TERRAFORM

INTRO:

- Terraform is an open source "Infrastructure as Code" tool, created by HashiCorp.
- A *declarative* coding tool, Terraform enables developers to use a high-level configuration language called HCL (HashiCorp Configuration Language) to describe the desired "end-state" cloud or on-premises infrastructure for running an application.
- Terraform uses a simple syntax, can provision infrastructure across multiple cloud and on-premises.

IAAC:

- Infrastructure as a Code (IaaC) is the managing and provisioning of infrastructure through code instead of through manual processes.
- With IaaC, configuration files are created that contain your infrastructure specifications, which makes it easier to edit and distribute configurations.
- IaC allows you to meet the growing needs of infrastructure changes in a scalable and trackable manner.
- The infrastructure terraform could handle low-level elements like networking, storage, compute instances, also high-level elements like **SaaS features, DNS entries**, etc.
- It is famous for easy to use but not true for complex environments it is not easy.
- Terraform is not fully cloud agnostic

WHY:

- It is a server orchestration tool (chef, ansible and puppet are configuration tools).
- Declarative code
- mutable code
- Reuseable
- All cloud providers are supported
- Its free and open source.

CLOUD ALTERNATES:

- AWS -- > CloudFormation templates (JSON/YAML)
- AZURE -- > ARM TEMPLATES (JSON)
- GCP -- > GDE

ADVANTAGES:

- Readable code.
- Dry run.
- Importing of Resources is easy.
- Creating of multiple resources.
- Can create modules for repeatable code.

DIS ADVANTAGES:

- It is 3rd party tool. It takes time to accommodate new services.
- BUGS
- Need to trouble shoot.

TERRAFORM SETUP

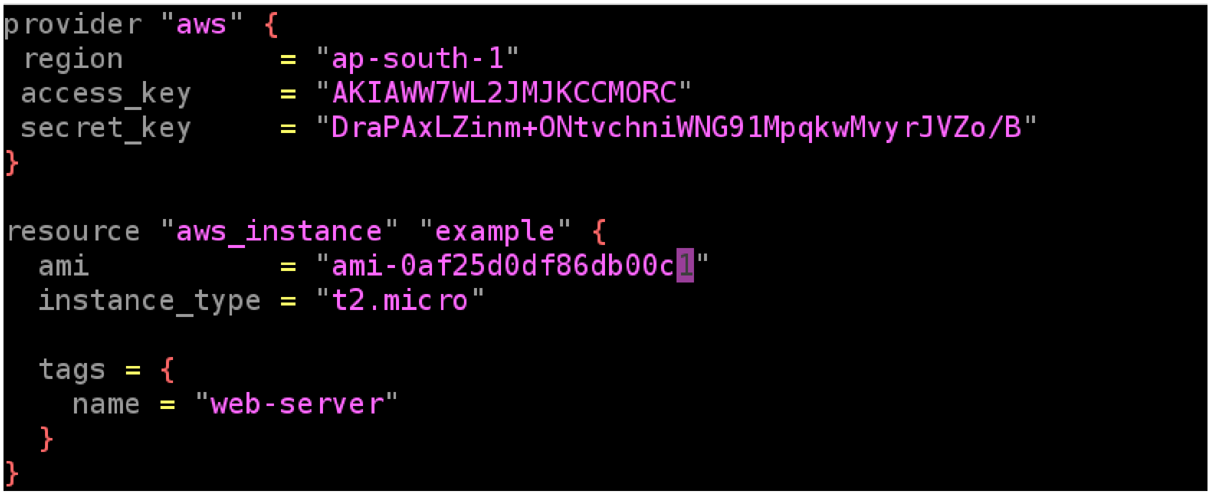
- wget https://releases.hashicorp.com/terraform/1.1.3/terraform_1.1.3_linux_amd64.zip
- sudo apt-get install zip -y
- Unzip terraform
- mv terraform /usr/local/bin/
- terraform version
- cd ~
- mkdir terraform & vim main.tf
- write the basic code
- Go to IAM andcreate a user called terraform and give both access give admin access.

CREATING AN INSTANCE

```
provider "aws" {
 region= "ap-south-1"
 access_key     = "AKIAWW7WL2JMJKCCMORC"
 secret_key     = "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"
}

resource "aws_instance" "web" {
  ami= "ami-08e4e35cccc6189f4"
  instance_type = "t2.micro"

  tags = {
    name = "web-server"
  }
}
```

Note:
We need not to give access key and secret key directly, we have some more options
1. Use -- > aws configure
2. Use -- >   shared_credentials_files = ["/.aws/credentials"] (git this inside file under region)

- terraform init : now terraform will be initialized
- Now see the hidden files you will find a terraform directory
- terraform plan : Read config file and compare local state file.
- Terraform apply:
- You will get an error think logically to get it.
- You need to give your ami-id on ap-south-1 and instance will be created there only.
- Now terraform.tfstate file will be created which consist all the metadata.
- Terraform destroy : kill the instances



```
provider "aws" {
  region          = "ap-south-1"
  access_key      = "AKIAWW7WL2JMJKCCMORC"
  secret_key      = "DraPAxLZinm+ONtvchniWNG91MpqkwMvyrJVZo/B"
}

resource "aws_instance" "example" {
  ami             = "ami-0af25d0df86db00c1"
  instance_type = "t2.micro"

  tags = {
    name = "web-server"
  }
}
```

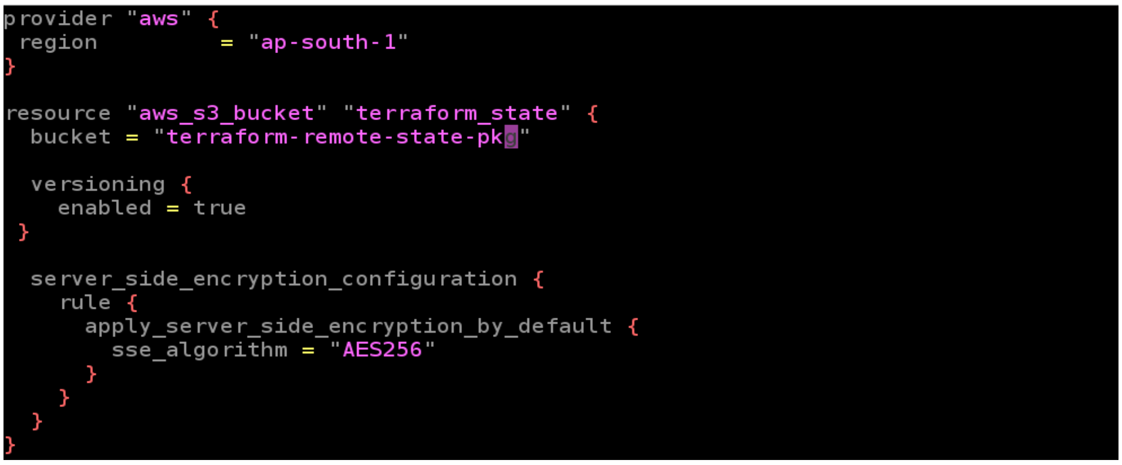
EC2-ROLE BASED AUTHENTICATION:

- By using this without access key and secret access key we can perform the actions
- IAM -- > Roles -- > Create -- > AWS Services : EC2 -- > AdministratorAccess -- > Name : EC2-Access -- > Role name : Terraform-role-base -- > Create
- Select instance -- > Actions -- > Security -- > Modify IAM Role -- > Select Role -- > Save
- Now remove both Access key and Secret Acess key and save the main.tf file.
- Terraform plan and terraform apply.
- Now the instance will be created.

S3 BACKEND SETUP FOR REMOTE STATE FILE

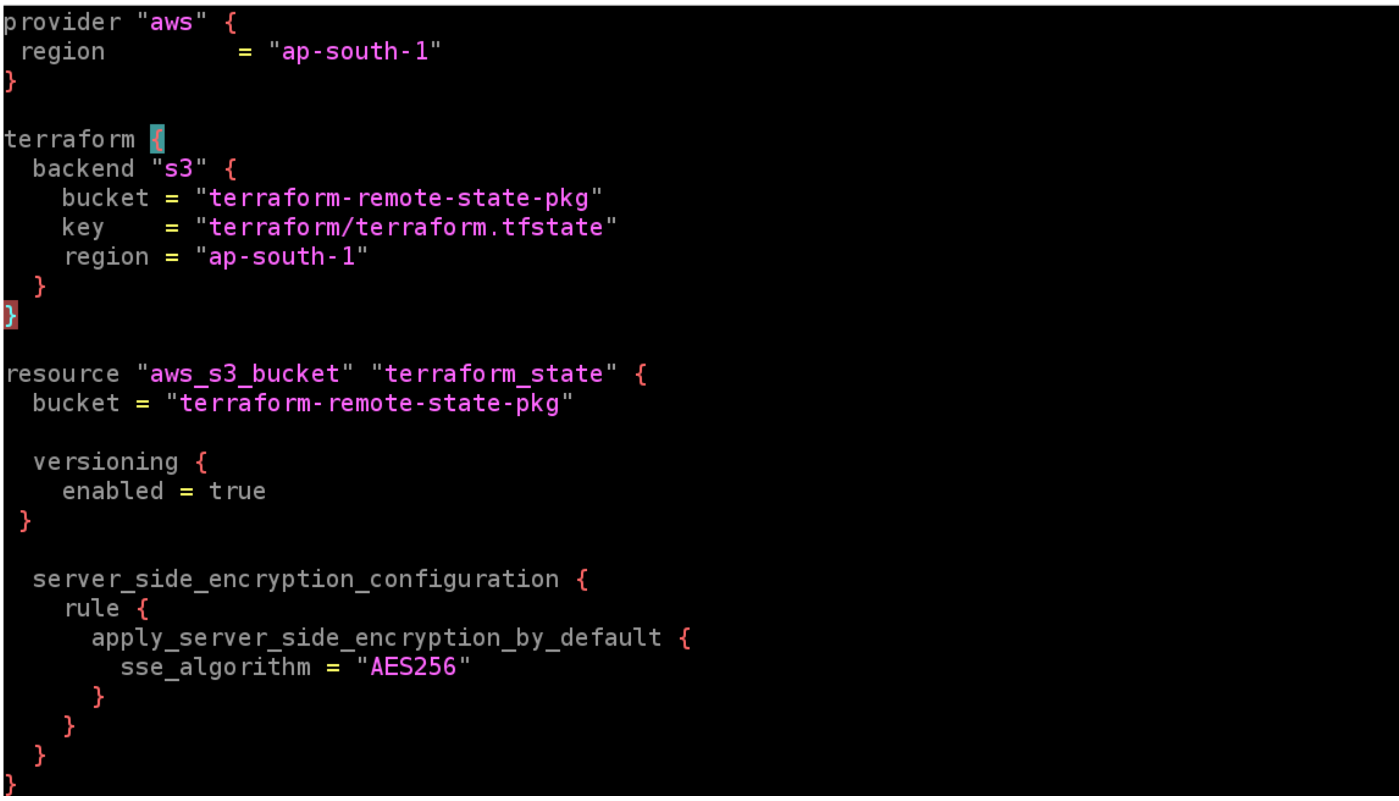
In terraform we have two state files one is local state file and another is remote state file.
We use Local state file when we there is no involvement of other person.

- Create a bucket with versioning enable
- Initialize the backend with S3 using Terraform.
- Launch the resources using terraform to validate the remote state file and Versioning.



