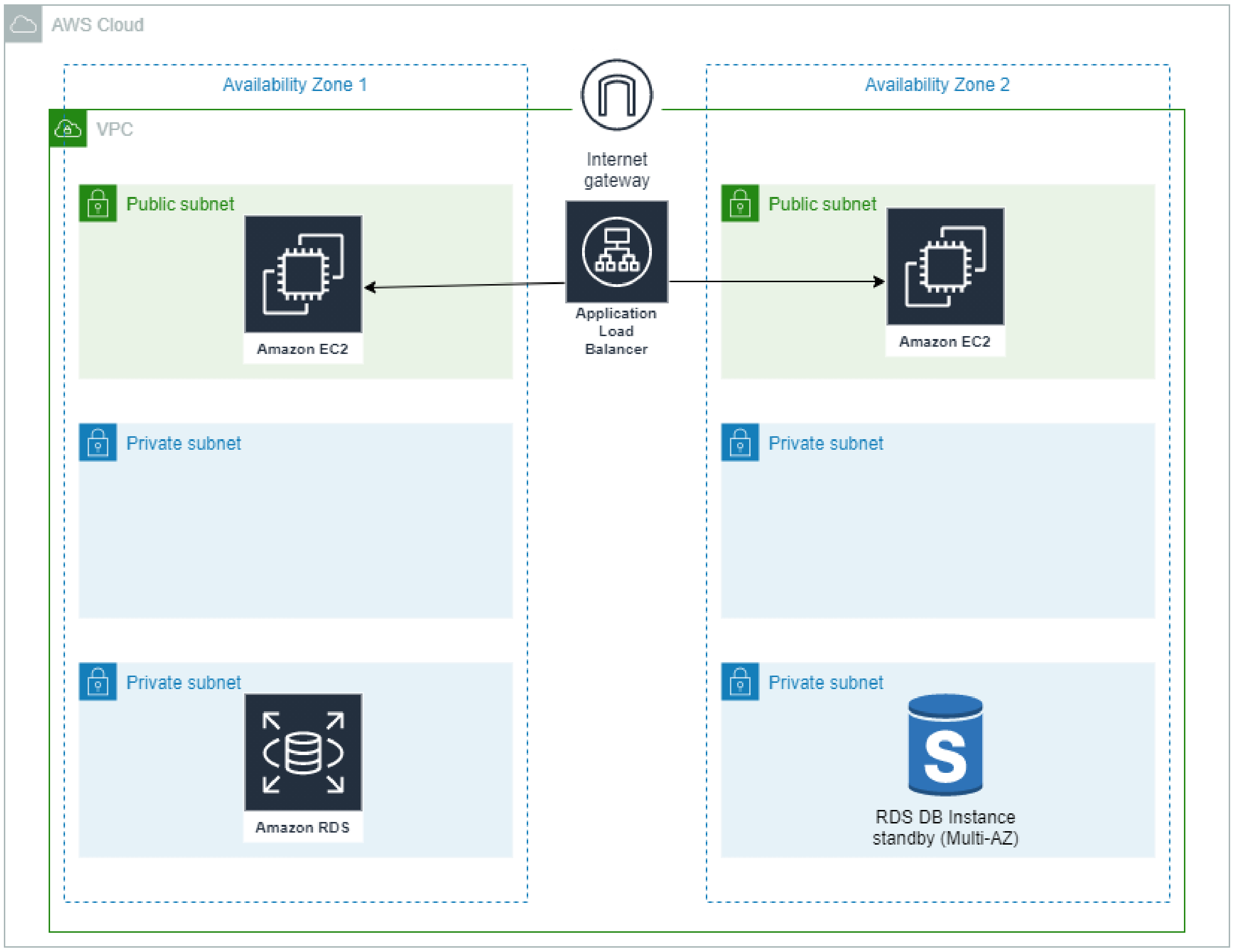
Terraform: Deploy A Three-Tier Architecture in AWS



Infrastructure as Code (IaC)

- The cloud gives us the ability to create our environments quickly, but the problem that arises is how to configure and manage the environments.
- Manually updating from the console may be acceptable for a small organization in a single region, but what if you must create and maintain environments in multiple regions?
- Not only is it an inefficient use of time to create and maintain everything, but it’s also error prone.
- Imagine that you are asked to create an environment in a single Region.
- Not really a big deal and you are able to complete the task with relative ease.
- Now you need to do the exact same thing for five more Regions.

- Not only that, once you have completed the excruciatingly repetitive task, your leadership asks you to make a change that then needs to be applied to all Regions.
- This example is inefficiency at its finest.
- Think about infrastructure as code as a scalable blueprint for your environment.
- It allows you to provision and configure your environments in a reliable and safe way.
- By using code to deploy your infrastructure you gain the ability to use the same tools as developers such as version control, testing, code reviews, and CI/CD.

