
So we will go to the security group and open these two ports on our server .

Inorder to create security group =>

```
// in order to create security group
resource "aws_security_group" "myapp-sg"{
 name = "myapp-sg"
 // we need to associate the security gropu with the vpc so that the server
 //inside the vpc can be associated with the security group
 vpc_id = aws_vpc.myapp-vpc.id
 //now definign the firewall rules in the security group
 // two type of request 1. inbound , 2.outbound traffic
 // inbound = ssh into EC2 ,access from the browser
 ingress{//inbound
   from_port = 22 // range of ports u want to open from -:> to
   to_port = 22
   protocol = "TCP"
   cidr_blocks = [var.my_ip] //list of ip address that are allowed to access port
defined above
   // who is allowed to access resources in port 22 .
   // mention your laptop ipaddress
}
 ingress{
   from_port = 8080
   to_port = 8080
   protocol = "TCP"
   cidr_blocks = ["0.0.0.0/0"]//we want any one to access this from the browser
  // any ip address can access that server on port 8080
}

 //exiting traffic rule .outgoing traffic
 egress{// installation or fetching docker images .
   // request going out of out private server
   //allow that request to leave the vpc and we dont want to restrict it on any
port

   from_port = 0
   to_port = 0
   protocol = "-1"
   cidr_blocks = ["0.0.0.0/0"]
   prefix_list_ids = []//allowing VPc end points
}
 tags = {// naming our resouces for better understand of whole does this
belong
     Name : "${var.env_prefix}-sg"
   }
}
```

This task of creating the security group is also done !!!

When u go to the AWS - security group => u will see 1 security group that is default sg for that region vpc , 2nd one the default sg of the created VPC , and third one is our custom SG .
When ever we create a new VPC a default security group is created .

Yes u can use the default VPC security group ->

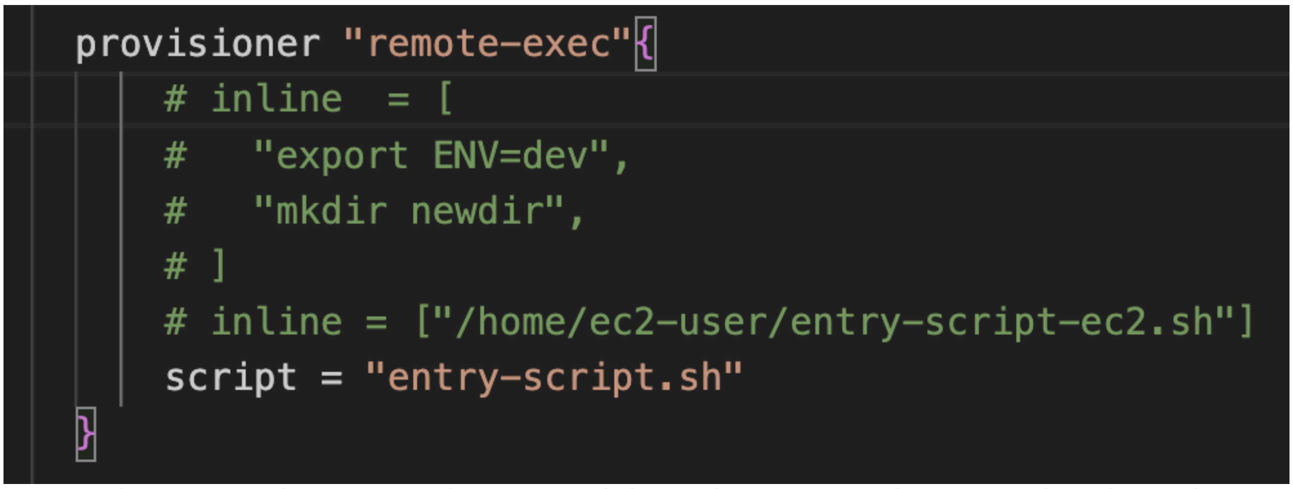
Using default security group ->

```
resource “aws_default_security_group" "default-sg"{
```

```
// we need to associate the security gropu with the vpc so that the server
//inside the vpc can be associated with the security group
vpc_id = aws_vpc.myapp-vpc.id
//now definign the firewall rules in the security group
// two type of request 1. inbound , 2.outbound traffic
// inbound = ssh into EC2 ,access from the browser
ingress{//inbound
  from_port = 22 // range of ports u want to open from -:> to
  to_port = 22
  protocol = "TCP"
  cidr_blocks = [var.my_ip] //list of ip address that are allowed to access port
defined above
  // who is allowed to access resources in port 22 .
  // mention your laptop ipaddress
}
ingress{
  from_port = 8080
  to_port = 8080
  protocol = "TCP"
  cidr_blocks = ["0.0.0.0/0"]//we want any one to access this from the browser
 // any ip address can access that server on port 8080
}

//exiting traffic rule .outgoing traffic
egress{// installation or fetching docker images .
  // request going out of out private server
  //allow that request to leave the vpc and we dont want to restrict it on any
port

  from_port = 0
  to_port = 0
  protocol = "-1"
  cidr_blocks = ["0.0.0.0/0"]
  prefix_list_ids = []//allowing VPc end points
}
tags = {// naming our resouces for better understand of whole does this
belong
    Name : "${var.env_prefix}-sg"
  }
}
```

