# Mastering Terraform

## My Complete DevOps Journey in One Guide!

- Terraform Basics & IaC Concepts
- Setup & Installation on Ubuntu, Amazon Linux
- Lifecycle Management: init → plan → apply → destroy
- State Management: S3 Backend + DynamoDB
- Terraform Modules & Providers

By Brij Mohan Singh

# TERRAFORM

- Terraform is an open source "Infrastructure as a Code" tool, created by HashiCorp.
- It was developed by Mitchell Hashimoto with Go Language in th year 2014 which
- All the configuration files used (HashiCorp Configuration Language) language for the code.
- Terraform uses a simple syntax, can provision infrastructure across multiple clouds & On premises.
- It is Cloud Agnostic it means the systems does not depends on single provider.

**Infrastructure as Code (IaC):** Terraform is a tool used for implementing Infrastructure as Code. It allows you to define and manage infrastructure configurations in a declarative manner.

**Multi-Cloud Support:** Terraform is cloud-agnostic and supports multiple cloud providers such as AWS, Azure, Google Cloud, and others. It also works with on-premises and hybrid cloud environments.

**Declarative Configuration:** Users describe the desired state of their infrastructure in a configuration file (usually written in HashiCorp Configuration Language - HCL), and Terraform takes care of figuring out how to achieve that state.

**Resource Provisioning:** Terraform provisions and manages infrastructure resources like virtual machines, storage, networks, and more. It creates and updates resources based on the configuration provided.

**State Management:** Terraform maintains a state file that keeps track of the current state of the infrastructure. This file is used to plan and apply changes, ensuring that Terraform can update resources accurately.

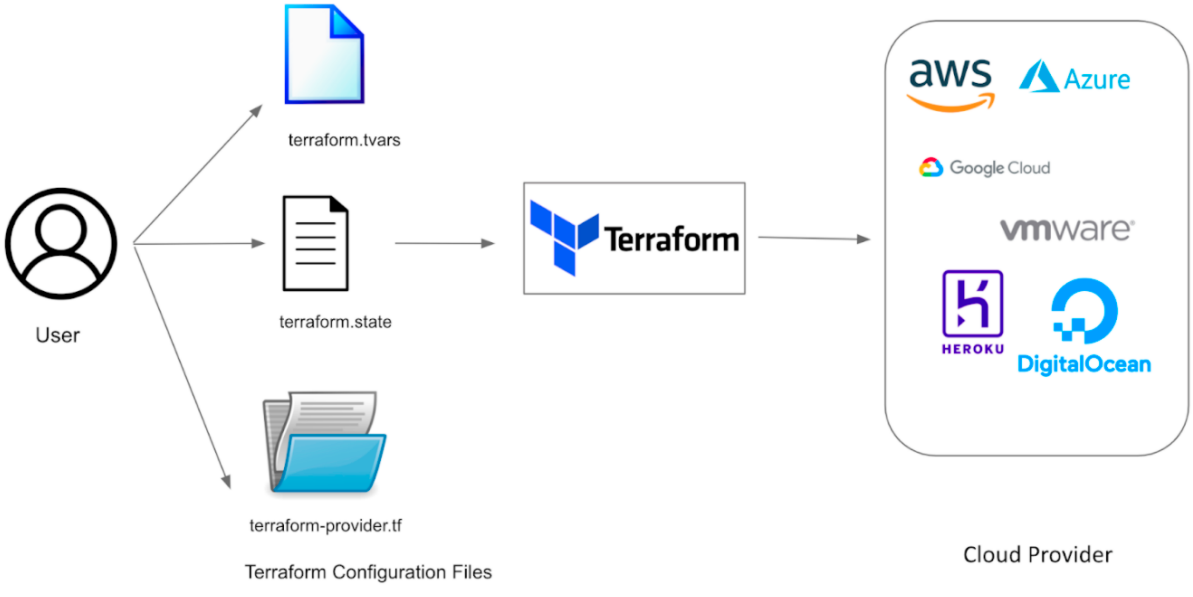
**Plan and Apply Workflow:** Before making changes, Terraform generates an execution plan, showing what actions it will take. Users review the plan and then apply it to make the changes to the infrastructure.

**Version Control Integration:** Terraform configurations can be versioned using version control systems like Git. This allows for collaboration, code review, and tracking changes over time.

**Modular Configuration:** Infrastructure configurations can be organized into modules, making it easier to reuse and share components across different projects.

**Community and Ecosystem:** Terraform has a vibrant community and a rich ecosystem of modules and providers contributed by the community, making it easier to leverage pre-built solutions for common infrastructure components.

**Immutable Infrastructure:** Terraform encourages the concept of immutable infrastructure, where changes to infrastructure are made by replacing existing resources rather than modifying them in place.




## WHAT IS IAAC:

- Infrastructure as Code (IaC) is a practice in DevOps that involves managing and provisioning infrastructure resources using code and automation.
- Server automation and configuration management tools can often be used to achieve IaC. There are also solutions specifically for IaC.
- By using these IAAC we can automate the creation of Infrastructure instead of manual process.
- IaC brings the principles of software development to infrastructure management, allowing for more streamlined and agile operations.
- IaC tools, such as Terraform or Ansible, automate the provisioning and management of infrastructure resources. By defining infrastructure as code, you can create scripts or playbooks that automatically create, configure, and manage your infrastructure in a consistent and repeatable manner.