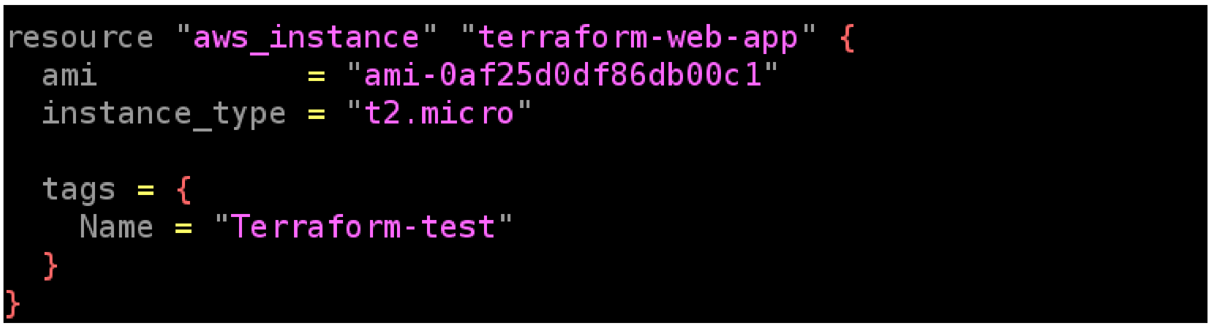
```
provider "aws" {
 region          = "ap-south-1"
}

resource "aws_s3_bucket" "terraform_state" {
  bucket = "terraform-remote-state-pkg"

  versioning {
    enabled = true
 }

  server_side_encryption_configuration {
    rule {
      apply_server_side_encryption_by_default {
        sse_algorithm = "AES256"
      }
    }
  }
}
```

- Terraform plan and terraform apply
- Now the bucket will be created on that region



```
provider "aws" {
 region          = "ap-south-1"
}

terraform {
  backend "s3" {
    bucket = "terraform-remote-state-pkg"
    key    = "terraform/terraform.tfstate"
    region = "ap-south-1"
  }
}

resource "aws_s3_bucket" "terraform_state" {
  bucket = "terraform-remote-state-pkg"

  versioning {
    enabled = true
 }

  server_side_encryption_configuration {
    rule {
      apply_server_side_encryption_by_default {
        sse_algorithm = "AES256"
      }
    }
  }
}
```

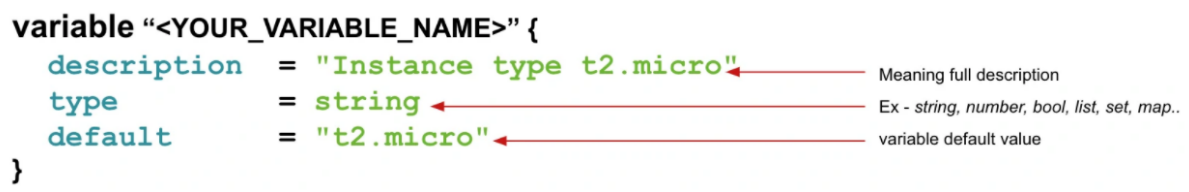
- ➢ Terraform init : It will be successful
- ➢ Now add EC2 Resource in same code.



```
resource "aws_instance" "terraform-web-app" {
  ami           = "ami-0af25d0df86db00c1"
  instance_type = "t2.micro"

  tags = {
    Name = "Terraform-test"
  }
}
```

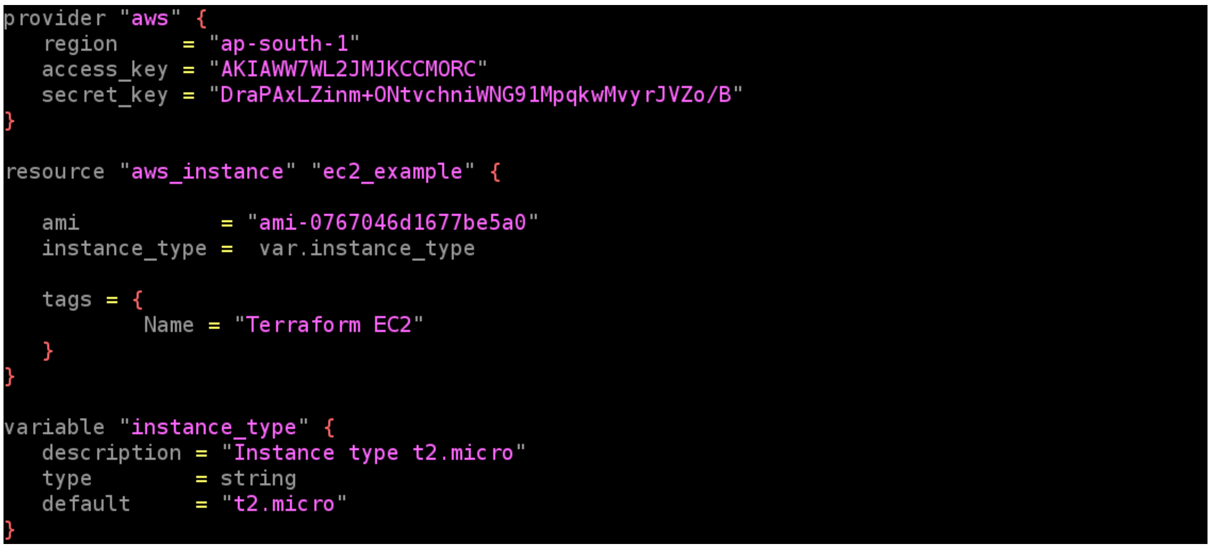
- ➢ Add this part on the end
- ➢ Terraform plan and terraform apply.
- ➢ Now verify the versioning on that bucket you can two versions and new instance will be created.
- ➢ If you give a new tag then you can see the new version (Terraform plan and apply)

TERRAFORM TYPES (VARIABLE TYPE)



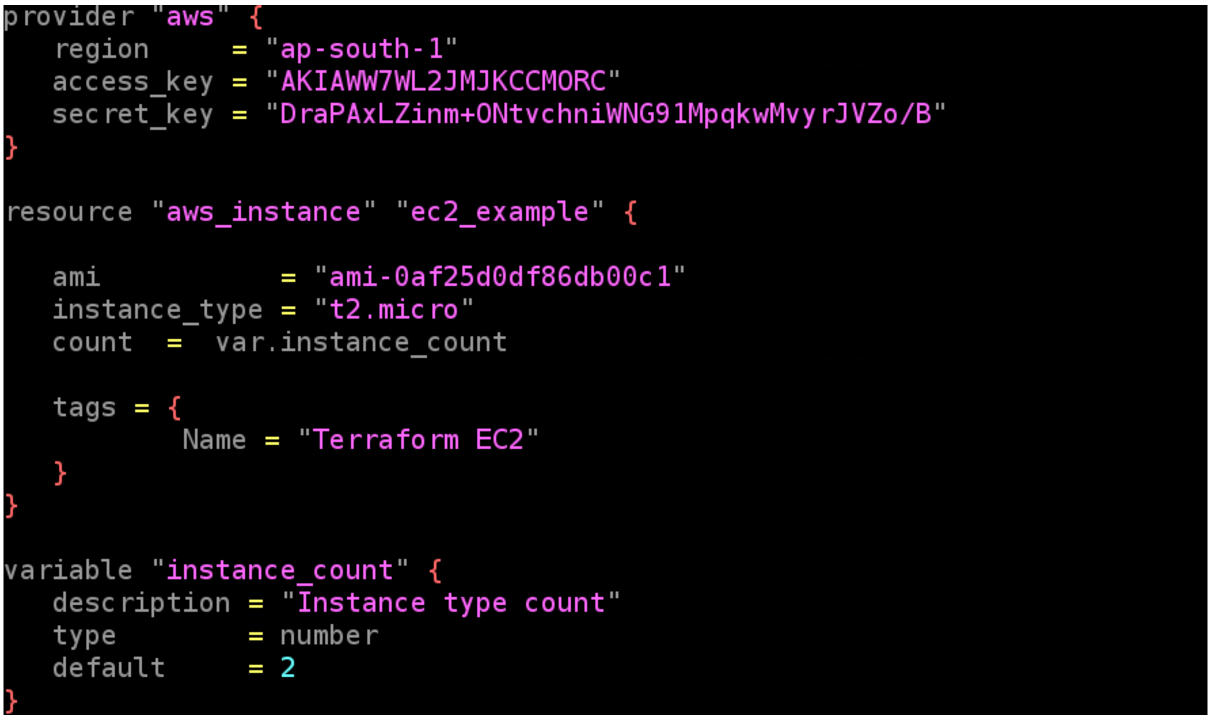
```
variable "<YOUR_VARIABLE_NAME>" {
  description   = "Instance type t2.micro"
  type          = string
  default       = "t2.micro"
}
```

- string: a sequence of Unicode characters representing some text, like "hello". (terraform init, plan, apply, destroy)



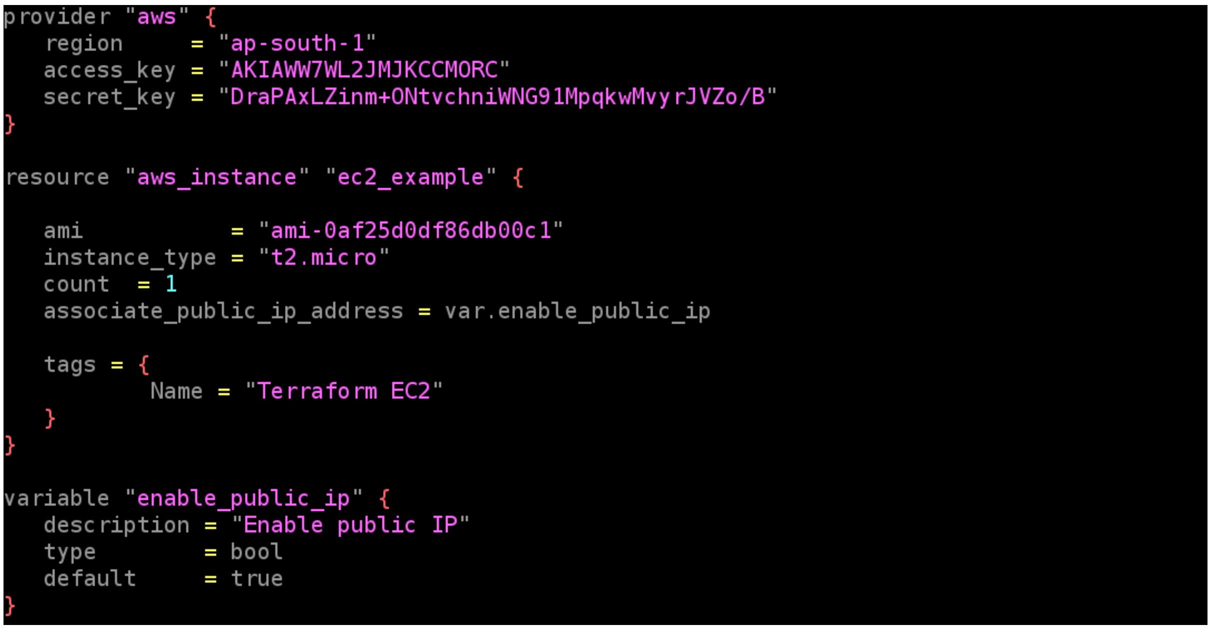
```
provider "aws" {
   region     = "ap-south-1"
   access_key = "AKIAWW7WL2JMJKCCMORC"
   secret_key = "DraPAxLZinm+0NtvchniWNG91MpqkwMvyrJVZo/B"
}

resource "aws_instance" "ec2_example" {

   ami           = "ami-0767046d1677be5a0"
   instance_type =  var.instance_type

   tags = {
           Name = "Terraform EC2"
   }
}

variable "instance_type" {
   description = "Instance type t2.micro"
   type        = string
   default     = "t2.micro"
}
```

- number: a numeric value. The number type can represent both whole numbers like 15 and fractional values like 6.283185.