1. till now we have a vpc

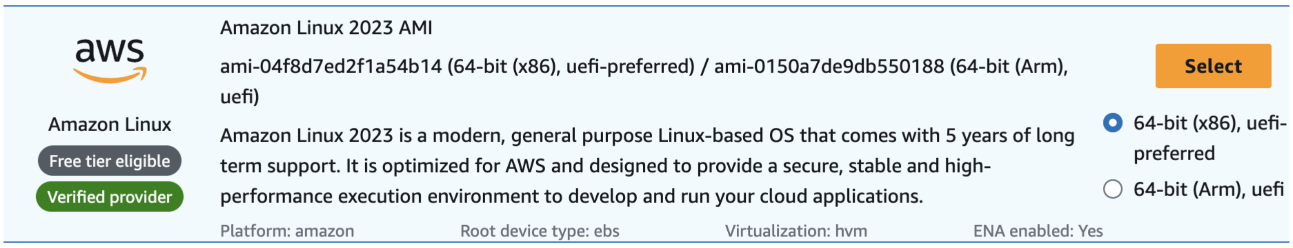
2. That has A subnet inside
3. We have connect the vpc thourhg IG and configure it with route table



```
provisioner "remote-exec"{
    # inline  = [
    #   "export ENV=dev",
    #   "mkdir newdir",
    # ]
    # inline = ["/home/ec2-user/entry-script-ec2.sh"]
    script = "entry-script.sh"
}
```

4. We have security groups that opens inbound port 22 and 8080 and outbound all traffic

Now we need to configure a resource which will create a EC2 instance in our AWS



Amazon Linux 2023 AMI

ami-04f8d7ed2f1a54b14 (64-bit (x86), uefi-preferred) / ami-0150a7de9db550188 (64-bit (Arm), uefi)

Amazon Linux 2023 is a modern, general purpose Linux-based OS that comes with 5 years of long term support. It is optimized for AWS and designed to provide a secure, stable and high-performance execution environment to develop and run your cloud applications.

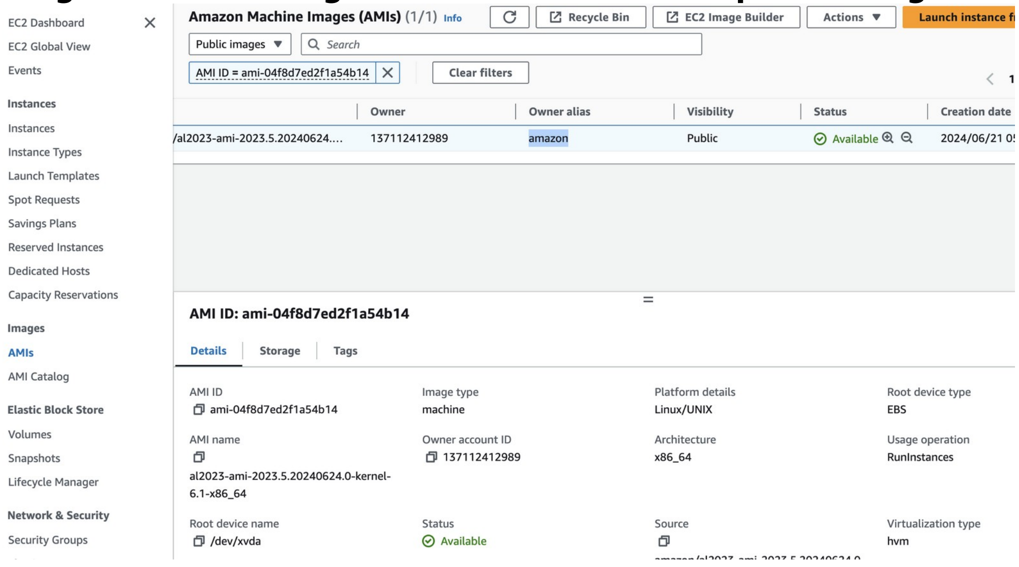
Platform: amazon   Root device type: ebs   Virtualization: hvm   ENA enabled: Yes

Go to the Amazon Machine Image that the attribute -> each AMI has a id

```
resource "aws_instance" "myapp-server"{
 ami = //is the image whcih the ec2 server will be based on or OS image

}
```

If we directly hard coad the id of image -> if may change in the future .so **set AMI dynamically**

So go to the -> images -> AMI -> search for public images .