## ALTERNATIVES OF TERRAFORM:

- AWS -- > CFT (JSON/YAML)

- AZURE -- > ARM TEMPLATES (JSON)
- GCP -- > CLOUD DEPLOYMENT MANAGER (YAML/ PYTHON)
- PULUMI -- (PYTHON, JS, C#, GO & TYPE SCRIPT)
- ANSIBLE -- > (YAML)
- PUPPET
- CHEF
- VAGRANT
- CROSSPLANE

## TERRAFORM SETUP IN UBUNTU:

- 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

## TERRAFORM SETUP IN AMAZON LINUX:

- sudo yum-config-manager --add-repo https://rpm.releases.hashicorp.com/AmazonLinux/hashicorp.repo
- sudo yum -y install terraform

## TERRAFORM LIFECYCLE:

The Terraform lifecycle refers to the sequence of steps and processes that occur when working with Terraform to manage infrastructure as code. Here's an overview of the typical Terraform lifecycle:

**Write Configuration:**

- Users define their infrastructure in a declarative configuration language, commonly using HashiCorp Configuration Language (HCL).

**Initialize:**

- Run terraform init to initialize a Terraform working directory. This step downloads the necessary providers and sets up the backend.

**Plan:**

- Run terraform plan to create an execution plan. Terraform compares the desired state from the configuration with the current state and generates a plan for the changes

required to reach the desired state.

Review Plan:

- Examine the output of the plan to understand what changes Terraform intends to make to the infrastructure. This is an opportunity to verify the planned changes before applying them.

Apply:

- Execute terraform apply to apply the changes outlined in the plan. Terraform makes the necessary API calls to create, update, or delete resources to align the infrastructure with the desired state.

Destroy (Optional):

- When infrastructure is no longer needed, or for testing purposes, run terraform destroy to tear down all resources created by Terraform. This is irreversible, so use with caution.

## CREATING EC2 INSTANCE:

```
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"
  }
}
```

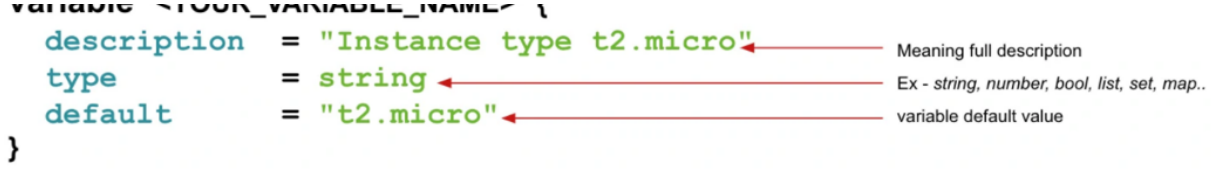
Lets assume if we have multiple instances/resources in a terraform, if we want to delete a single instance/resorce first we have to check the list of resources present in

main.tf file using **terraform state list**so it will gives the list of entire resources

to delete particular resorce: **terraform destroy -target=aws_instance.key[0]**

## TERRAFORM VARIABLE TYPES:

variable "<YOUR_VARIABLE_NAME>" {




Input Variables serve as parameters for a Terraform module, so users can customize behavior without editing the source.

Output Values are like return values for a Terraform module. Local Values are a convenience feature for assigning a short name to an expression.

## TERRAFORM STRING:

It seems like your question might be incomplete or unclear. If you are looking for information about working with strings in Terraform, I can provide some guidance.

In Terraform, strings are used to represent text data and can be manipulated using various functions and operators

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

**TERRAFORM NUMBER:** The number type can represent both whole numbers and fractional 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"
```

```
    count = var.instance_count

    tags = {
            Name = "Terraform EC2"
    }
}

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

**TERRAFORM BOOLEAN:** a boolean represents a binary value indicating either true or false. Booleans are used to express logical conditions, make decisions, and control the flow of Terraform configurations. In HashiCorp Configuration Language (HCL), which is used for writing Terraform configurations, boolean values are written as true or false.

```
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"
    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/TUPLE:

```
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"
    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/OBJECT:

```
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"
  }
}
```

## 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.

We are going to take the same example by declaring a list(string) and adding three users to it - user1, user2, user3

Use the above ec2 block if you want

```
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"]
}
```

## 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.

```
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"]
}
```

## LOOPS WITH COUNT:

we need to use count but to use the count first we need to declare collections inside our file.

```
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"]
}
```

## LAUNCH EC2 INSTANCE WITH SG:

```
resource "aws_security_group" "demo-sg" {
  name = "sec-grp"
  description = "Allow HTTP and SSH traffic via Terraform"

  ingress {
    from_port   = 80
```

```
    to_port     = 80
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }
  ingress {
    from_port   = 22
    to_port     = 22
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }

  egress {
    from_port   = 22
    to_port     = 22
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }
  egress {
    from_port   = 0
    to_port     = 0
    protocol    = "-1"
    cidr_blocks = ["0.0.0.0/0"]
  }
}
```

```
provider "aws" {
region = "us-east-1"
access_key = "AKIARSPNELGYCQNIC7XU"
secret_key = "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"
}

resource "aws_instance" "key" {
ami = "ami-0aa7d40eeae50c9a9"
instance_type = "t2.micro"
vpc_security_group_ids = [aws_security_group.demo-sg.id]
tags = {
Name = "auto-instance"
}
}
```

**TERRAFORM CLI:** to pass values form command line during run time

```
provider "aws" {

}

resource "aws_instance" "two" {

      ami = "ami-0715c1897453cabd1"

      instance_type = var.instance_type

      tags = {

          Name = "web-server"

      }

}

variable "instance_type" {

}
```

terraform apply --auto-approve -var="instance_type=t2.micro"

terraform destroy --auto-approve -var="instance_type=t2.micro"

**TERRAFORM OUTPUTS:** used to show the properties/metadata of resources

```
provider "aws" {

}

resource "aws_instance" "two" {

    ami = "ami-0715c1897453cabd1"

    instance_type = "t2.micro"

    tags = {

        Name = "web-server"

    }

}

output "abc" {

    value = [aws_instance.two.public_ip, aws_instance.two.public_dns,
    aws_instance.two.private_ip]

}
```

## ALIAS & PROVIDERS:

```
provider "aws" {

region = "us-east-1"

}

resource "aws_instance" "one" {

ami = "ami-0715c1897453cabd1"

instance_type = "t2.micro"

tags = {

Name = "web-server"

}

}

provider "aws" {

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

