**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 IaC, 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
* Immutable code

**CLOUD ALTERNATES:**

* AWS -- > CloudFormation templates (JSON/YAML)
* AZURE -- > ARM TEMPLATES (JSON)
* TERRAFORM(Car) -- > AWS(IOL), AZURE(BP), GCP(HP), VMWARE
* FUEL -- > CODE

**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

**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"

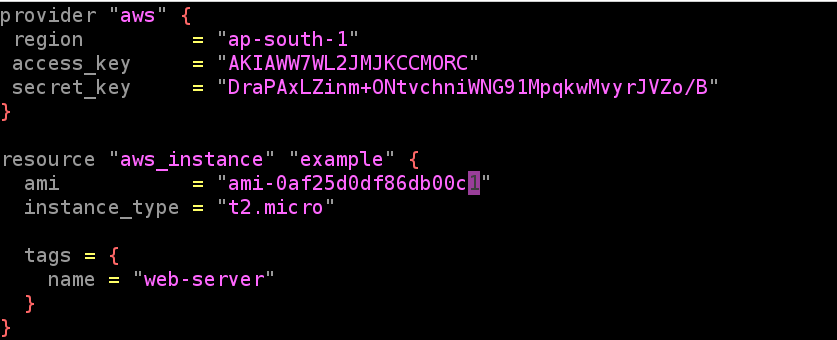
tags = {

name = "web-server"

}

}

* 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



**EC2-ROLE BASED AUTHENTICATION:**

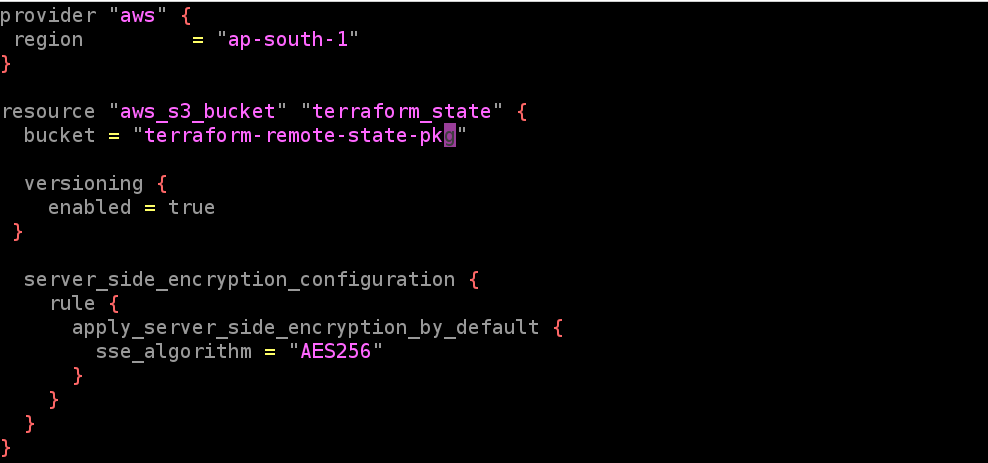
* By using this without access key and secret access key we can perform the actions
* IAM -- > Roles -- > Create -- > AWS Services : EC2 -- > [AdministratorAccess](https://console.aws.amazon.com/iam/home#/policies/arn%3Aaws%3Aiam%3A%3Aaws%3Apolicy%2FAdministratorAccess) -- > Name : EC2-Access -- > Role 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**

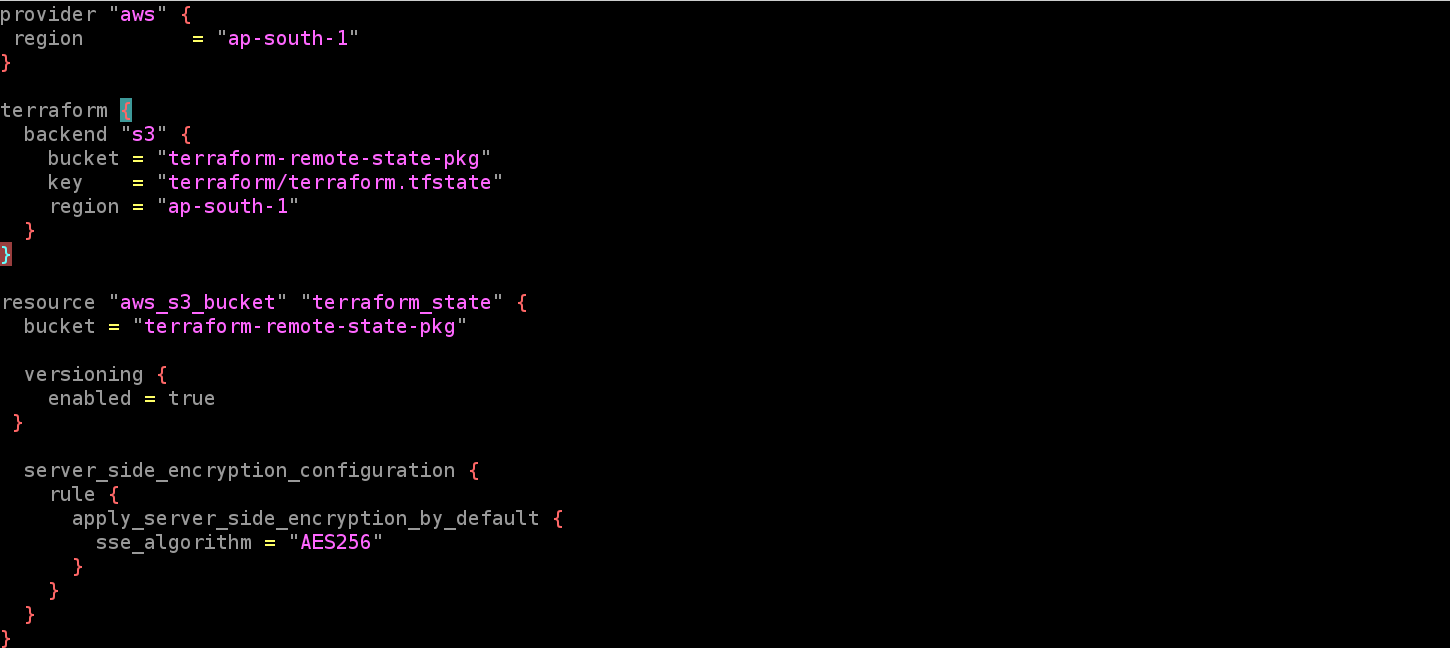
In terraform we have two state files one is local state file and another is remote state file.

