TERRAFORM PROJECT

AWS INFRASTRUCTURE AUTOMATION USING TERRAFORM

TERRAFORM :

Terraform is an open-source Infrastructure as Code (IaC) tool developed by HashiCorp. It allows you to define and manage your infrastructure (servers, databases, networking, etc.) using a declarative configuration language. With Terraform, you can automate the provisioning, scaling, and management of resources across multiple cloud providers (like AWS, Azure, Google Cloud) and on-premises environments.

Simple Use Cases of Terraform:

- **Cloud Infrastructure Management:**
  Automate the setup of servers, databases, networks, and storage across cloud providers like AWS, Azure, or GCP.
- **Multi-Cloud Deployments:**
  Manage resources in multiple cloud platforms simultaneously, such as hosting an app on AWS while storing data in Azure.
- **Auto-Scaling Applications:**
  Set up rules to automatically sacle your servers up or down based on demand, ensuring performance and cost-efficiency.
- **Infrastructure as Code (Iac):**
  Write code to define your infrastructure, making it east to replicate environmets(e.g., dev, test, prod) consistently.
- **Resource Management:**
  Terraform excels at resource management, allowing users to create, update, and delete infrastructure resources programmatically and consistently.

Terraform Commands:

- **Terraform init:**
  Terraform initialization downloads the required plugins for the providers defined in the configuration files. Without this step terraform wouldn't know how to interact with the desired infrastructure platform.
- **Terraform validate:**
  It ensures that the files are syntactically correct. This includes checking for common mistakes like missing braces or incorrect argument formats.

- **Terraform plan (terraform plan -lock=false):**

  It helps to ensure that the changes you intend to make to your infrastructure are accurate and safe before applying them.

- **Terraform apply:**

  It is used to apply the changes to your infrastructure like creating, updating, or destroying resources as specified in our configuration files.

- **Terraform destroy:**

  It is used to remove all the resources managed by terraform. It helps to delete resources that are no longer needed.

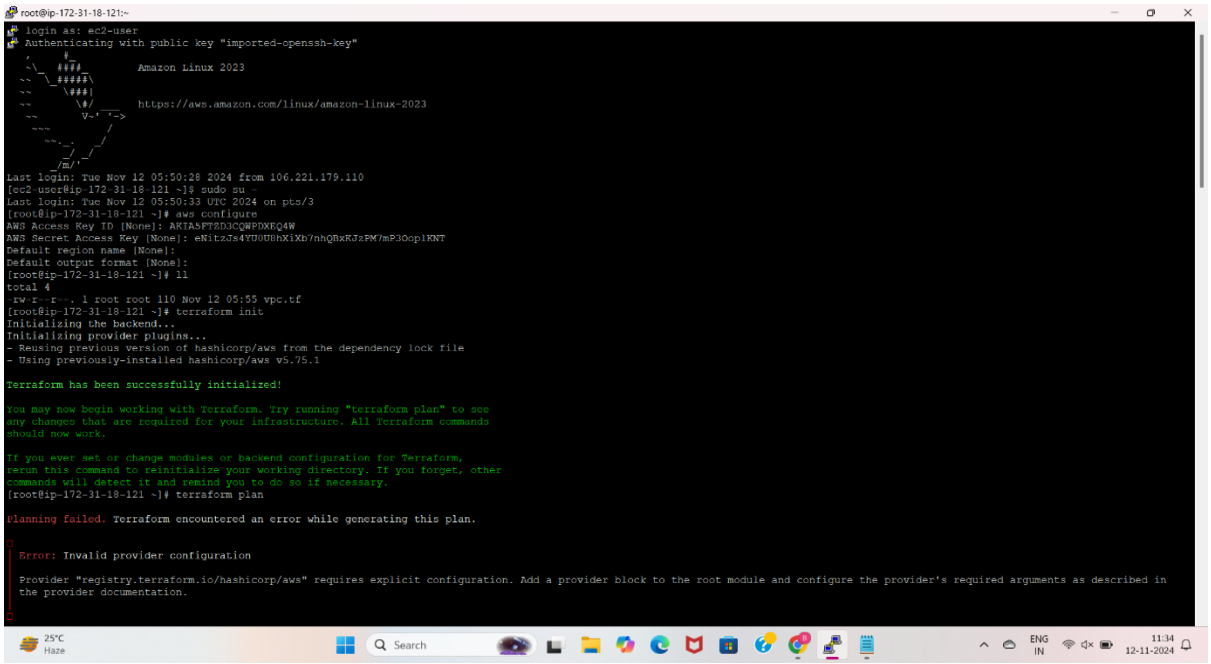
PREREQUISITES:

- ✓ Basic knowledge of aws & terraform
- ✓ Aws account
- ✓ Iam user
- ✓ Github account
- ✓ Aws access & secret key

Basic knowledge of aws & terraform

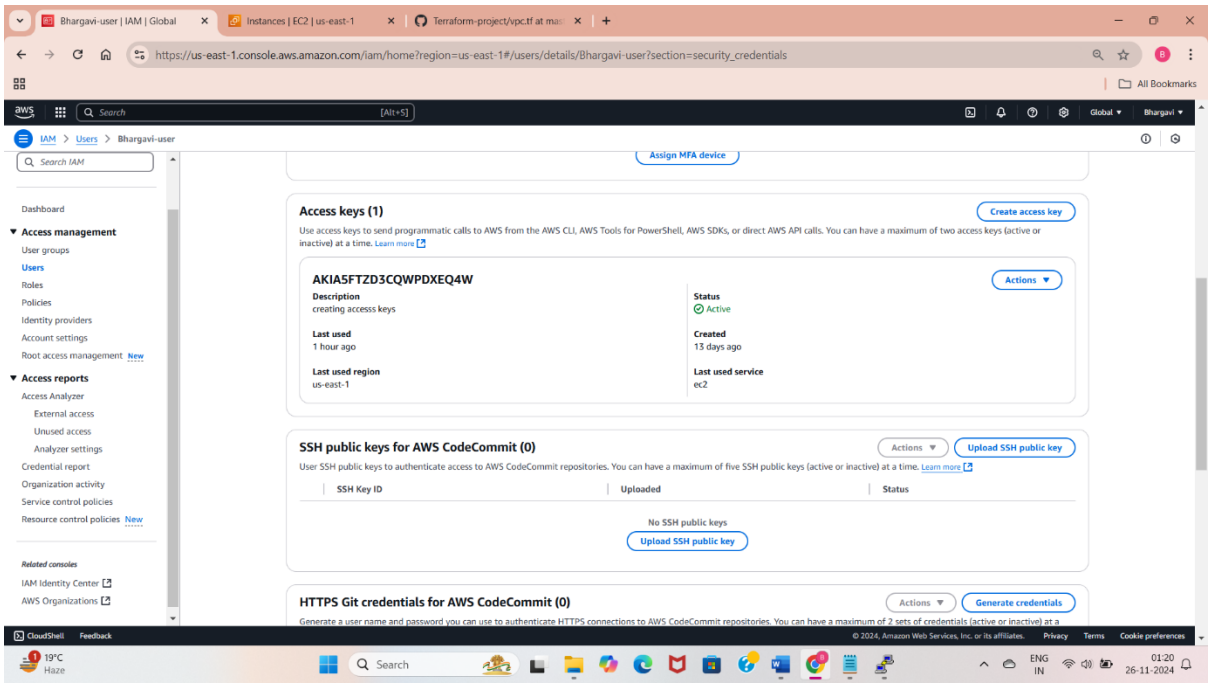
To create a files using terraform first we need to install terraform using the below commands

- sudo yum install -y yum-utils shadow-utils
- sudo yum-config-manager --add-repo https://rpm.releases.hashicorp.com/AmazonLinux/hashicorp.repo
- sudo yum -y install terraform



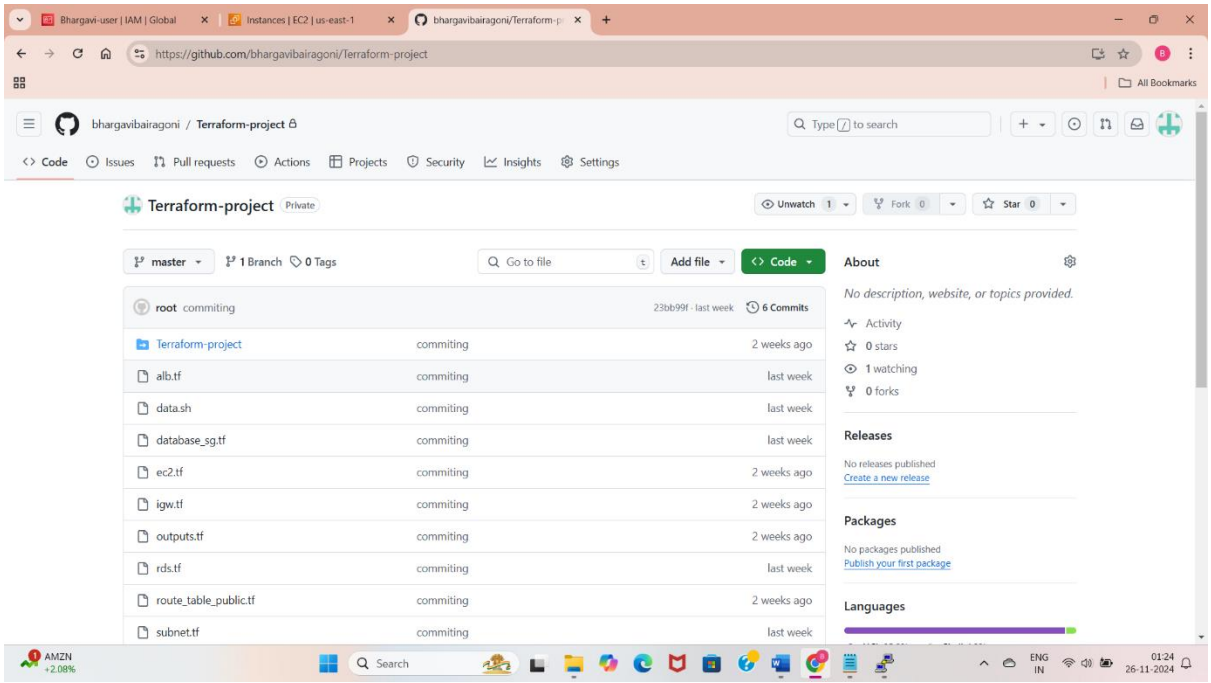
IAM USER

With the help of IAM user I have created access keys and secret access keys to provide security for my AWS account from unauthorized usage and unintended data loss.



GITHUB ACCOUNT

I have created a github account to store the data and code related to the terraform, so that I can access the code immediatedly whenever I want to access.



AWS ACCESS & SECRET KEY

- ✓ Access keys and secret access keys are necessary when using Terraform to manage AWS resources because they provide authentication and authorization for Terraform to interact with AWS on your behalf. AWS uses these credentials to verify that you have the appropriate permissions to create, modify, or delete resources in your account.
- ✓ Use <”aws configure”> to set access and secret keys.

STEP 1: CREATE A FILE FOR VPC

A Virtual Private Cloud (VPC) is a logically isolated section of the AWS cloud where we can launch AWS resources in a virtual isolated network. Here I am creating a file for vpc with the name of “vpc.tf”, and with the specific cidr.