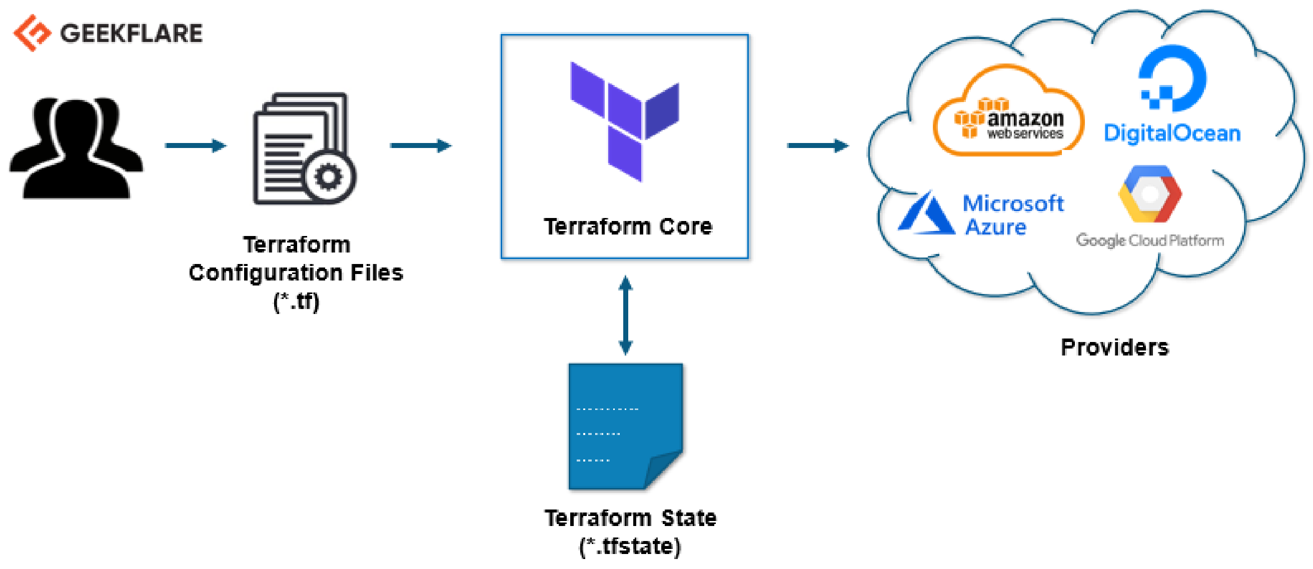
Terraform

- HashiCorp Terraform is a tool for building, changing, and versioning infrastructure that has an open-source and enterprise version.
- Terraform is cloud agnostic and can be used to create multi-cloud infrastructure.
- It allows IaC in a human readable language called HashiCorp Configuration Language (HCL).

Prerequisites – we did this already

- Install Terraform - check the installation by running terraform -v
- Install the AWS CLI – check installation by running the command aws - - v
- Sign up for an AWS Account – make sure you have an account
- Preferred IDE (Visual Studio Code) -VSC

Terraform Components you should know



Providers – This is an essential component of terraform. Terraform provisions infrastructure in the cloud. Many different clouds terraform supports such as Google, AWS, Azure, etc

Terraform state – represents the current state of terraform. This current state is usually stored in the terraform file.

Terraform core: This is where we run the terraform lifecycle commands. Terraform lifecycle commands are [terraform init, terraform plan, terraform apply, and terraform destroy]

Configuration files: These are the files that contain the infrastructure code or terraform code. This code usually describes the desired state we, as cloud practitioners, would like to provision in the cloud.

The configuration files are written and stored as a git repository in Github. The files are usually divided into names such as provider. tf, and resource. tf, output.tf, etc., make making changes when needed and keeping our desired state infrastructure code organized easier.

Important Use Cases for Terraform

- Terraform is used to automate the process of provisioning infrastructure in the companies.

- Automating this process is very beneficial to many businesses and helps to reduce the human errors that occur when the provisioning is manual and tedious.
- Terraform allow us to quicken the process of provisioning infrastructure in the cloud. One of the main goals for many companies is to create an infrastructure through a repeatable process.
- This is only possible using declarative tools like terraform. Repeatability of the process is possible because we store the desired state of the infrastructure code in git, which allows us to create the same resources in another region, saving the company many resources in terms of human labor, time, and capital.
- We can use terraform to solve the business's needs to modify its infrastructure frequently. Since the infrastructure code is stored as a git repository, we can always make modifications when necessary.
- It is important to note that modifications can be minor or significant. A minor modification can involve just changing the name of the VPC, while a significant modification can involve creating new instances inside the existing infrastructure.
- Terraform allows us to automate modifying our infrastructure provisioned in the cloud.
- Customers are very dynamic. A business might need to run different versions of the same infrastructure to solve its needs.
- To accomplish this, terraform becomes very important as it can allow us to automate the process of different provisioning versions of our infrastructure to meet the different needs of our customers.
- Businesses need to clean up their outdated and no longer needed infrastructure so that a space can be created to provide new infrastructure.
- Terraform is very important as it allows the company to automate cleaning up the outdated or no longer needed infrastructure in the cloud.

Main Common Terraform Commands

I - init

P – plan

A – apply

D – destroy

Other common Terraform Commands

Terraform fmt

Terraform validate

Terraform show

Project 1 – Terraform on premise

We are going to create infrastructure using the terraform installed in your laptop. DO NOT USE THE CLOUD.

Configure Provider

Information regarding the provider is stored in provider.tf

Providers are plugins that Terraform requires so that it can install and use for your Terraform configuration. Terraform offers the ability to use a variety of **Providers,** so it doesn't make sense to use all of them for each file. We will declare our **Provider** as AWS.

