Sammy Cespedes
Kura Labs Cohort 3
November 4, 2022

Deployment 4

In this Deployment we are using Infrastructure as Code, specifically Terraform to create an infrastructure for deploying a Flask Application

The Set Up:

- I first began by installing Jenkins on an EC2 instance
    - I will be using a previous EC2 instance that I have create for my prior deployments
    - In this EC2 I have automated the process of installing Jenkins through inserting a bash script within the instance configurations that will automatically install Jenkins and it's dependencies such as the Java RunTime Environment which is essential for the installation of Jenkins
    - Also, I configured this instance's security groups to have ports 80, 8080 and 22
        - Port 22 will be used for SSH purposes into the instance
        - Port 8080 will be used alongside the public IP address of this instance in order to access the Jenkins server
    - This EC2 instance is located within the Default VPC

Script Used to Install Jenkins on EC2

```
#!/bin/bash

# Download JRE
sudo apt update && sudo apt install default-jre -y

# Get the key to access Jenkins package
wget -q -O - https://pkg.jenkins.io/debian-stable/jenkins.io.key | sudo gpg --dearmor -o/usr/share/keyrings/jenkins.gpg

# Using the key to authenticate the package
sudo sh -c 'echo deb [signed-by=/usr/share/keyrings/jenkins.gpg] http://pkg.jenkins.io/debian-stable binary/ > /etc/apt/sources.list.d/jenkins.list'

# Update and install Jenkins
sudo apt update && sudo apt install jenkins -y

# Start Jenkins and check the status to make sure it is running
sudo systemctl start jenkins
sudo systemctl status jenkins >> ~/file.txt
```

- I proceeded to SSH into my EC2 instance and ran a < systemctl status Jenkins > on the terminal to see if my Jenkins server is running properly
- Following that, I proceeded with the installation of Terraform

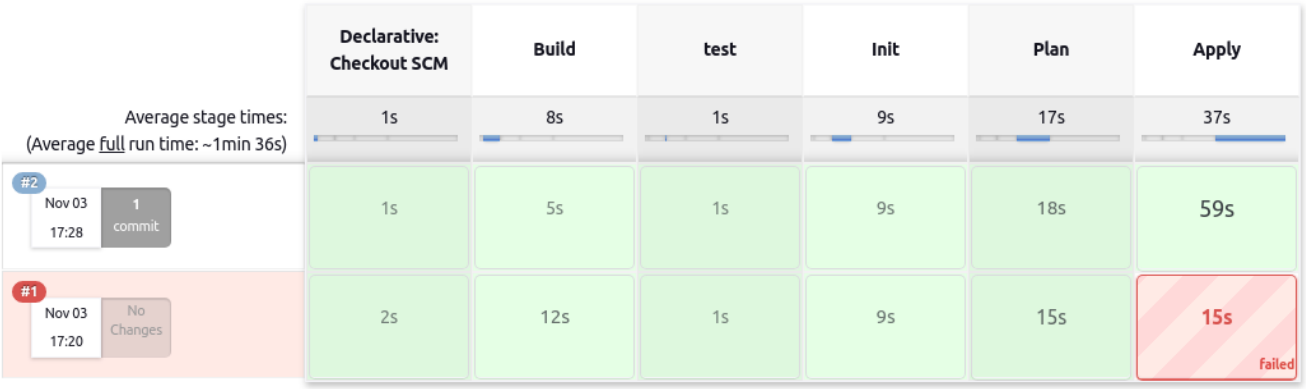
Commands used to Install Terraform

```
$ wget -O- https://apt.releases.hashicorp.com/gpg | gpg --dearmor | sudo tee /usr/share/keyrings/hashicorp-archive-keyring.gpg
$ echo "deb [signed-by=/usr/share/keyrings/hashicorp-archive-keyring.gpg] https://apt.releases.hashicorp.com $(lsb_release -cs) main" | sudo tee /e
$ sudo apt update && sudo apt install terraform
```

- Following the installation of Terraform, I accessed my Jenkins server on my web browser and logged into my account

- After logging into my Jenkins account, I configured my credentials on Jenkins
    - Manage Jenkins —--> Manage Credentials —--> Selected Global —--> Add Credential
        - This process allowed me to connect my AWS Access Key and Secret Key to Jenkins so I am able to Build/Test my application

Build Results



- Initially the Apply stage within my pipeline failed because my Terraform file located in my remote GitHub Repository did not have my specific Key Pair Name that I use within AWS Console to be able to SSH into my ec2 instances
    - Therefore, after specifying my key pair name I ran the build again and it successfully ran all of my stages within my Jenkins file on my GitHub Repo
    - Terraform Apply in my pipeline is responsible for creating or updating my infrastructure depending on the configuration files
        - However, a Terraform Plan command must be ran prior to a Terraform Apply and what that does is reports back to you any changes within your infrastructure, providing visibility to those modifications so you are aware of them prior to applying them to your entire infrastructure