```
alias = "south"

}

resource "aws_instance" "two" {

provider = "aws.south"

ami = "ami-0607784b46cbe5816"

instance_type = "t2.micro"

tags = {

Name = "web-server"

}

}
```

## TERRAFORM WORKSPACE:

In Terraform, a workspace is a way to manage multiple instances of your infrastructure configurations. Workspaces allow you to maintain different sets of infrastructure within the same Terraform configuration files. Each workspace has its own state, variables, and resources, allowing you to manage and deploy distinct environments or configurations.

**Default Workspace:**

- When you initialize a Terraform configuration without explicitly creating a workspace, you are in the default workspace. The default workspace is often used for the main or production environment.

**Create a Workspace:**

- You can create additional workspaces using the terraform workspace new

**List Workspaces:**

- To see a list of available workspaces, you can use: terraform workspace list

**Select a Workspace:**

- Use the terraform workspace select command to switch between workspaces: terraform workspace select dev

**Destroy Specific Workspace:**

- You can destroy resources for a specific workspace using: terraform workspace select dev && terraform destroy

## TERRAFORM CODE TO CREATE S3 BUCKET:

```
resource "aws_s3_bucket" "one" {
 bucket = "my-bucket-name"
}

resource "aws_s3_bucket_ownership_controls" "two" {
 bucket = aws_s3_bucket.one.id
 rule {
  object_ownership = "BucketOwnerPreferred"
 }
}

resource "aws_s3_bucket_acl" "three" {
 depends_on = [aws_s3_bucket_ownership_controls.two]

 bucket = aws_s3_bucket.one.id
 acl    = "private"
}

resource "aws_s3_bucket_versioning" "three" {
bucket = aws_s3_bucket.one.id
versioning_configuration {
status = "Enabled"
}
```

```
}
```

## TERRAFORM CODE TO CREATE VPC:

```
resource "aws_vpc" "abc" {
   cidr_block = "10.0.0.0/16"
   instance_tenancy = "default"
   enable_dns_hostnames = "true"
   tags = {
      Name = "my-vpc"
   }
}

resource "aws_subnet" "mysubnet" {
   vpc_id = aws_vpc.abc.id
   cidr_block = "10.0.0.0/16"
   availability_zone = "ap-south-1a"
   tags = {
      Name = "subnet-1"
   }
}

resource "aws_internet_gateway" "igw" {
   vpc_id = aws_vpc.abc.id
   tags = {
      Name = "my-igw"
   }
```

```
}


resource "aws_route_table" "myrt" {
   vpc_id = aws_vpc.abc.id
   route {
      cidr_block = "0.0.0.0/0"
      gateway_id = aws_internet_gateway.igw.id
   }
   tags = {
      Name = "my-route-table"
   }
}
```

## TERRAFORM CODE TO CREATE EBS:

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


 tags = {
   Name = "Volume-1"
 }
}
```

## TERRAFORM CODE TO CREATE EFS:

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

```
resource "aws_efs_file_system" "foo" {
 creation_token = "my-product"
 tags = {
 Name = "swiggy-efs"
 }
}
```

## TERRAFORM MODULES:

is a container where you can create multiple resources. Used to create .tf files in the directory structure.

**main.tf**

```
module "my_instance_module" {
 source = "./modules/instances"
 ami = “ami-0a2457eba250ca23d"
 instance_type = "t2.micro"
 instance_name = " rahaminstance"
}
module "s3_module" {
source = "./modules/buckets"
bucket_name = "rahamshaik009988"
}
```