We need to configure the aws for authentication. AWS configuration provides Terraform with the necessary permissions to provision and manage these resources.

If we don’t provide access key and secret access key terraform shows error while generating plan.

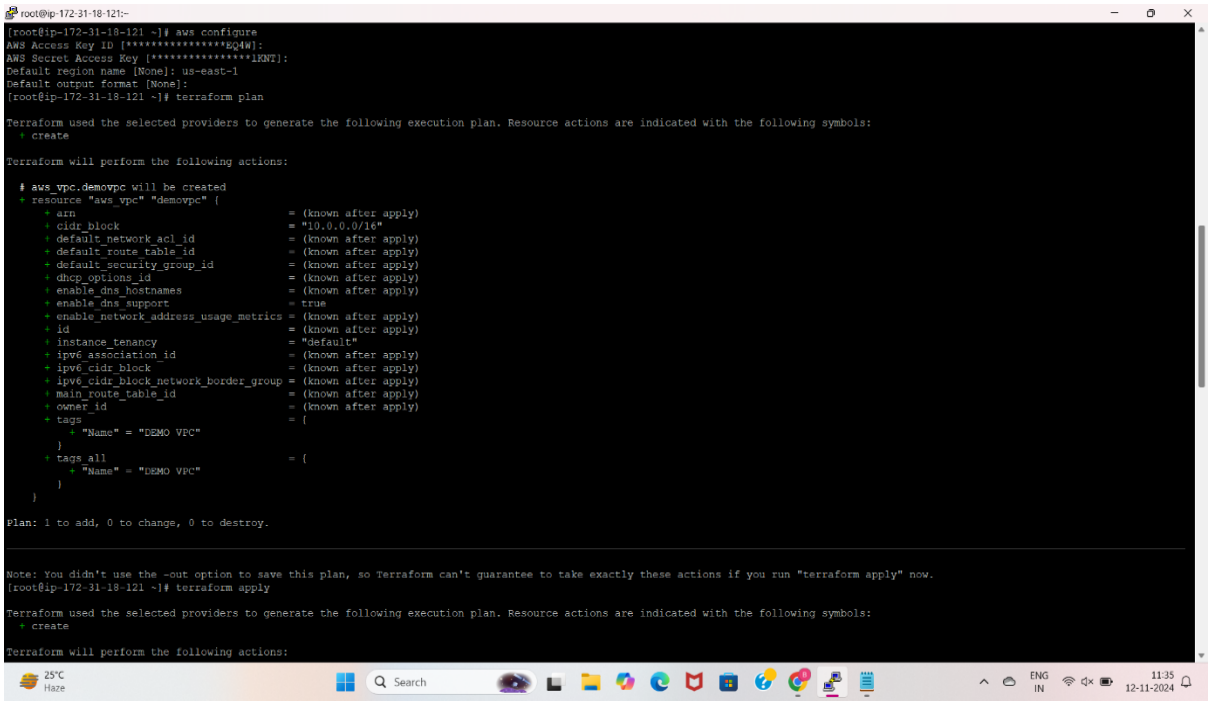
Here I did not provided any authentication so it is showing error. To resolve this I need to generate access keys using the user from IAM.

root@ip-172-31-80-26:~

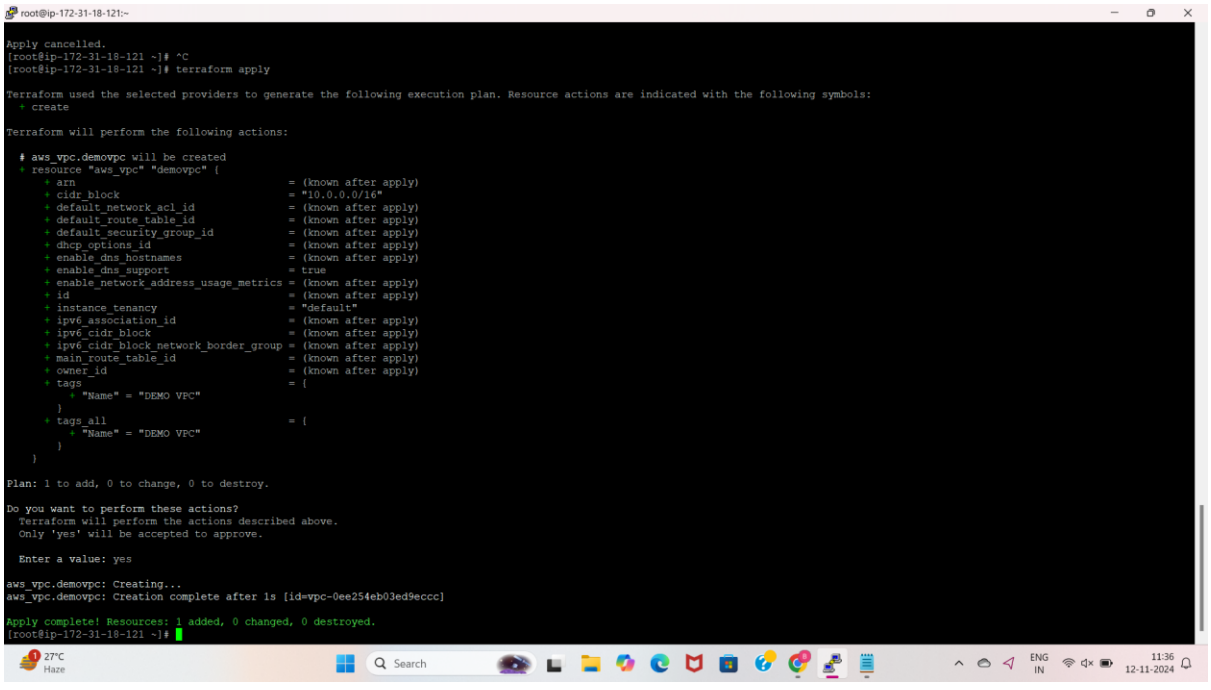
```
provider "aws" {
  access_key = "AKIA5FTZD3CQWPDXEQ4W"
  secret_key = "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"
  region     = "us-east-1"
}
resource "aws_vpc" "demovpc" {
cidr_block="10.0.0.0/16"
instance_tenancy="default"
tags={
Name="DEMO VPC"
}
}
~
```

When using Terraform to create a Virtual Private Cloud (VPC) in AWS, you need to ensure proper authentication and configuration of the AWS provider. The process involves defining the VPC in a Terraform configuration file (vpc.tf) and authenticating Terraform with AWS to enable resource provisioning.

This Terraform configuration sets up a provider for AWS and creates a Virtual Private Cloud (VPC) within the AWS environment. Cidr block defines the IP range for the VPC. This allows for a maximum of 65,536 IP addresses

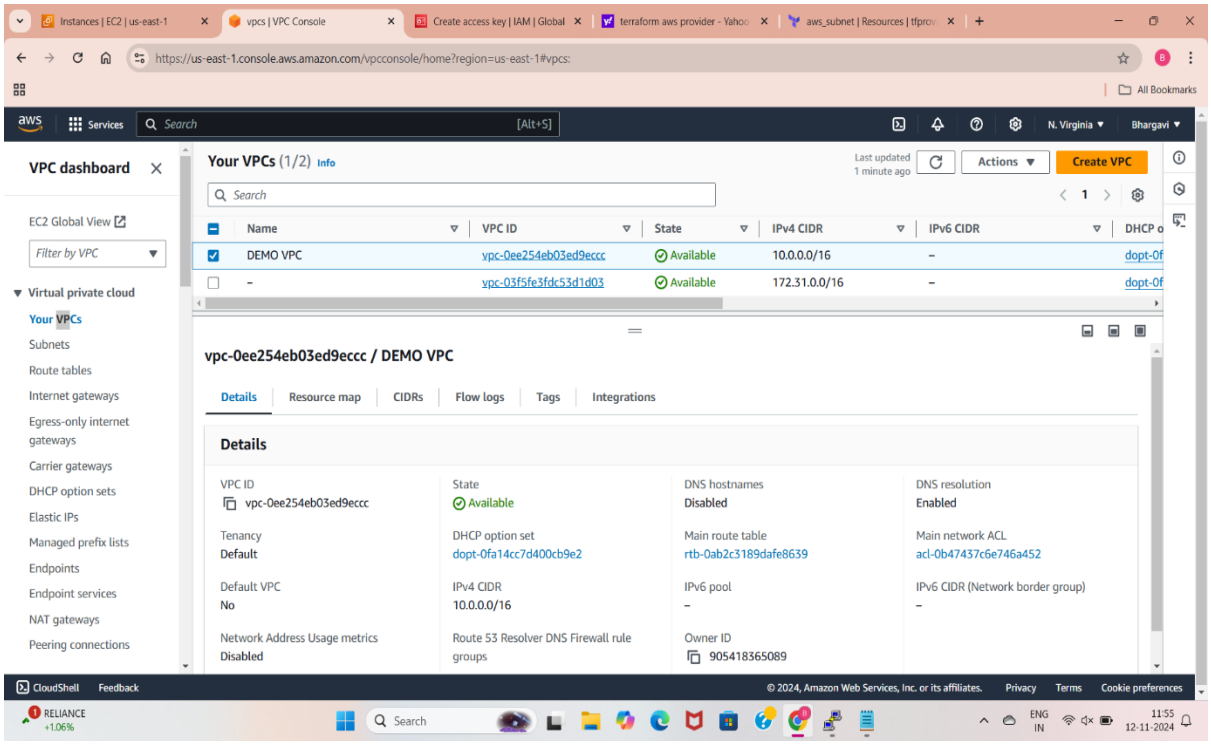


Plan for creating VPC is successful. Resource is used to create the service whatever we mention with the name. I have provided the resource with “aws_vpc” ,”demovpc” is the name given to the vpc to reference our vpc while using it. For example if we want to create subnets from vpc we can use this reference name.



After planning to create a vpc we need to apply the changes so that vpc or any services /infrastructure is created automatically. Here we need to run the command

“<terraform apply>”. It is showing whatever we mentioned in the code and there is no error, so it is asking for confirmation , we need to say “yes” if we want to create the vpc otherwise say “no”.



We can see DEMO VPC is created successfully in aws management console.

STEP 2: CREATE A FILE FOR THE SUBNET

A subnet is a subdivision of the IP address range within a VPC. Subnets allow you to allocate smaller, more manageable address spaces for groups of resources.