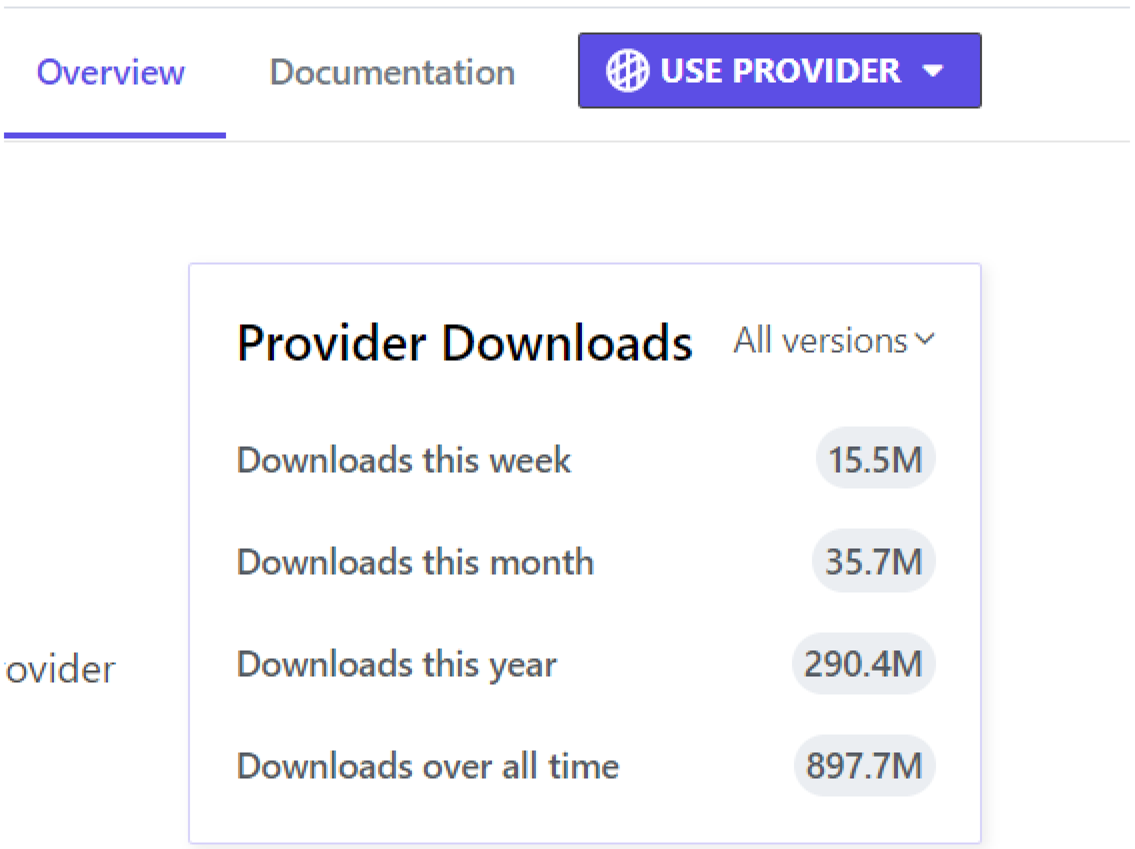
1. Create a **main.tf** file and add each of the following sections to the **main.tf** file.

2. From the terminal in the Terraform directory containing install_apache.sh and main.tf run `terraform init`

3. Use the below code to set our **Provider** to AWS and set our Region to us-east-2 (This is my region)

Link: https://registry.terraform.io/providers/hashicorp/aws/latest

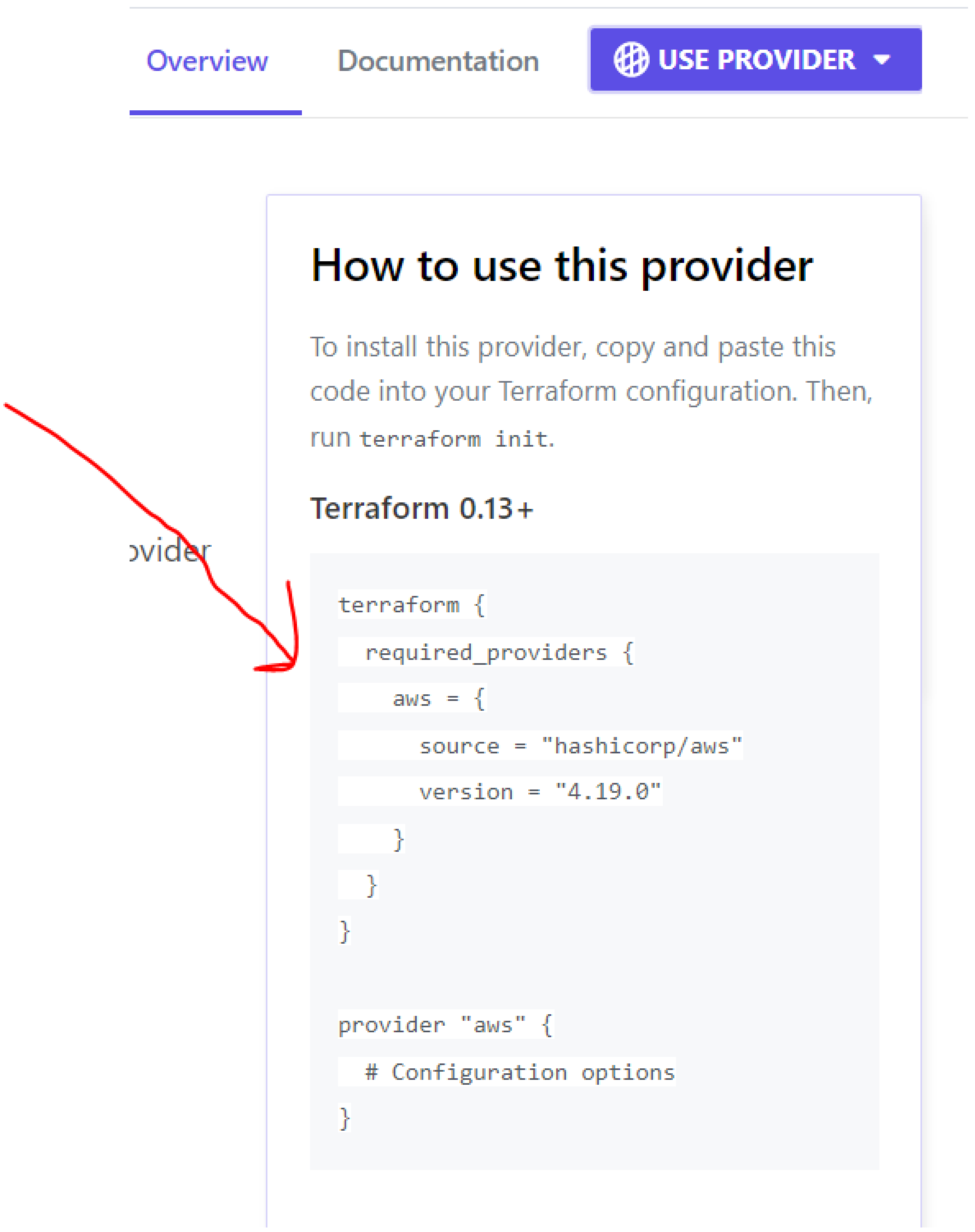
Go ➔ Provider



Provider Downloads

Downloads this week	15.5M
Downloads this month	35.7M
Downloads this year	290.4M
Downloads over all time	897.7M