Amazon Machine Images (AMIs) (1/1) Info

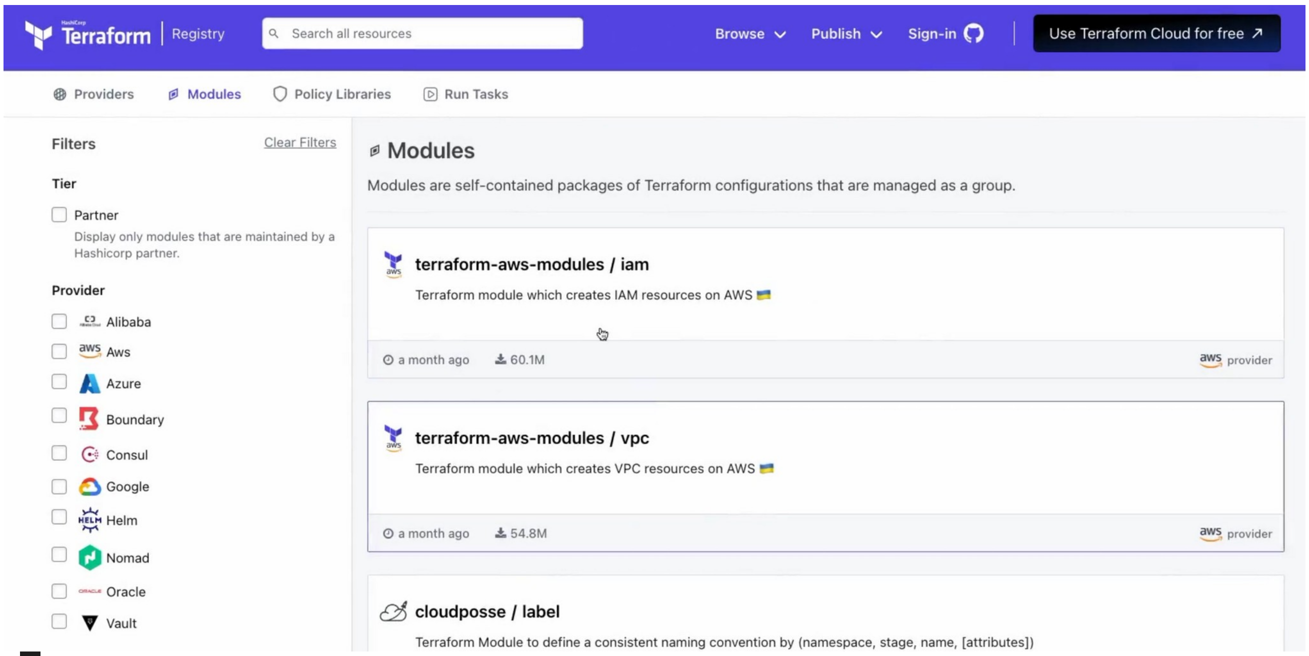
AMI ID = ami-04f8d7ed2f1a54b14

	Owner	Owner alias	Visibility	Status	Creation date
/al2023-ami-2023.5.20240624....	137112412989	amazon	Public	Available	2024/06/21 0

AMI ID: ami-04f8d7ed2f1a54b14

AMI ID ami-04f8d7ed2f1a54b14	Image type machine	Platform details Linux/UNIX	Root device type EBS
AMI name al2023-ami-2023.5.20240624.0-kernel-6.1-x86_64	Owner account ID 137112412989	Architecture x86_64	Usage operation RunInstances
Root device name /dev/xvda	Status Available	Source	Virtualization type hvm

```
data "aws_ami" "latest-amazon-linux-image"{
 most_recent = true
 owners = ["amazon"]
 filter {
   name = "name"//
   values = ["amzn2-ami-kernel-*-x86_64-gp2"]// for name what values we want
to filter on .
 }// this will give us all the images which matches this regular expression .
 //and give me the most recent of those .
 filter{
   name = "virtulization-type"
   values = ["hvm"]
 }// this way we dont need to worry about hardcoding the value of what image i
need
```



Display only modules that are maintained by a Hashicorp partner.

Modules

Modules are self-contained packages of Terraform configurations that are managed as a group.

terraform-aws-modules / iam

Terraform module which creates IAM resources on AWS

a month ago 60.1M aws provider

terraform-aws-modules / vpc

Terraform module which creates VPC resources on AWS

a month ago 54.8M aws provider

cloudposse / label

Terraform Module to define a consistent naming convention by (namespace, stage, name, [attributes])

```
}
resource "aws_instance" "myapp-server"{
 ami = data.aws_ami.latest-amazon-linux-image.id//is the image whcih the ec2
server will be based on or OS image
}

output "aws_ami_id"{// we can see the output of aws_ami // use terraform
plan= to see the image id
 value = data.aws_ami.latest-amazon-linux-image.id
instance_type = var.instance_type
/ first we have subnetid
 // we want our ec2 endup in our subnet
 subnet_id = aws_subnet.myapp-subnet-1

}
```

Create EC2 instance :

Subnet id , security group are optional if we do not specify them explicitly this ec2 instance will be launched in default VPC in one of the Availability zone in one of the subnets .

: create a private - public key pair => best practice is to store that .pem file to your .ssh folder
: mv ~/Downloads/terraform-server-key-pair.pem ~/.ssh/
: ls ~/.ssh/terraform-server-key-pair.pem
This is a secure place in your machine to store the private keys .

Now restrict permission on .pem file

:ls -l ~/.ssh/terraform-server-key-pair.pem
chmod 400 ~/.ssh/terraform-server-key-pair.pem // only the user has the read permission
This above option is must to save your pem file .

AWS rejects ssh request if the permission are not set correctly .

Yes created !!!!

// now ssh into the created instance ; ==
: ssh -I ~/.ssh/terraform-server-key-pair.pem ec2-user@publicIpaddress
// we have provided the private key
: ec2-user is the default user for amazon linux images .

Note : here I was not able to log in to my instance by ssh , I have given the wrong ip address inbound traffic of my pc to ssh port 22 ;)

Automate ssh key pair
Creating ssh key pair for the server is not optimal if we have to create is all the time.
Now we need to automate this using terraform .

public_key = //a key pair must already exist locally on our machine
: ls ~/.ssh/id_rsa // this is a private key
: ls ~/.ssh/id_rsa.pub // this is a public key

If u don't have this .ssh folder => u can create it using : ssh-keygen

```
resource "aws_key_pair" "ssh-key"{
 key_name = "dev-server-key"
 public_key = //a key pair must already exist locally on our machine
}
```
Note : h we need rsa.pub as a value in public_key .

cat ~/.ssh/id_ed25519.pub
NOTE : id_rsa or id_ed.. are same id_ed is more secure and strong.

pasting the content of pub key in public_key attribute . We don't want this to be openly seen by any one every user will have there own public key .