**provider.tf**

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

```
}
```

**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

}
```

validate: will check only configuration

plan: will check errors on code

apply: will check the values

## TERRAFORM ADVANTAGES:

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

## TERRAFORM DISADVANTAGES:

- Currently under development. Each month, we release a beta version.
- There is no error handling
- There is no way to roll back. As a result, we must delete everything and re-run code.
- A few things are prohibited from import.
- Bugs

HashiCorp

# Terraform

- TERRAFORM WAS FIRST INTRODUCED IN JULY 2014 WHICH SUPPORTED AWS & DIGITAL OCEAN OUT THAT TIME.
- INVENTED BY MITCHEL HASHIMOTO, WRITTEN IN GO LANGUAGE
- TERRAFORM IS USEFUL IN CASE OF MULTI-CLOUD AND HYBRID CLOUD
- TERRAFORM USES HASHICORP CONFIGURE LANGUAGE (HCL), IT IS SIMILAR TO JSON BUT EASY AND HUMAN READABLE.
- IAC: INFRASTRUCTURE AS A CODE (IAC) IS THE MANAGING AND PROVISIONING OF INFRASTRUCTURE THROUGH CODE INSTEAD OF THROUGH MANUAL PROCESS

- THERE ARE TWO WAYS TO APPROACH IAC

  1. IMPERATIVE  2. DECLARATIVE

IMPERATIVE: APPROACH DEFINES THE SPECIFIC COMMANDS NEEDED TO ACHEIVE THE DESIRED CONFIGURATION AND THOSE COMMANDS THEN NEED TO BE EXECUTED IN THE CORRECT ORDER

IN MY LANGUAGE: IMPERATIVE KE ANDAR CHEEZE SEQUENCE ME PROPER CONFIGURE HONI CHAHIYE, JAISE KHANA BANANA FIR KHANA NAKI PEHLE KHANA FIR BANANA WRONG

DECLARATIVE: APPROACH DEFINES THE DESIRED STATE OF THE SYSTEM, INCLUDING WHAT RESOURCES YOU NEED AND ANY PROPERTIES THEY SHOULD HAVE AND AN IAC TOOL WILL CONFIGURE IT FOR YOU

TERRAFORM IS A DECLARITIVE APPROACH: NO NEED FOR SEQUENCE WO APNE AAP SAMAJHLEGA

IT IS IMPORTANT TO UNDERSTAND THE DIFFERENCE BETWEEN CONFIGURATION MANAGEMENT TOOL AND IAC TOOL:

> ANSIBLE, CHEF, PUPPET ARE CONFIGURATION MANAGEMENT TOOL WHICH MEANS THEY ARE PRIMARILY DESIRED TO INSTALL AND MANAGE SOFTWARE ON EXISTING SERIES

> TERRAFORM AND CLOUDFORMATION ARE IAC TOOLS WHICH ARE DESIGNED TO PROVISION SERVERS AND INFRASTRUCTURE THEMSELVES

> YOU CAN USE IAC AND CMT ALLTOGETHER, FOR EG. YOU COULD USE TERRAFORM TO CREATE A NEW EC2 INSTANCE ON AWS, THEN TERRAFORM CAN CALL ANSIBLE TO INSTALL AND CONFIGURE SOFTWARE AND APPLICATIONS ON THE EC2 INSTANCES

BENEFITS OF TERRAFORM

1. SUPPPORTS ALMOST ALL CLOUD PROVIDERS AND CAN MANAGE INFRASTRUCTURE ON ALL CLOUDS

2. TERRAFORM HAS A SIMPLE LANGUAGE KNOWN AS HCL (HASHICORP LANGUAGE)

3. EASY TO INTEGRATE WITH CONFIGURATION MANAGEMENT TOOLS

4. IT IS EASILIY EXTENSIBLE WITH PLUGINNS

5. IT IS FREE

6. TERRAFORM KEEP TRACKS OF YOUR REAL INFRASTRUCTURE IN A STATE FILE

7. RE-USE CODE (CAN RE-USE CODES IN TERRAFORM)

## INSTALLATION OF TERRAFORM

1. GO TO TERRAFORM OFFICIAL WEBSITE (TERRAFORM.IO) DOWNLOAD TERRAFORM SELECT WINDOWS AND DOWNLOAD 386 AND A ZIP FILE WILL DOWNLOAD

2. GO TO FILE MANAGER THIS PC C DRIVE, GO IN PROGRAM FILES

3. IN THE PROGRAM FILES RIGHT CLICK OF YOUR MOUSE AND MAKE A NEW FOLDER NAME AS TERRAFORM

4. DOWNLOAD WINRAR

5. OPEN THE DOWNLOAD ZIP FILE IT WILL OPEN IN WINRAR THEN CLICK ON EXTRACT TO

6. CLICK ON PLUS SIGN OF C DRIVE AND THEN CLICK ON PROGRAM FILES THEN SELECT TERRAFORM FOLDER AND CLICK ON OK

7. COPY THE PATH > C:\Program Files\TERRAFORM

8. NOW GO TO ENVIRONMENT VARIABLES

9. NOW IN PATH PASTE THE PATH

10. OPEN CMD CHECK terraform version

output> Terraform v1.7.3

on windows_386

11. WE WILL USE THE GITBASH TO RUN THE COMMANDS

# TERRAFORM COMMANDS AND EXECUTION

PROVIDERS:-

A PROVIDER IS RESPONISBLE FOR UNDERSTANDING API INTERACTIONS AND EXPOSING RESOURCES. IF AN API IS AVAILABLE, YOU CAN CREATE A PROVIDER. A PROVIDER CREATE A PLUGINN, IN ORDER TO MAKE A PROVIDER AVAILABLE ON TERRAFORM WE NEED TO MAKE A TERRAFORM INIT, THIS COMMAND DOWNLOAD ANY PLUGINNS WE NEED FOR OUR PROVIDERS.

## BASIC COMMANDS OF TERRAFORM:-

1. TERRAFORM INIT: WORKING DIRECTORY OR WILL CREATE A SETUP

2. TERRAFORM PLAN: MAKING A PLAN, WHAT TO DO, WHAT IS THE PLAN, EVEN USEFUL FOR TEAMWORK

3. TERRAFORM VALIDATE: TO CHECK THE CODE IS CORRECT OR NOT, LIKE A DRY RUN COMMAND

4. TERRAFORM APPLY: APPLY PLAN

5. TERRAFORM DESTROY: DELETE

1. TERRAFORM INIT:-

THE TERRAFORM INIT COMMAND IS USED TO INITIALIZE A WORKING DIRECTORY CONTAINING TERRAFORM CONFIGURATION FILES, IT IS SAFE TO RUN THIS COMMAND MULTIPLE TIMES, THIS COMMAND WILL NEVER DELETE YOUR EXISTING CONFIGURATION OR STATE DURING INIT, THE ROOT CONFIGURATION DIRECTORY IS CONSULTANT FOR BACKEND CONFIGURATION AND THE CHOOSEN BACKEND IS INTIALIZED USING THE GIVEN CONFIGURATION SETTING.

2. TERRAFORM PLAN:-

THE TERRAFORM PLAN COMMAND IS USED TO CREATE AN EXECUTION PLAN. TERRAFORM PERFORMS A REFRESH, UNLESS EXPLICITLY DISABLED AND THEN DETRMINES WHAT ACTIONS ARE NECCESSARY TO ARCHIVE THE DESIRED STATE SPECIFIED IN THE CONFIGURATION FILES.

3. TERRAFORM VALIDATE:-

THE TERRAFORM VALIDATE COMMAND VALIDATE THE CONFIGURATION FILES IN A DIRECRORY, REFERRING ONLY TO THE CONFIGURATION AND NOT ACCESSING ANY REMOTE SERVER SUCH AS REMOTE STATE, PROVIDER API ETC. VALIDATE RUNS CHECKS THAT VERIFY WHETHER A CONFIGURATION IS SYNTATICALLY VALID AND INTERNALLY CONSISTENT, REGARDLESS OF ANY PROVIDER VARIABLES OR EXISTING STATE. IT IS THUS USEFUL IN GENERAL VERIFIACTION OF REUSABLE MODULES INCLUDING CORRECTNESS OF ATTRIBUTES NAMES AND VALUE TYPE.

4. TERRAFORM APPLY:-

THE TERRAFORM APPLY COMMAND IS USED TO APPLY THE CHANGES REQUIRED TO REACH THE DESIRED STATE OF THE CONFIGURATION OR THE PRE-DETERMINED SET OF ACTIONS GENERATED BY A TERRAFORM PLAN EXECUTION PLAN.

5. TERRAFORM DESTROY:-

THE TERRAFORM DESTROY COMMAND IS USED TO DESTROY THE TERRAFORM MANAGED INFRASTRUCTURE.

- HCL SYNTAX IS BASIC AND SHOULD BE READABLE BY THOSE FAMILIAR WITH OTHER SCRIPTING LANGUAGES

IT HAS 3 PARTS:

1. BLOCK: - BLOCKS GROUP EXPRESSION, ARGUEMENTS AND OTHER BLOCKS INTO A LABLE STRUCTURE, WHICH EXTERNAL BLOCKS CAN THEN REFERNECE ITS CURLY BRACKET SYNTAX IS SHARED BY MOST OBJECT-ORIENTED LANGUAGES

2. ARGUMENTS: - ARGUMENTS ARE AN ABSTRACTION THAT ENABLE IT ADMINS TO ASSIGN VALUES TO DESCRIPTIVE NAMES WHICH CAN REPRESENT OR COMPUTE VALUES THEY CAN BE SIMPLE SUCH AS, A STRING OR NUMERIC VALUE , OR MORE COMPLICATED SUCH AS ARITHMATIC OR LOGICAL EXPRESSION.

3. EXPRESSIONS: - EXPRESSIONS EITHER REPRESENT OR COMPUTE VALUES THEY CAN BE SIMPLE, SUCH AS A STRING OR NUMERIC VALUE, OR MORE COMPLICATED, SUCH AS ARTHEMETIC OR LOGICAL EXPRESSIONS.

**********

PRACTICAL:

**********

- OPEN GITBASH

$ terraform version

Terraform v1.7.3

$ cd documents/

$ mkdir terraform

$ cd terraform  ( we willwork here )

$ mkdir dir1

$ cd dir1

SO LET'S CREATE HERE .tf FILES IN WHICH WE WILL WRITE OUR CODE (.tf) IS MANDATORY OR ELSE TERRAFORM CAN'T READ IT

$ vi file1.tf

IN THIS FILE WE FIRST NEED A CODE TO CONNECT WITH US PROVIDER WHICH IS AWS, FOR THAT WE HAVE TO GO TO TERRAFORM WEBSITE

- GO TO GOOGLE TYPE: aws terraform provider : OPEN THE WEBSITE AND THERE RELATED TO AWS ALL THE WORK CODE WILL BE THERE, LIKE THIS WE CAN SEE ALL PROVIDERS CODES

**********