We use Local state file when we there is no involvement of other person.

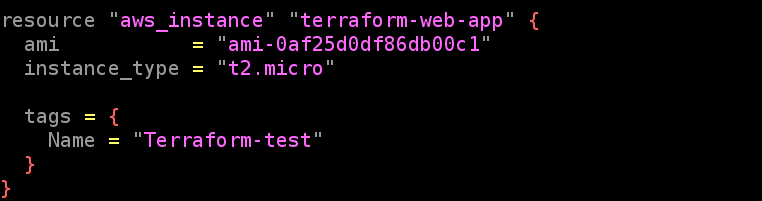
* Create a bucket with versioning enable
* Initialize the backend with S3 using Terraform.
* Launch the resources using terraform to validate the remote state file and Versioning.



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

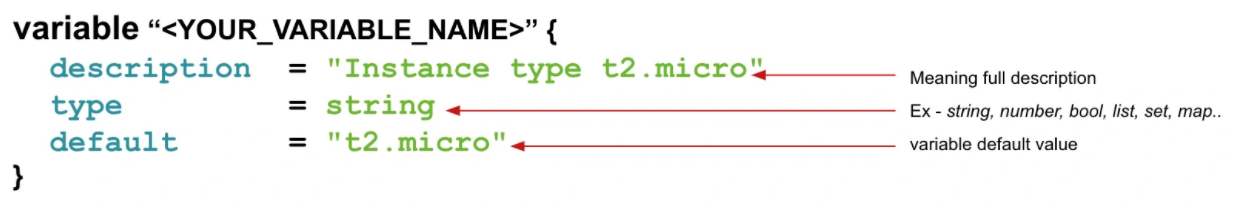


* Terraform init : It will be successful
* Now add EC2 Resource in same code.

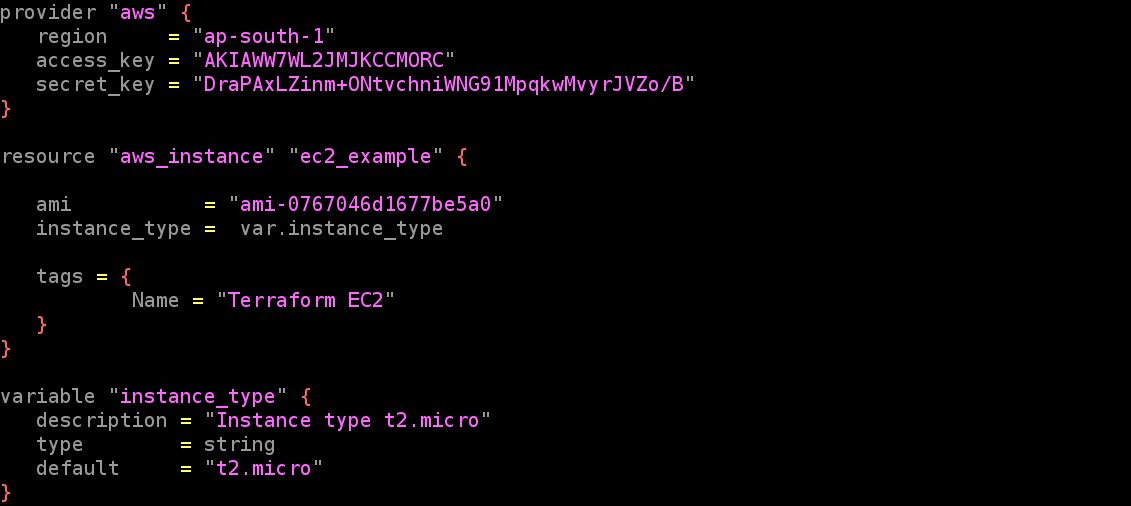


* Add this part on the end
* Terraform plan and terraform apply.
* Now verify the versioning on that bucket you can two versions and new instance will be created.
* If you give a new tag then you can see the new version (Terraform plan and apply)