Therefor extracting it to a variable =>

We can reference it from the aws_instance

```
 resource "aws_instance" "myapp-server" {
 ami = data.aws_ami.latest-amazon-linux-image.id
 instance_type = var.instance_type
 subnet_id=aws_subnet.myapp-subnet-1.id
 vpc_security_group_ids = [aws_security_group.myapp-sg.id]
 availability_zone = var.avail_zone

 associate_public_ip_address = true

 key_name = aws_key_pair.ssh-key.key_name

 tags = {// naming our resouces for better understand of whole does this
belong
    Name : "${var.env_prefix}-server"
   }
}

resource "aws_key_pair" "ssh-key"{
 key_name = "dev-server-key"
 public_key = file(var.my_public_key_location)//a key pair must already exist
locally on our machine
}
```

```
my_public_key_location = "/Users/vishaldwivedi/.ssh/id_ed25519.pub"
```

:terraform plan
: terraform apply --auto-approve

Now the server must be replaced so it must be recreated .

The old instance will be deleted automatically and new is running now .

: terraform state show => will print out the attribute of ec2 instance .

So now we are gonna use the private key to ssh to the server =>

```
ssh -i ~/.ssh/id_ed25519 ec2-user@13.232.190.205
```

Or we we just use -> ssh ec2-user@publicIpAddressOfInstance
As -I ~/.ssh/id_ed25519 this is a default location

Note : this is the one advantage of using your local own public private key pair .

Now we can : rm ~/.ssh/terraform_server_key_pair.pem that we have downloaded .

NOTE = automate as much as possible and do little as possible manually .

Advantage is => u forget the clean up the resources when creating new one

, now u can replicate in diff environment , replication is difficult with manual configuration ,you have to document the manual steps , collaboration is difficult .

Run entry point script to run docker container .

Now till now we have ec2 server configured , networking configured , but nothing is running on the server .

: user_data => this attribute will help us to configure any app on our server.
Entry point script that get executed in ec2 instance when server is instantiated .

```
//below adding ec2 user in docker group , port 8080 on host bind with port 80 on nginx
 // we need to ensure that our instance get destroyed and recreated when we start our instace every time.
 // for that we have documentaion aws_instance on terraform -> user_data_replace_on_change
 // this block below will only get executed once .
 user_data = <<EOF
                #!/bin/bash
                sudo yum update -y && sudo yum install -y docker
                sudo systemctl start docker
                sudo usermod -aG docker ec2-user
                docker run -p 8080:80 nginx
            EOF

 user_data_replace_on_change = true

 tags = {// naming our resouces for better understand of whole does this belong
    Name : "${var.env_prefix}-server"
   }
```

Here when we create a user data more make any changes the instance will be recreated every time when we apply .

: now u can log in to the instance and see the docker container is running : docker ps
And we have already configured security group for that so the port 8080 is now accessible from the browser .
Therefore : instance public ip address on port 8080 we can see nginx welcome page ->

What we have done in this project :

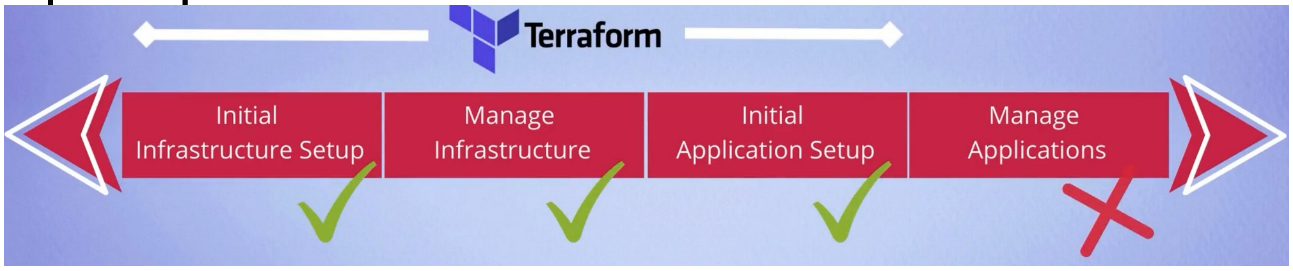
1. Creating custom VPC
2. Creating custom subnet
3. Creating route table and internet gateway
4. Provision Ec2 instance

5. Create security group
6. Deploy nginx docker container.

Extract the shell script :
If u want to execute a longer entry point script when the server starts u can reference it from a file instead of having it in a terraform configuration .

But this change will cause ec2 instance to be recreated.

Important points : =>

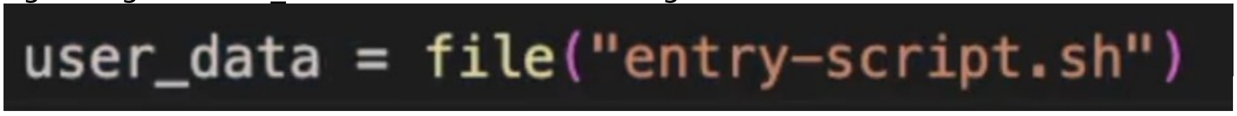


Installing packages and and configuring them once the infrastructure is managed and configured should be done with another tool .