- After creating vpc we need to create subnets, here am creating 6 subnets. Creating multiple subnets in a Virtual Private Cloud (VPC) allows you to organize and secure your resources based on different requirements, such as availability, security, and access control. Each subnet will be allocated a specific range of IP addresses, and you can deploy your resources (like EC2 instances) into these subnets.
- Subnets in AWS are defined by CIDR blocks, and we need to decide the size of each subnet. A common practice is to divide the VPC into subnets with /24 CIDR blocks, each providing 256 IP addresses (of which 251 are usable). You can adjust the sizes as needed, such as /25 or /23, depending on the number of IPs required per subnet.
- It's recommended to distribute your subnets across different Availability Zones (AZs) to ensure high availability and fault tolerance. AWS regions have multiple AZs (e.g., us-east-1a, us-east-1b, us-east-1c), and each subnet should be assigned to one AZ.

root@ip-172-31-80-26:~

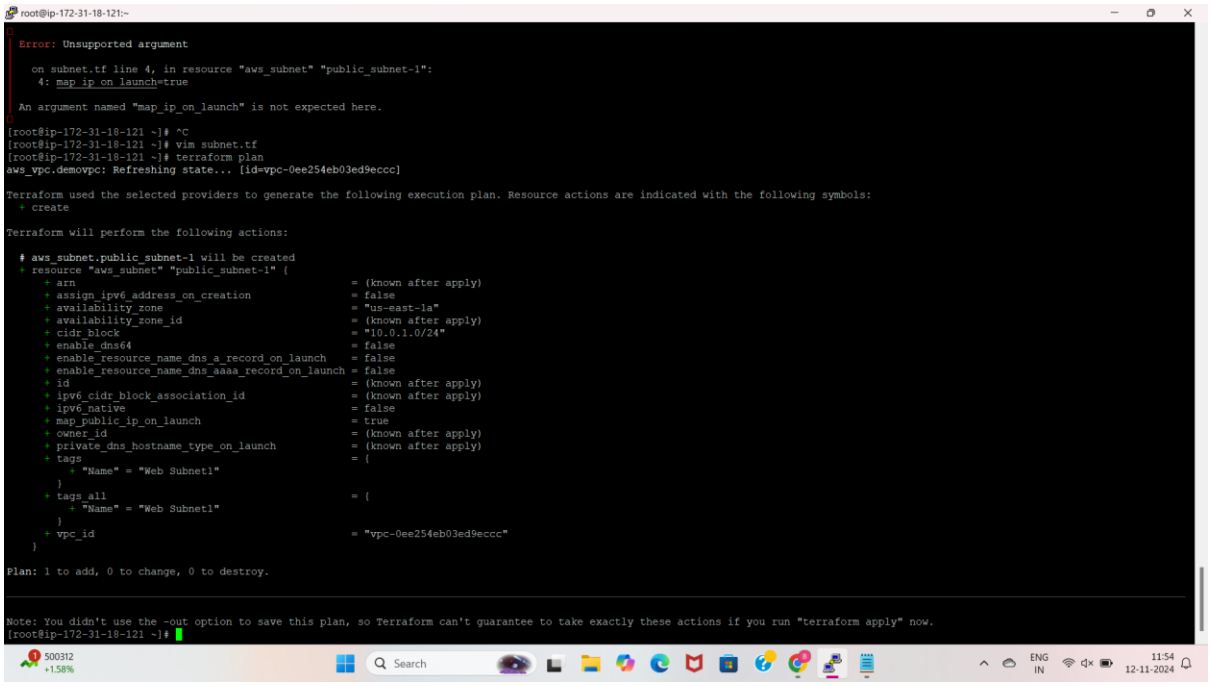
```
resource "aws_subnet" "public_subnet-1" {
vpc_id="${aws_vpc.demovpc.id}"
cidr_block="10.0.1.0/24"
map_public_ip_on_launch=true
availability_zone="us-east-1a"
tags={
Name="Web Subnet 1"
}
}
#creating public subnet
resource "aws_subnet" "public_subnet-2" {
vpc_id="${aws_vpc.demovpc.id}"
cidr_block="10.0.2.0/24"
map_public_ip_on_launch=true
availability_zone="us-east-1b"
tags={
Name="WEB SUBNET2"
}
}
#creating application subnet 1
resource "aws_subnet" "applicatiom-subnet-1" {
vpc_id="${aws_vpc.demovpc.id}"
cidr_block="10.0.3.0/24"
map_public_ip_on_launch=false
availability_zone="us-east-1a"
tags={
Name="Application Subnet 1"
}
}
#creating application subnet
resource "aws_subnet" "application-subnet-2" {
vpc_id="${aws_vpc.demovpc.id}"
cidr_block="10.0.4.0/24"
map_public_ip_on_launch=false
availability_zone="us-east-1b"
tags={
Name="Application Subnet 2"
}
}

resource "aws_subnet" "database-subnet-1" {
vpc_id="${aws_vpc.demovpc.id}"
cidr_block="10.0.5.0/24"
availability_zone="us-east-1a"
tags={
Name="Database Subnet 1"
}
}
-- INSERT --
```

```
resource "aws_subnet" "database-subnet-2" {
  vpc_id="${aws_vpc.demovpc.id}"
  cidr_block="10.0.6.0/24"
  availability_zone="us-east-1b"
  tags={
    Name="Database Subnet 2"
  }
}
-- INSERT --
```

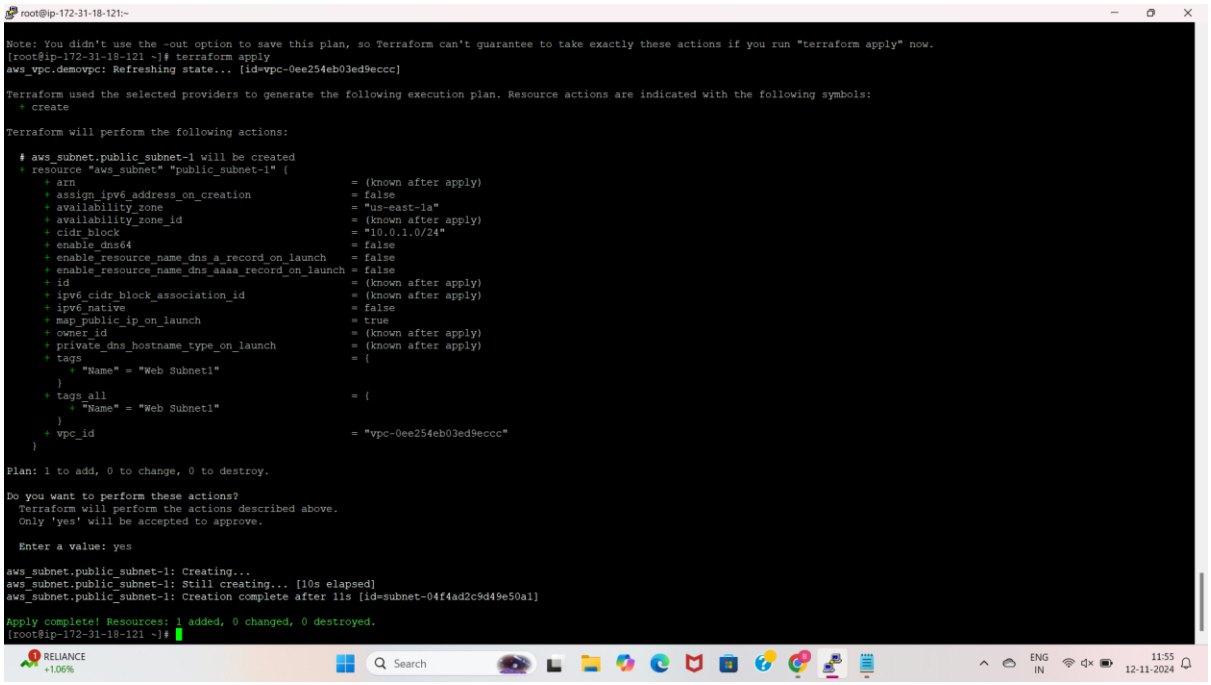
This Terraform configuration defines **six subnets** within a Virtual Private Cloud (VPC) ,each subnet is configured with a CIDR block, availability zone, and attributes suitable for its purpose.

- ✓ Frontend Tier (Public Subnets): Accessible from the internet.

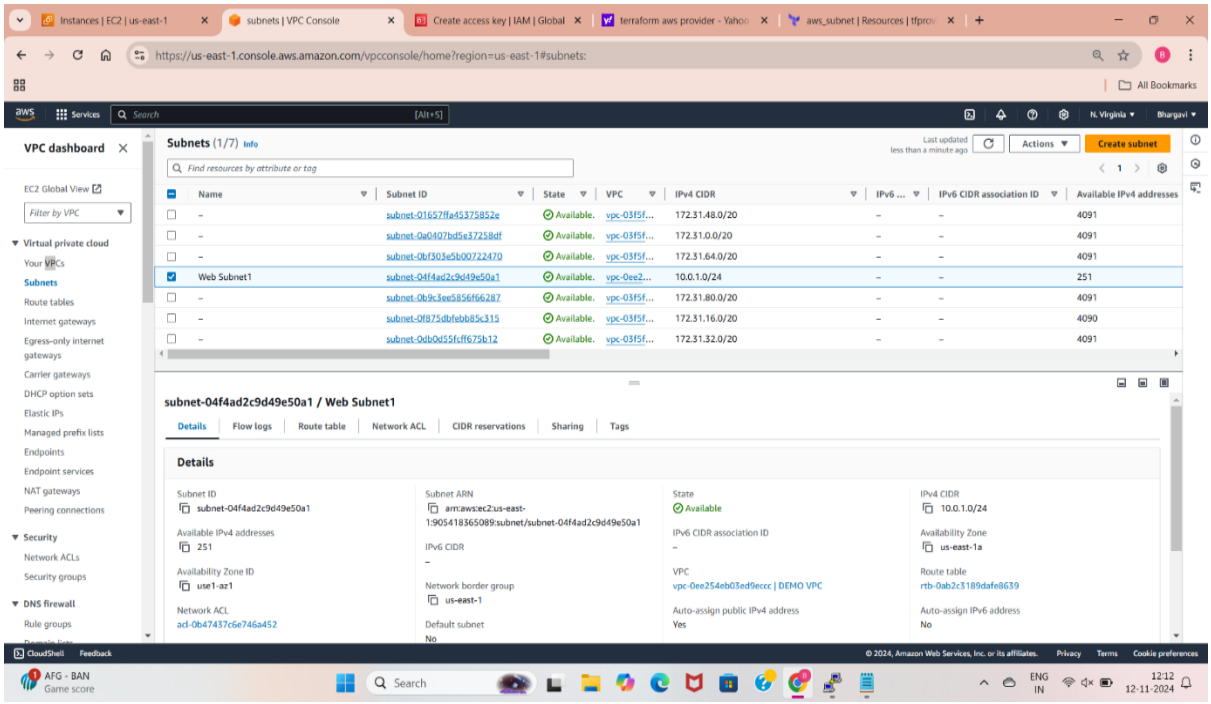
- ✓ Application Tier (Private Application Subnets): For internal processing, no direct internet access.
- ✓ Database Tier (Private Database Subnets): Fully private, isolated from the internet for security.



