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.

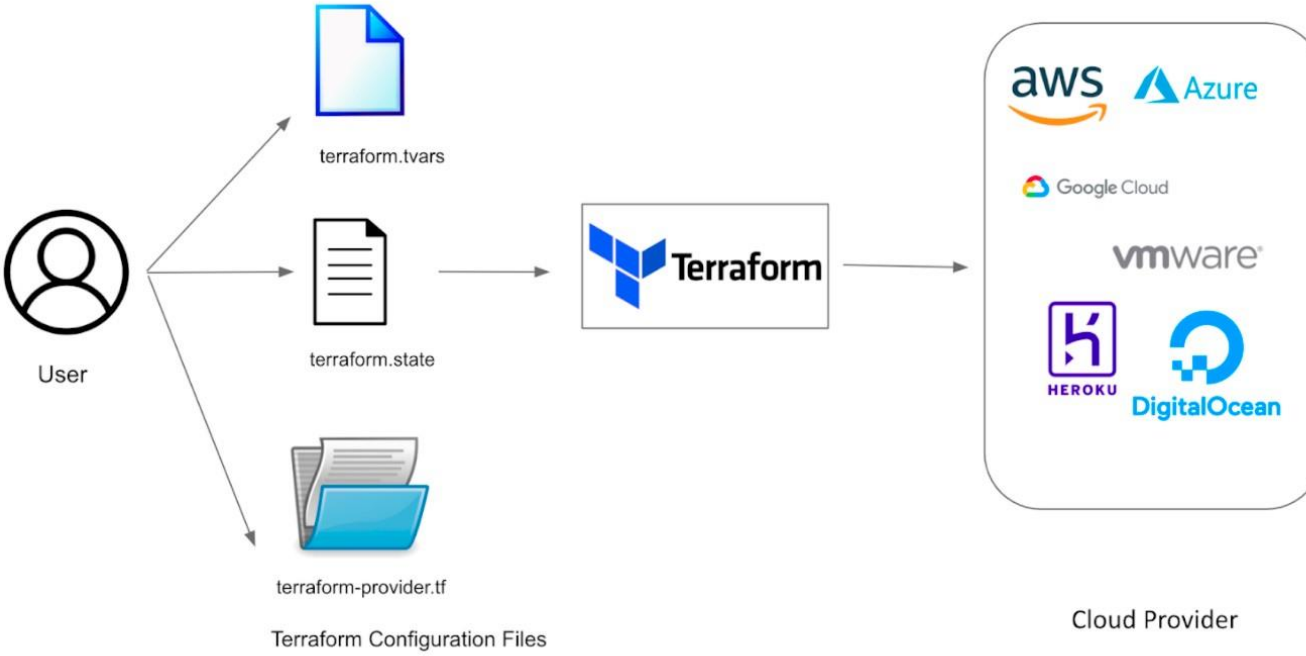
Plan and Apply Workflow: Before making changes, Terraform generates an execution plan, showing what actions it will take. Users review the plan and then apply it to make the changes to the infrastructure.

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 di erent 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"   <------- Meaning full description
  type          = string                     <------- Ex - string, number, bool, list, set, map..
  default       = "t2.micro"                 <------- 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 di erence 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"]
}
```

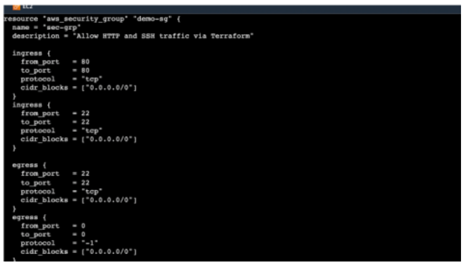
LOOPS WITH COUNT:

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"]
```

```
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 OUTPUTS: used to show the properties/metadata of resources

provider "aws" {

}

resource "aws_instance"

"two" { ami = "ami-

0715c1897453cabd

1" 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-

0715c1897453cabd

1" instance_type =

"t2.micro" tags = {

Name = "web-server"

}

}

provid

er

"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 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_hostname
  s = "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

    }

    t

    a

    g

    s
```

```
  =
  {
    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 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
-

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"]
  }



egr
ess
{
fro
m_
port
= 0
to_
port
= 0
prot
ocol
= "-
1"
cidr
_blo
cks
=
["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.mysub
net2.id]

   listener {
instance_port
= 8000
instance_protoc
ol = "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]
 }
```