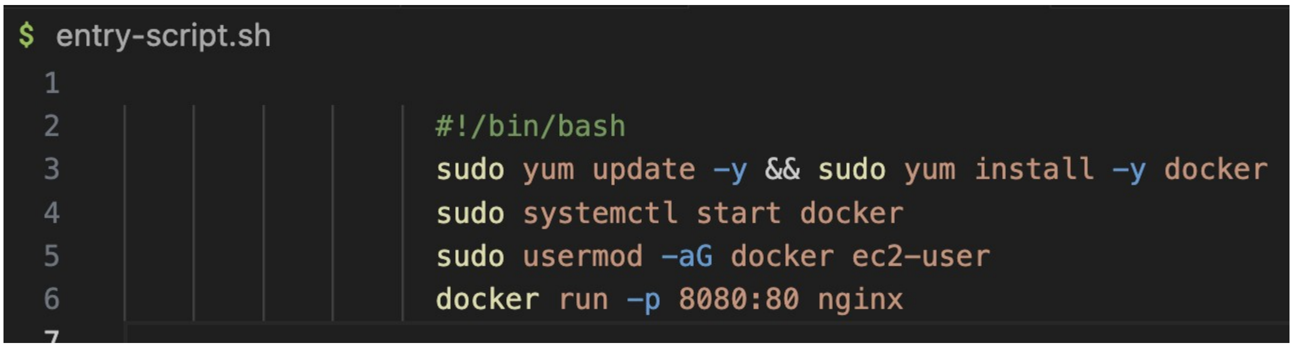
So there are another configuration management tools that are made for this tasks ->
Eg: chef , puppet , ansible . Are used for configuring server , deploying app , install/update packages . Etc ..

PROVISIONERS :

Eg: using => user_data attribute to executing cmds on virtual servers .



```
user_data = file("entry-script.sh")
```



```
$ entry-script.sh

#!/bin/bash
sudo yum update -y && sudo yum install -y docker
sudo systemctl start docker
sudo usermod -aG docker ec2-user
docker run -p 8080:80 nginx
```

This is one of the way for executing the commands on the ec2 servers . By passing in the user -data as a initial data to be executed on server .

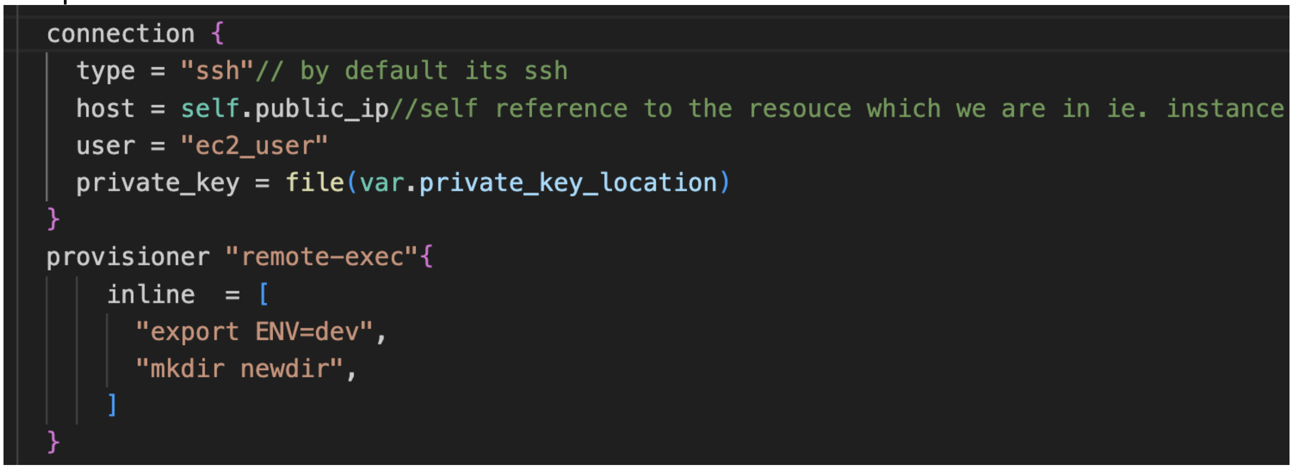
All these data will be handed over from the terraform to the cloud provider .

Even though we have configured the initial setup of the server .

Terraform has a concept of provisioners to execute the script on server.
(base) vishaldwivedi@vishals-MacBook-Air-2 terraform % git checkout -b feature/provisioners
// creating a new branch and switching it .

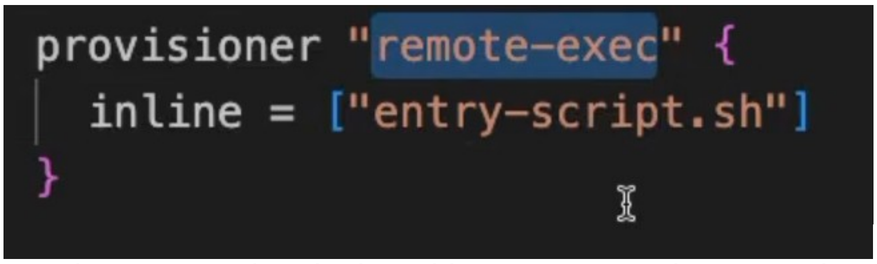
1. **: remote-exec is a provisioner =** which invokes script on a remote resource after it is created .it has a inline [// where u can execute it] attribute . Where u can list all the commands .