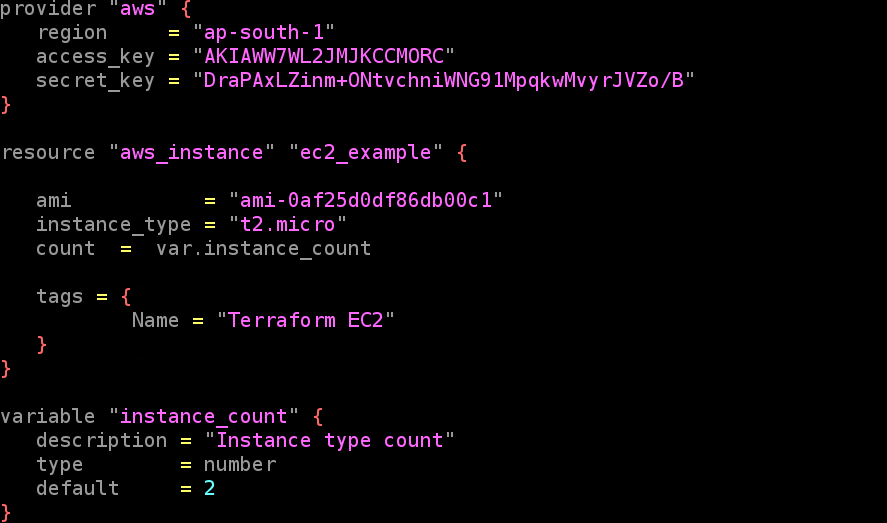
TERRAFORM TYPES (VARIABLE TYPE)



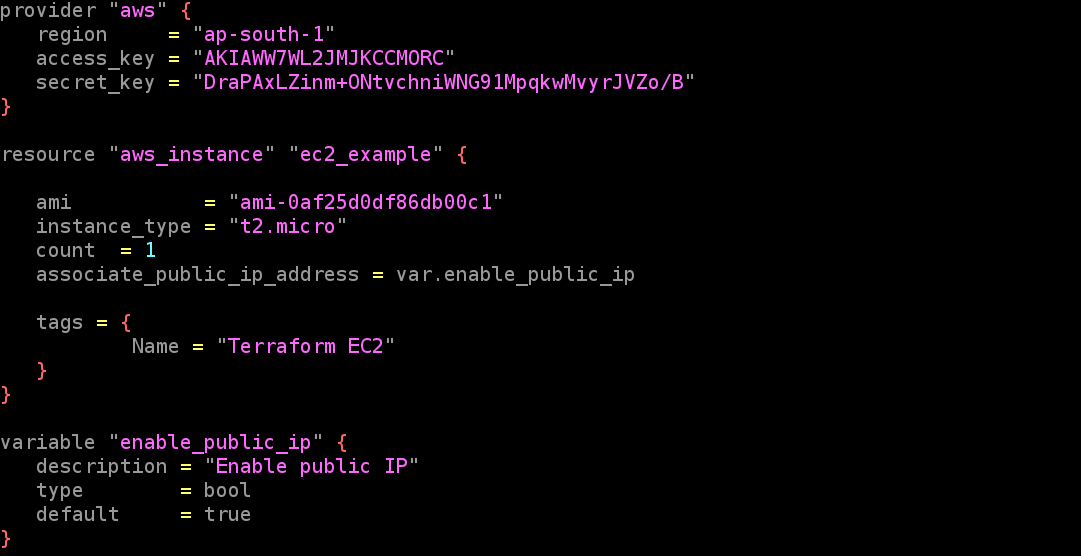
* [string](https://www.terraform.io/language/expressions/types#string): a sequence of Unicode characters representing some text, like "hello". (terraform init, plan, apply, destroy)