Copy this code



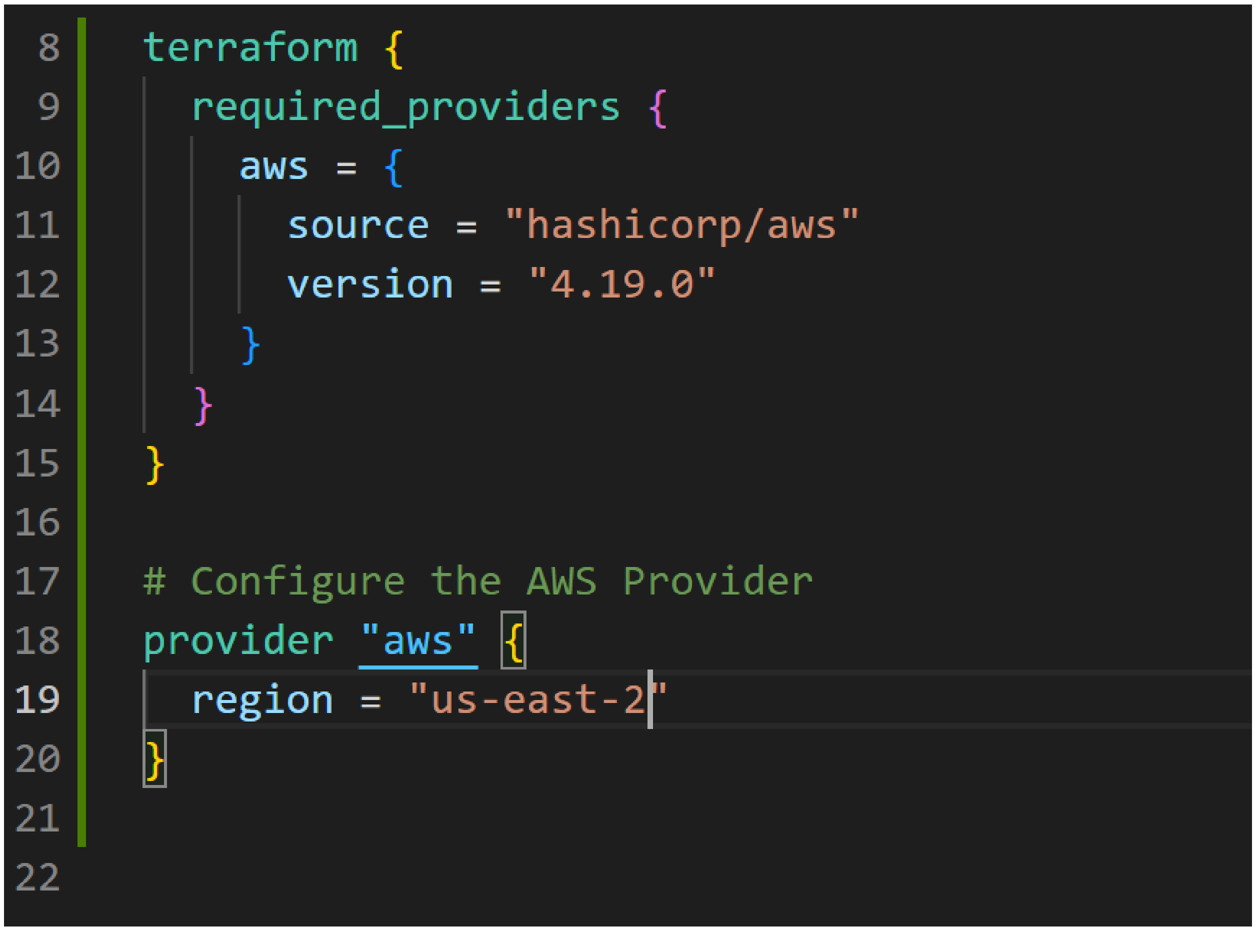
How to use this provider

To install this provider, copy and paste this code into your Terraform configuration. Then, run `terraform init`.

Terraform 0.13+

```
terraform {
  required_providers {
    aws = {
      source = "hashicorp/aws"
      version = "4.19.0"
    }
  }
}

provider "aws" {
  # Configuration options
}
```

Go ➔ to IDE (Visual Studio Code) and create a file called provider.tf



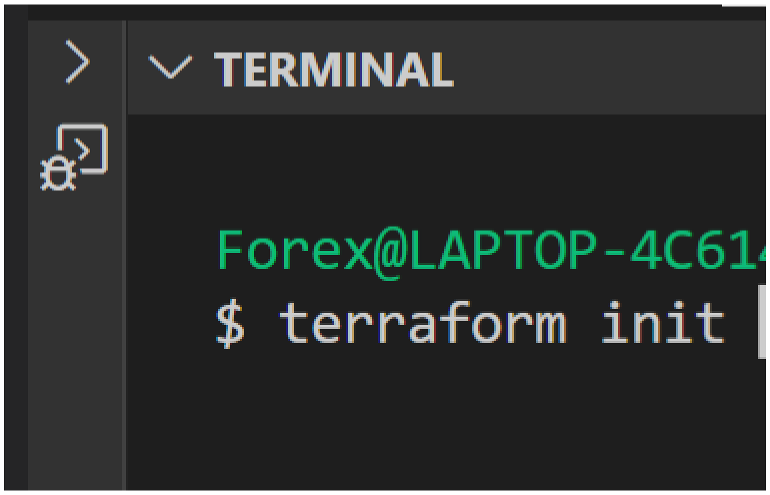
```
terraform {
  required_providers {
    aws = {
      source = "hashicorp/aws"
      version = "4.19.0"
    }
  }
}

# Configure the AWS Provider
provider "aws" {
  region = "us-east-2"
}

```

Save using Ctrl + S

We are done with providing information about our provider (AWS).

