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.

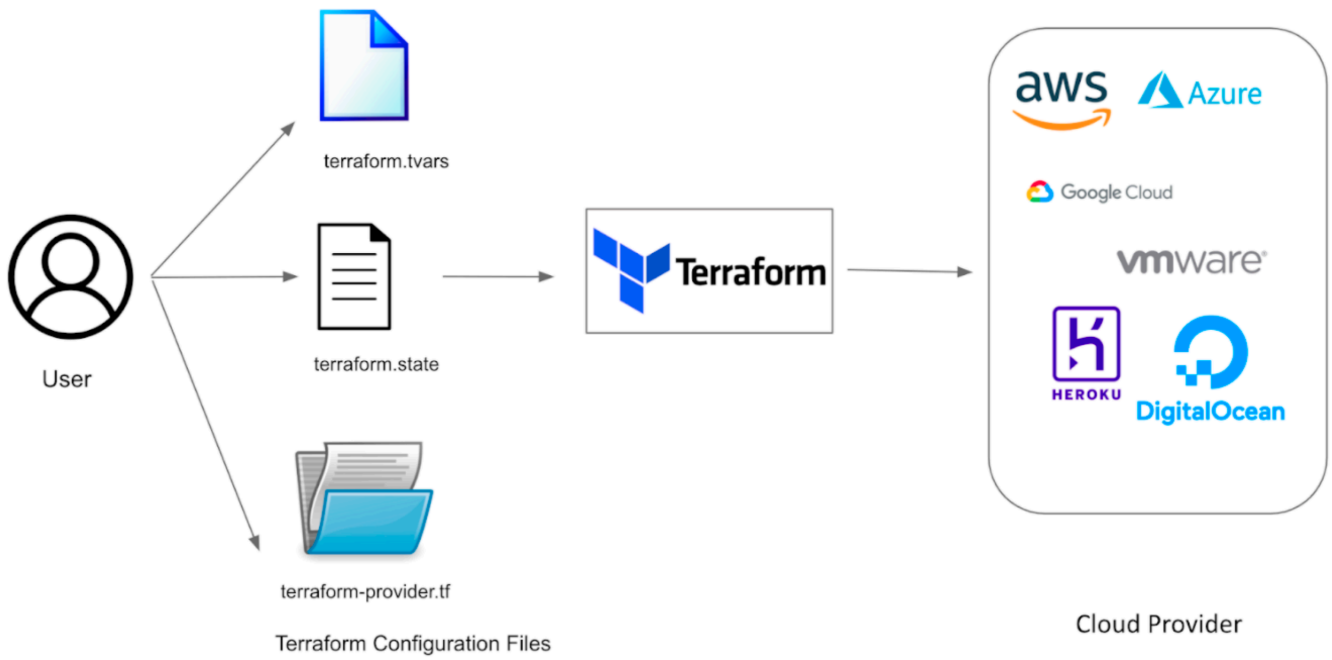
Plan and Apply Workflow: Before making changes, Terraform generates an execution plan, showing what actions it will take. Users review the plan and then apply it to make the changes to the infrastructure.

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

Meaning full description

Ex - *string, number, bool, list, set, map..*

variable default value

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:

```
    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
```

```
n = "us-east-1"
s_key = "AKIARSPNELGYCQNIC7XU"
t_key = "BWoyjrs3M7xnPbWSi7ouirtEaikeCNOdh8WFxbzB"


rce "aws_instance" "key" {
 "ami-0aa7d40eeae50c9a9"
nce_type = "t2.micro"
ecurity_group_ids = [aws_security_group.demo-sg.id]
= {
= "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

HANDS ON WORKSPACE:

vim main.tf

```
provider "aws" {

  region = var.region

  access_key = ""
```

```
  secret_key = ""
}
resource "aws_instance" "example" {
  ami           = var.ami
  instance_type = var.size
  tags = {
    Name = "demo-server"
  }
}

variable "region" {
  default = "us-east-1"
}

variable "ami" {
  default = "ami-00b8917ae86a424c9"
}

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

terraform init
terraform plan
terraform apply

This will create instance in default workspace

Create new workspace called **dev**

terraform init
terraform plan
terraform apply

This will create one more instance with same config on **dev** env

Even though there is one instance already up & running on AWS, terraform will create one more instance because we run the terraform commands from different workspace!