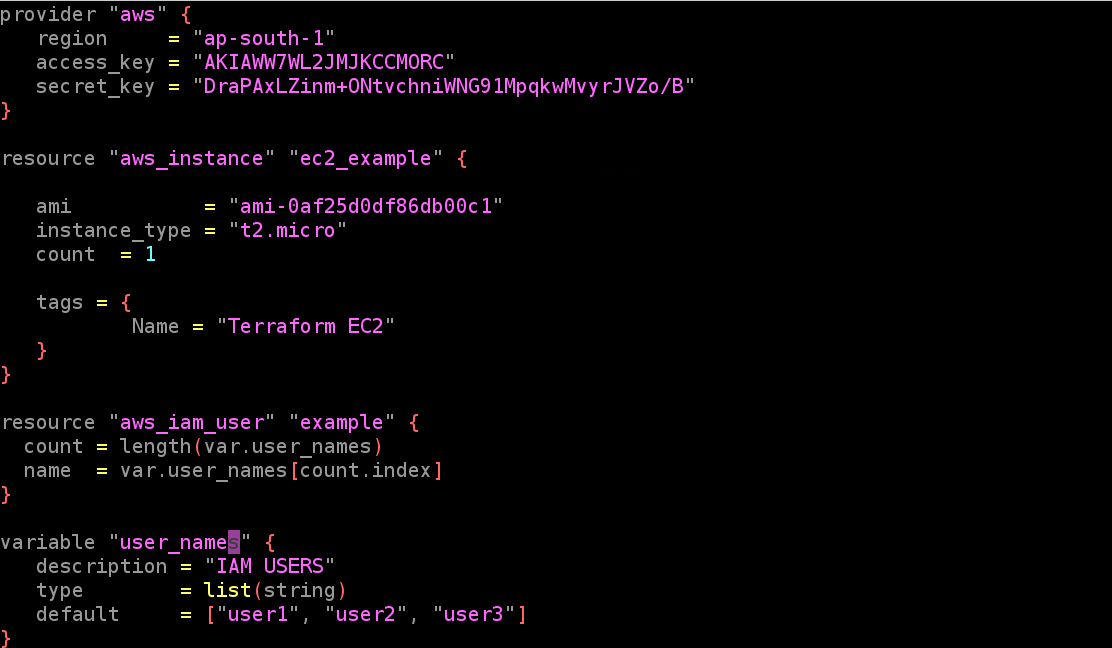
* [number](https://www.terraform.io/language/expressions/types#number): a numeric value. The number type can represent both whole numbers like 15 and fractional values like 6.283185.



* [bool](https://www.terraform.io/language/expressions/types#bool): a boolean value, either true or false. [null](https://www.terraform.io/language/expressions/types#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.



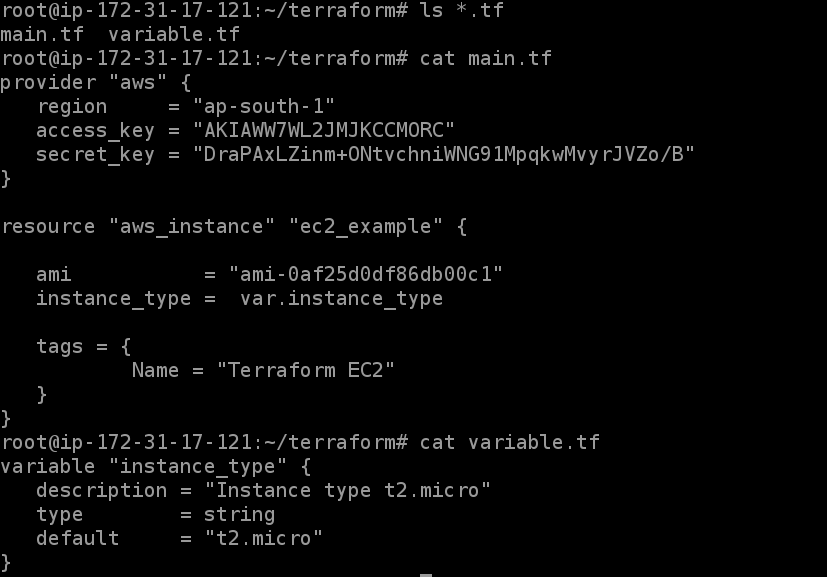
* [list](https://www.terraform.io/language/expressions/types#list) (or tuple): a sequence of values, like ["user1", "user2", "user3"]. Elements in a list or tuple are identified by consecutive whole numbers, starting with zero.



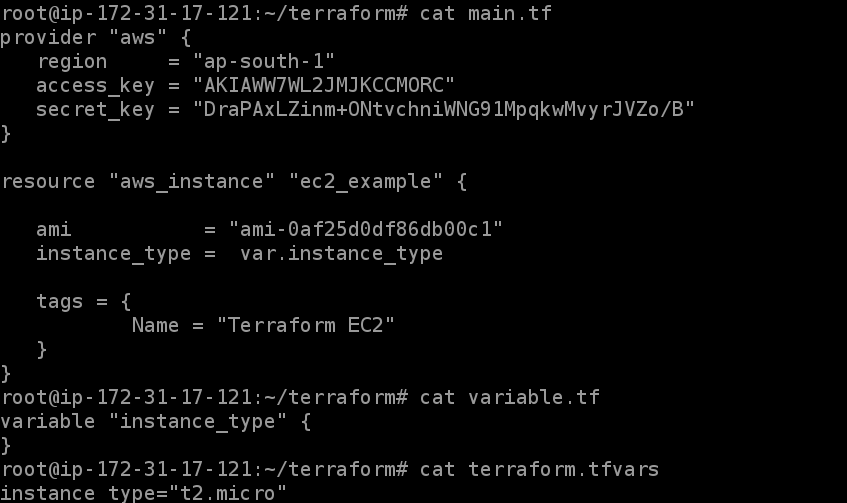
* [map](https://www.terraform.io/language/expressions/types#map) (or object): a group of values identified by named labels, like {project = "project-plan", environment = “dev”}.