Before moving forward, let us validate whether our provider information is correct.

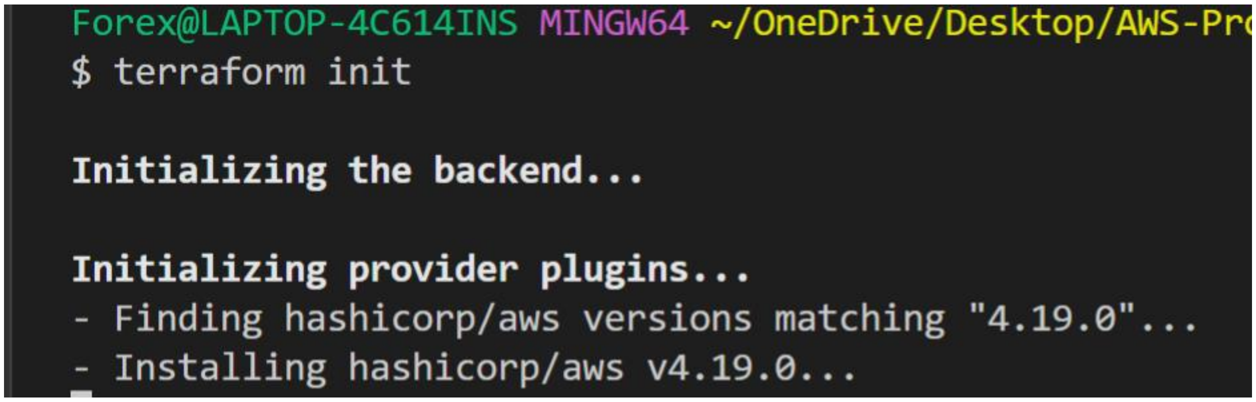
Run the init command in your terminal



```
Forex@LAPTOP-4C61
$ terraform init
```

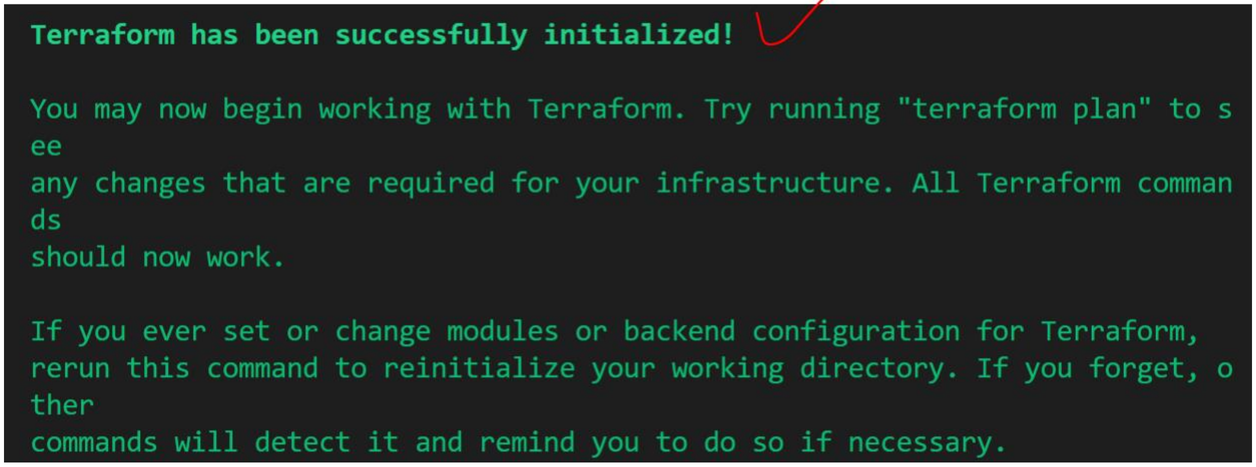
$ terraform init

Output



```
Forex@LAPTOP-4C614INS MINGW64 ~/OneDrive/Desktop/AWS-Pr
$ terraform init

Initializing the backend...

Initializing provider plugins...
- Finding hashicorp/aws versions matching "4.19.0"...
- Installing hashicorp/aws v4.19.0...
```

When completed, you should view the information below.



```
Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan" to s
ee
any changes that are required for your infrastructure. All Terraform comman
ds
should now work.

If you ever set or change modules or backend configuration for Terraform,
rerun this command to reinitialize your working directory. If you forget, o
ther
commands will detect it and remind you to do so if necessary.
```

Great!!

Next, let us create two more resources

VPC and Subnets

Let start with VPC

Go ➔ VSC and create a file called resource.tf

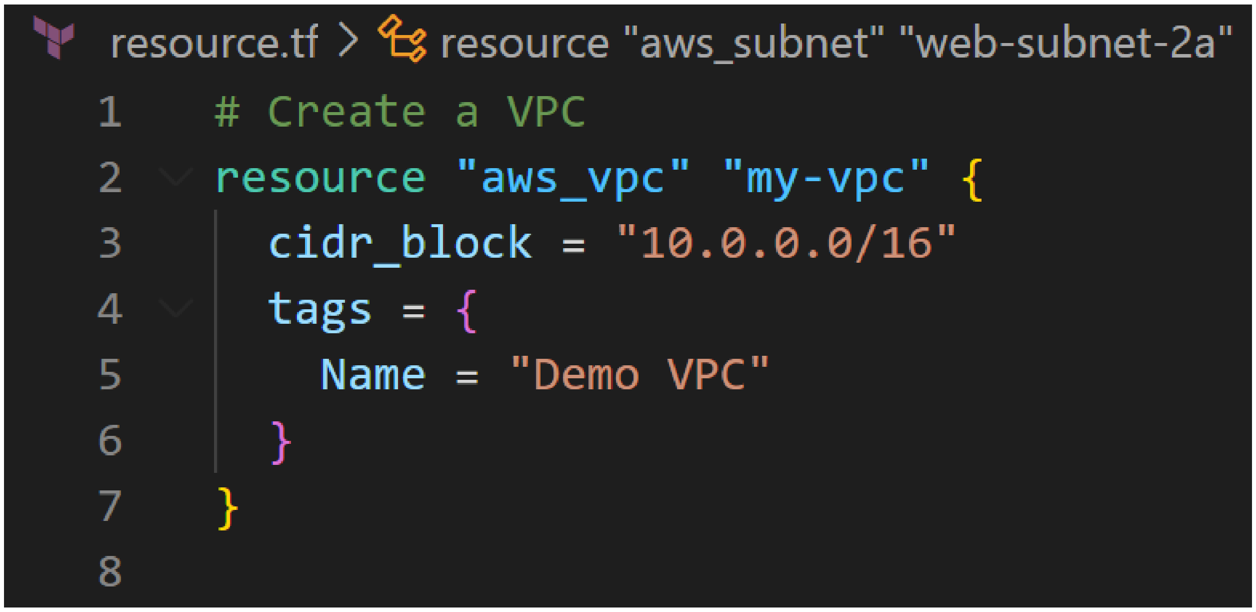
This is where we are going to store the infratsryuacture code for both the vpc and subnet.