Create new workspace called **prod**

terraform init
terraform plan
terraform apply

This will create one more instance with same config on **prod** env

As per the above code, all the instances will get same instances names, if you wish to change it then change the instance name like below code:

```
tags = {
   Name = "example-server-${terraform.workspace}"
 }
```

If we apply the changes, that will creates the instances like below names:

- demo-server-stg
- demo-server-dev
- demo-server-prod

If i want to give different size of instance for different environment also, for that we need to modify our main.tf file as below

```
provider "aws" {
 region = var.region
}

locals {
 instance_types = {
  dev   = "t2.micro"
```

```
    test = "t2.small"
    prod  = "m4.large"
  }
}

resource "aws_instance" "demo_server" {
  ami           = var.ami
  instance_type = local.instance_types[terraform.workspace]
  tags = {
    Name = "example-server-${terraform.workspace}"
  }
}
```

If we execute the above code, that will create different sizes for instances.

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"
}
}
```

STORE ALL THE STATE FILES IN S3 BUCKET:

We can also store state file in S3 bucket, and for that create **backend.tf** file and add below code in it. Add he below code in main.tf file

```
terraform {
  backend "s3" {
    bucket = "sm7243.ccit"
    key    = "prod/terraform.tfstate"
    region = "us-east-1"
  }
}
```

Note: Bucket must be created before & Prod folder will automatically in s3 bucket.

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 state list : To list the terraform state files

terraform state show state name : used to see a particular module state file

terraform destroy -target state name : used to destroy a particular module

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

IMPORTING EXISITING RESOURCES TO TERRAFORM:

If the infra is already created manually, we can import all the data in terraform.

create **provider.tf** file

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

now use **terraform init** command

create **main.tf** file:

```
resource "aws_instance" "my-server" {
  ami           = "ami-0578f2b35d0328762"
  instance_type = "t2.micro"

  tags = {
    Name = "my-server"
  }
}
```

Note: Make sure that we have to mention existing EC2 AMI & Name on **main.tf** file

Now use **terraform import aws_instance.<instancename> <instanceid>** command to import all the data.

TERRAFORM CODE FOR ASG, VCP, SUBNETS, RT, IGW, SECURITY GROUP & EC2

vim provider.tf

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

vim vpc.tf

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

vim subnets.tf

```
resource "aws_subnet" "mysubnet1" {
    vpc_id = aws_vpc.abc.id
    cidr_block = "10.0.1.0/24"
    availability_zone = "us-east-1a"
```

```
    map_public_ip_on_launch = "true"
    tags = {
        Name = "subnet-1"
    }
}

resource "aws_subnet" "mysubnet2" {
  vpc_id     = aws_vpc.abc.id
  cidr_block = "10.0.2.0/24"
  availability_zone = "us-east-1b"
  map_public_ip_on_launch = "true"
  tags = {
    Name = "subnet-2"
  }
}
```

vim igw.tf

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

vim rt.tf

```
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"
    }
}

resource "aws_route_table_association" "public_subnet_association_1a" {
  subnet_id      = aws_subnet.mysubnet1.id
  route_table_id = aws_route_table.myrt.id
}
resource "aws_route_table_association" "public_subnet_association_1b" {
  subnet_id      = aws_subnet.mysubnet2.id
  route_table_id = aws_route_table.myrt.id
}
```

vim security.tf

```
resource "aws_security_group" "web_server" {
  name_prefix = "web-server-sg"
  vpc_id      =  aws_vpc.abc.id
```

```
ingress {
  from_port   = 22
  to_port     = 22
  protocol    = "tcp"
  cidr_blocks = ["0.0.0.0/0"]
}
ingress {
  from_port   = 80
  to_port     = 80
  protocol    = "tcp"
  cidr_blocks = ["0.0.0.0/0"]
}
 ingress {
  from_port   = 443
  to_port     = 443
  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"]
```

```
  }
}
```

vim main.tf

```
resource "aws_launch_configuration" "web_server_as" {
   image_id        = "ami-00b8917ae86a424c9"
   instance_type = "t2.micro"
   security_groups = [aws_security_group.web_server.id]

   user_data = <<-EOF
   #!/bin/bash
   sudo yum update -y
   sudo yum install httpd -y
   sudo systemctl start httpd
   sudo systemctl enable httpd
   sudo systemctl restart httpd
   sudo chmod 766 /var/www/html/index.html
   sudo echo "<html><body><h1>Welcome to Terraform Scaling.</h1></body></html>"
>/var/www/html/index.html
  EOF
 }

 resource "aws_elb" "web_server_lb"{
    name = "web-server-lb"
    security_groups = [aws_security_group.web_server.id]
    subnets = [aws_subnet.mysubnet1.id,aws_subnet.mysubnet2.id]
```