```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"        #(This will pickup aws related provider from hashicorp website)
      version = "~> 5.0"
    }
  }
}
```

:wq

**********

$ ls - a

(we can see there is no file but after we will do init here it will initialize all plugins need for aws)

$ terraform init

output>

Initializing provider plugins..

Terraform has been successfully initialized!  (ok done)

$ ls -a

output>  .terraform/  .terraform.lock.hcl  file1.tf  (can see some plugins and files came and these are hidden files)

(all the plugins are saved in .terraform file)

$ cd .terraform/presstab button till the .exe file

$ cd .terraform/providers/registry.terraform.io/hashicorp/aws/5.36.0/windows_386/terraform-provider-aws_v5.36.0_x5.exe

$ cd dir1

# NOW LETS CREATE A LOGIN WITH THE AWS ACCOUNT

GO TO YOUR AWS ACCOUNT GO IN IAM ROLE CLICK ON USER AND ADD USER > Name it  > terraform user

#  THEN CLICK ON (Attach policies directly)

# GIVE THE ADMINISTRATOR-ACCESS AND NEXT AND CREATE USER

# CREATE A ACCESS KEY AND SECRET KEY

# OPEN FILE vim file1.tf

paste below the provider

```
provider "aws" {
  region     = "ap-south-1"                ( can give the region where you want to do work )
  access_key = "my-access-key"             (give the access key here)
  secret_key = "my-secret-key"             (give the secret key here)
}
```

FOR EXAMPLE:

**********

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

provider "aws" {                          #( THE FILE SHOULD LOOK LIKE THIS )
  region     = "us-west-2"
  access_key = "AKIA23YWPX7YRKN5XH4I"
  secret_key = "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"
}                                         ( terraform will login to aws from this )
```

**********

$ terraform plan

OUTPUT> No changes. Your infrastructure matches the configuration.  (OK PERFECT)

$ terraform apply

OUTPUT> Apply complete! Resources: 0 added, 0 changed, 0 destroyed.

# NOW LET'S SEE IS MY TERRAFORM IS LOGIN WITH THE AWS

# NOW WE WILL CREATE AN EC2 INSTANCE FROM THE TERRAFORM SO WE NEED AN AMI ID OF THE O/S ON WHICH WE WANT TO CREATE

# COPY THE AMI ID OF THE O/S LIKE LINUX / UBUNTU AND THEN OPEN THE VIM FILE1.TF

<LINUX AMI ID> : ami-0449c34f967dbf18a

THEN OPEN THE VIM FILE1.TF

**********

```
resource "aws_instance" "s1" {
  ami           = "ami-0449c34f967dbf18a"
  instance_type = "t2.micro"

  tags = {
    Name = "server1"                          #(IF YOU WANT MORE THINGS GO IN THE TERRAFORM ARGUEMENTS AND USE)
  }
}
```

PASTE THIS CODE BELOW THE PREVIOUS CODE

**********

FINAL FILE CODE :

**********

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

provider "aws" {
  region     = "ap-south-1"
  access_key = "AKIA23YWPX7Y262MS6VW"
  secret_key = "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"
}
```