Create VPC and Subnets

1. Our first Resource is creating our VPC with CIDR 10.0.0.0/16.
2. **web-subnet-1** and **web-subnet-2** resources create our web layer in two availability zones. Notice that we have `map_public_ip_on_launch = true`
3. **application-subnet-1** and **application-subnet-2** resources create our application layer in two availability zones. This will be a private subnet.

4. **database-subnet-1** and **database-subnet-2** resources create our database layer in two availability zones. This will be a private subnet.

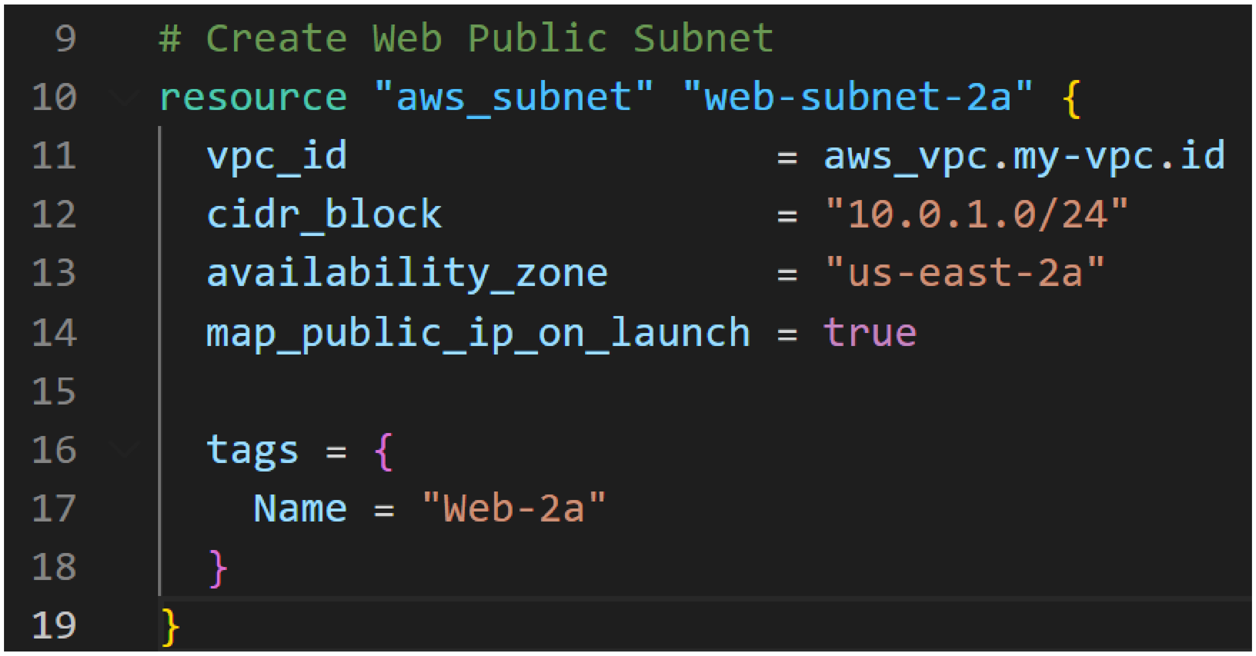
VPC terraform code – in Visual Studio Code



resource.tf > resource "aws_subnet" "web-subnet-2a"

```
# Create a VPC
resource "aws_vpc" "my-vpc" {
  cidr_block = "10.0.0.0/16"
  tags = {
    Name = "Demo VPC"
  }
}

```

Subnet-2a terraform code – in Visual Studio Code



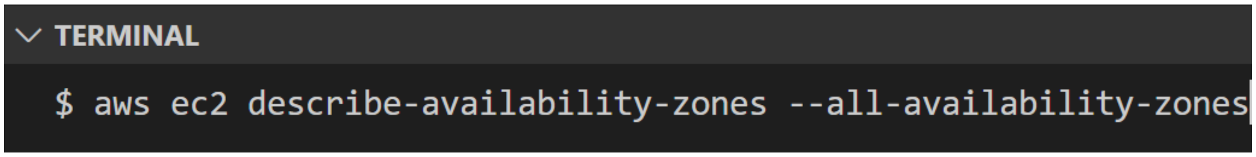
```
# Create Web Public Subnet
resource "aws_subnet" "web-subnet-2a" {
  vpc_id                  = aws_vpc.my-vpc.id
  cidr_block              = "10.0.1.0/24"
  availability_zone       = "us-east-2a"
  map_public_ip_on_launch = true

  tags = {
    Name = "Web-2a"
  }
}
```

Great!!

Knowing the availability zones in your region.