**VARIABLE.TF**



**TERRAFORM.TFVARS**

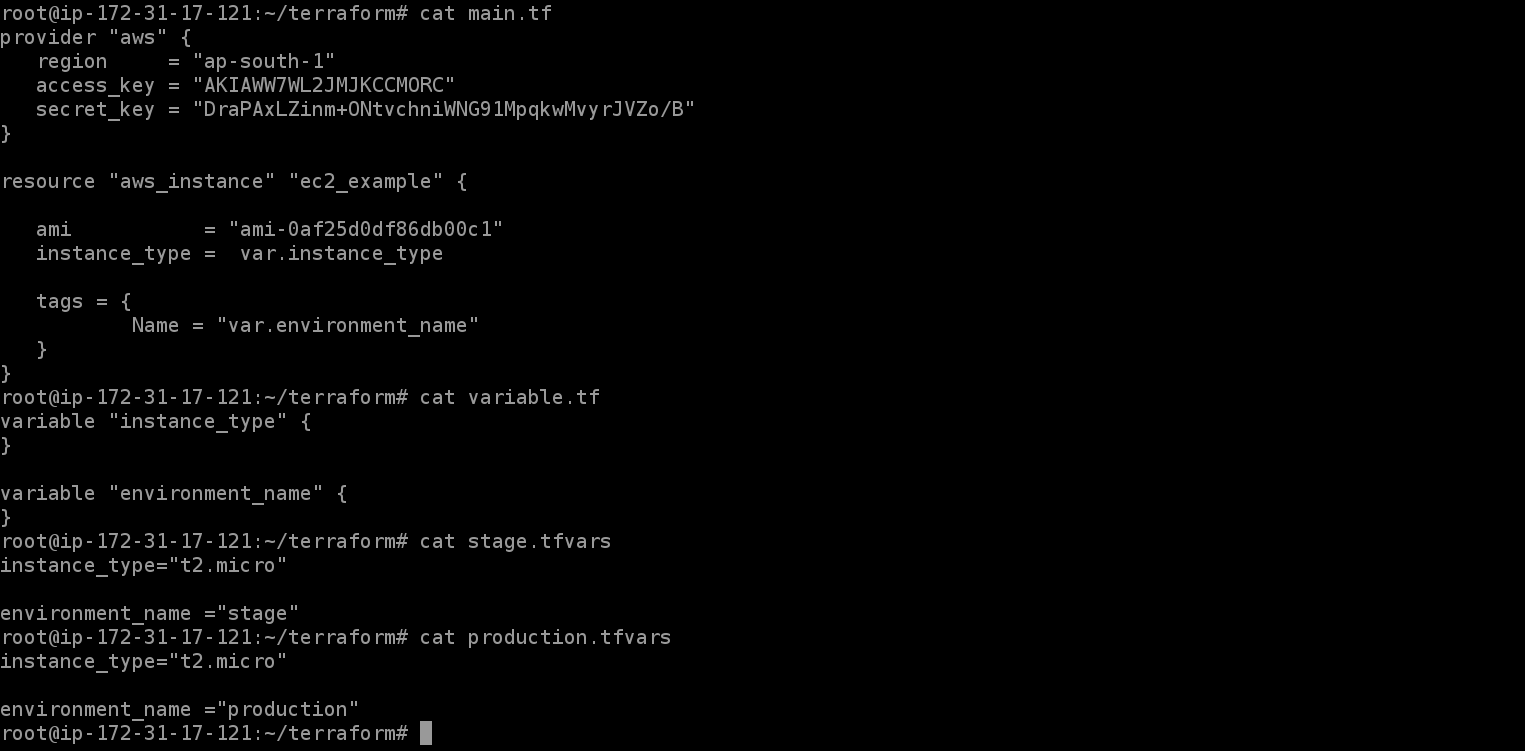
****

**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

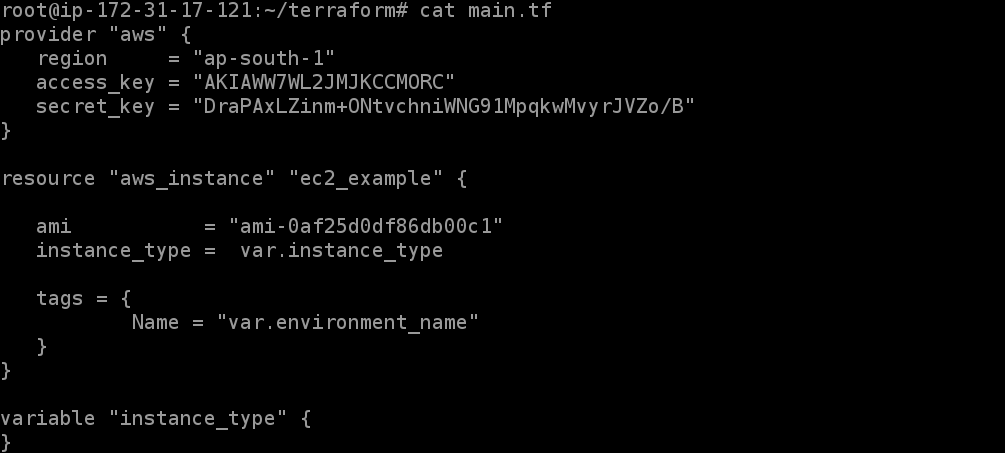


terraform plan -var-file="stage.tfvars"

terraform apply -var-file="stage.tfvars"

terraform destroy -var-file="stage.tfvars"