```
resource "aws_instance" "s1" {
  ami           = "ami-0449c34f967dbf18a"
  instance_type = "t2.micro"

  tags = {
    Name = "server1"
  }
}
```

**********

# NOW LETS DRY RUN THE CODE FIRST

$ terraform validate

OUTPUT> Success! The configuration is valid.  (OK PERFECT CODE IS SUCCES)

$ terraform plan  (IT WILL SHOW WHAT WILL HAPPEN WHAT THINGS WILL BE DONE IN + SIGN)

# NOW WE WILL APPLY THIS PLAN

$ terraform apply

$ type yes

OUTPUT>

Apply complete! Resources: 1 added, 0 changed, 0 destroyed  (ok the instance is created)

# NOW IF YOU WANT TO CREATE TWO/THREE INSTANCES SO WE JUST HAVE TO ADD IN THE SCRIPT LETS DO IT

# LETS DESTROY THE WORK FIRST

$ terraform destroy  ( we can even specifically delete the work we will do that later )

# INSTANCE WILL BE DELETED AUTOMATICALLY

# NOW CREATING 2 INSTANCES

SIMPLY COPY THE INSTANCE MAKING CODE AND PASTE BELOW IN THE VIM FILE.TF

**********

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

provider "aws" {
  region     = "ap-south-1"
  access_key = "AKIA23YWPX7Y262MS6VW"
  secret_key = "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"
}

resource "aws_instance" "s1" {
  ami           = "ami-0449c34f967dbf18a"
  instance_type = "t2.micro"

  tags = {
    Name = "server1"
  }
}
resource "aws_instance" "s2" {
  ami           = "ami-0449c34f967dbf18a"
  instance_type = "t2.micro"

  tags = {
    Name = "server2"
```

```
 }
}
```

**********

# NOW WE WILL LAUNCH 3 INSTANCES WITH JUST COUNT SYNTAX IN VIM.TF FILE

**********

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

provider "aws" {
 region     = "ap-south-1"
 access_key = "AKIA23YWPX7Y262MS6VW"
 secret_key = "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"
}

resource "aws_instance" "s1" {
 ami           = "ami-0449c34f967dbf18a"
 instance_type = "t2.micro"
 count         = 3
 }
}
```

**********

FINAL CODE IN THE FILE WILL LOOK LIKE THIS

NOW THE THREE INSTANCES WILL CREATE

$ terraform plan

$ terraform apply

OUTPUT> Apply complete! Resources: 3 added, 0 changed, 0 destroyed  (ok 3 instances are created)

---

- LET'S CREATE A GITHUB REPOSITORY USING GITHUB AS A PROVIDER
- GO TO GITHUB ACCOUNT AND FOLLOW THESE STEPS THEN CLICK ON SETTING > DEVELOPER SETTING > PERSONAL ACCESS TOKENS > TOKEN CLASSIC > CREATE NEW TOKEN > EXPIRATION=7 DAYS > admin.org, repo, delete repo > GENERATE TOKEN

$ vim providers.tf

**********

provider "aws" {

region = "ap-south-1"

}

provider "github"  {

token = "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"

}

**********

$ terraform init  ( it will then add github as a provider )

NOW RESOURCE FILE CREATE

$ nano resources.tf

**********

resource "github_repository" "myrepo" {

name  = "myrepo1"

description = "My repo for terraform"

visibility = "public" }

**********

$ terraform plan

o/p:> github_repository.myrepo will be created

$ terraform apply

o/p:> github_repository.myrepo: Creation complete after 5s [id=myrepo1]

(NOW IF YOU WILL SEE THE GITHUB YOU CAN SEE THE NEW REPO IS CREATED)

---------****---------

# VARIABLES IN TERRAFORM

---------****---------

## TYPES:

1. INPUT VARIABLES:
- input variables in terraform let users specify values when creating their infrastructure.
- input variables are declared through "variables" block, but there are many other ways to define a variable
- variables can hold data of different datatypes.
2. OUTPUT VARIABLES
3. LOCAL VARIABLES

VARIABLE BLOCK ATTRIBUTES:

1. TYPE - to identify the type of the variable being declared.

2. DEFAULT - default value in case the value is not provided explicitly

3. DESCRIPTION - a description of the variable, this description is also used to generate documentation for the module.

4. VALIDATION - to define validation rules.

5. SENSITIVE - a boolean value, if true, Terraform masks the variable's value anywhere it displays the variable.

NOTE: all the attributes are optional

EXAMPLE ON INPUT VARIABLES: WE DON'T WANT THE VALUES OF THIS INSTANCE, WE ARE CREATING SHOULD BE THE HARDCORED SO LET'S DO THIS WITH VARIABLES

$ vim variable.tf

**********

```
resource "aws_instance" "s10" {
  ami           = var.os
  instance_type = var.size

  tags = {
    Name = var.name
}
}

variable "os" {
  type = string
  default = "ami-0449c34f967dbf18a"
  description = "this is my ami id"
 }

variable "size" {
  default = "t2.micro"
}

variable "name" {
  default = "terraformEC2"
}
```

**********

$ terraform plan

$ terraform apply -auto-approve

(YOU CAN SEE A INSTANCE WILL BE CREATED AND THE VALUES ARE PICKED UP FROM OUR VARIABLE)

(THAT MESNS THESE ARE THE SOFT CORE VALUES WE GAVE AND CAN REUSE IT AGAIN AND AGAIN)

---

- NOW LETS CREATE THE S3 BUCKET USING THE TERRAFORM

$ vim resource.tf

**********

```
resource "aws_instance" "s10" {
  ami           = var.os
  instance_type = var.size

  tags = {
    Name = var.name
}
}

resource "aws_s3_bucket" "bucket" {
    bucket = var.bucketname
}
```

**********

- NOW LET'S ADD THE VARIABLE  THE VARIABLE FOR THIS IN THE FILE OF VARIABLE

**********