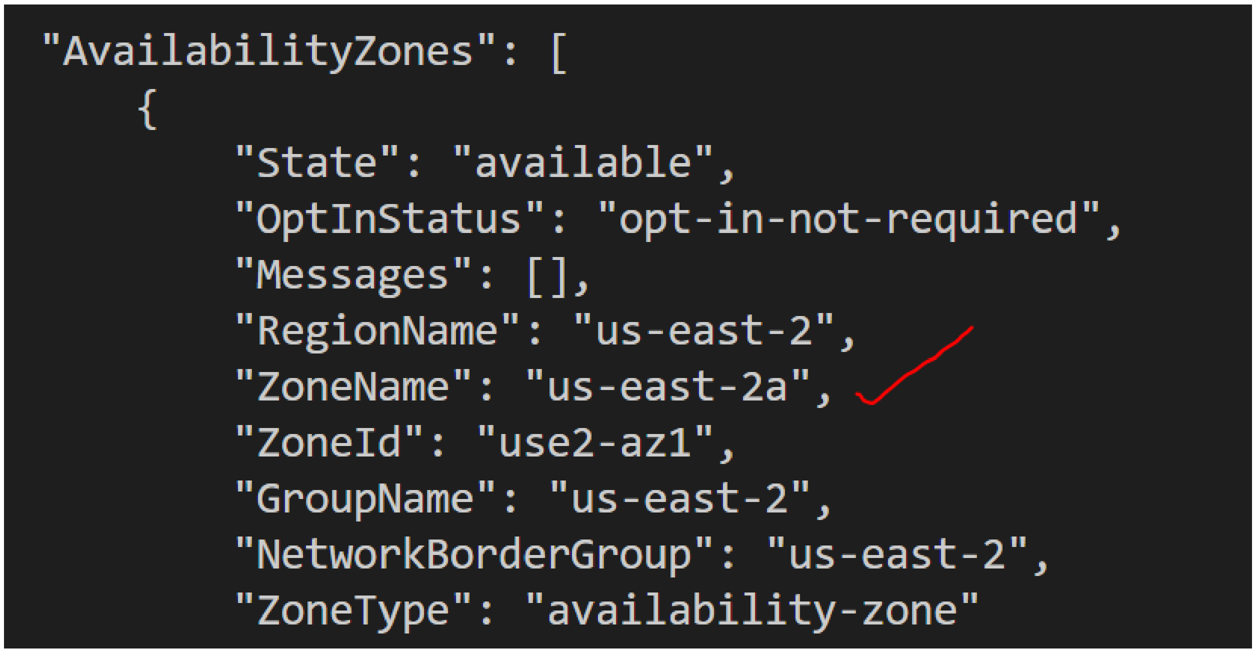
To do this, run the command,

$ aws ec2 describe-availability-zones --all-availability-zones

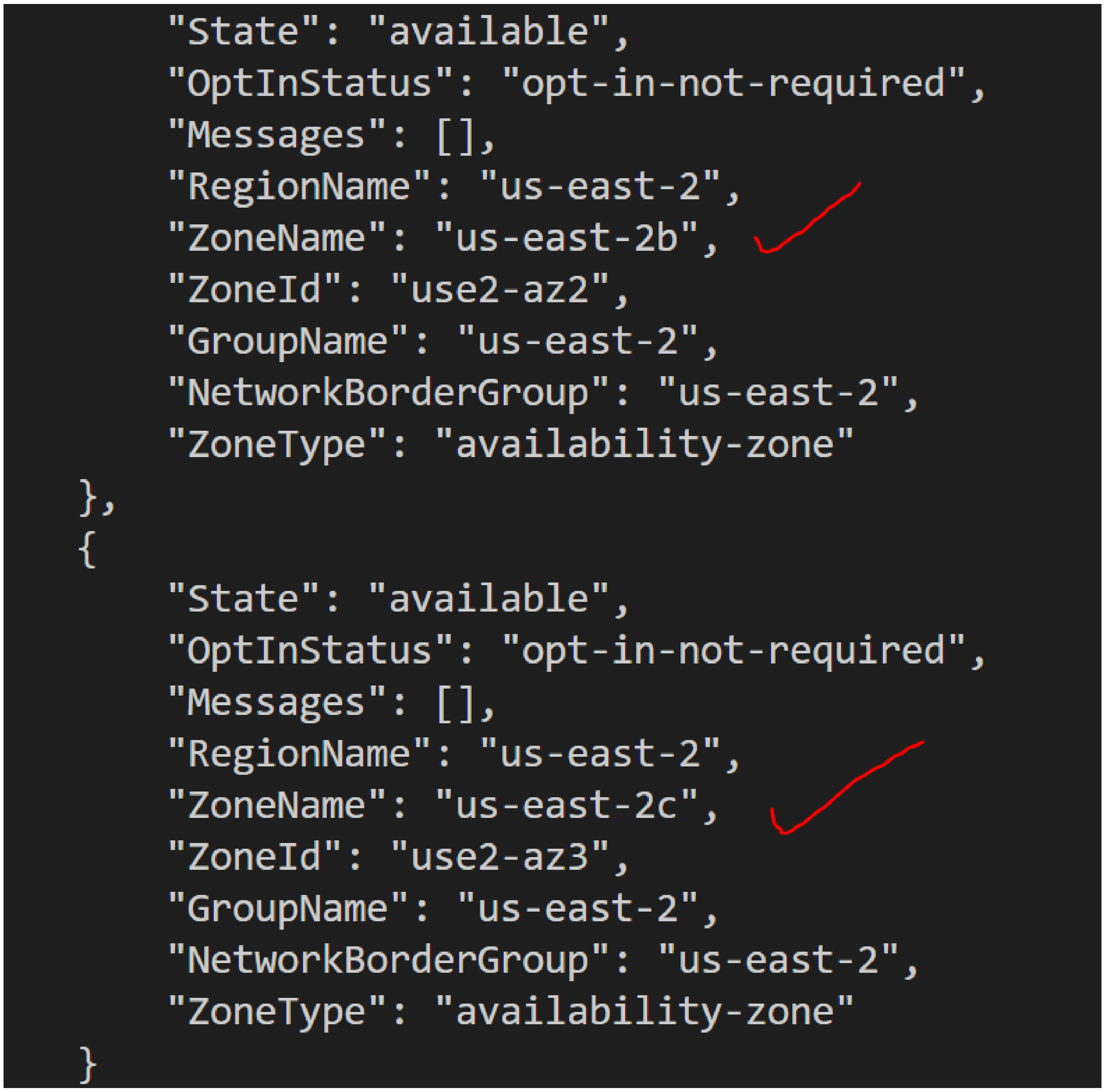


```
$ aws ec2 describe-availability-zones --all-availability-zones
```