**TERRAFROM COMMANDLINE VARIABLE**

****

terraform plan -var="instance\_type=t2.micro"

terraform apply -var="instance\_type=t2.micro"

terraform destroy -var="instance\_type=t2.micro"

**TERRAFORM LOCALS**

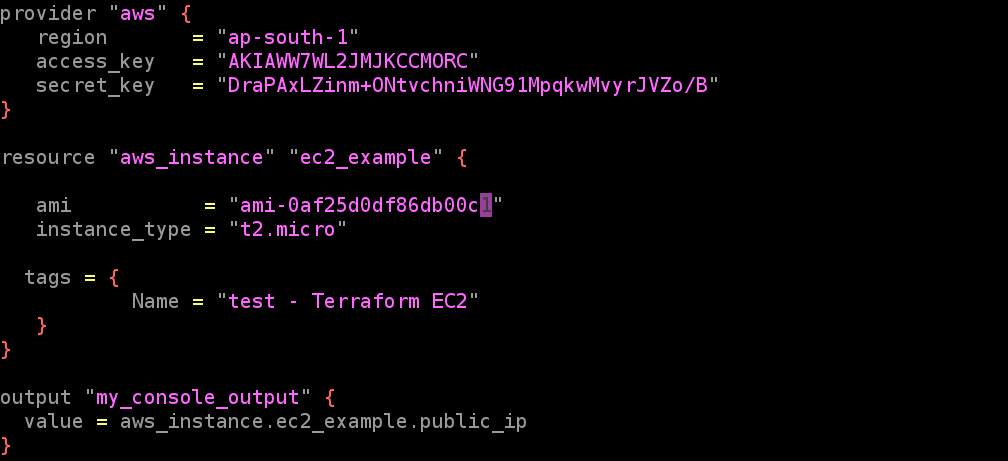
Terraform locals are quite similar to terraform variables but Terraform locals do not change their value. On the other hand, if you talk about Terraform input variables then it is dependent on user input and it can change its value. So if you have a very large Terraform file where you need to use the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

**NOTE:** Give the Entire Provide block as usually.

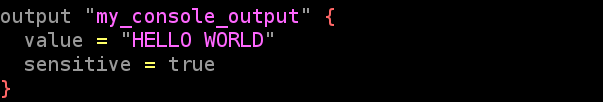


**TERRAFORM OUTPUT VALUES**

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.

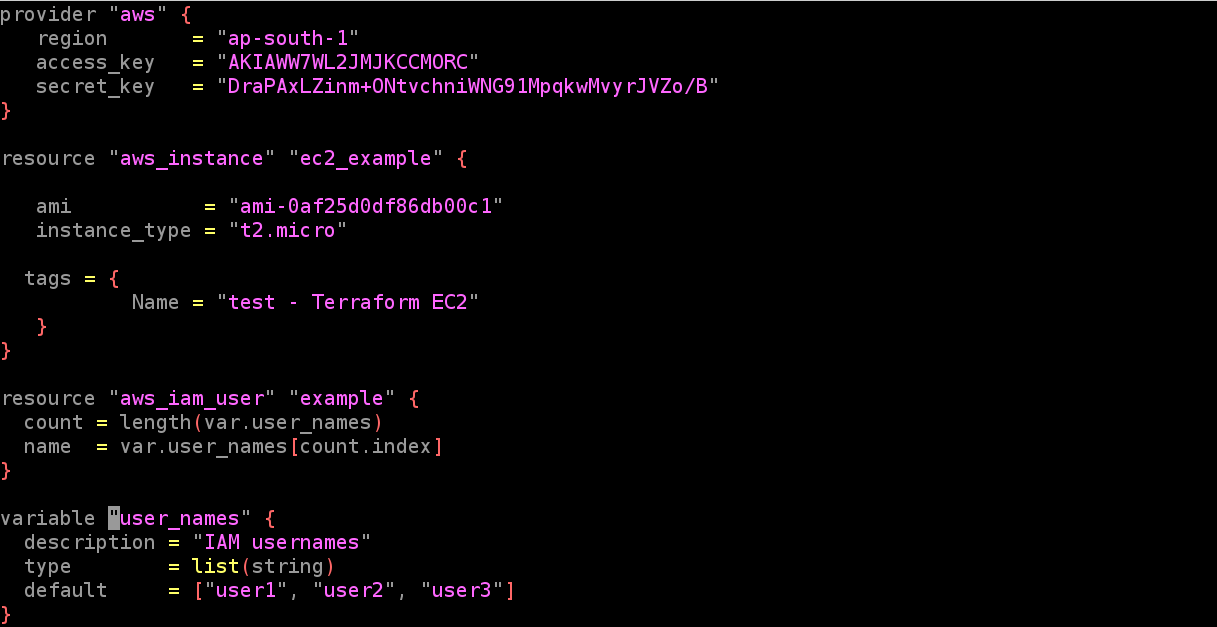


Now if you want to hide the sensitive info (like IP) use the key called sensitive.