By default it does not connect to the remote server even though it is inside the aws instance resource and we can connect it using a **connection** attribute it is specific for the provisioners .



```
connection {
  type = "ssh"// by default its ssh
  host = self.public_ip//self reference to the resouce which we are in ie. instance
  user = "ec2_user"
  private_key = file(var.private_key_location)
}
provisioner "remote-exec"{
    inline  = [
      "export ENV=dev",
      "mkdir newdir",
    ]
}
```

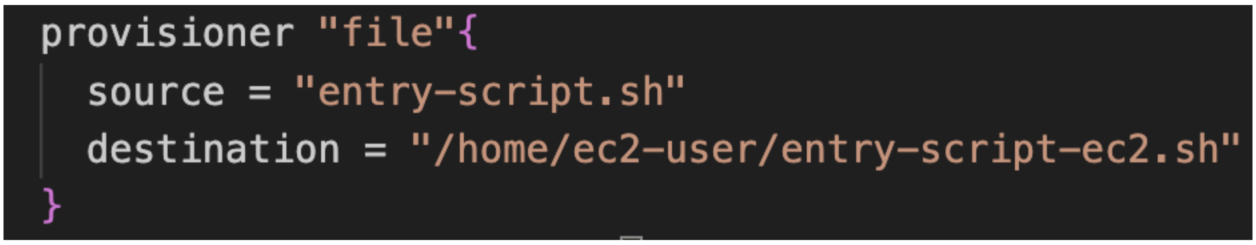
Remote exec means it execute on the remote server not in our local machine .



```
provisioner "remote-exec" {
  inline = ["entry-script.sh"]
}
```

so the below command to run the entry-script.sh should be present on the server itself.

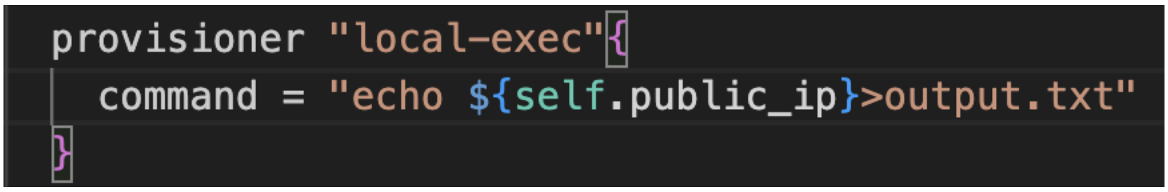
So therefore we need to copy that entry-script.sh file to the server first -=>
For that there is another provisioner > “file”
Copy Form the local machine to the remote server .



```
provisioner "file"{
  source = "entry-script.sh"
  destination = "/home/ec2-user/entry-script-ec2.sh"
}
```

Next provisioner ->
:provisioner "local-exec"

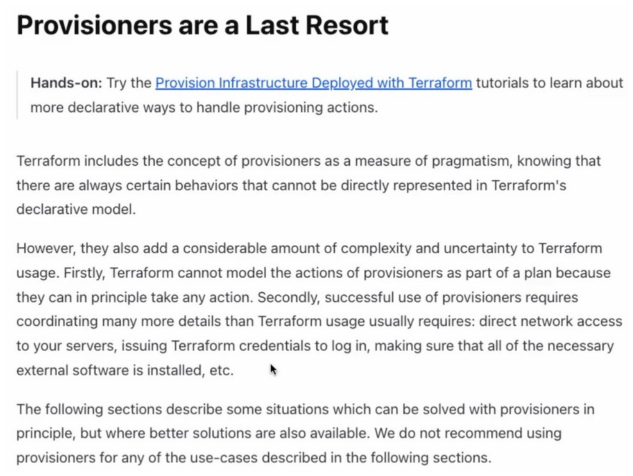
These command will be executed locally after a resource is created .



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

So these are the three type of provisioner

Note : U might thing provisioners are great but they are not recommend by terraform .



Provisioners are a Last Resort

Hands-on: Try the Provision Infrastructure Deployed with Terraform tutorials to learn about more declarative ways to handle provisioning actions.

Terraform includes the concept of provisioners as a measure of pragmatism, knowing that there are always certain behaviors that cannot be directly represented in Terraform's declarative model.

However, they also add a considerable amount of complexity and uncertainty to Terraform usage. Firstly, Terraform cannot model the actions of provisioners as part of a plan because they can in principle take any action. Secondly, successful use of provisioners requires coordinating many more details than Terraform usage usually requires: direct network access to your servers, issuing Terraform credentials to log in, making sure that all of the necessary external software is installed, etc.

The following sections describe some situations which can be solved with provisioners in principle, but where better solutions are also available. We do not recommend using provisioners for any of the use-cases described in the following sections.

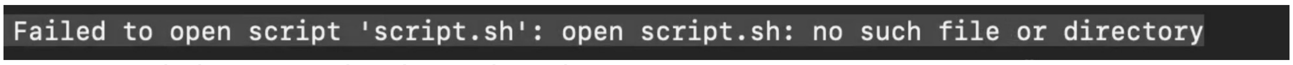
Provisioners do not work very well -> unexpected behaviour. Etc ..

Provisioner breaks the idempotency concept => that describes some thing gives u the same output no matter how many time u execute that command -> terraform does not know what are u executing if the script has executed or not . It breaks the current state desired state comparison .