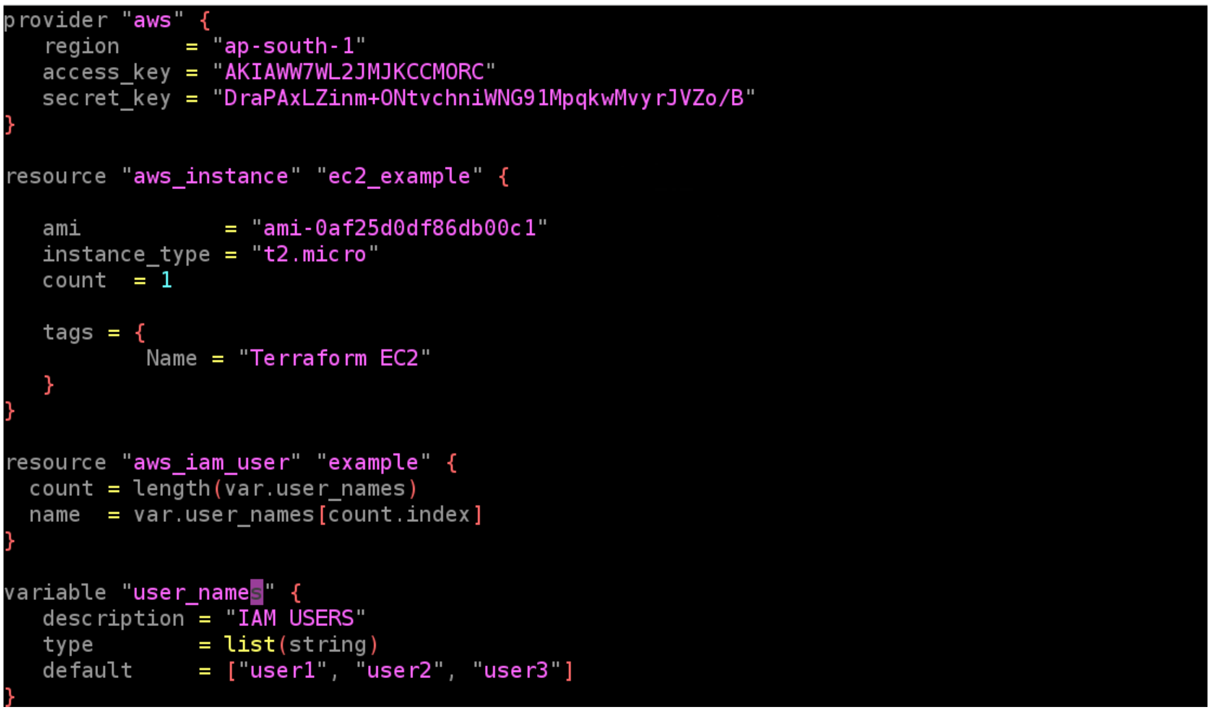
```
provider "aws" {
   region     = "ap-south-1"
   access_key = "AKIAWW7WL2JMJKCCMORC"
   secret_key = "DraPAxLZinm+0NtvchniWNG91MpqkwMvyrJVZo/B"
}

resource "aws_instance" "ec2_example" {

   ami           = "ami-0af25d0df86db00c1"
   instance_type = "t2.micro"
   count   =  var.instance_count

   tags = {
           Name = "Terraform EC2"
   }
}

variable "instance_count" {
   description = "Instance type count"
   type        = number
   default     = 2
}
```

- bool: a boolean value, either true or false. null: a value that represents *absence* or *omission.* If you set an argument of a resource to null, terraform behaves as though you had completely omitted it — it will use the argument's default value if it has one, or raise an error if the argument is mandatory. null is most useful in conditional expressions, so you can dynamically omit an argument if a condition isn't met.



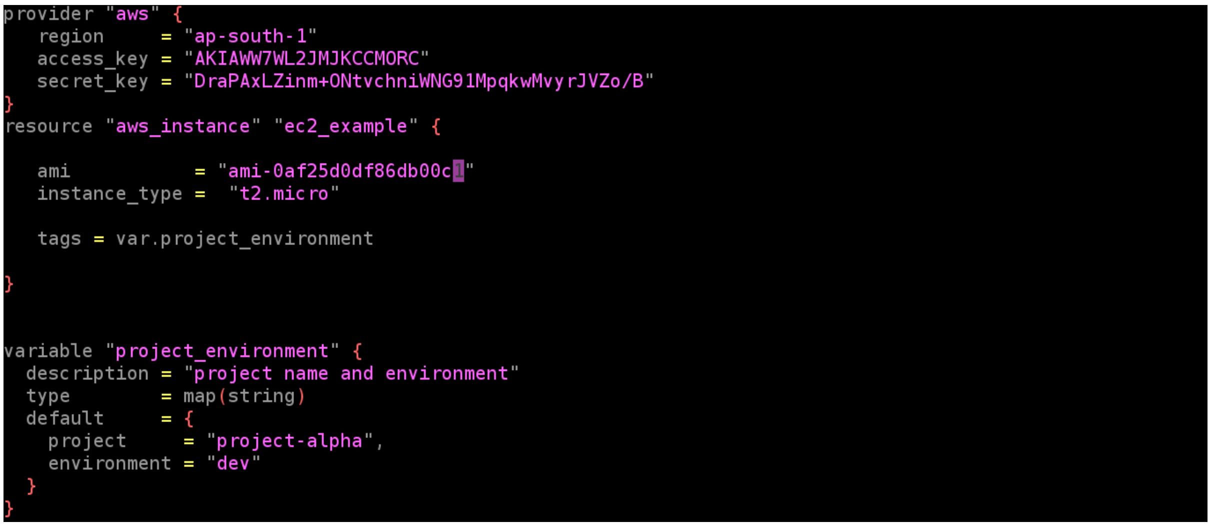
```
provider "aws" {
   region     = "ap-south-1"
   access_key = "AKIAWW7WL2JMJKCCMORC"
   secret_key = "DraPAxLZinm+0NtvchniWNG91MpqkwMvyrJVZo/B"
}

resource "aws_instance" "ec2_example" {

   ami           = "ami-0af25d0df86db00c1"
   instance_type = "t2.micro"
   count  = 1
   associate_public_ip_address = var.enable_public_ip

   tags = {
           Name = "Terraform EC2"
   }
}

variable "enable_public_ip" {
   description = "Enable public IP"
   type        = bool
   default     = true
}
```

- list (or tuple): a sequence of values, like ["user1", "user2", "user3"]. Elements in a list or tuple are identified by consecutive whole numbers, starting with zero.



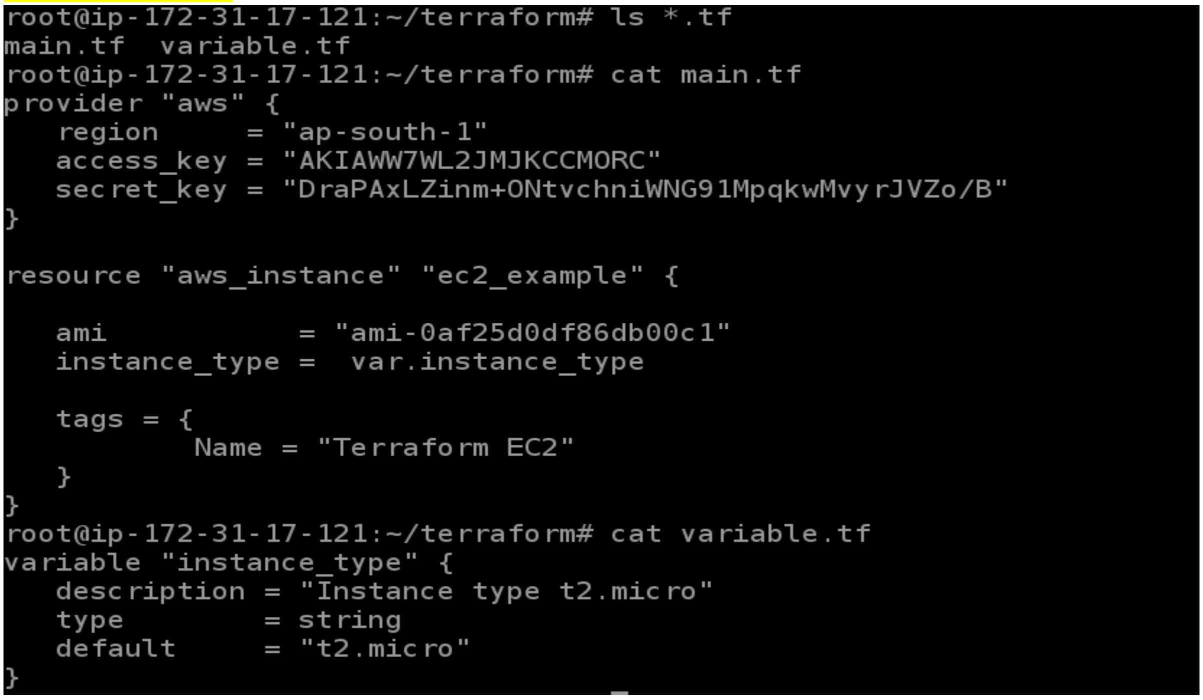
```
provider "aws" {
   region     = "ap-south-1"
   access_key = "AKIAWW7WL2JMJKCCMORC"
   secret_key = "DraPAxLZinm+0NtvchniWNG91MpqkwMvyrJVZo/B"
}

resource "aws_instance" "ec2_example" {

   ami           = "ami-0af25d0df86db00c1"
   instance_type = "t2.micro"
   count  = 1

   tags = {
           Name = "Terraform EC2"
   }
}

resource "aws_iam_user" "example" {
  count = length(var.user_names)
  name  = var.user_names[count.index]
}

variable "user_names" {
   description = "IAM USERS"
   type        = list(string)
   default     = ["user1", "user2", "user3"]
}
```

- map (or object): a group of values identified by named labels, like {project = "project-plan", environment = “dev”}.



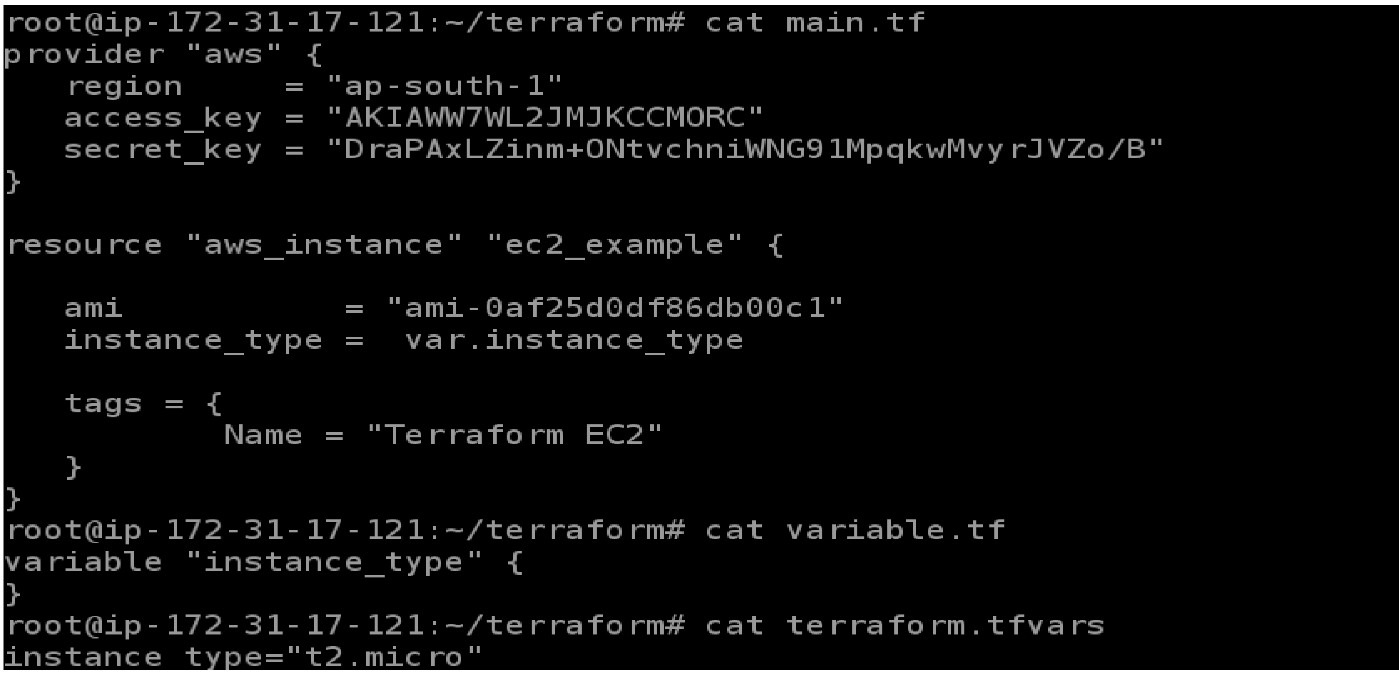
```
provider "aws" {
   region     = "ap-south-1"
   access_key = "AKIAWW7WL2JMJKCCMORC"
   secret_key = "DraPAxLZinm+ONtvchniWNG91MpqkwMvyrJVZo/B"
}
resource "aws_instance" "ec2_example" {

   ami           = "ami-0af25d0df86db00c1"
   instance_type =  "t2.micro"

   tags = var.project_environment

}


variable "project_environment" {
  description = "project name and environment"
  type        = map(string)
  default     = {
    project     = "project-alpha",
    environment = "dev"
  }
}
```