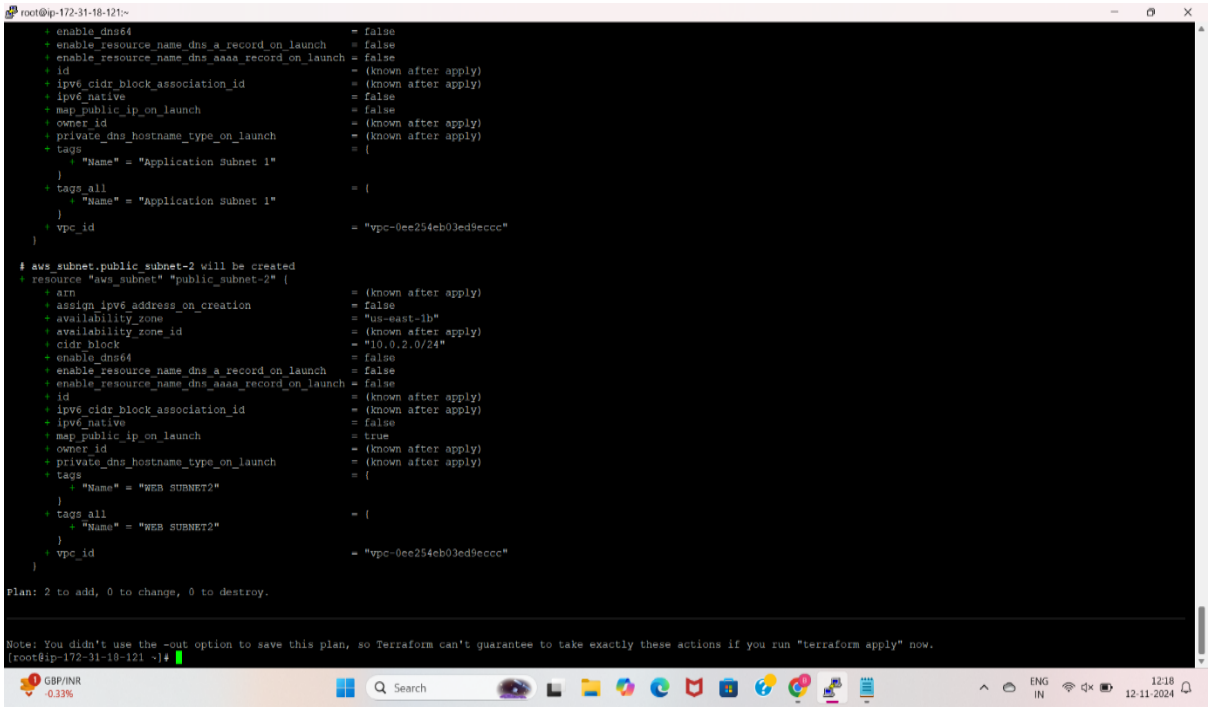
Creating plan for the subnet by using “terraform plan”. We can what we are configured in the code and what is going to create in the first subnet.



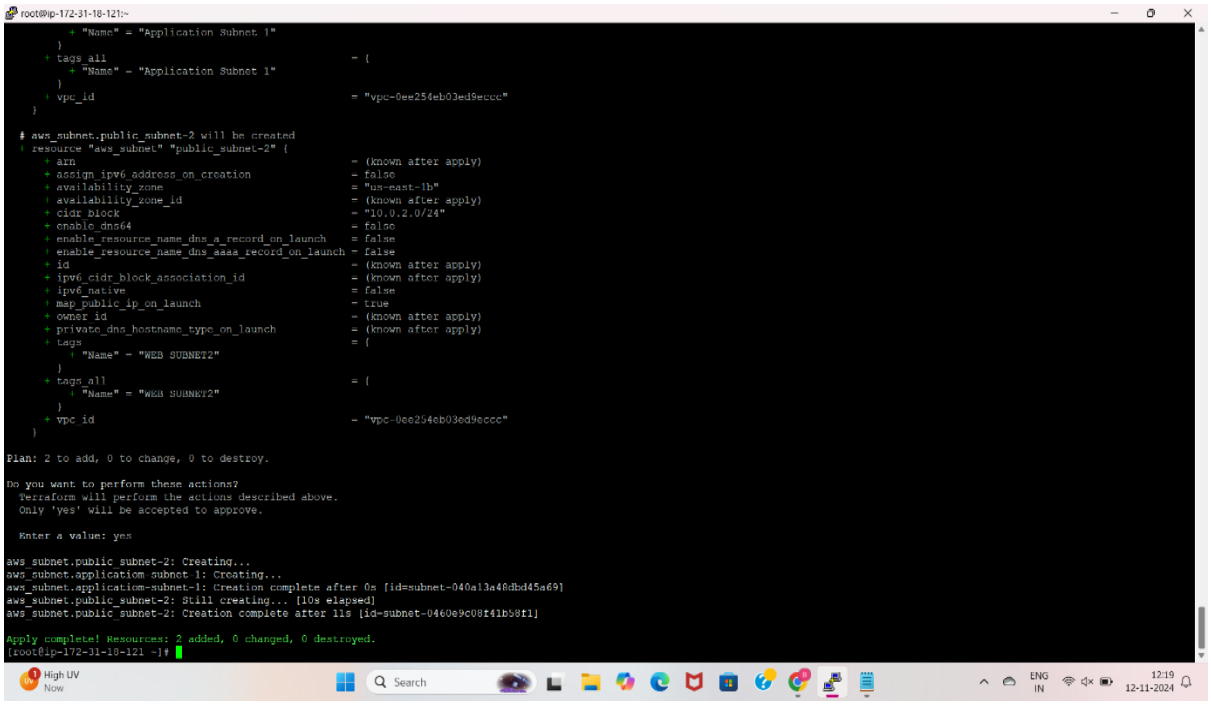
After using “terraform apply” we can see subnet1 is creating.



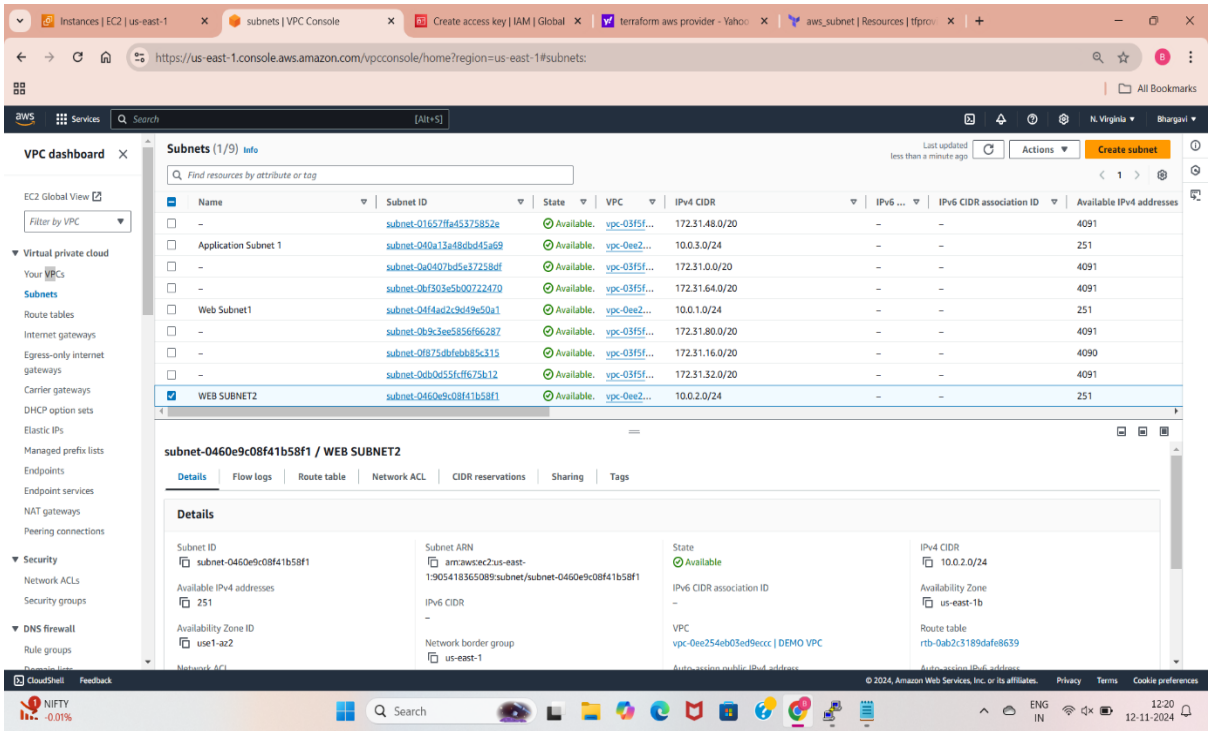
The first subnet with the name “Web Subnet1” is created



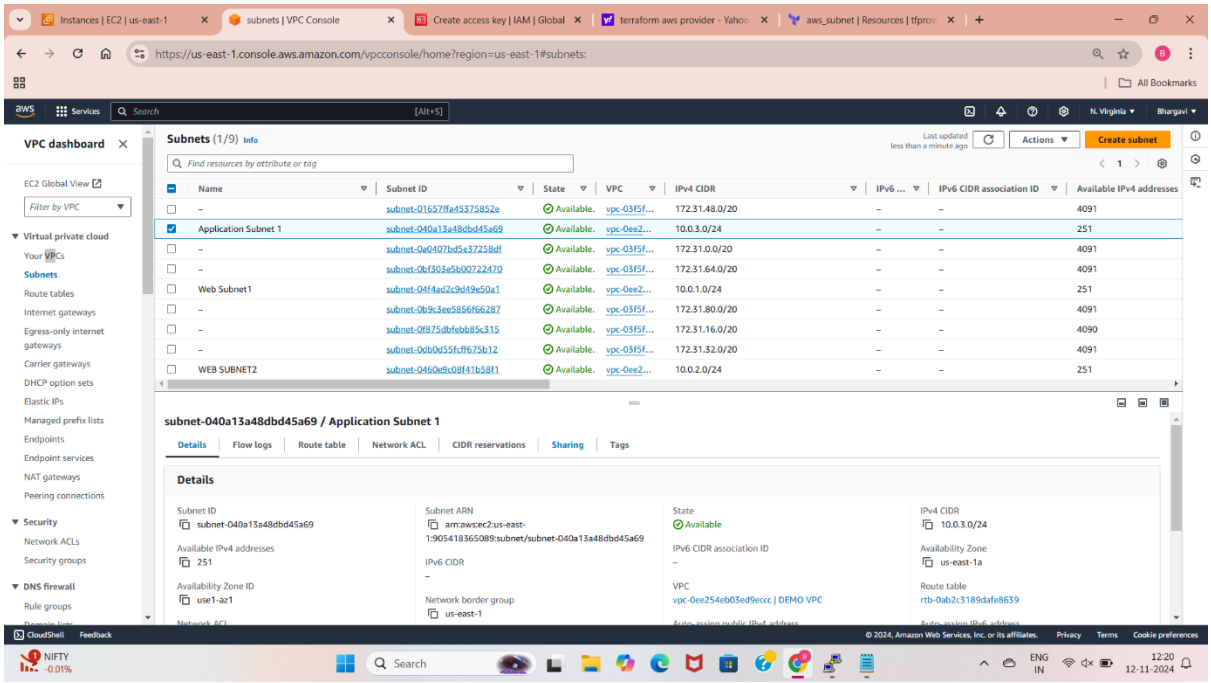
Here I have planned to create second subnet by writing code for second subnet.