Destroy Stage

```
stage('Destroy') {
    steps {
     withCredentials([string(credentialsId: 'AWS_ACCESS_KEY', variable: 'aws_access_key'),
                      string(credentialsId: 'AWS_SECRET_KEY', variable: 'aws_secret_key')]) {
                         dir('intTerraform') {
                           sh 'terraform destroy -auto-approve -var="aws_access_key=$aws_access_key"-var="aws_secret_key=$aws_secret_key"'
```

- This stage was added to my Jenkins file to destroy my infrastructure once my application deployment has run successfully
    - This will tear down my entire infrastructure

Infrastructure as Code: Terraform

- I will be deploying an application using Terraform to create the infrastructure
    - Terraform is a Infrastructure as Code software tool that is main purpose is to automate various infrastructure tasks
    - It is also meant for provisioning cloud resources
- Within this deployment, I will be creating a VPC, an EC2 instance and 2 public subnets
    - Also, I will be specifying my availability zones which will house my ec2 instance
    - US-EAST-1 Northern Virginia will be my Availability Zone

```
variable "AWS_REGION" {
    default = "us-east-1"
}

variable "AMI" {
    type = "map"

    default {
        us-east-1 = "ami-08c40ec9ead489470"
    }
}
```

- This is my 'variables.tf' file where I will be calling my variables from within my VPC.tf and Network.tf terraform file that will contain all of my configurations necessary to create my VPC and also my route table and internet gateway that I will be using to connect to my public subnets
- In this file specifically, you can see the location of my availability zone as well as the AMI

- After specifying my variables, I then ran < terraform init > whitch initialized terraform within my terminal
    - Upon running this command, a < .terraform > file was created within my deployment 4 terraform folder

```
resource "aws_vpc" "deployment4-vpc" {
    cidr_block = "10.0.0.0/16"
    enable_dns_support = "true" #gives you an internal domain name
    enable_dns_hostnames = "true" #gives you an internal host name
    enable_classiclink = "false"
    instance_tenancy = "default"

    tags {
        Name = "deployment4-vpc"
    }
}

resource "aws_subnet" "deployment4-subnet-public-1" {
    vpc_id = "${aws_vpc.deployment4-vpc.id}"
    cidr_block = "10.0.1.0/24"
    map_public_ip_on_launch = "true"
    availability_zone = "us-east-1"


    tags {
        Name = "deployment4-subnet-public-1"
    }
```

- This is my VPC.tf file where I have configured all of the resources needed to create my VPC
    - I specified the CIDR block for both my VPC and Public Subnet
- The “map_public_ip_on_launch” configuration is saying that if this is true, my subnet will always be public and if it is false, my subnet will be private

```
resource "aws_internet_gateway" "deployment4-igw" {
    vpc_id = "${aws_vpc.deployment4-vpc.id}"
    tags {
        Name = "deployment4-igw"
    }
}
```

- The internet gateway allows my VPC to be able to connect to the internet

```
resource "aws_route_table" "deployment4-public-crt" {
    vpc_id = "${aws_vpc.main-vpc.id}"

    route {
        cidr_block = "0.0.0.0/0"
        gateway_id = "${aws_internet_gateway.deployment4-igw.id}"
    }

    tags {
        Name = "deployment4-public-crt"
    }
}

resource "aws_route_table_association" "deployment4-crta-public-subnet-1" {
    subnet_id = "${aws_subnet.deployment4-subnet-public-1.id}"
    route_table_id = "${aws_route_table.deployment4-public-crt.id}"
}
```

- I created a custom routing table that will interact with the public subnet and it allows the public subnet to reach the internet gateway
- A route table is used to determine where network traffic is directed
    - In this deployment the public subnet it directing outbound traffic using the route table to the internet gateway and subsequently the internet so the user is able to access the frontend of the URL-Shortener
    - I also needed to associate the created route table to my public subnet

```
resource "aws_security_group" "ssh-allowed" {
    vpc_id = "${aws_vpc.deployment4-vpc.id}"

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

    ingress {
    from_port = 8000
    to_port = 8000
    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"]
    }

    tags {
        Name = "ssh-allowed"
```

- I used these Security Groups within my terraform file to specify port 22 for SSH purposed
    - This port allows me to access my EC2 via SSH and that is also why I opened up the port under the 0.0.0.0 IP address, meaning that it is telling the server to listen for and accept any connections that are incoming from any IP Address
- Port 8000 is the Gunicorn port which is acting as our Web Server and is listening for traffic and routing it to the internet gateway through the route table

- With Gunicorn installed within my created EC2 instance, it's main purpose is to pass data requests to the application which is listening for on port 8000 and also to receive response data
    - Gunicorn is distributing incoming requests across my instances using the route table I configured within my terraform infrastructure and is also communicating with the web server

<u>The Power of Terraform</u>

- Using infrastructure as code, Terraform makes creating resources and building an infrastructure much quicker and efficient
    - The implementation of terraform modules also helps group up resources into a specific file or "module", reducing the amount of code needed to develop

- Modules also promote reusability within your own infrastructure, you are able to call upon specific modules that contain certain resources needed for a task throughout your terraform infrastructure