```
variable "os" {
  type = string
  default = "ami-0449c34f967dbf18a"
  description = "this is my ami id"
 }

variable "size" {
```

```
  default = "t2.micro"
}

variable "name" {
  default = "terraformEC2"
}

variable "bucketname" {

}
```

**********

$ terraform plan  (NOW THE TERRAFORM WILL ASK YOU THE BUCKET NAME AS YOU KEPT THE VARIABLES ATTRIBUTES EMPTY)

(USING THE CLI PROMPT)

O/P:>

```
var.bucketname
  Enter a value: mycloudvrushank
```

O/P:>  + bucket  = "mycloudvrushank"

(YOU CAN SEE THE BUCKET WILL BE CREATE IF I RUN THE TERRRAFORM APPLY)

WHAT IF I RUN THIS COMMAND AGAIN SO IT WILL ASK ME THE SAME LIKE GIVE THE NAME SO IF I WANT TO KEEP IT HARD-CORE VALUE:

```
$ terraform plan -var="bucketname=mycloudvrushank"
```

(NOW YOU WILL SEE THE TERRAFORM WILL NOT ASK US THE BUCKET NAME AND WILL SHOW US THE PLAN)

```
$ terraform apply -auto-approve -var="bucketname=mycloudvrushank"
```

(NOW CHECK IN THE AWS S3 YOU WILL FIND A BUCKET THERE)

---

- NOW LET'S CREATE A AWS IAM USER WITH THE HELP OF TERRAFORM

$ vim iam.tf

**********

resource "aws_iam_user" "myuser" {

name = var.username

}

**********

NOW DEFINE VARIABLE

**********

variable "username" {

}

**********

$ terraform plan

$ terraform apply

---

## OUTPUT VARIABLE:-

---

- TERRAFORM OUTPUTS: GET ENDPOINTS, IP ADDRESSES, DATABASES USER CREDENTIALS IN TERMINAL USING OUTPUTS
- TO GET ALL OUTPUTS DECLARE RUN: TERRAFORM OUTPUT

(IF I WANT AN OUTPUT OF ANYTHING LIKE IPADDRESS DNS ETC OF AN INSTANCE OR SOME OTHER SERVICES SO I CAN USE THE OUTPUT I CAN SEE THOSE THINGS ON THE CLI)

LETS DO AN PRACTICAL ON IT:

**********

output "IPaddress"  {

 value = aws_instance.s10.public_ip

}

**********

$ terraform plan

o/p:> + IPaddress = 10.0.0.12      (this is the way you can get the ouput)

--

. NOW LETS TRY TO GET THE DNS FROM THE OUTPUT VARIABLE

$ vim out.tf

**********

output "DNS"  {

 value = aws_instance.s10.public_dns

}

**********

$ terraform plan

o/p:>  Changes to Outputs:

  + DNS =

**********

# STATE FILE TERRAFORM

**********

WHAT IS STATE FILE: -

- State file in Terraform is where Terraform records the information of the infrastructure it has created.
- State file is important because it helps Terraform update and manage existing infrastructure instead of creating new instances or resources.
- Advantages of using the state file include updating existing infrastructure and destroying infrastructure in a controlled manner.
- One drawback of the state file is that it records sensitive information, such as passwords, by default.

Terraform State File

---

Terraform is an Infrastructure as Code (IaC) tool used to define and provision infrastructure resources. The Terraform state file is a crucial component of Terraform that helps it keep track of the resources it manages and their current state. This file, often named terraform.tfstate, is a JSON or HCL (HashiCorp Configuration Language) formatted file that contains important information about the infrastructure's current state, such as resource attributes, dependencies, and metadata.

---

Advantages of Terraform State File:

---

Resource Tracking: The state file keeps track of all the resources managed by Terraform, including their attributes and dependencies. This ensures that Terraform can accurately update or destroy resources when necessary.

Concurrency Control: Terraform uses the state file to lock resources, preventing multiple users or processes from modifying the same resource simultaneously. This helps avoid conflicts and ensures data consistency.

Plan Calculation: Terraform uses the state file to calculate and display the difference between the desired configuration (defined in your Terraform code) and the current infrastructure state. This helps you understand what changes Terraform will make before applying them.

Resource Metadata: The state file stores metadata about each resource, such as unique identifiers, which is crucial for managing resources and understanding their relationships.

---

Disadvantages of Storing Terraform State in Version Control Systems (VCS):

---

Security Risks: Sensitive information, such as API keys or passwords, may be stored in the state file if it's committed to a VCS. This poses a security risk because VCS repositories are often shared among team members.

Versioning Complexity: Managing state files in VCS can lead to complex versioning issues, especially when multiple team members are working on the same infrastructure.

Overcoming Disadvantages with Remote Backends (e.g., S3):

A remote backend stores the Terraform state file outside of your local file system and version control. Using S3 as a remote backend is a popular choice due to its reliability and scalability. Here's how to set it up:

---

WHY WE NEED TO STORE TERRAFORM STATE FILE IN THE S3 BUCKET :->

---

SO AS WE CAN'T UPLOAD IT ON THE GITHUB BECAUSE THE STATE FILE CONTAINS SENSITIVE INFORMATION LIKE OUR ACCESS KEYS AND INFRASTRUCTURE DETAILS SO WE DONT WANT TO SHARE THIS WITH ANYONE AND LOCALLY IT'S NOT A GOOD PRACTICE TO KEEP IT SO THATS WE KEEP IT TO THE S3 BUCKET IN OUR AWS SO IT WOULD BE VERY SAFE THERE ONLY WE CAN ACCESS OUR AWS AND THE TERRAFORM.TFSTATE FILE

---

PRACTICAL: SO LETS SETUP THE BACKEND AS S3 WITH THE PROVIDERS

---

$ terraform version

SO WHEN YOU DO ANY KIND OF APPLY IN THE AWS INFRASTRUCTURE A TERRAFORM STATE FILE CREATES SO YOU CAN CHECK IT BY

# ls

OUTPUT> terrafrom.tfstate

# cat terrafrom.tfstate > IT WILL SHOW U ALL THE DETAILS ABOUT THE INFRASTRUCTURE YOU CREATED SO WHEN YOU WILL CREATE THE .tf FILE FOR THE MAKING OF THE INSTANCE THEN YOU CAN DO THE TERRAFORM APPLY YOU CAN SEE THAT THE .tfstate FILE IS GENERATED AND THEN IT CARRIES ALL THE INFORMATION IN DETAILS OPEN IT AND SEE.

SO NOW I CANNOT PUSH IT TO THE GITHUB AS THEN I HAVE TO PUSH THE ENTIRE .tf FILE WHICH IS WRONG AS BECAUSE IT CARRIES THE SENSITIVE INFORMATION

SO THE STATE FILE IS THE HEART OF TERRAFORM AS IF I WILL DELETE THE STATE FILE AND WILL DO THE terraform apply COMMAND SO IT WILL AGAIN MAKE THE NEW EC2 INSTANCE BUT INSTEAD OF THIS IT COULD SAY THE INSTANCE IS ALREADY THERE SO THATS WHY STATE FILE IS VERY IMPORTANT.

NOW STORING THE STATE FILE IN THE S3 BUCKET

Storing Terraform state files in an S3 bucket is a recommended best practice because it provides a central location for storing and managing your infrastructure's state files.

PRACTICAL:

# WE WILL CREATE THE S3 BUCKET SO THAT WE CAN USE IT WITH THE CONFIGURATION DETAILS TO STORE THE STATE FILE IN THE S3 BUCKET.

vi main.tf

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