**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.



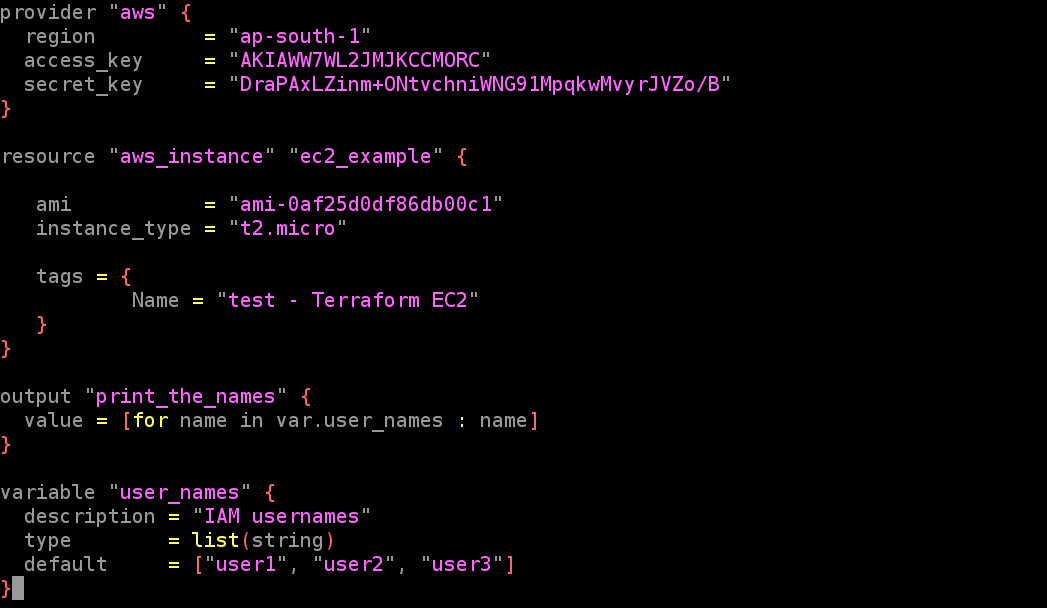
**LOOPS WITH FOR\_EACH**

* 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.



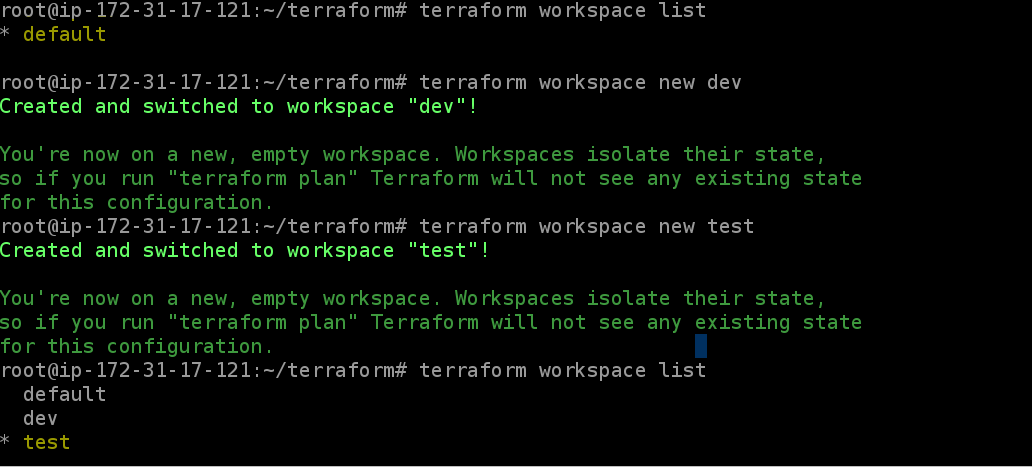
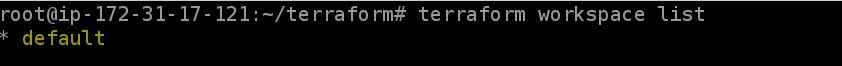
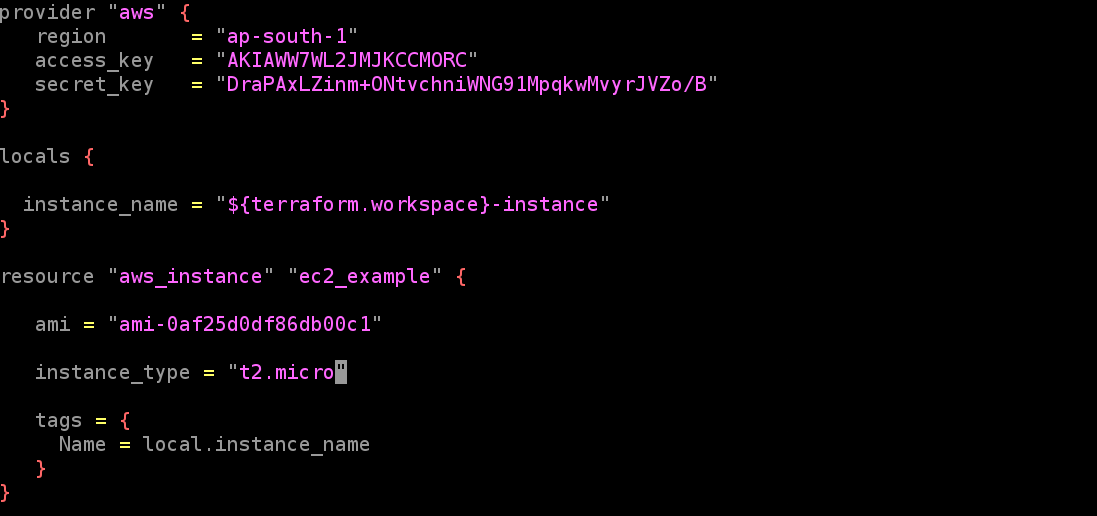
**FOR LOOP**