VARIABLE.TF



```
root@ip-172-31-17-121:~/terraform# ls *.tf
main.tf  variable.tf
root@ip-172-31-17-121:~/terraform# cat main.tf
provider "aws" {
   region     = "ap-south-1"
   access_key = "AKIAWW7WL2JMJKCCMORC"
   secret_key = "DraPAxLZinm+ONtvchniWNG91MpqkwMvyrJVZo/B"
}

resource "aws_instance" "ec2_example" {

   ami           = "ami-0af25d0df86db00c1"
   instance_type =  var.instance_type

   tags = {
           Name = "Terraform EC2"
   }
}
root@ip-172-31-17-121:~/terraform# cat variable.tf
variable "instance_type" {
   description = "Instance type t2.micro"
   type        = string
   default     = "t2.micro"
}
```

TERRAFORM.TFVARS



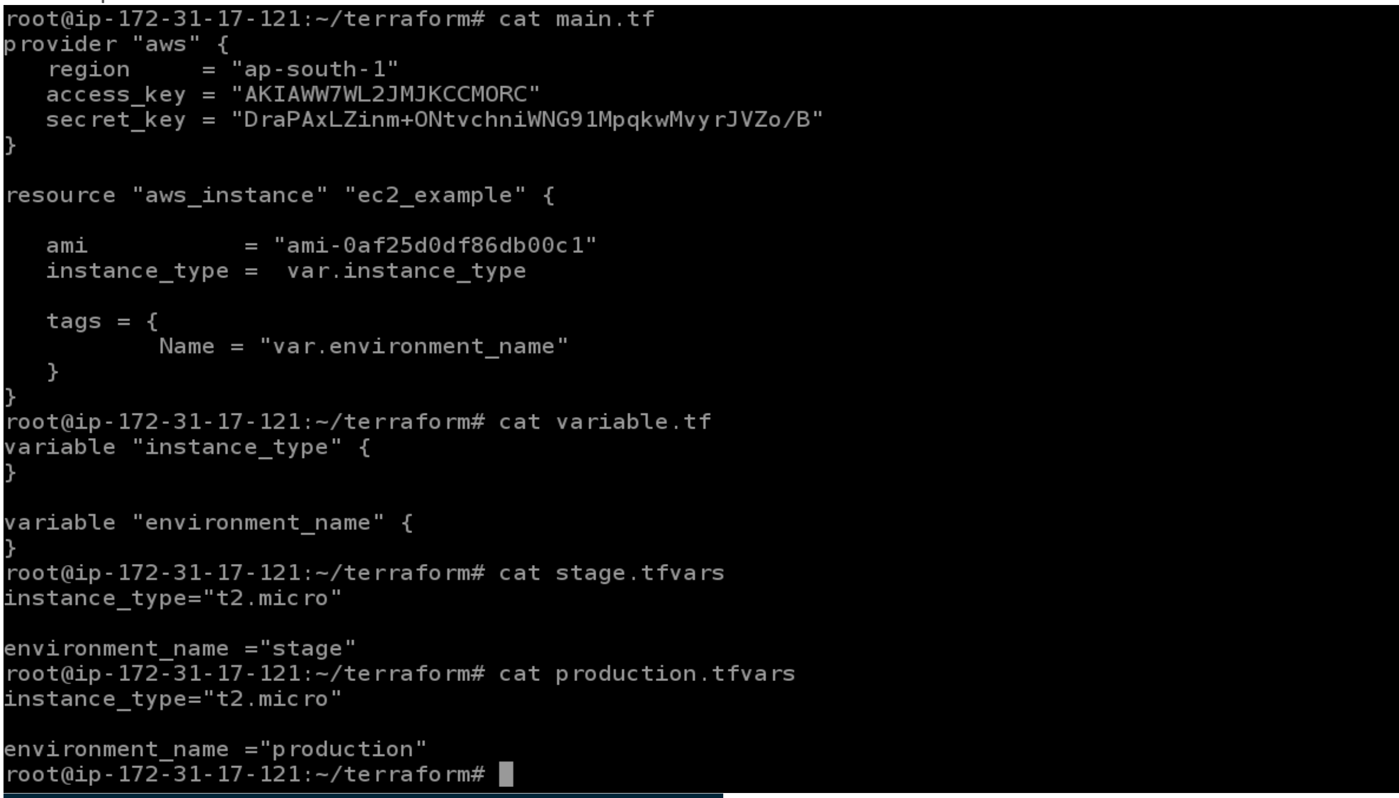
```
root@ip-172-31-17-121:~/terraform# cat main.tf
provider "aws" {
   region     = "ap-south-1"
   access_key = "AKIAWW7WL2JMJKCCMORC"
   secret_key = "DraPAxLZinm+ONtvchniWNG91MpqkwMvyrJVZo/B"
}

resource "aws_instance" "ec2_example" {

   ami            = "ami-0af25d0df86db00c1"
   instance_type =  var.instance_type

   tags = {
           Name = "Terraform EC2"
   }
}
root@ip-172-31-17-121:~/terraform# cat variable.tf
variable "instance_type" {
}
root@ip-172-31-17-121:~/terraform# cat terraform.tfvars
instance_type="t2.micro"
```

MULTIPE TFVAR FILES

There can be situation where you need create multiple tfvars files based on the environment like stage, production.

So in such scenario you can create one tfvars file for each environment -

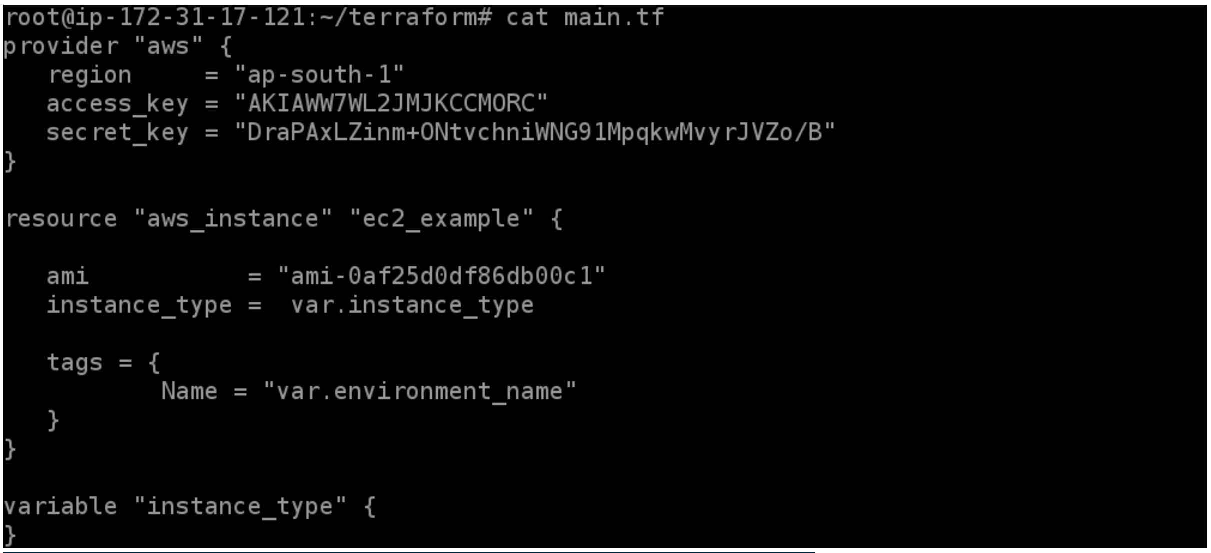
1. stage.tfvars
2. production.tfvars



```
root@ip-172-31-17-121:~/terraform# cat main.tf
provider "aws" {
   region     = "ap-south-1"
   access_key = "AKIAWW7WL2JMJKCCMORC"
   secret_key = "DraPAxLZinm+ONtvchniWNG91MpqkwMvyrJVZo/B"
}

resource "aws_instance" "ec2_example" {

   ami            = "ami-0af25d0df86db00c1"
   instance_type =  var.instance_type

   tags = {
           Name = "var.environment_name"
   }
}
root@ip-172-31-17-121:~/terraform# cat variable.tf
variable "instance_type" {
}

variable "environment_name" {
}
root@ip-172-31-17-121:~/terraform# cat stage.tfvars
instance_type="t2.micro"

environment_name ="stage"
root@ip-172-31-17-121:~/terraform# cat production.tfvars
instance_type="t2.micro"

environment_name ="production"
root@ip-172-31-17-121:~/terraform#
```

```
terraform plan -var-file="stage.tfvars"
terraform apply -var-file="stage.tfvars"
terraform destroy -var-file="stage.tfvars"
```

TERRAFROM COMMANDLINE VARIABLE



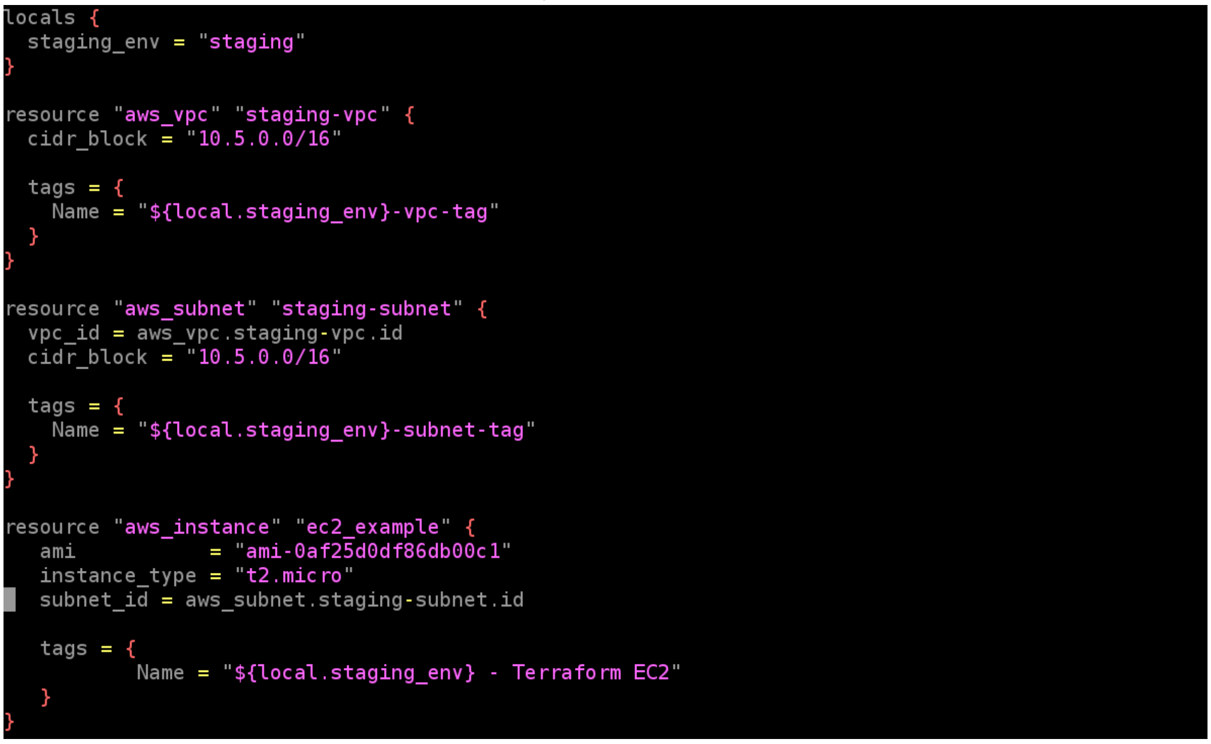
```
root@ip-172-31-17-121:~/terraform# cat main.tf
provider "aws" {
   region     = "ap-south-1"
   access_key = "AKIAWW7WL2JMJKCCMORC"
   secret_key = "DraPAxLZinm+ONtvchniWNG91MpqkwMvyrJVZo/B"
}

resource "aws_instance" "ec2_example" {

   ami           = "ami-0af25d0df86db00c1"
   instance_type =  var.instance_type

   tags = {
           Name = "var.environment_name"
   }
}

variable "instance_type" {
}
```

```
terraform plan -var="instance_type=t2.micro"
terraform apply -var="instance_type=t2.micro"
terraform destroy -var="instance_type=t2.micro"
```

TERRAFORM LOCALS

Terraform locals are quite similar to terraform variables but Terraform locals do not change their value. On the other hand, if you talk about Terraform input variables then it is dependent on user input and it can change its value. So if you have a very large Terraform file where you need to use the same values or expressions multiple times then Terraform local can be useful for you.