```
terraform {
  backend "s3" {
    bucket         = "MYBUCKET"                    # paste the bucket name you created
    key            = "terraform/terraform.tfstate"       # terraform is the folder and inside that folder the terraform.tfstate will be stored
    region         = "ap-south-1"
  }
}
```

```
provider "aws" {
```

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

resource "aws_instance" "s1" {
  ami           = "ami-0449c34f967dbf18a"
  instance_type = "t2.micro"
}

:wq
```

# NOW WE NEED TO PROVIDE THE AWS CONFIGURATION ACCESS ABD SECRET KEY OF THE IAM USER WHICH WE MADE TO ACCESS THE AWS

TO PROVIDE THAT USE THIS COMMAND>

```
export AWS_ACCESS_KEY_ID="your-access-key"
export AWS_SECRET_ACCESS_KEY="your-secret-key"
```

NOW INITIALIZE THE TERRAFORM

```
# terraform init

# terraform apply
```

# IF YOU DONT HAVE THE BUCKET THEN YOU HAVE TO FIRST CREATE THE BUCKET THEN YOU CAN USE THAT NAME OF THE BUCKET WHERE THE STATE FILE WILL BE UPLOADED.

SO GO IN THE SAME REGION AND THEN CREATE THE BUCKET.

and you backend file will run and the tf.state file will be store in the s3 bucket

SECURING THE STATE FILE ACCESS
**********

# NOW WE WILL SECURE THE CHANGES IN THE INFRASTRUCTURE>

# WHY WE NEED TO SECURE TERRAFORM STATE FILE :->

---

SO IN THE SINGLE TERRAFORM PROJECT THERE CAN BE MULTIPLE DEVELOPERS WORK SO EVERY DEVELOPER CAN DO THE CHANGES IN THE TERRAFORM INFRA OF OUR AWS SO TO KEEP THAT DATA OF THE TERRAFORM INFRASTRUCTURE IS AUTOMATICALLY UPDATE IN THE TERRAFORM.STATE FILE SO IT ALWAYS STAYS AUTOMCATICALLY IN THE LOCALLY MEANS ON OUR LAPTOP BUT TO KEEP THAT DATA SAFE AND SECURE WE KEEP THAT FILE IN THE S3 BUCKET SO THAT THE DATA WOULD ALWAYS SAFELY STORED.

BUT ANY DEVELOPER CAN MAKE CHANGES IN OUR STATE FILE AND CHANGE OUR INFRA SO THAT IS NOT GOOD FOR US SO WHAT WE DO FOR THAT # SO WE DO LOCK THAT STATE FILE WITH THE HELP OF THE DYNAMODB TABLE.

# Terraform uses the DynamoDB table for state locking.

---

# WE WILL CREATE A TABLE OF THE DYNAMO DB AN WE WILL LOCK OUR S3 BUCKET ACCESS AND ALSO WE WILL LOCK THE CHANGES TO OUR INFRA

---

PRACTICAL :->

---

# GO TO AWS > SEARCH FOR THE DYNAMODB AND THERE U WILL SEE AND OPTION CREATE TABLESO CLICK ON IT

# GIVE THE TABLE NAME > (anyname)

# IN <Partition key> GIVE :> LockID >THIS IS MANDATORY TO GIVE

# NOW CLICK ON CREATE TABLE AND COPY THE TABLE NAME

vi main.tf

```
terraform {
  required_providers {
```

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

terraform {
  backend "s3" {
    bucket         = "MYBUCKET"
    key            = "terraform/terraform.tfstate"
    region         = "ap-south-1"
    dynamodb_table = "mytable"              # give ur table name here
  }
}

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

resource "aws_instance" "s1" {
  ami           = "ami-0449c34f967dbf18a"
  instance_type = "t2.micro"
  count = "3"
}

:WQ

$  terraform init  -reconfigure

$ terraform apply
```

NOW YOUR STATE FILE ONLY YOU CAN ACCESS AND YOU ONLY CAN MAKE THE CHANGES IT IS LOCKED

# GO TO YOUR DYNAMODB TABLE AND IN ITEMS YOU CAN SEE YOUR STATE FILE

# SO OTHER DEVELOPER CANNOT MAKE CHANGES UNTILL YOU WILL REMOVE THIS LOCK

As Terraform uses the DynamoDB table for state locking.