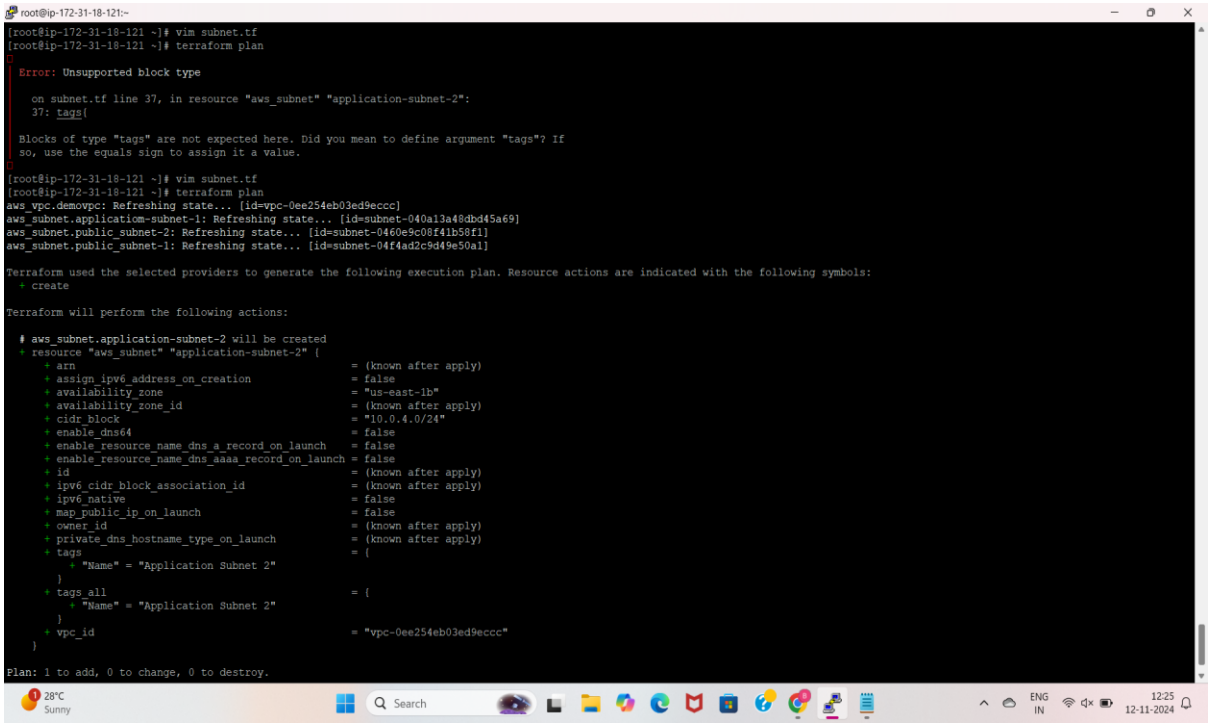
I have created second subnet with the command “terraform apply”.



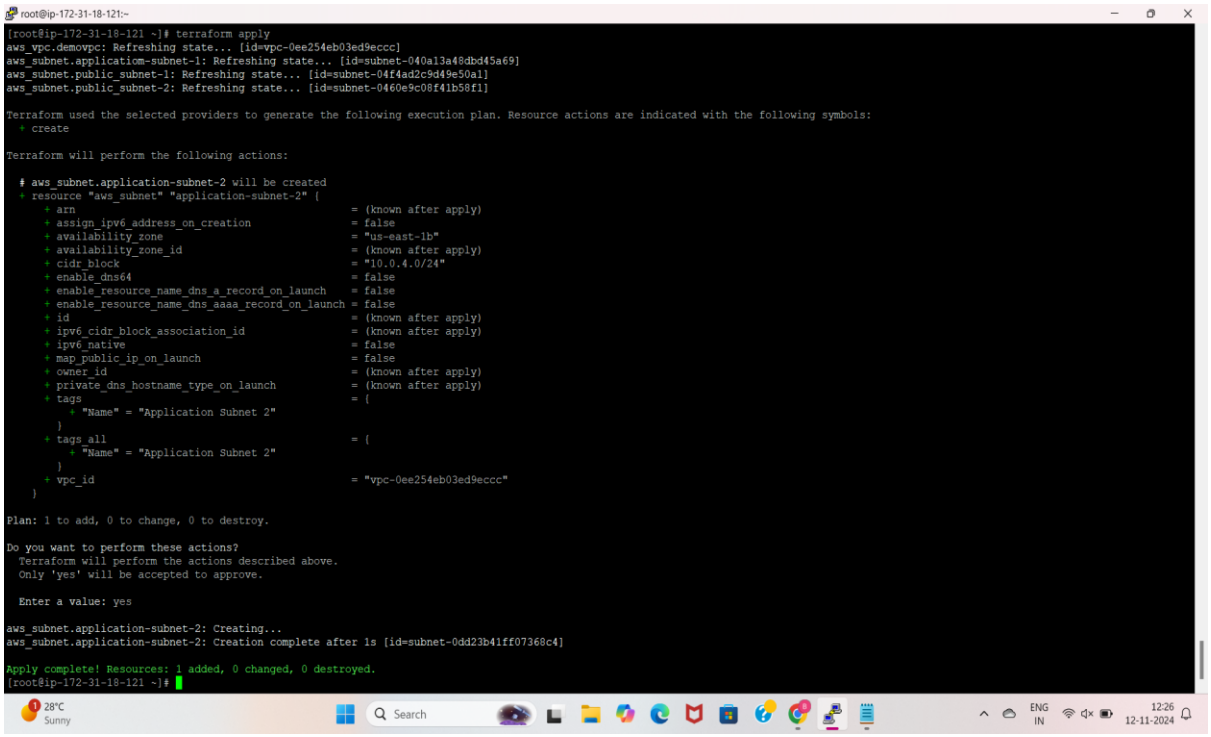
We can see the second subnet in the aws management console.



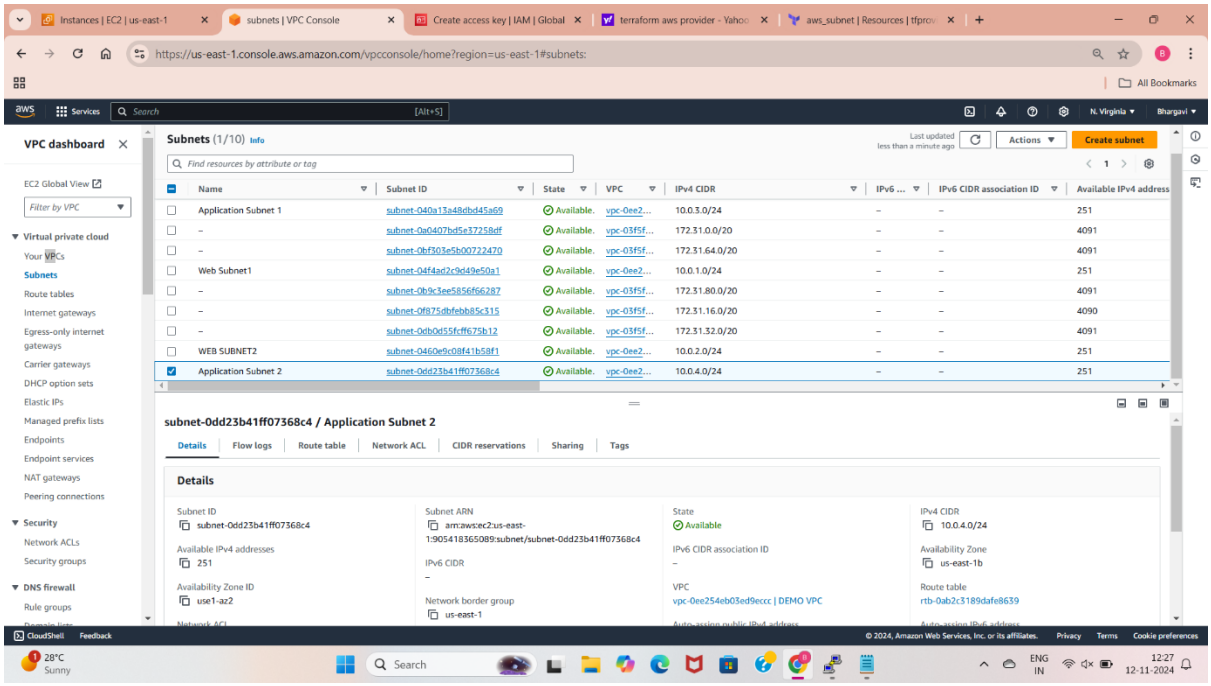
Created third subnet with the name of “application subnet 1”.



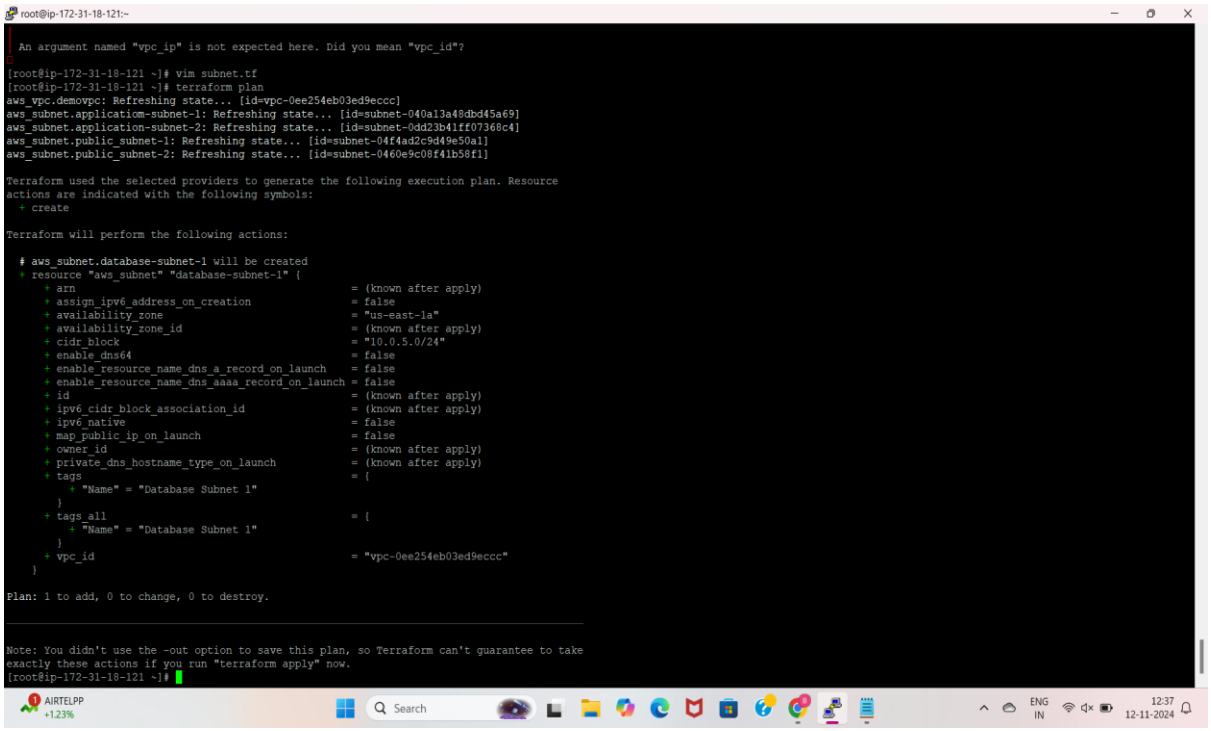
Creating fourth subnet with the help of code and used “terraform plan” to see the configurations that are going to create in the aws console are showing in the terminal.