Alternative to execute command remotely -> use the configuration management tool .eg: puppet , ansible , chef .

Alternative to execute local-exec -> u can use a “local” provider managed by hashicorp.which is used to handle local files on terraform.

Provisioner failure :



```
Failed to open script 'script.sh': open script.sh: no such file or directory
```

Instance will be create by the script will not .

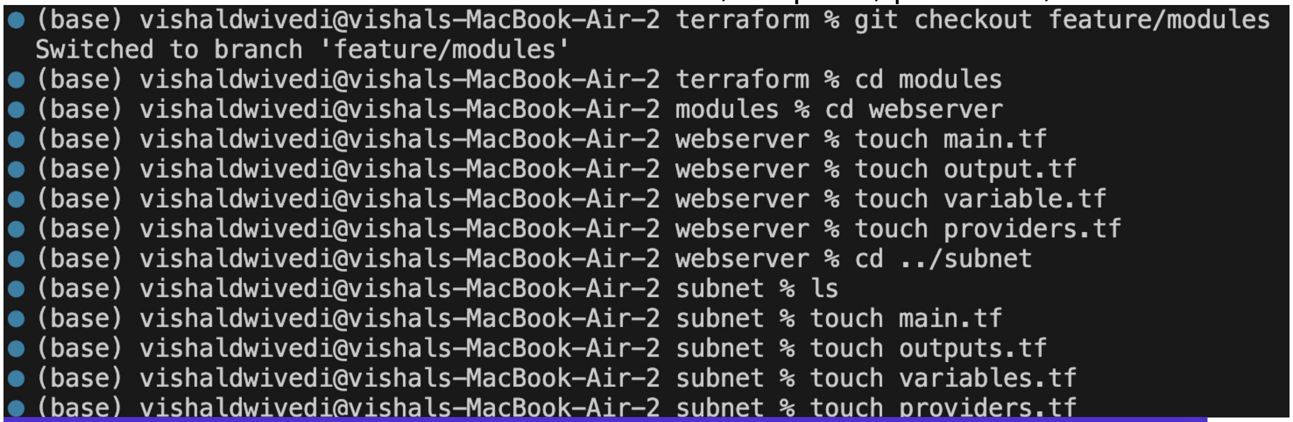
MODULES in terraform

Without modules => complex configuration , huge file, no overview
We have a monolithic file like very thing in one file . -?>
So we will -> Organize and group configuration , encapsulate into distinct logical components , Re-use easy .

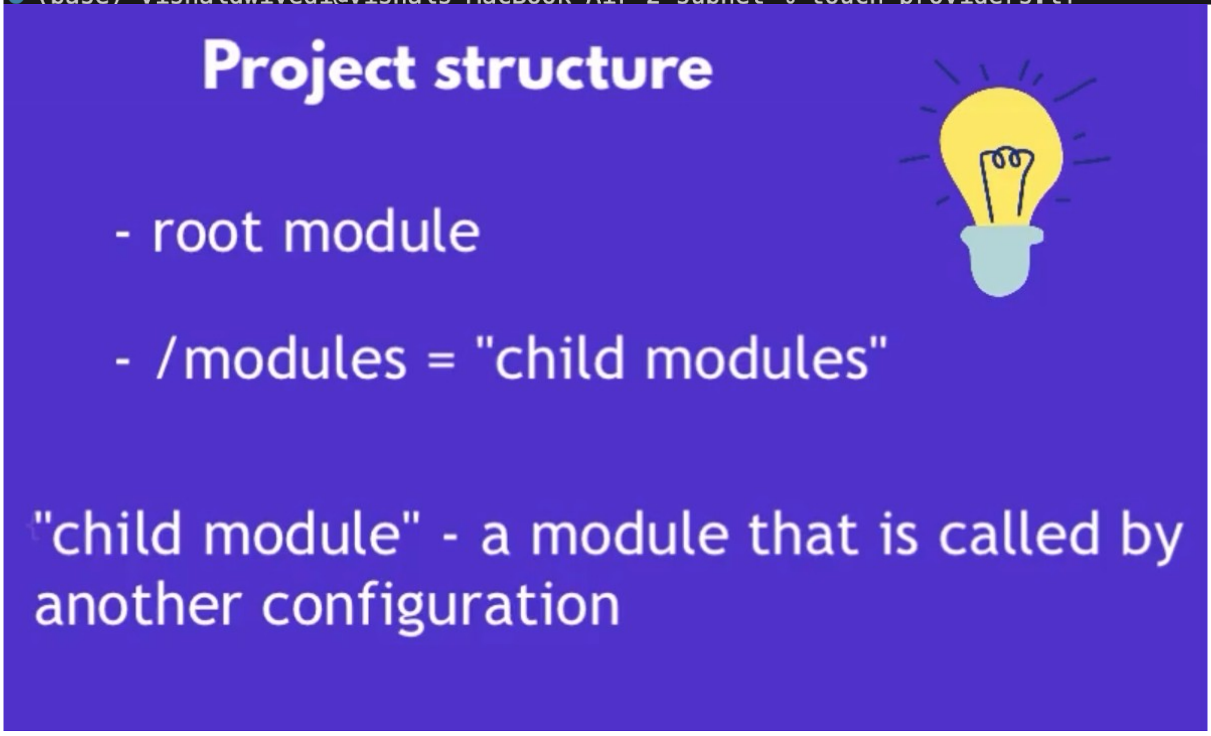
Modules are like function that we write once and can reuse it later .