* 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



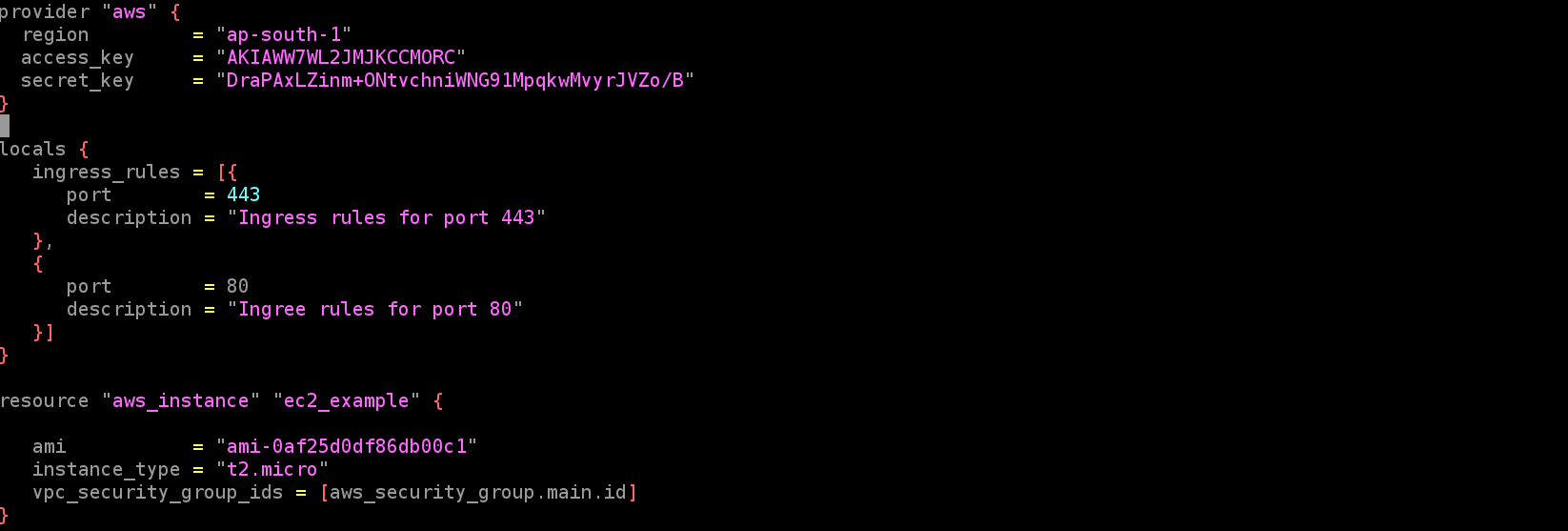
**TERRAFORM WORKSPACE**

* 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



**DYNAMIC BLOCK**

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



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

} ]

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.2.03.2"

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

database\_name = "mydb"

master\_username = "foo"

master\_password = "bar"

backup\_retention\_period = 5

preferred\_backup\_window = "07:00-09:00"

}

ALIAS & PROVIDER:

provider "aws" {

alias = "east"

region = "us-east-1"

}

resource "aws\_instance" "example" {

ami = "ami-0e6329e222e662a52"

instance\_type = "t2.micro"

}

resource "aws\_instance" "example1" {

ami = "ami-0c4e4b4eb2e11d1d4"

instance\_type = "t2.medium"

provider = "aws.east"

}

LOCAL:

resource "local\_file" "abc" {

filename = “/root/abc.txt”

}

resource "local\_filw" "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

}