VASANTDADA PATIL PRATISHTHAN'S COLLEGE OF ENGINEERING AND VISUAL ARTS

ISO 9001:2015 Certified Institute

Department of Information Technology

NBA Accredited Course (Dated 01/07/2024 to 30/06/2027)

EXPERIMENT - 6

Aim: To Build, change, and destroy AWS / GCP /Microsoft Azure/ Digital Ocean infrastructure Using Terraform.

Theory:

Infrastructure as code (IaC) tools allow you to manage infrastructure with configuration files rather than through a graphical user interface. IaC allows you to build, change, and manage your infrastructure in a safe, consistent, and repeatable way by defining resource configurations that you can version, reuse, and share.

Terraform is HashiCorp’s infrastructure as code tool. It lets you define resources and infrastructure in human-readable, declarative configuration files, and manages your infrastructure’s lifecycle. Using Terraform has several advantages over manually managing your infrastructure.

Terraform can manage infrastructure on multiple cloud platforms. The human-readable configuration language helps you write infrastructure code quickly. Terraform's state allows you to track resource changes throughout your deployments. You can commit your configurations to version control to safely collaborate on infrastructure.
Manage any infrastructure Terraform plugins called providers let Terraform interact with cloud platforms and other services via their application programming interfaces (APIs). HashiCorp and the Terraform community have written over 1,000 providers to manage resources on Amazon Web Services (AWS), Azure, Google Cloud Platform (GCP), Kubernetes, Helm, GitHub, Splunk, and DataDog, just to name a few. Find providers for many of the platforms and services you already use in the Terraform Registry. If you don't find the provider you're looking for, you can write your own.

To deploy infrastructure with Terraform:

- Scope - Identify the infrastructure for your project.
- Author - Write the configuration for your infrastructure.
- Initialize - Install the plugins Terraform needs to manage the infrastructure.
- Plan - Preview the changes Terraform will make to match your configuration.
- Apply - Make the planned changes. Build Infrastructure

Prerequisites
Before we proceed we need some thing to be setup.

- The Terraform installed.
- The AWS CLI installed.
- An AWS account.
- AWS access key credentials.

1. We can configure aws cli

```
> aws configure
AWS Access Key ID [None]: AKIAUEA4PENWWEXX52MW
AWS Secret Access Key [None]: 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
Default region name [None]: us-east-1
Default output format [None]:
```

2. Then go with terraform, create a directory in which main.tf will live

```
> mkdir myTerraInfra
```

3. Create main.tf which contains our main configuration for infrastructure

```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = " > 4.16"
    }
  }
  required_version = " = 1.2.0"
}

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

resource "aws_instance" "app_server" { ami    =
  "ami-0453898e98046c639"
  instance_type = "t2.micro"

  tags = {
    Name = var.instance_name
  }

  provisioner "remote-exec" { inline = [
        "sudo apt-get -y update",
        "sudo apt-get -y install nginx", "sudo
        service nginx start"
    ]
  }
}
```

4. Once the configuration file is created, initialize the terraform so It can install required providers defined in main.tf

```
> terraform init Initializing the
backend  .
Initializing provider plugins  .
-  Finding hashicorp/aws versions matching " > 4.16"  .
-  Installing hashicorp/aws v4.67.0  .
-  Installed hashicorp/aws v4.67.0 (signed by HashiCorp)
Terraform has created a lock file .terraform.lock.hcl to record the provider selections it made
above. Include this file in your version control repository so that Terraform can guarantee to
make the same selections by default when you run "terraform init" in the future.

Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan" to see any changes
that are required for your infrastructure.
```

5. Now, we can go with formatting and validating our configuration files.

```
> terraform  fmt
> terraform validate
Success!  The  configuration  is  valid
```

6. Next is the main part, here we create actual infrastructure

```
> terraform  apply
```

It will show dry run plan and will ask to continue

```
Do  you  want  to  perform  these  actions?
   Terraform will perform the actions described above.
   Only  'yes' will be  accepted  to  approve.

   Enter  a  value:  yes
```

After this it will create your infrastructure, you can verify by login to aws console. Or using aws cli.

You can modify you code and rerun this to update your infrastructure. There can be different files like variables.tf, or anyother.

When the infrastructure building gets completed, 2 files will be generated:

- terraform.tfstate : It stores the status of current infrastructure applied
- terrafrom.tfstate.backup : It is backup if above file incase of mess.
- .terraform : This directory contains the providers packages.

7. Now to destroy it run the following command

```
> terraform destroy
```

It will again ask to continue, type yes and It will start destroying

```
.
aws_instance.app_server: Destroying  . [id=i-04701c6a4dd4c7364] aws_instance.app_server: Still
destroying  . [id=i-04701c6a4dd4c7364, 10s elapsed] aws_instance.app_server: Still destroying  .
[id=i-04701c6a4dd4c7364, 20s elapsed] aws_instance.app_server: Still destroying  . [id=i-
04701c6a4dd4c7364, 30s elapsed] aws_instance.app_server: Still destroying  . [id=i-
04701c6a4dd4c7364, 40s elapsed] aws_instance.app_server: Still destroying  . [id=i-
04701c6a4dd4c7364, 50s elapsed] aws_instance.app_server: Still destroying  . [id=i-
04701c6a4dd4c7364, 1m0s elapsed] aws_instance.app_server: Destruction complete after 1m2s
Destroy complete! Resources: 1 destroyed.
```

Conclusion: Thus, we have successfully Build, changed, and destroyed AWS / GCP /Microsoft Azure/ Digital Ocean infrastructure Using Terraform.