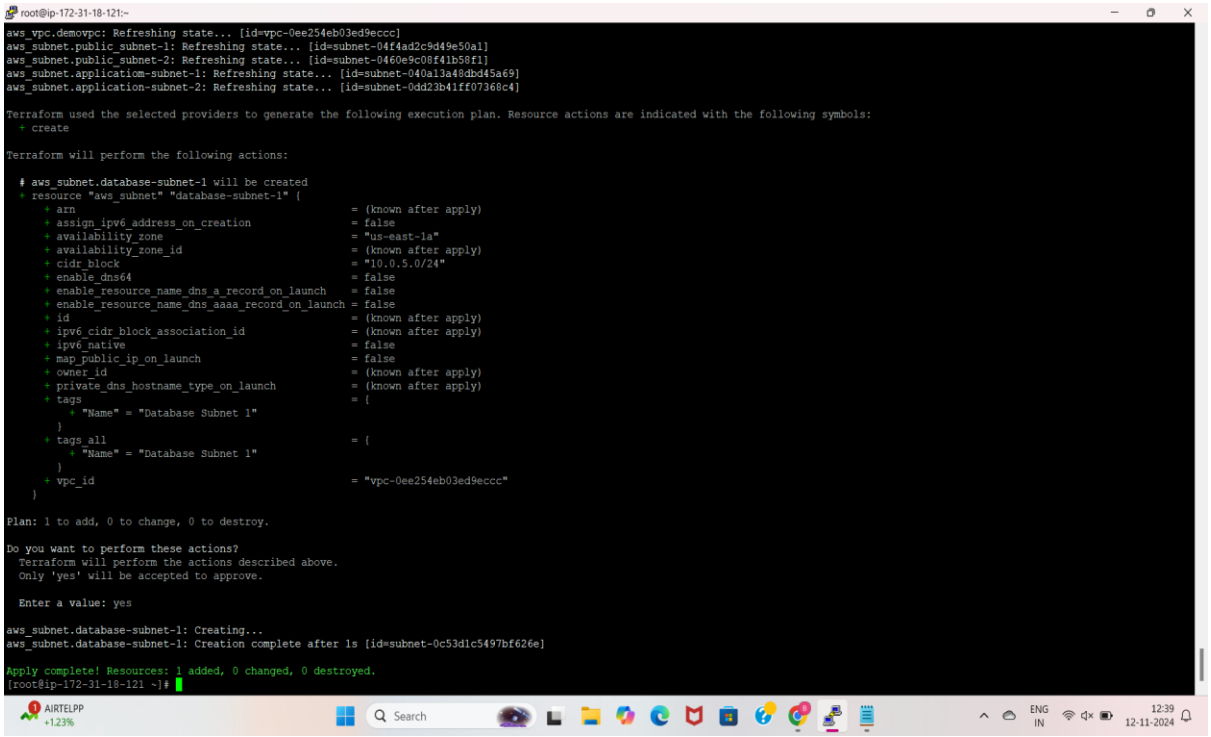
I have used “terraform apply” to create fourth subnet with the specified configurations.



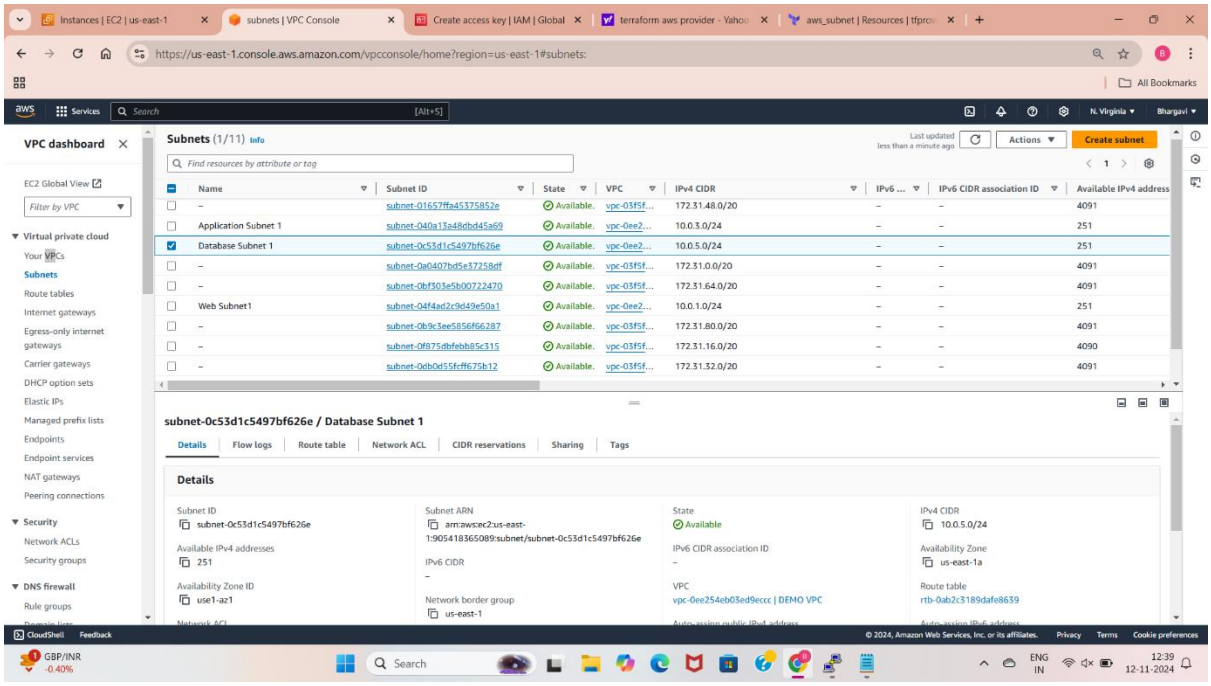
Fourth subnet with the name of “Application Subnet 2” is created successfully with the defined cidr block.



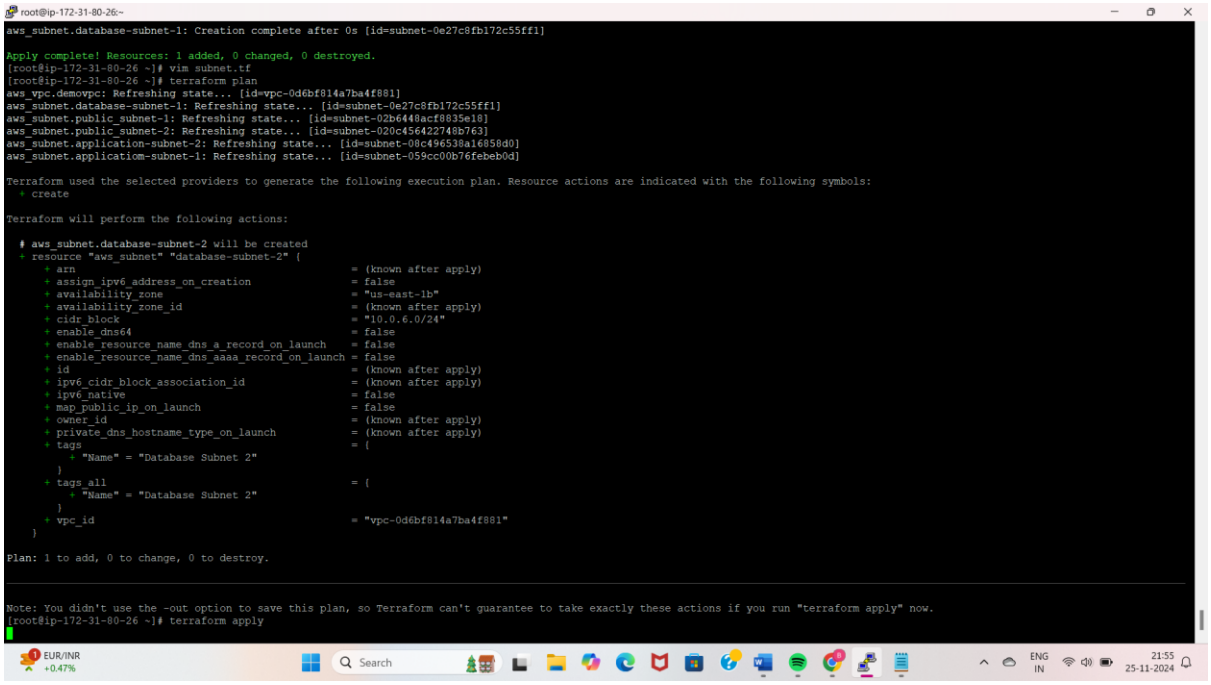
Creating a new subnet with different cidr 10.0.5.0/24, the name of the subnet is “database-subnet-1”.



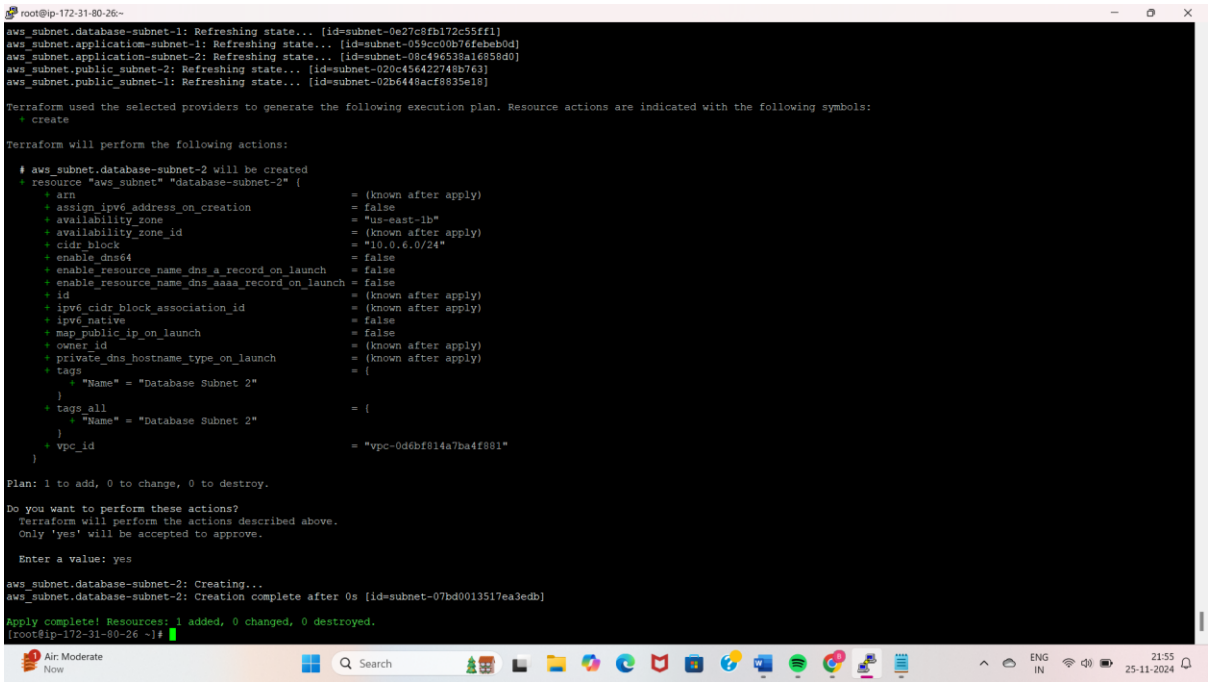
To apply the above mentioned configurations we need use terraform apply, then the subnet is going to create. We can see the creation of the subnet is successful.