There are modules provided by terraform which u can re use . =>

Creating modules folder -> inside it creating web server and subnet folder -> and such of these folder will have files main , outputs , providers, variables.tf ..



```
● (base) vishaldwivedi@vishals-MacBook-Air-2 terraform % git checkout feature/modules
  Switched to branch 'feature/modules'
● (base) vishaldwivedi@vishals-MacBook-Air-2 terraform % cd modules
● (base) vishaldwivedi@vishals-MacBook-Air-2 modules % cd webserver
● (base) vishaldwivedi@vishals-MacBook-Air-2 webserver % touch main.tf
● (base) vishaldwivedi@vishals-MacBook-Air-2 webserver % touch output.tf
● (base) vishaldwivedi@vishals-MacBook-Air-2 webserver % touch variable.tf
● (base) vishaldwivedi@vishals-MacBook-Air-2 webserver % touch providers.tf
● (base) vishaldwivedi@vishals-MacBook-Air-2 webserver % cd ../subnet
● (base) vishaldwivedi@vishals-MacBook-Air-2 subnet % ls
● (base) vishaldwivedi@vishals-MacBook-Air-2 subnet % touch main.tf
● (base) vishaldwivedi@vishals-MacBook-Air-2 subnet % touch outputs.tf
● (base) vishaldwivedi@vishals-MacBook-Air-2 subnet % touch variables.tf
● (base) vishaldwivedi@vishals-MacBook-Air-2 subnet % touch providers.tf
```



Project structure

- root module

- /modules = "child modules"

"child module" - a module that is called by another configuration

Keep in mind when ever u are creating a module it should store at least 3-4 resources .

1. Grab -> subnet , route table and internet gateway to the subnet module main.tf .

Now how do we reference the subnet module from other root configuration file . ->
Using a module key work in root configuration file .

How do u access the resource output of one module to other module . =>

Output values .=>

2. Now create a web server module . ==> we have key pair , instance , aws ami , security group which configure firewall for the instance .
Grab all of these from the root main.tf to the web server module main.tf
So we need to fix the reference to all the values here .

We have modularised our project .

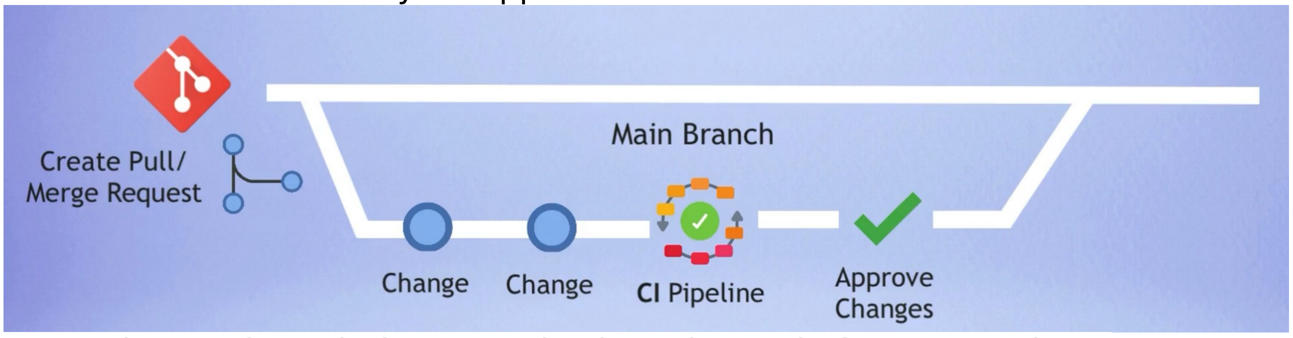
Remote state in terraform :

TERRAFORM BEST PRACTICES :

1. State in terraform is a json file which keep track of the current state and terraform compare your current state with actual state . Then make changes in the infrastructure creating a execution plan : -> **u can change this state file but only manipulate state only through TF commands do not manually manipulate it otherwise u may get some unexpected results .**
2. Where does this tfstate file comes from -> when u do terraform apply , terraform automatically create a state file locally . But if u are in team other team member also need a terraform.tfstate file and they would need the latest state file before making there own changes . **So 2nd best practice is to configure a shared remote state file** . It can be amazon s3 , azure blob storage , google cloud storage, terraform cloud.
3. But what if two team member do terraform apply at a same time -.> which may lead to conflicts and state file corruption .**so u must lock the state file**. Eg: u can configure it in your storage backend like S3 supports state locking and consistency checking via Dynamo DB .**but not all storage backend support locking : so choose the remote storage that supports state file.**
4. What happened when u loose your state file **so u need to setup backup for it and enable versioning to it eg: in s3 u can turn on the versioning feature .**
5. Usually u have multiple environment eg: testing , development , production so which environment do this state file belongs to . So best practice is to

have a separate state file for each environment and each state file will have versioning , backup in its own storage backend .

6. Just like your application code u should host terraform code in its own git repo . It is effective collaboration in a team and version control for infrastructure as a code concept.
7. How is allowed to make change is terraform code : u should maintain your terraform code same as your application code .



8. Apply change through the CICD pipeline , instead of team members manually apply ing changes from there own computes it should be only done from a automated build this way u will have only single location from where u can apply change in your infrastructure .