NOTE: Give the Entire Provide block as usually.



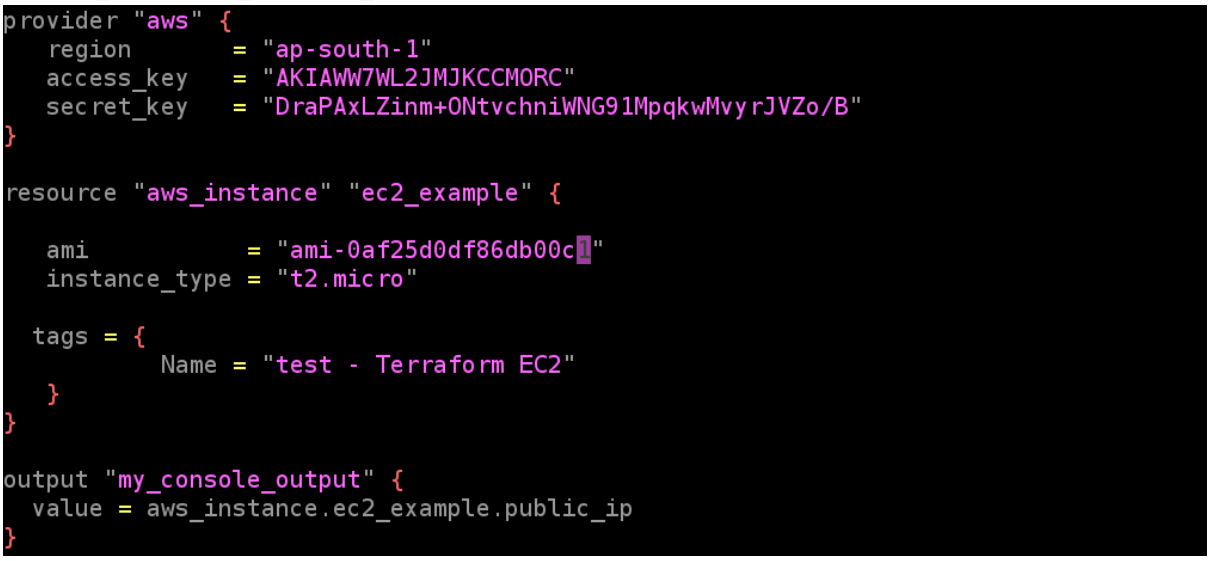
```
locals {
  staging_env = "staging"
}

resource "aws_vpc" "staging-vpc" {
  cidr_block = "10.5.0.0/16"

  tags = {
    Name = "${local.staging_env}-vpc-tag"
  }
}

resource "aws_subnet" "staging-subnet" {
  vpc_id = aws_vpc.staging-vpc.id
  cidr_block = "10.5.0.0/16"

  tags = {
    Name = "${local.staging_env}-subnet-tag"
  }
}

resource "aws_instance" "ec2_example" {
   ami           = "ami-0af25d0df86db00c1"
   instance_type = "t2.micro"
   subnet_id = aws_subnet.staging-subnet.id

   tags = {
           Name = "${local.staging_env} - Terraform EC2"
   }
}
```

provider "aws" {

```
  region = "ap-south-1"
}

locals {
staging_env = "prod"
}

resource "aws_vpc" "one" {
cidr_block = "10.5.0.0/16"
tags = {
Name = "${local.staging_env}-vpc"
}
}

resource "aws_subnet" "two" {
vpc_id = aws_vpc.one.id
cidr_block = "10.5.0.0/16"
availability_zone = "ap-south-1a"
tags = {
Name = "${local.staging_env}-subnet"
}
}

resource "aws_instance" "three" {
  ami           = "ami-074dc0a6f6c764218"
  instance_type = "t2.medium"
  subnet_id = aws_subnet.two.id
  tags = {
    Name = "${local.staging_env}-instance"
  }
}
```

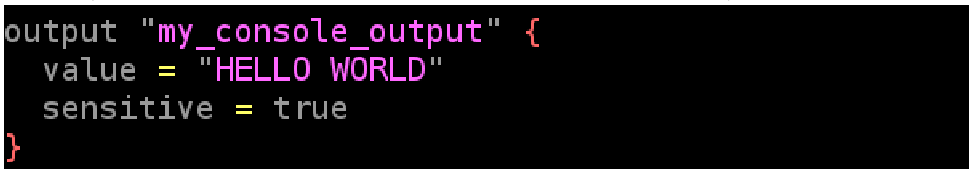
TERRAFORM OUTPUT VALUES

Terraform output values will be really useful when you want to debug your terraform code. Terraform output values can help you to print the attributes reference(arn, instance_state, outpost_arn, public_ip, public_dns etc) on your console.



```
provider "aws" {
   region       = "ap-south-1"
   access_key   = "AKIAWW7WL2JMJKCCMORC"
   secret_key   = "DraPAxLZinm+ONtvchniWNG91MpqkwMvyrJVZo/B"
}

resource "aws_instance" "ec2_example" {

   ami           = "ami-0af25d0df86db00c1"
   instance_type = "t2.micro"

  tags = {
          Name = "test - Terraform EC2"
   }
}

output "my_console_output" {
  value = aws_instance.ec2_example.public_ip
}
```

Now if you want to hide the sensitive info (like IP) use the key called sensitive.

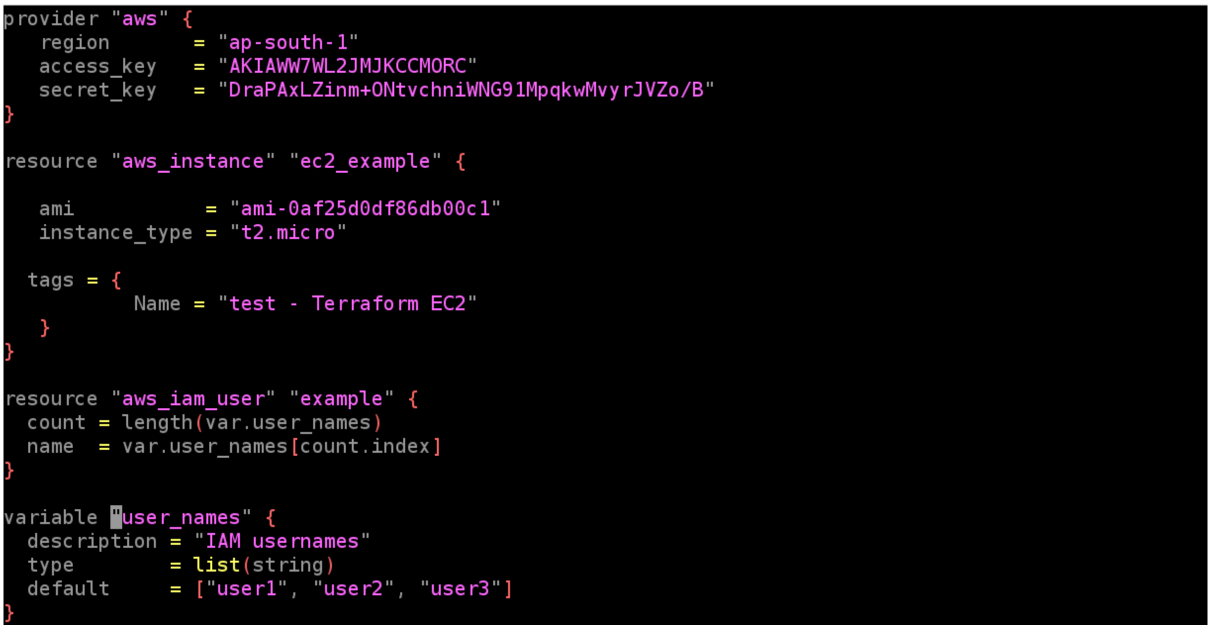


```
output "my_console_output" {
  value = "HELLO WORLD"
  sensitive = true
}
```

TO PRINT MULTIPLE VALUES:
output "my_console" {
value = [aws_instance.one.private_dns,aws_instance.one.public_ip, aws_instance.one.private_ip]
}

LOOPS WITH COUNT

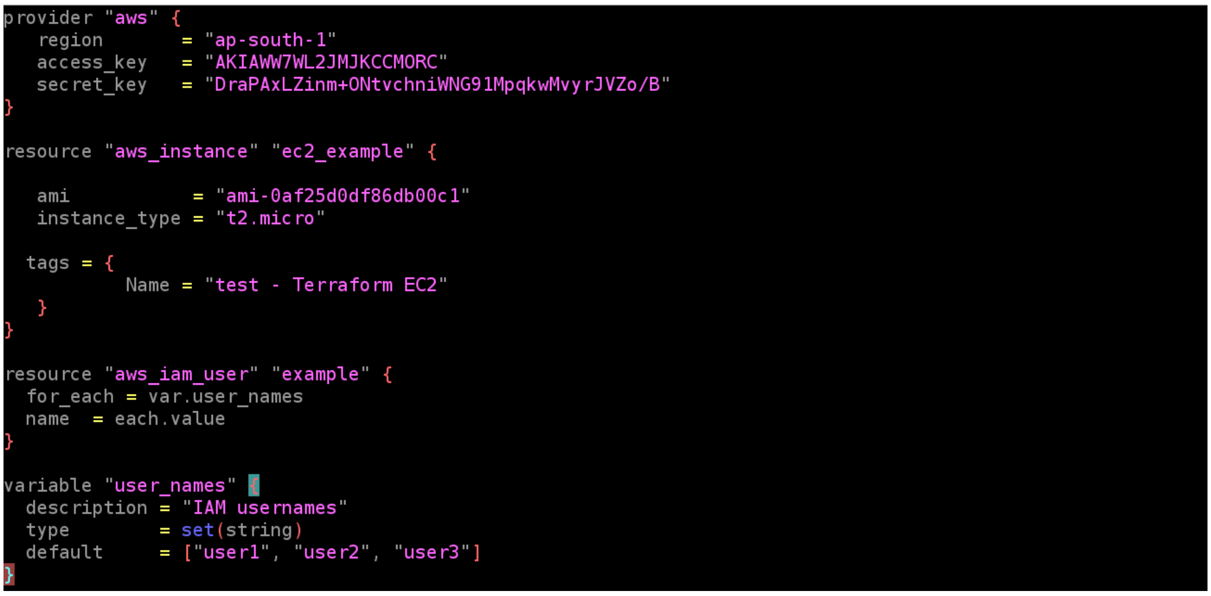
- As the name suggests we need to use count but to use the count first we need to declare collections inside our terraform file.



```
provider "aws" {
   region       = "ap-south-1"
   access_key   = "AKIAWW7WL2JMJKCCMORC"
   secret_key   = "DraPAxLZinm+ONtvchniWNG91MpqkwMvyrJVZo/B"
}

resource "aws_instance" "ec2_example" {

   ami           = "ami-0af25d0df86db00c1"
   instance_type = "t2.micro"

  tags = {
          Name = "test - Terraform EC2"
   }
}

resource "aws_iam_user" "example" {
  count = length(var.user_names)
  name  = var.user_names[count.index]
}

variable "user_names" {
  description = "IAM usernames"
  type        = list(string)
  default     = ["user1", "user2", "user3"]
}
```