Output



```
"AvailabilityZones": [
    {
        "State": "available",
        "OptInStatus": "opt-in-not-required",
        "Messages": [],
        "RegionName": "us-east-2",
        "ZoneName": "us-east-2a", ✓
        "ZoneId": "use2-az1",
        "GroupName": "us-east-2",
        "NetworkBorderGroup": "us-east-2",
        "ZoneType": "availability-zone"
```

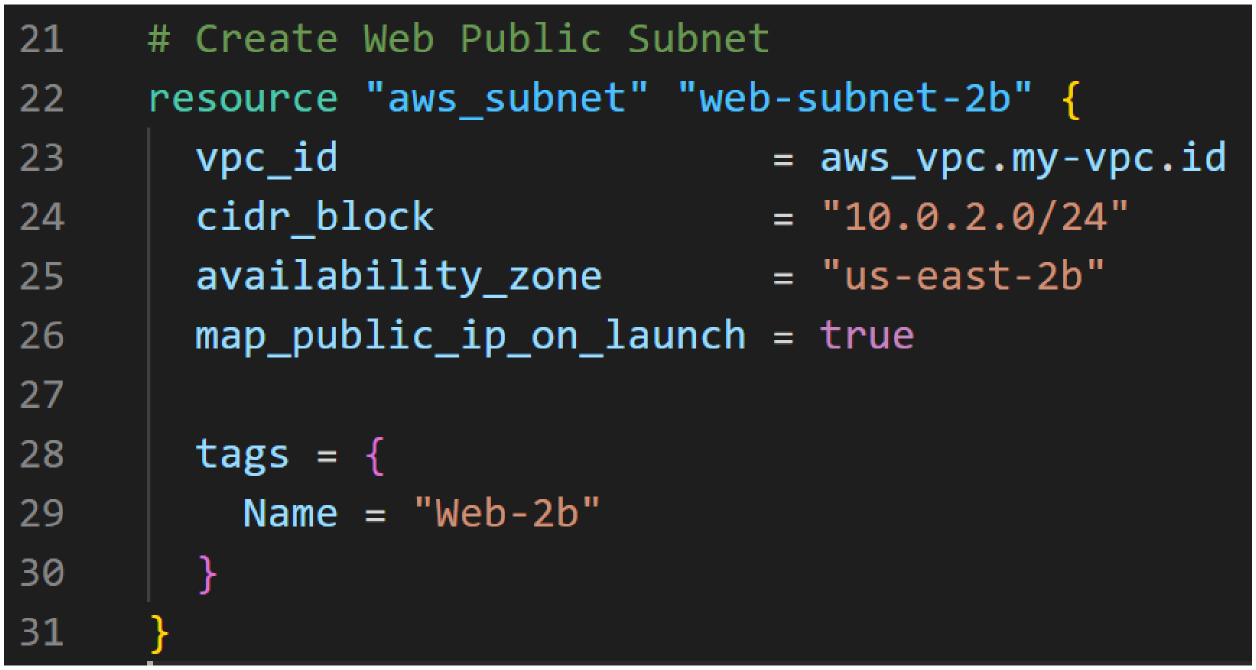


```
        "State": "available",
        "OptInStatus": "opt-in-not-required",
        "Messages": [],
        "RegionName": "us-east-2",
        "ZoneName": "us-east-2b",   ✓
        "ZoneId": "use2-az2",
        "GroupName": "us-east-2",
        "NetworkBorderGroup": "us-east-2",
        "ZoneType": "availability-zone"
    },
    {
        "State": "available",
        "OptInStatus": "opt-in-not-required",
        "Messages": [],
        "RegionName": "us-east-2",
        "ZoneName": "us-east-2c",   ✓
        "ZoneId": "use2-az3",
        "GroupName": "us-east-2",
        "NetworkBorderGroup": "us-east-2",
        "ZoneType": "availability-zone"
    }
```

In my region, which is us-east-2, I have three availability zones

Go ➔ Let add 2 more subnets

Terraform code for subnet-2b



```
# Create Web Public Subnet
resource "aws_subnet" "web-subnet-2b" {
  vpc_id                  = aws_vpc.my-vpc.id
  cidr_block              = "10.0.2.0/24"
  availability_zone       = "us-east-2b"
  map_public_ip_on_launch = true

  tags = {
    Name = "Web-2b"
  }
}
```

Terraform code – subnet-2c

Go ➔ Do some research on how to write terraform code for Internet gateway and Route table

Create Internet Gateway and Route Table

1. Our first resource block will create an **Internet Gateway**. We will need an **Internet Gateway** to allow our public subnets to connect to the Internet.

2. Just saying that our subnets are public does not make it so. We will need to create a route table and Associate our Web Layer subnets.

3. The **web-rt** route table creates a route in our **VPC** to our **Internet Gateway** for CIDR 0.0.0.0/0.

4. The next two blocks are associating **web-subnet-1** and **web-subnet-2** with the **web-rt** route table. I feel like you should be able to associate more than one subnet in a single block, but I kept getting errors and wasn’t able to find any helpful documentation.

Note: We only create one new public route table and do not need to create a private route table. By default, all subnets are associated with the default route table which is set to private by default.