```
  listener {
   instance_port     = 8000
   instance_protocol = "http"
   lb_port           = 80
   lb_protocol       = "http"
  }
  tags = {
   Name = "terraform-elb"
  }
 }
resource "aws_autoscaling_group" "web_server_asg" {
  name                 = "web-server-asg"
  launch_configuration = aws_launch_configuration.web_server_as.name
  min_size             = 1
  max_size             = 3
  desired_capacity     = 2
  health_check_type    = "EC2"
  load_balancers       = [aws_elb.web_server_lb.name]
  vpc_zone_identifier  = [aws_subnet.mysubnet1.id, aws_subnet.mysubnet2.id]
 }
```

TERRAFORM REFRESH:

Lets assume that we created EC2 and S3 bucket using terraform, but i deleted that s3 bucket manually. now what will happen?

you can see, i have created 2 resources by using the below code and both services are running.

```
resource "aws_instance" "one" {
ami = "ami-02d7fd1c2af6eead0"
instance_type = "t2.micro"
tags = {
Name = "terraform-instance"
}
}

resource "aws_s3_bucket" "two" {
bucket = "mustafa.terraform.bucket.refresh"
}
```

here is my state list

```
[root@ip-172-31-19-166 mustafa]# terraform state list
aws_instance.one
aws_s3_bucket.two
```

now i will delete s3 bucket manually, now if i perform the same command **terraform state list**

it is giving both s3 and ec2 details

```
[root@ip-172-31-19-166 mustafa]# terraform state list
aws_instance.one
aws_s3_bucket.two
```

Now if you want to update the state list perform this command **terraform apply -refresh-only**

This will update the running resources only.

Now check the state list again

```
[root@ip-172-31-19-166 mustafa]#
[root@ip-172-31-19-166 mustafa]# terraform state list
aws_instance.one
```

now we can see it is giving only EC2 details, So if you want to create s3 bucket agian

terraform apply --auto-approve

TERRAFORM PROVISIONERS:

Terraform provisioners are used to perform custom actions and tasks on local or remote machines during resource creation or destruction. They can be used to run shell scripts, copy files, or install software. There are two types of provisioners: **generic provisioners** (file, local-exec, and remote-exec) and **vendor provisioners** (chef, habitat, puppet, salt-masterless).

1. **Local-exec Provisioner**

This provisioner executes a command on the machine from where terraform code is run.

```
resource "aws_instance" "one" {
ami = "ami-02d7fd1c2af6eead0"
instance_type = "t2.micro"
tags = {
Name = "terraform-instance"
}
provisioner "local-exec" {
    command = "echo 'Instance created with ID: ${aws_instance.one.id}' >> instance-id.txt"
  }
}
```

The above code will create a file called instance-id.txt file on our terraform server that contains new created instance id.

2. **Resource Provisioner**

This is similar to a local provisioner, which will be run on the machine where terraform is running. The only difference is that it will be run at the end of the terraform execution, while a local provisioner will be run for each resource after resource creation.

3. **Remote-exec Provisioner**

This provisioner connects to a remote machine and executes commands inside that machine. This can be helpful to configure software or install packages on the remote machine

```
resource "aws_key_pair" "sample_aws_key" {
  key_name   = "mustafa"
  public_key = file("~/.ssh/id_rsa.pub")
}
resource "aws_instance" "three" {
  ami           = "ami-0f403e3180720dd7e"
  instance_type = "t2.micro"
  key_name      = aws_key_pair.sample_aws_key.key_name
  provisioner "remote-exec" {
    connection {
      type        = "ssh"
      user        = "ec2-user"
      host        = self.public_ip
      private_key = file("~/.ssh/id_rsa")
    }
    inline = [
      "echo 'Hello, World!'",
      "echo 'This is a remote-exec provisioner.'",
      "sudo yum install httpd -y",
      "sudo systemctl start httpd",
    ]
  }
  tags = {
    Name = "abc-server"
  }
}
```

4. **File Provisioner**

Helps to copy file from local machine to the remote machine, helpful for copying configuration files etc.