LOOPS WITH FOR_EACH

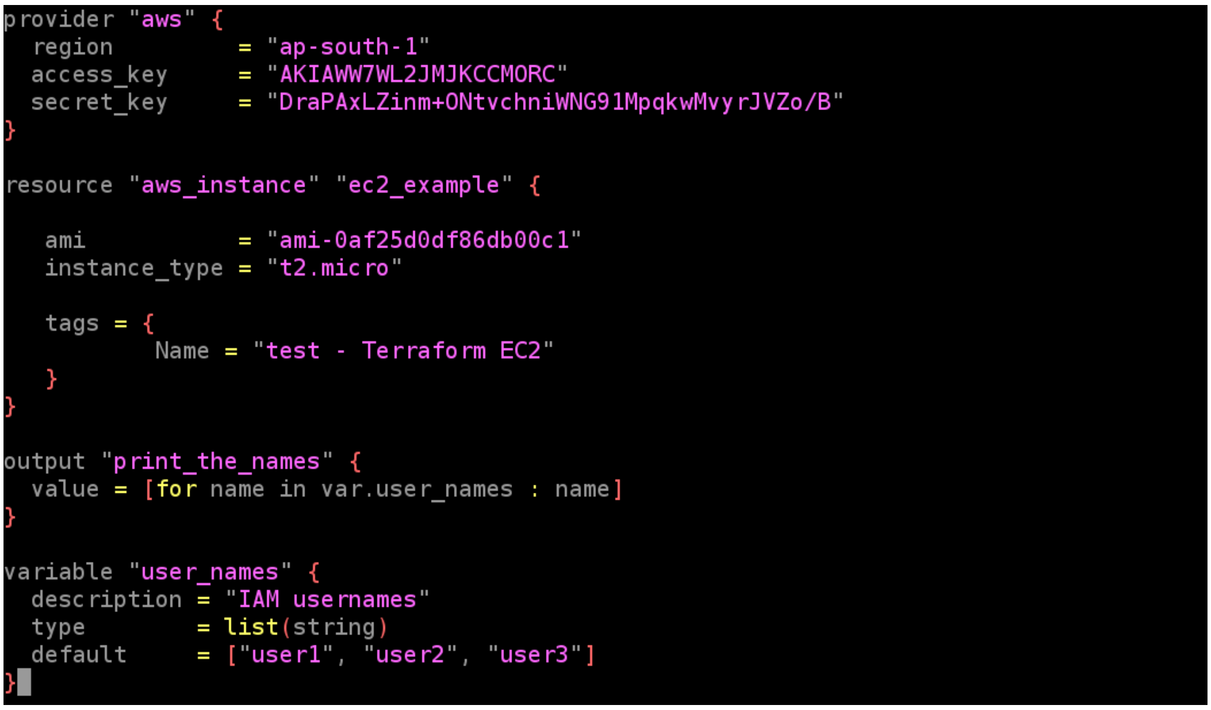
- The for_each is a little special in terraforming and you can not use it on any collection variable.
- *Note : - **It can only be used on** set(string) **or** map(string)**.***
- The reason why for_each does not work on list(string) is because a list can contain duplicate values but if you are using set(string) or map(string) then it does not support duplicate values.



```
provider "aws" {
   region       = "ap-south-1"
   access_key   = "AKIAWW7WL2JMJKCCMORC"
   secret_key   = "DraPAxLZinm+ONtvchniWNG91MpqkwMvyrJVZo/B"
}

resource "aws_instance" "ec2_example" {

   ami           = "ami-0af25d0df86db00c1"
   instance_type = "t2.micro"

  tags = {
          Name = "test - Terraform EC2"
   }
}

resource "aws_iam_user" "example" {
  for_each = var.user_names
  name  = each.value
}

variable "user_names" {
  description = "IAM usernames"
  type        = set(string)
  default     = ["user1", "user2", "user3"]
}
```

FOR LOOP

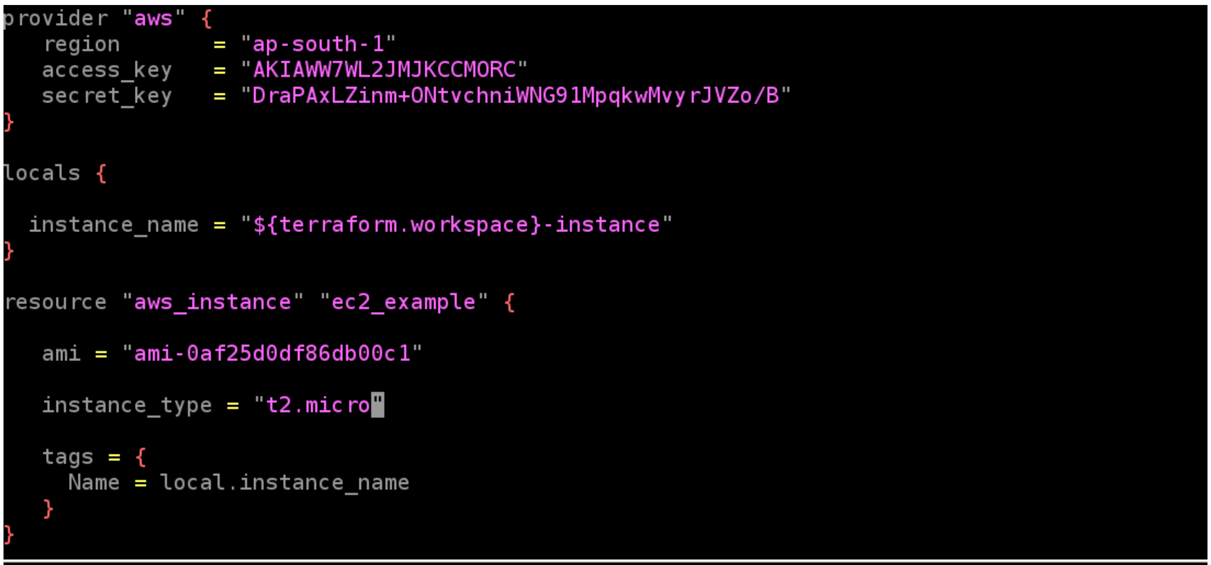
➢ The for loop is pretty simple and if you have used any programming language before then I guess you will be pretty much familiar with the for loop.
Only the difference you will notice over here is the syntax in Terraform.
➢ I am going to take the same example by declaring a list(string) and adding three users to it - user1, user2, user3



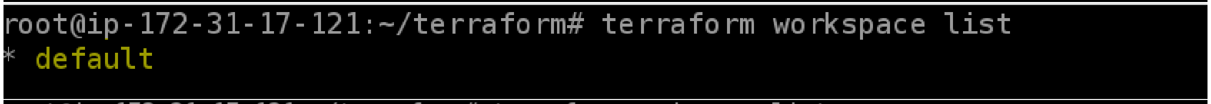
```
provider "aws" {
  region          = "ap-south-1"
  access_key      = "AKIAWW7WL2JMJKCCMORC"
  secret_key      = "DraPAxLZinm+ONtvchniWNG91MpqkwMvyrJVZo/B"
}

resource "aws_instance" "ec2_example" {

   ami           = "ami-0af25d0df86db00c1"
   instance_type = "t2.micro"

   tags = {
           Name = "test - Terraform EC2"
   }
}

output "print_the_names" {
  value = [for name in var.user_names : name]
}

variable "user_names" {
  description = "IAM usernames"
  type        = list(string)
  default     = ["user1", "user2", "user3"]
}
```

TERRAFORM WORKSPACE

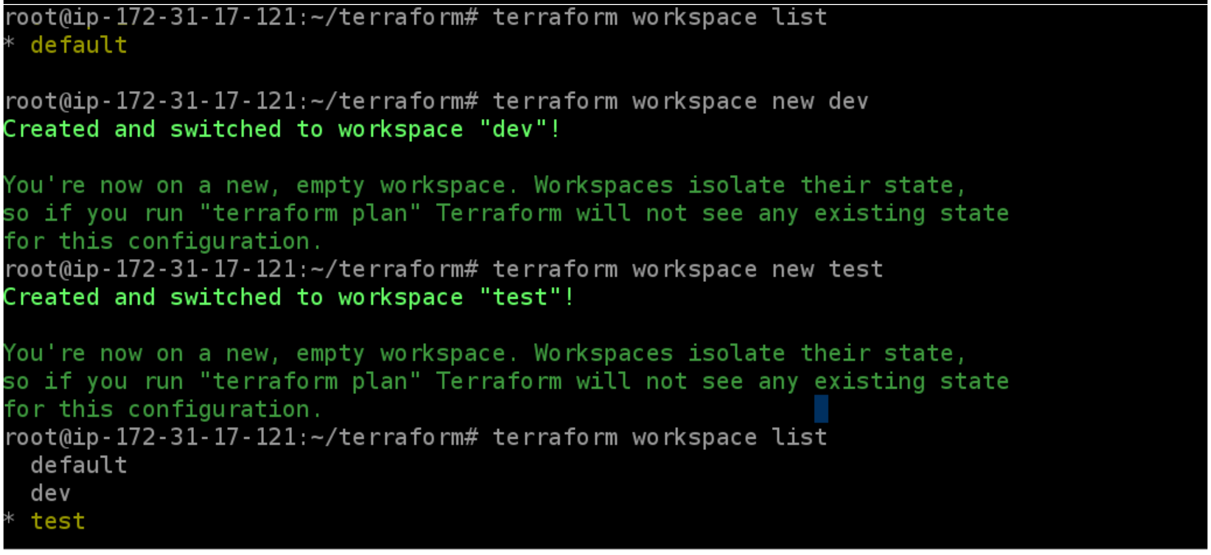
➢ To create a new workspace	: terraform workspace new workspace_name
➢ To list the workspace	: terraform workspace list
➢ To show current workspace	: terraform workspace show
➢ To switch workspace	: terraform workspace select workspace_name



```
provider "aws" {
   region       = "ap-south-1"
   access_key   = "AKIAWW7WL2JMJKCCMORC"
   secret_key   = "DraPAxLZinm+ONtvchniWNG91MpqkwMvyrJVZo/B"
}

locals {

  instance_name = "${terraform.workspace}-instance"
}

resource "aws_instance" "ec2_example" {

   ami = "ami-0af25d0df86db00c1"

   instance_type = "t2.micro"

   tags = {
     Name = local.instance_name
   }
}
```