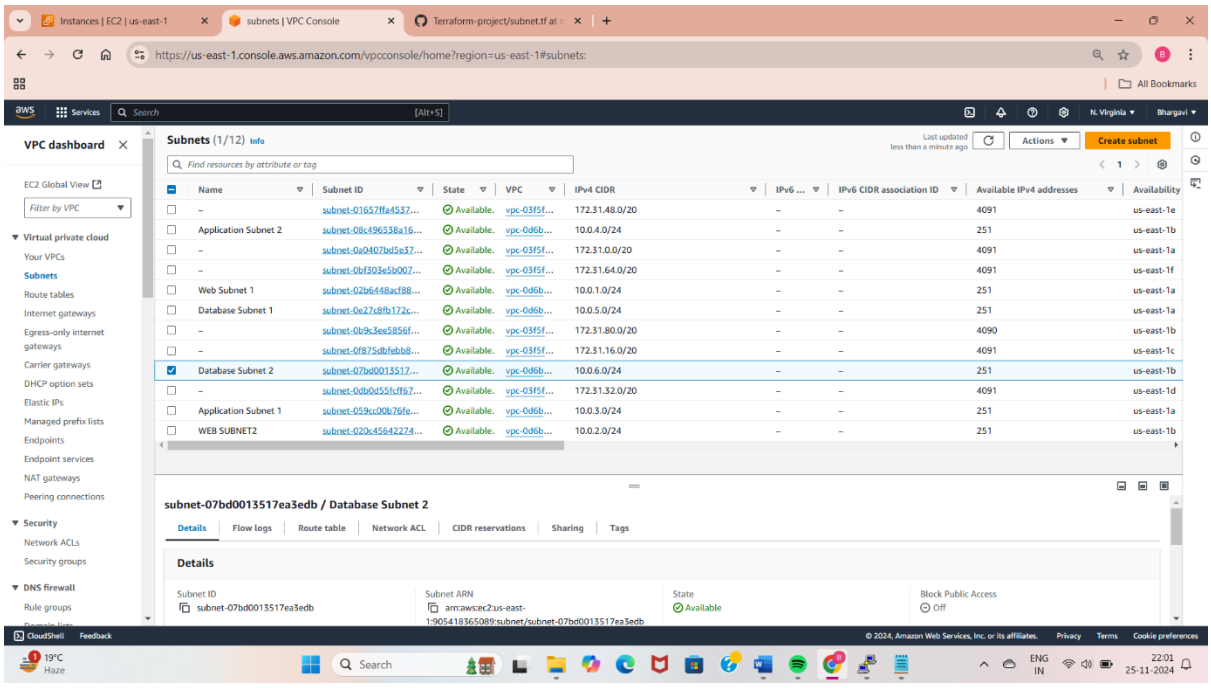
Database Subnet 1 is created in aws management console.



Creating 6th subnet with availability zone us-east-1b, cidr block of 10.0.6.0/24. The Terraform plan command executed successfully with no mistake in the code.



After using “terraform apply” the database subnet is creating with the sunet id.



Aws management console shows that the 6th subnet is created successfully.

STEP 3: CREATE INTERNET GATE WAY

Internet Gateway is used to connect our vpc to the internet, so that we can send and receive data from the resources of virtual private cloud.

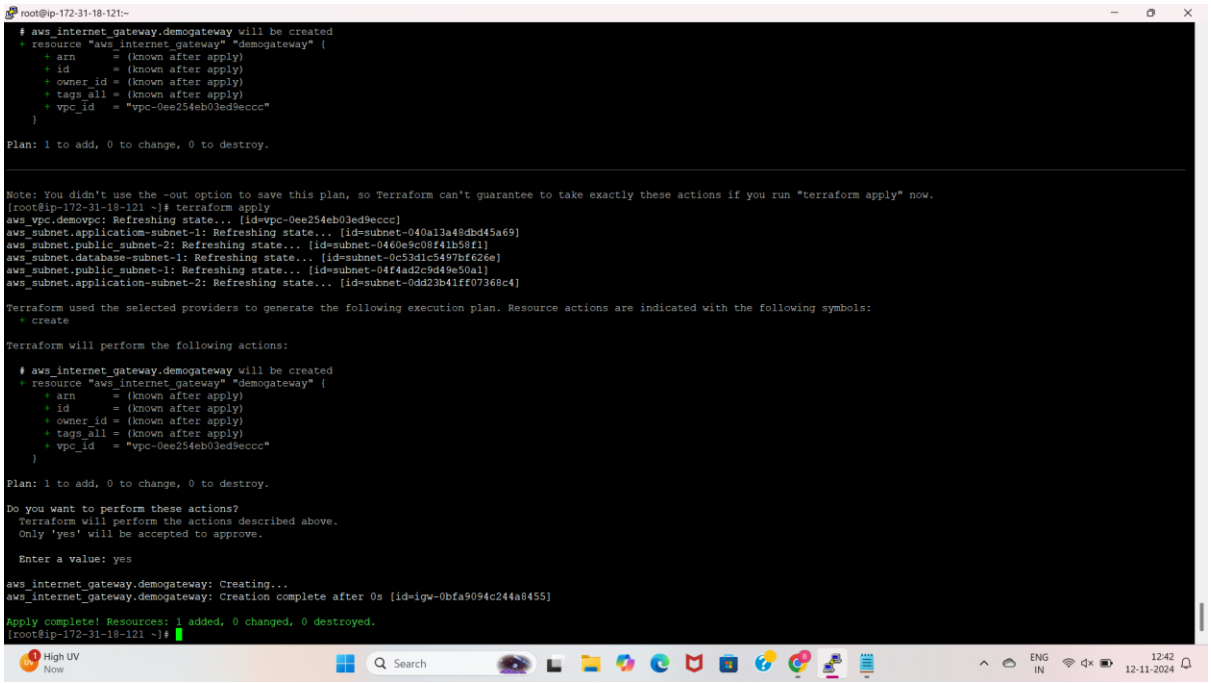
root@ip-172-31-80-26:~

```
resource "aws_internet_gateway" "demogateway" {
vpc_id="${aws_vpc.demovpc.id}"
tags={
Name="Internet Gateway"
}
}
~
~
~
```

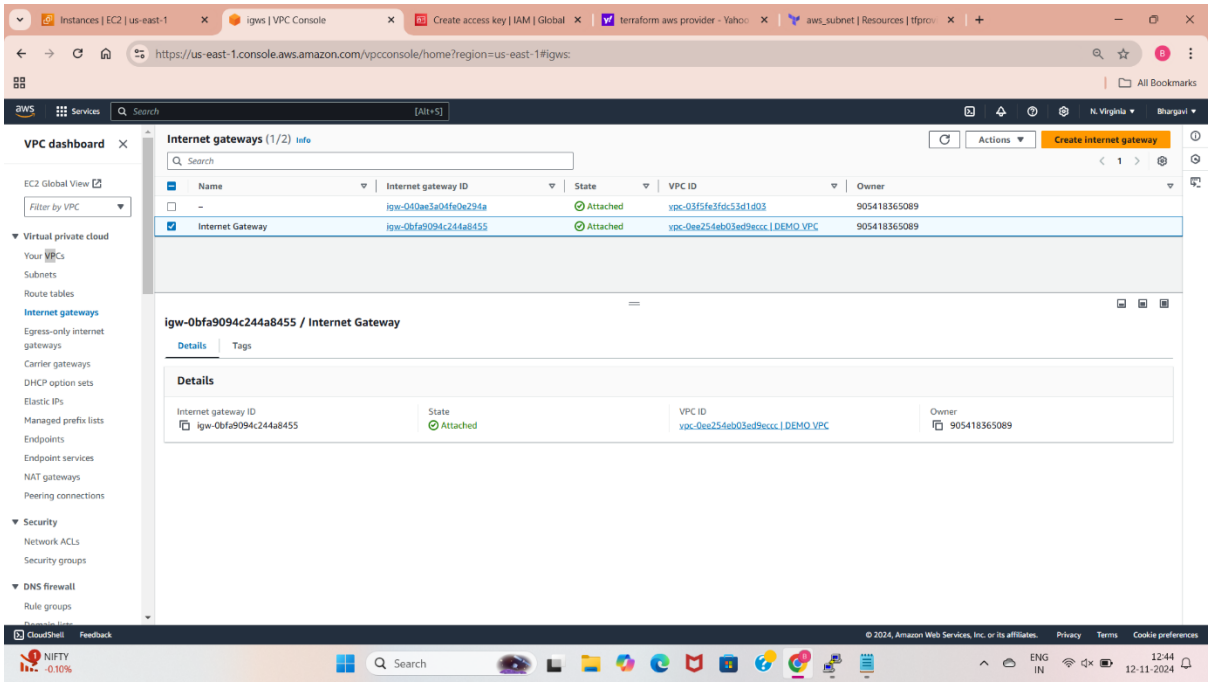
This is the code for creating internet gateway. Here we use the aws_internet_gateway resource type to create internet gateway for sending and receiving data from the resources.

```
root@ip-172-31-80-26:~
Plan: 1 to add, 0 to change, 0 to destroy.

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

aws_subnet.database-subnet-2: Creating...
aws_subnet.database-subnet-2: Creation complete after 0s [id=subnet-07bd0013517ea3edb]

Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
[root@ip-172-31-80-26 ~]# vim subnet.tf
[root@ip-172-31-80-26 ~]# vim vpc.tf
[root@ip-172-31-80-26 ~]# vim igw.tf
[root@ip-172-31-80-26 ~]# terraform plan
aws_vpc.demovpc: Refreshing state... [id=vpc-0d6bf814a7ba4f881]
aws_subnet.public_subnet-1: Refreshing state... [id=subnet-02b6448acf8835e18]
aws_subnet.applicatiom-subnet-1: Refreshing state... [id=subnet-059cc00b76febeb0d]
aws_subnet.application-subnet-2: Refreshing state... [id=subnet-08c496538a16858d0]
aws_subnet.database-subnet-1: Refreshing state... [id=subnet-0e27c8fb172c55ff1]
aws_subnet.database-subnet-2: Refreshing state... [id=subnet-07bd0013517ea3edb]
aws_subnet.public_subnet-2: Refreshing state... [id=subnet-020c456422748b763]

Terraform used the selected providers to generate the following execution plan. Resource actions are indicated with the following symbols:
  + create

Terraform will perform the following actions:

  # aws_internet_gateway.demogateway will be created
  + resource "aws_internet_gateway" "demogateway" {
      + arn      = (known after apply)
      + id       = (known after apply)
      + owner_id = (known after apply)
      + tags     = {
          + "Name" = "Internet Gateway"
        }
      + tags_all = {
          + "Name" = "Internet Gateway"
        }
      + vpc_id   = "vpc-0d6bf814a7ba4f881"
    }

Plan: 1 to add, 0 to change, 0 to destroy.

Note: You didn't use the -out option to save this plan, so Terraform can't guarantee to take exactly these actions if you run "terraform apply" now.
[root@ip-172-31-80-26 ~]#
```

This terminal shows that “terraform plan” command is performed to create internet gate way. There is no any mistake in the configurations so the plan is successful.



Whatever internet gateway is going to create in aws console, the configurations are shown here.



Internet gateway is created successfully.