```
resource "aws_key_pair" "sample_aws_key" {
  key_name   = "mustafa"
  public_key = file("~/.ssh/id_rsa.pub")
}
resource "aws_instance" "three" {
  ami           = "ami-0f403e3180720dd7e"
  instance_type = "t2.micro"
  key_name      = aws_key_pair.sample_aws_key.key_name
  tags = {
    Name = "abc-server"
  }
  provisioner "file" {
    connection {
      type        = "ssh"
      user        = "ec2-user"
      host        = self.public_ip
      private_key = file("~/.ssh/id_rsa")
    }
    source      = "config_files/nginx.conf"
    destination = "nginx.conf"
  }
}
```

5. **Null Resource Provisioner**

This provisioner does not create any resource, but can be used to trigger local-exec or remote-exec based on a condition.

A common use case for the null_resource provisioner is to execute a local script or command after creating an AWS EC2 instance. In the below example after creating an aws_instance, we use null_resource to perform a local-exec provisioner. The depends_on parameter ensures that the null_resource is created only after aws_instance resource has been successfully created. The null_resource is not a real resource and it is used just to perform a provisioner based on a dependency.

```
resource "aws_instance" "example" {
  ami           = "ami-0f403e3180720dd7e"
  instance_type = "t2.micro"
}

resource "null_resource" "example" {
  depends_on = [aws_instance.example]

  provisioner "local-exec" {
    command = "echo 'Resource ${aws_instance.example.id} is created.'"
  }
}
```

TERRAFORM TAINT:

This command will recreate the infra in same workspace

- terraform taint aws_instance.one
- terraform apply --auto-approve

To remove the resource from taint : terraform untaint aws_instance.one

TERRAFORM REPLACE:

To recreate any service we will use terraform taint, but as per the new update instead of taint we will directly use terraform replace command.

terraform apply --auto-approve -replace="aws_instance.three”)

TERRAFORM LIFECYCLE:

1. **PREVENT DESTROY:** When we create a resource on Terraform if we apply terraform destroy it will delete automatically but if you want to prevent a resource to without deleting we can use prevent destroy in life cycle.

```
resource "aws_instance" "three" {
  ami           = "ami-0d8667b0f72471655"
  instance_type = "t2.micro"
  tags = {
    Name = "abc-server"
  }
  lifecycle {
    prevent_destroy = true
  }
}
```

create_before_destroy :

By default when you modify any argument in Terraform and run apply it will destroy the existing resource and then it will create the new resource but if we put create_before_destroy that means the new resource will be created first and the old resource is going to delete later.

```
resource "aws_instance" "three" {
  ami           = "ami-0d8667b0f72471655"
  instance_type = "t2.micro"
  tags = {
    Name = "abc-server"
  }
  lifecycle {
    create_before_destroy = true
  }
}
```

IGNORE CHANGES:

When we create a resource with the help of Terraform it will convert the desired set into current state if we modify anything on current state then if you run terraform apply or plan it will reflect the changes if you don't want to reflect the changes if you want to ignore them we can use this one

```
resource "aws_instance" "three" {
  ami           = "ami-0d8667b0f72471655"
  instance_type = "t2.micro"
  tags = {
    Name = "abc-server"
  }
  lifecycle {
    ignore_changes = [tags]
  }
}
```

TERRAFORM LOCALS:

If we need to use common values in repeated blocks then we go for local blocks.all the values can be declared in a block called locals{}.

```
locals {
  common_tags = {
    users = "devteam"
  }
}
resource "aws_instance" "myec2" {
  ami           = "ami-96fda3c22c1c990a"
  instance_type = "t2.small"
  tags          = local.common_tags
}
resource "aws_instance" "myec3" {
  ami           = "ami-96fda3c22c1c990a"
  instance_type = "t2.small"
  tags          = local.common_tags
}
```

TERRAFORM DYNAMIC BLOCKS: Dynamic blocks helps to reduce repetitive codes.

```
resource "aws_security_group" "mysec" {
  name = "prodsecgroups"

  dynamic "ingress" {
    for_each = var.ports
    content {
      from_port   = ingress.value
      to_port     = ingress.value
      protocol    = "tcp"
      cidr_blocks = ["10.0.0.0/0"]
    }
  }
}

variable "ports" {
  type    = list(any)
  default = [22, 2049, 80]
}
```