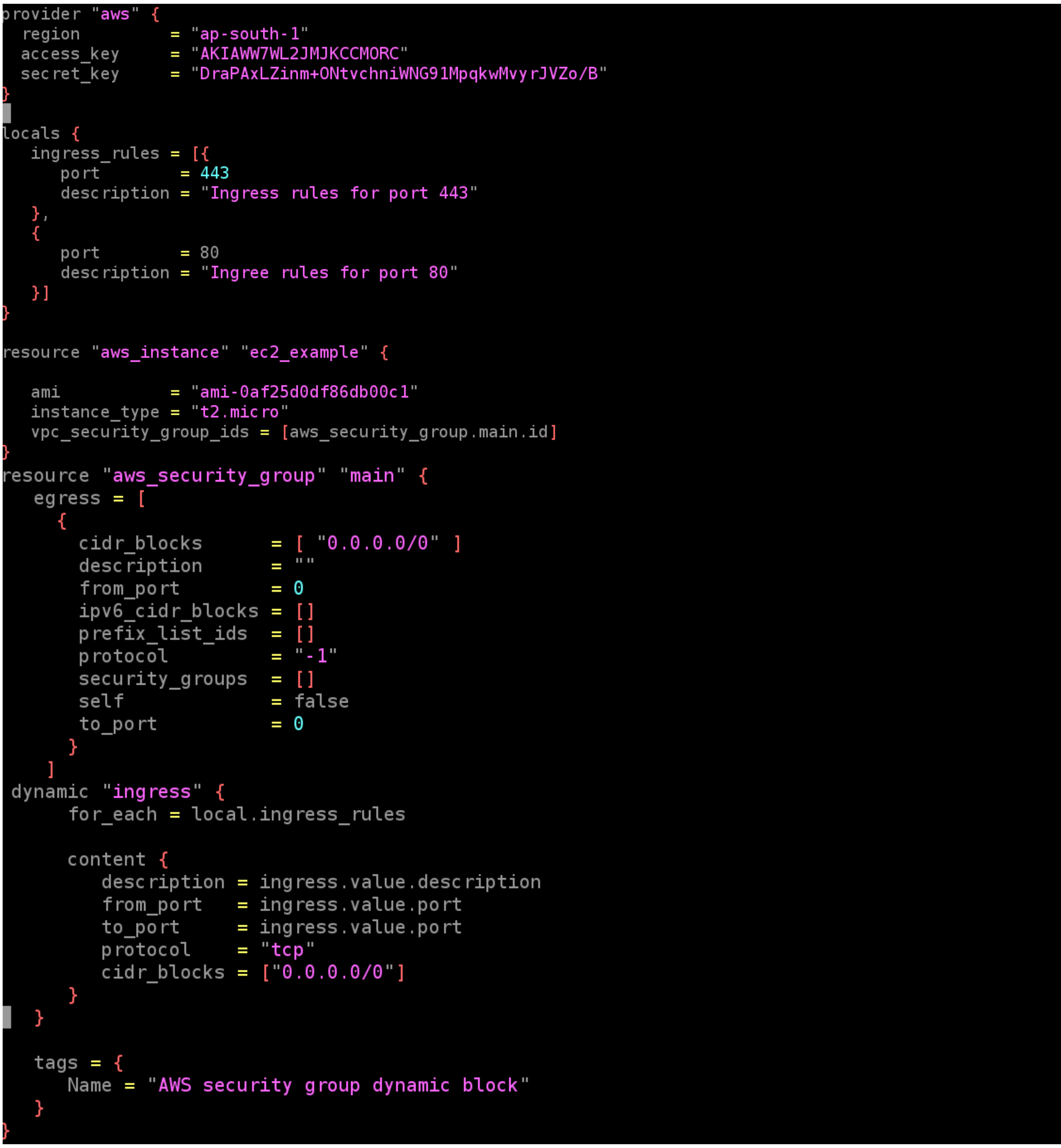
```
root@ip-172-31-17-121:~/terraform# terraform workspace list
* default
```



```
root@ip-172-31-17-121:~/terraform# terraform workspace list
* default

root@ip-172-31-17-121:~/terraform# terraform workspace new dev
Created and switched to workspace "dev"!

You're now on a new, empty workspace. Workspaces isolate their state,
so if you run "terraform plan" Terraform will not see any existing state
for this configuration.
root@ip-172-31-17-121:~/terraform# terraform workspace new test
Created and switched to workspace "test"!

You're now on a new, empty workspace. Workspaces isolate their state,
so if you run "terraform plan" Terraform will not see any existing state
for this configuration.
root@ip-172-31-17-121:~/terraform# terraform workspace list
  default
  dev
* test
```

DYNAMIC BLOCK

- Reduces the line of the code and makes the code reusable for us.



```
provider "aws" {
  region          = "ap-south-1"
  access_key      = "AKIAWW7WL2JMJKCCMORC"
  secret_key      = "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"
}

locals {
   ingress_rules = [{
      port        = 443
      description = "Ingress rules for port 443"
   },
   {
      port        = 80
      description = "Ingree rules for port 80"
   }]
}

resource "aws_instance" "ec2_example" {

   ami           = "ami-0af25d0df86db00c1"
   instance_type = "t2.micro"
   vpc_security_group_ids = [aws_security_group.main.id]
}
resource "aws_security_group" "main" {
   egress = [
     {
       cidr_blocks      = [ "0.0.0.0/0" ]
       description      = ""
       from_port        = 0
       ipv6_cidr_blocks = []
       prefix_list_ids  = []
       protocol         = "-1"
       security_groups  = []
       self             = false
       to_port          = 0
     }
   ]
 dynamic "ingress" {
      for_each = local.ingress_rules

      content {
         description = ingress.value.description
         from_port   = ingress.value.port
         to_port     = ingress.value.port
         protocol    = "tcp"
         cidr_blocks = ["0.0.0.0/0"]
      }
   }

   tags = {
      Name = "AWS security group dynamic block"
   }
}
```

CODE:

```
locals {
  ingress_rules = [{
port= 443
description = "Ingress rules for port 443"
  },
  {
port= 80
description = "Ingree rules for port 80"
  }]
}

resource "aws_instance" "ec2_example" {
   ami = "ami-0c02fb55956c7d316"
   instance_type = "t2.micro"
   vpc_security_group_ids = [aws_security_group.main.id]
   tags = {
Name = "Terraform EC2"
   }
}

resource "aws_security_group" "main" {

egress = [
{
cidr_blocks      = [ "0.0.0.0/0" ]
description      = "*"
from_port= 0
ipv6_cidr_blocks = []
prefix_list_ids  = []
protocol= "-1"
security_groups  = []
self= false
to_port= 0
} ]

dynamic "ingress" {
for_each = local.ingress_rules

content {
description = "*"
from_port = ingress.value.port
to_port = ingress.value.port
protocol = "tcp"
cidr_blocks = ["0.0.0.0/0"]
}
}

tags = {
Name = "terra sg"
}
}
```

EBS:

```
resource "aws_ebs_volume" "example" {
  availability_zone = "us-west-2a"
  size              = 40

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

EFS:

```
resource "aws_efs_file_system" "foo" {
  creation_token = "my-product"

  tags = {
    Name = "MyProduct"
  }
}
```

S3:

```
resource "aws_s3_bucket" "example" {
  bucket = "my-tf-example-bucket"
}
```

```
resource "aws_s3_bucket_acl" "example_bucket_acl" {
  bucket = aws_s3_bucket.example.id
  acl    = "private"
}
```

RDS:

```
resource "aws_rds_cluster" "default" {
  cluster_identifier      = "aurora-cluster-demo"
  engine                  = "aurora-mysql"
  engine_version          = "5.7.mysql_aurora.2.10.2"
  availability_zones      = ["us-east-1a", "us-east-1b", "us-east-1c"]
  database_name           = "mydb"
  master_username         = "foo"
  master_password         = "Raham@#444555"
  backup_retention_period = 5
  preferred_backup_window = "07:00-09:00"
}
```

ALIAS & PROVIDER:

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

provider "aws" {
alias = "south"
region = "ap-south-1"
}

resource "aws_instance" "one" {
ami = "ami-0b0dcb5067f052a63"
instance_type = "t2.micro"
tags = {
Name = "nvirginia"
}
}

resource "aws_instance" "two" {
ami = "ami-0a2457eba250ca23d"
instance_type = "t2.micro"
provider = "aws.south"
tags = {
Name = "mumbai"
}
}
```

LOCAL:

```
resource "local_file" "abc" {
filename = “/root/abc.txt”
}
```

```
resource "local_file" "abc" {
filename = “/root/abc.txt”
content = “hai all”
}
```

```
resource "local_file" "abc" {
filename = “/root/abc.txt”
content = “hai all”
file_permission = “777”
}
```

MULTIPLE PROVIDERS:
Lets work with multiple providers now

RANDOM PROVIDER:
The "random" provider allows the use of randomness within Terraform configurations. This is a logical provider, which means that it works entirely within Terraform's logic, and doesn't interact with any other services.
it provides resources that generate random values during their creation and then hold those values steady until the inputs are changed.

```
terraform plan -lock=false
```

Version Constraints:
If you want to work on specific version then u can use this one:
Provider – > hashicorp – > local – > version

```
Terraform {
required_providers {
local = {
source = “hashicorp/local”
version = “1.3.0”
}
}
}

resource "local_file" "abc" {
filename = “/root/abc.txt”
content = “hai all”
file_permission = “777”
}
```

```
terraform init -upgrade
```
Terraform plan & apply
Check the local version now by using terraform version

Greater than :
Replace 1.3.0 with > 1.3.0 : It will be upgraded to more than version 1.3.0
Less than :
Replace 1.3.0 with < 1.3.0 : It will be upgraded to less than version 1.3.0

In between :
Give “ > 1.2.0, <2.0.0, !=1.4.0”

Specific version/Incremental version:
“~>1.2” : It will work on the incremental values of last version (ie 1.4)

TERRAFORM MODULES
Main.tf

```
module "my_instance_module" {
      source = "./modules/instances"
      ami = "ami-069f1a13711c4eb69"
      instance_type = "t2.micro"
      instance_name = "myvm01"
}

module "s3_module" {
```

```
source = "./modules/buckets"
bucket_name = "abc"
}
=========================================
Provider.tf

provider "aws" {
region = "ap-south-1"
access_key = "AKIAT5FAP2W4MO3QEH6U"
secret_key = "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"
}
=============================================
Modules/instances/main.tf

resource "aws_instance" "my_instance" {
	ami = var.ami
	instance_type = var.instance_type
	tags = {
		Name = var.instance_name
	}
}
=====================================================
Modules/instances/variable.tf

variable "ami" {
  type        = string
}

variable "instance_type" {
  type        = string
}

variable "instance_name" {
  description   = "Value of the Name tag for the EC2 instance"
  type        = string
}
===================================================
Modules/buckets/main.tf

resource "aws_s3_bucket" "b" {
bucket = var.bucket_name
}
==================================================
Modules/buckets/variable.tf

variable "bucket_name" {
type = string
}
```

TERRAFORM IMPORT:
If you create a resource outside terraform ie which is not created by using terraform, but if you want to bring under terraform control like getting details into state file and manage it.
For this we can use terraform import.

```
terraform import aws_instance.one(block-labels) i-0ae501196b74b014c (resource-id)
```

so now try to give some input and apply the infra will be updated

TERRAFORM CONFIGURE REMOTE STATE FILE:
Terraform supports lot of resource states s3 is one of them.
Here we are going to store remote state files in s3.

When multiple developers working on a single project by storing this state file on remote laction they can use it for the reference

Create a bucket manually

```
resource "aws_instance" "one" {
  ami           = "ami-0574da719dca65348"
  instance_type = "t2.micro"

  tags = {
    name = "web-server"
  }
}

terraform {
  backend "s3" {
    bucket = "raham889977"
    key    = "swiggy/prod/terraform.tfstate"
    region = "us-east-1"
  }
}
```

Init plan and apply
```
terraform init -reconfigure
```

TERRAFORM REFRESH:

When we have resource created by terraforming and if someone manually deleted from the console at that time there will be some miss match between state file and manual resources.
By using this refresh command we can make sure that our state file and resource on console are in same state.
Create an normal ec2 instance by apply command and delete it manually.

Before delete: Terraform state list

After delete: Terraform refresh & Terraform state list
TERRAFORM VAULT:

Secure, store, and tightly control access to tokens, passwords, certificates, encryption keys for protecting secrets, and other sensitive data using a UI, CLI, or HTTP API.

How it works:
Vault works primarily with tokens and a token is associated to the client's policy. Each policy is path-based and policy rules constrains the actions and accessibility to the paths for each client. With Vault, you can create tokens manually and assign them to your clients, or the clients can log in and obtain a token. The illustration below displays Vault's core workflow.

Installation:
sudo yum install -y yum-utils
sudo yum-config-manager --add-repo
https://rpm.releases.hashicorp.com/AmazonLinux/hashicorp.repo
sudo yum -y install vault
vault -version

Modes:
There are mainly two modes
1. Development mode
2. server mode (prod)

Commands:
Step 1 : CLI Command for starting the server - vault server -dev

Port: so it indicates our server is running on port 8200
Storage: by default we will see inmem, full form is in memeory storage, when ever we are in dev it will show on in mem that means its going to store creds in memory. If its prod then we are going to store creds on disk or database.
Unseal key & Root token: these both later we are going to export as environment variables
Copy those on anywhere

Step 2 : Set VAULT_ADDR by exporting to environment variable
- export VAULT_ADDR='http://127.0.0.1:8200'
Step 3 : Set Root Token by exporting to environment variable
- export VAULT_TOKEN="hvs.6j4cuewowBGit65rheNoceI7"
Step 4 : Verify the status of vault server by running the command
- vault status

DEBUGGIN & CALIDATION:

Debugging and validation are always a very crucial part of development. Without debugging and validation it is really hard for a developer to do the troubleshooting

To enable the debugging in Terraform you need to set two environment variables -

1. TF_LOG
2. TF_FILE_PATH

In the TF_LOG you can set the log level to - DEBUG, INFO, TRACE, WARN

And in TF_FILE_PATH you set the location of the log file where logs will be saved.

```
export TF_LOG=DEBUG
echo $TF_LOG
export TF_LOG_PATH="/root/terraform/terraform.log"
echo $TF_LOG_PATH
```

create a EC2 resource and see the logs

VALIDATION:

VERSION MANAGER:
Managing different versions of Terraform is a bit hard because you need to remove the existing Terraform and install the new Terraform version.

But upgrading and rolling back terraform versions became a real necessity when you work with the latest stack of different cloud providers (AWS, Azure, Google Cloud) where they keep releasing new versions of their product and often it is noticed that some of the latest versions of those do not work well with Terraform so you often need either upgrade or rollback the version of your Terraform

In case you are using Terraform to provision and manage your infrastructure, you normally install a specific version on your machine (or on your CI servers).

But what if you wanted to install another terraform version to test it out?

In case you have multiple environment in the same codebase, let's say dev and prod, and you deployed both of them using a fixed terraform version (1.2.7).

After a while a new terraform version is available (1.3.1). You can update the version on your machine and test if it works fine, if not re-install the older version ... Which can be a bit cumbersome ...

Or, you can use tfenv which is a Terraform version manager.

After installing tfenv, you can use the tfenv list command to list the available terraform versions you have installed:

REPO LINK: https://github.com/tfutils/tfenv

SETUP:

```
git clone --depth=1 https://github.com/tfutils/tfenv.git ~/.tfenv
echo 'export PATH="$HOME/.tfenv/bin:$PATH"' >> ~/.bash_profile
echo 'export PATH=$PATH:$HOME/.tfenv/bin' >> ~/.bashrc
```

```
source .bashrc
tfenv -v
tfenv
tfenv list-remote
tfenv install 1.2.9
tfenv install 1.2.9
```
to verify go to this path: /root/.tfenv/versions
```
tfenv use 1.2.8 : * mark
tfenv list
tfenv use 1.2.7 : * mark
```

TERRAFORM CLOUD:

Terraform Cloud is the place where you can work from UI instead on your local machine. It securely stores state and secret data, and can connect to version control systems so that you can develop your infrastructure using a workflow similar to application development. The Terraform Cloud UI provides a detailed view into the resources managed by a Terraform project and gives enhanced visibility into each Terraform operation.

Terraform Cloud also has a private registry for sharing your organization's Terraform modules and providers. Paid features include access controls for approving changes to infrastructure, detailed policy controls for governing the contents of Terraform configurations, cost estimates for runs, and more.

Terraform Cloud helps you collaborate on each step of your infrastructure development process. For example, each time you plan a new change, your team can review and approve the plan before you apply it. It also automatically locks state during operations to prevent concurrent modifications that may corrupt the state file.

TO CREATE A CLOUD:

Go to the terrafor instance and give the command
```
terraform login
```

You can create and login from here:
https://app.terraform.io/app/settings/tokens?source=terraform-login
it will ask the token generate and paste it here

get start from the scratch and create a workspace
create a rep on github first

SETP-1:

name -- > private -- > readme -- > .gitignore: terraform -- > license: apache version 2.0
token: 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

clone this repo to terraform instance and craete an new folder swith sample name then add and commit and push it to github

STEP-2:
create a workspace -- > VCS -- > Github.com (custom) -- > Register a new OAuth application -- >
Application name: Terraform Cloud (Shaik_Raham)
Homepage URL: https://app.terraform.io -- > Authorization callback URL:
https://app.terraform.io/auth/f22b8a2d-8fa5-43e5-b1a4-7e0f1a31a22a/callback -- > Register
NOTE: All these you can see on the SetUp Provider page

Name: github-oauth-terraform -- > Client ID: c1997e0459a6e266e2ad -- > Client Secret:
3108a440034361ca4e2044fc766ec63ec1535413 -- > connect and continue
click on authorize you will get a mail from github that is is done.

STEP-3:
SSH-KEY is optional skip it

now start working:

Workspaces -- > create -- > select github -- > select the repo -- > Advance -- > Terraform Working Directory: sample -- > create workspace
go to workspace overview -- > if you want to change anything go to settings

create access and secret access key (NOTE: Set as environment varibles)
Variables -- > Add: KEY: AWS_ACCESS_KEY_ID: ******************* -- >SAVE -- > ADD -- >
AWS_SECRET_ACCESS_KEY: *************** -- > SAVE

Go to the server and write the code now on sample directory and write code
```
provider "aws" {
  region = "ap-southeast-2"
}

resource "aws_instance" "one" {
  ami           = "ami-051a81c2bd3e755db"
  instance_type = "t2.micro"
  tags = {
    Name = "raham"
  }
}
```

NOTE: Go to variables select terraform variable and give (optional)
Add variable -- > terraform variable -- > key : instance_type -- > t2.micro -- > save

add commit and push the files check it on github
it will treigger automatically
go to the runs and check it is triggred
we need to give apply manually because by default it is manual apply method is selected on settings
if you want to make it auto apply go to settings and change it

Go to github and write coe for bucket now it will be creating automatically

TEST CASE:
Now we have one ec2 instance in a workspace and using that statefile im going to create instance on another workspace

craete a new repo and write code and commit in on github

workspaces -- > new -- > name: prod -- > Apply Method: Auto apply -- > Run Triggers -- > Only trigger runs when files in specified paths change : Path: terraformcloud/sample2/ -- > save
go to overview configure vars now
After configure if you got to actions you can see all the basic opertaions of terrafom in UI
click on run once you will see it running from ui as auto approve

if you want to destroy go to settings -- > Destruction and Deletion -- > Destroy infrastructure -- > queue destroy plan -- > workspace name -- > save
now it will be going to delete

Delete Workspace
Deleting this workspace does not destroy these resources, and Terraform cannot manage or track remaining infrastructure after deletion.
Deleting this workspace permanently removes all of its variables, settings, alert history, run history, and Terraform state.

We strongly recommend destroying all resources before deleting this workspace. You can queue a destroy plan in the workspace Settings > Destruction and Deletion.
This workspace manages 1 resource. Deleting this workspace does not destroy these resources, and Terraform cannot manage or track remaining infrastructure after deletion.

DEPENDS_ON:
It is used when resource one is dependent on resource two, here we have bucket first but using depends_on the instance will be created first then the bucket will be created.

```
resource "aws_s3_bucket" "one" {
  bucket = "rahamshaik88662211-depends-on"
  depends_on = [
    aws_instance.two
  ]
}
resource "aws_instance" "two" {
  ami           = "ami-0b5eea76982371e91"
  instance_type = "t2.micro"
  tags = {
    name = "web-server"
  }
}
```

TERRAFORM LIFE_CYCLE:

create_before_destroy: When Terraform determines it needs to destroy an object and recreate it, the normal behavior will create the new object after the existing one is destroyed. Using this attribute will create the new object first and then destroy the old one. This can help reduce downtime. Some objects have restrictions that the use of this setting may cause issues with, preventing objects from existing concurrently. Hence, it is important to understand any resource constraints before using this option.

```
resource "aws_instance" "web" {
  ami           = "ami-0b5eea76982371e91"
  instance_type = "t2.micro"

  tags = {
    name = "web-server"
  }
  lifecycle {
       create_before_destroy = true
  }
}
```

prevent_destroy :

This lifecycle option prevents Terraform from accidentally removing critical resources. This is useful to avoid downtime when a change would result in the destruction and recreation of resource. This block should be used only when necessary as it will make certain configuration changes impossible.

If you give true then it will make prevent to destroy the instance.

If you give false then it will make prevent to destroy the instance

```
lifecycle {
       prevent _destroy = true
  }
```

Ignore_changes:

It is useful when attributes of a resource are updated outside of Terraform, for example, when an instance created with name automatically applies tags. When Terraform detects the changes the aws Policy has applied, it will ignore them and not attempt to modify the tag. Attributes of the resource that need to be ignored can be specified.

Create an instance with a name and add a new tag like Env = Prod manually.

Go back to the terminal and give terraform plan, it will show update in-place

Because we have added the Env tag manually but the information is not a state file.

Add thes new block and apply one more time it wont show now

```
lifecycle {
      Ignore_changes = [
         tags, …… (if you want to add anyother you can add here)
]
  }
```

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.
Check with adding few more new tags and apply it.

Meta-arguments are special constructs provided for resources.
Meta-arguments come in handy in situations like creating resources in the same cloud provider, but in different regions, or when creating multiple identical resources with different names, or when we have to declare implicit dependencies in places where Terraform is not able to identify the dependency itself.

TERRAFORM NULL RESOURCE:
As in the name you see a prefix null which means this resource will not exist on your Cloud Infrastructure(AWS, Google Cloud, Azure). The reason is there is no terraform state associated with it, due to you can update the null_resource inside your Terraform file as many times as you can.

```
resource "aws_instance" "one" {
  ami           = "ami-0b5eea76982371e91"
  instance_type = "t2.micro"

  tags = {
    Name = "web-server"
  }
}

resource "null_resource" "null_resource_simple" {
  provisioner "local-exec" {
    command = "echo Hello World"
  }
}
```

Give apply it will print hellow world and create ec2 instance, but if you give second time it won't come
Then destroy and give trigger on the code

The trigger is a block inside the null_resource which holds key-value pair.
But to understand the trigger please have look at the following points
- As the name suggest trigger, it will execute local-exec, remote-exec or data block.
- Trigger will only work when it detects the change in the key-value pair.
- Trigger can only work once if key-value is changed once but on the other hand if the key-value pair changes its value every time it will execute every time you run $terraform apply command.

```
resource "null_resource" "null_resource_simple" {
triggers = {
    id = aws_instance.one.id
  }
  provisioner "local-exec" {
    command = "echo Hello World"
  }
}
```

If you put the trigger inside null resource then null resource is going to execute once because it will get new id into trigger if we don’t get new id inside trigger then null resource will wont execute.

Advantages:

1. **Don't create any infrastructure resources.**
2. **Automate tasks and integrate with external systems**
3. **Modular and reusable infrastructure code**
4. **Variety of provisioners**
5. **Powerful and Flexible**

FILE PROVISONERS:

Terraform provisioners are of three types –
1. file
2. local-exec
3. remote-exec

1.File provisioner - The file provisioner will help you to copy the file securely from the local machine or development machine to the remote Ec2 instance.

2. Local provisioner - The local-exec provisioner invokes a local executable after a resource is created. This invokes a process on the machine running Terraform, not on the resource. It will help you to run the shell command onto your local machine instead of the remote EC2 instance. The local provisioners are

3. Remote provisioner - The remote-exec provisioner invokes a script on a remote resource after it is created. This can be used to run a configuration management tool, bootstrap into a cluster, etc. It will help you to run the shell command onto the remote Ec2 instance of AWS or virtual machine of Google Cloud