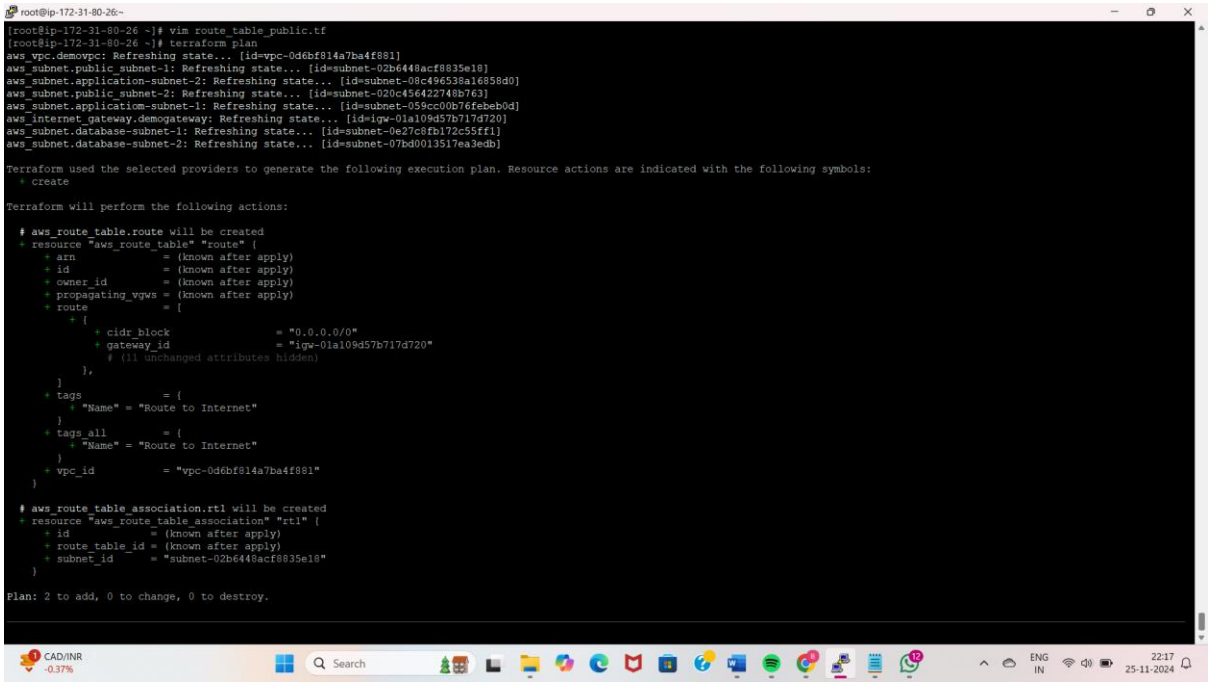
STEP 4: CREATING THE ROUTING TABLE

Route table is used to determine how the traffic is directed with in the vpc.

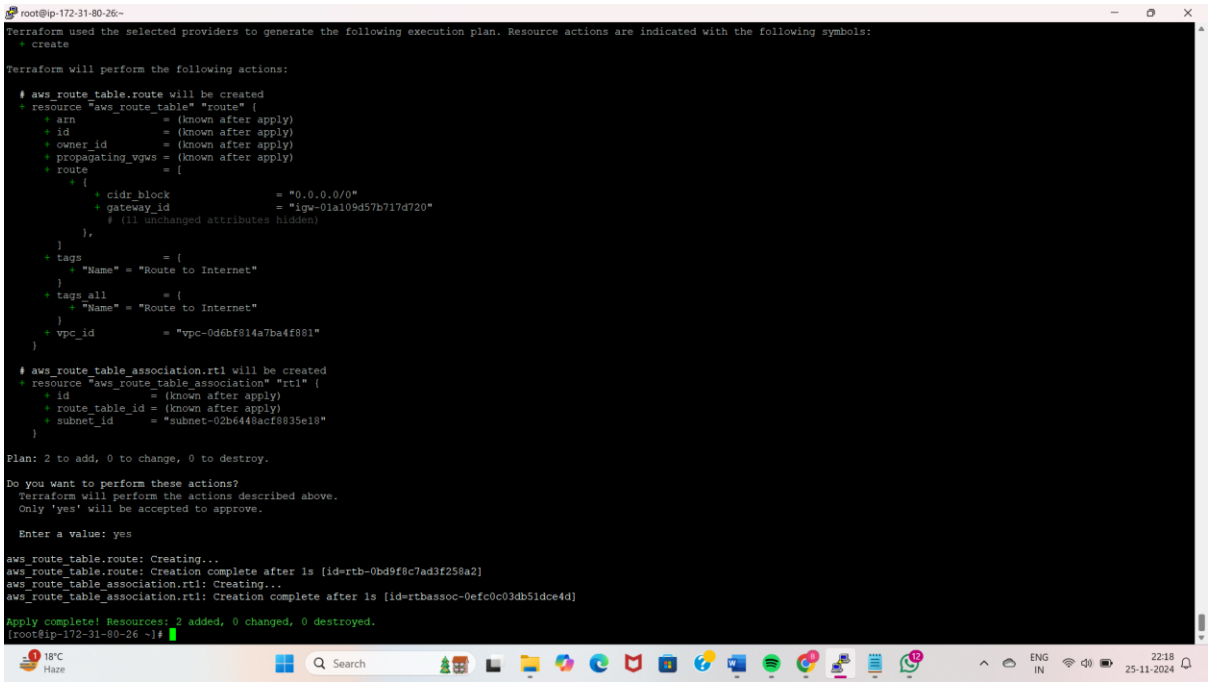
To create route table and associate it with the vpc and internet gate way we need to create a route table file with the name of route_table_public.tf.

root@ip-172-31-80-26:~

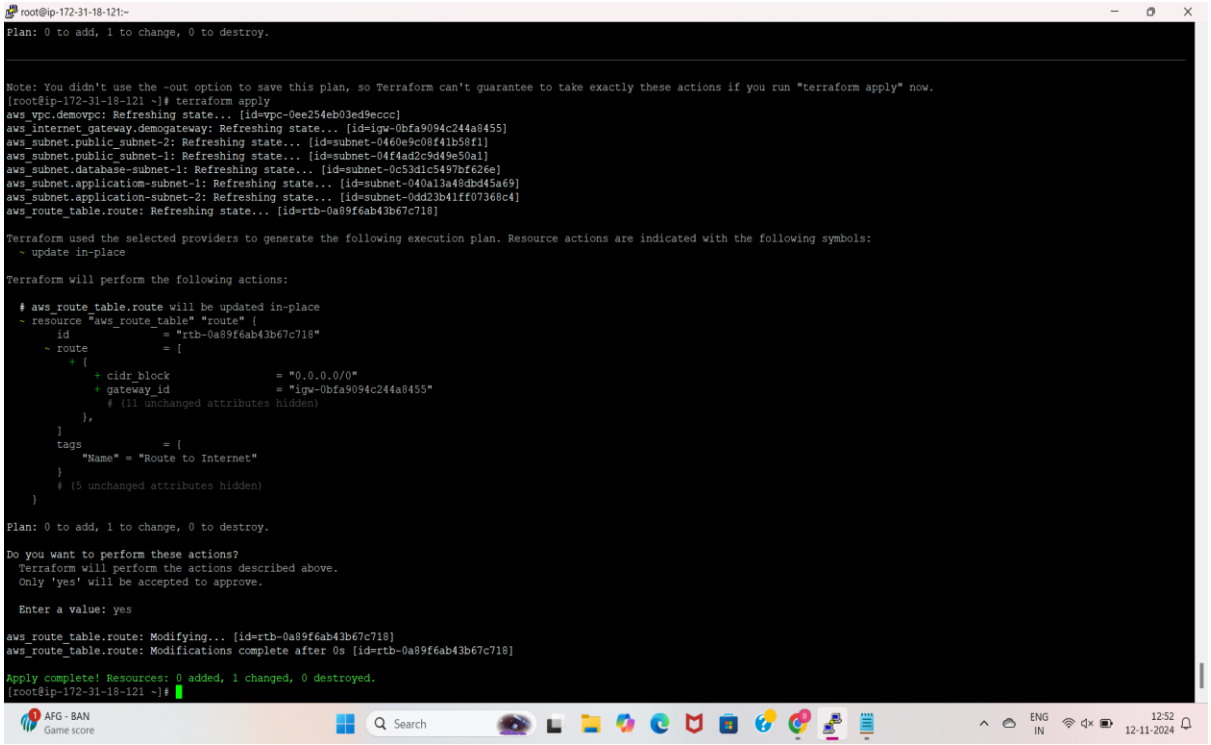
```
resource "aws_route_table" "route" {
vpc_id="${aws_vpc.demovpc.id}"
route{
cidr_block="0.0.0.0/0"
gateway_id="${aws_internet_gateway.demogateway.id}"
}
tags={
Name="Route to Internet"
}
}

resource "aws_route_table_association" "rt1" {
subnet_id="${aws_subnet.public_subnet-1.id}"
route_table_id="${aws_route_table.route.id}"
}

resource "aws_route_table_association" "rt2" {
subnet_id="${aws_subnet.public_subnet-2.id}"
route_table_id="${aws_route_table.route.id}"
}
~
~
~
```



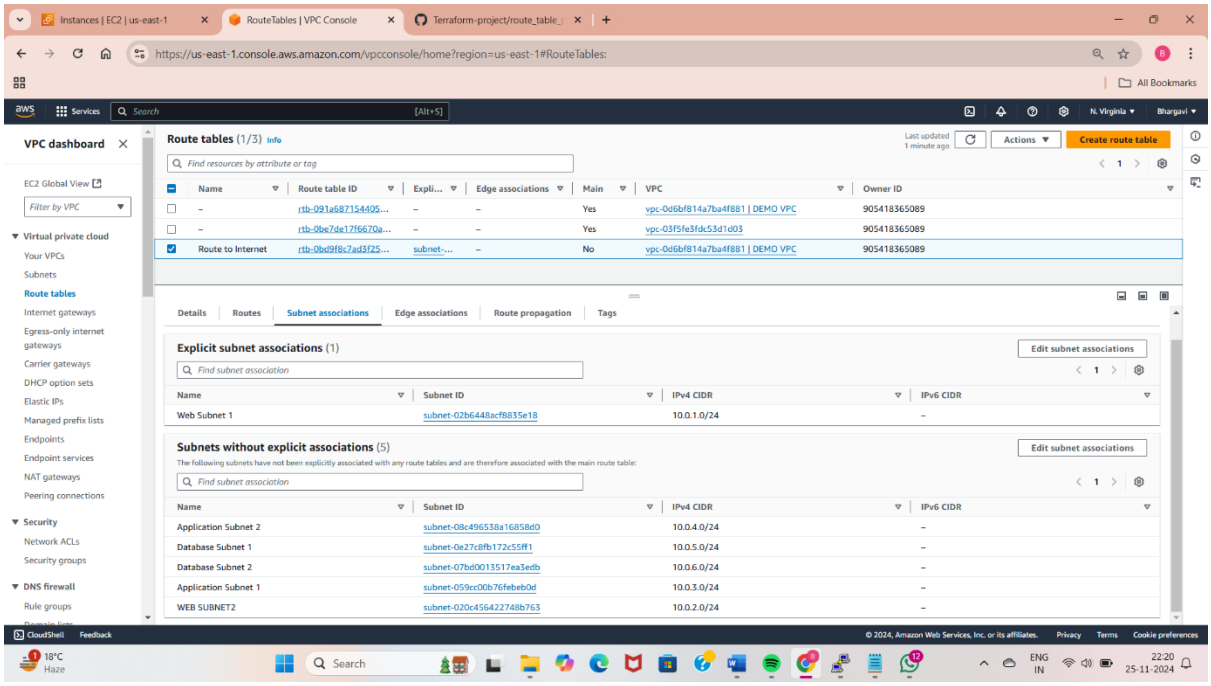
The plan for route table is created using the command “terraform plan”.



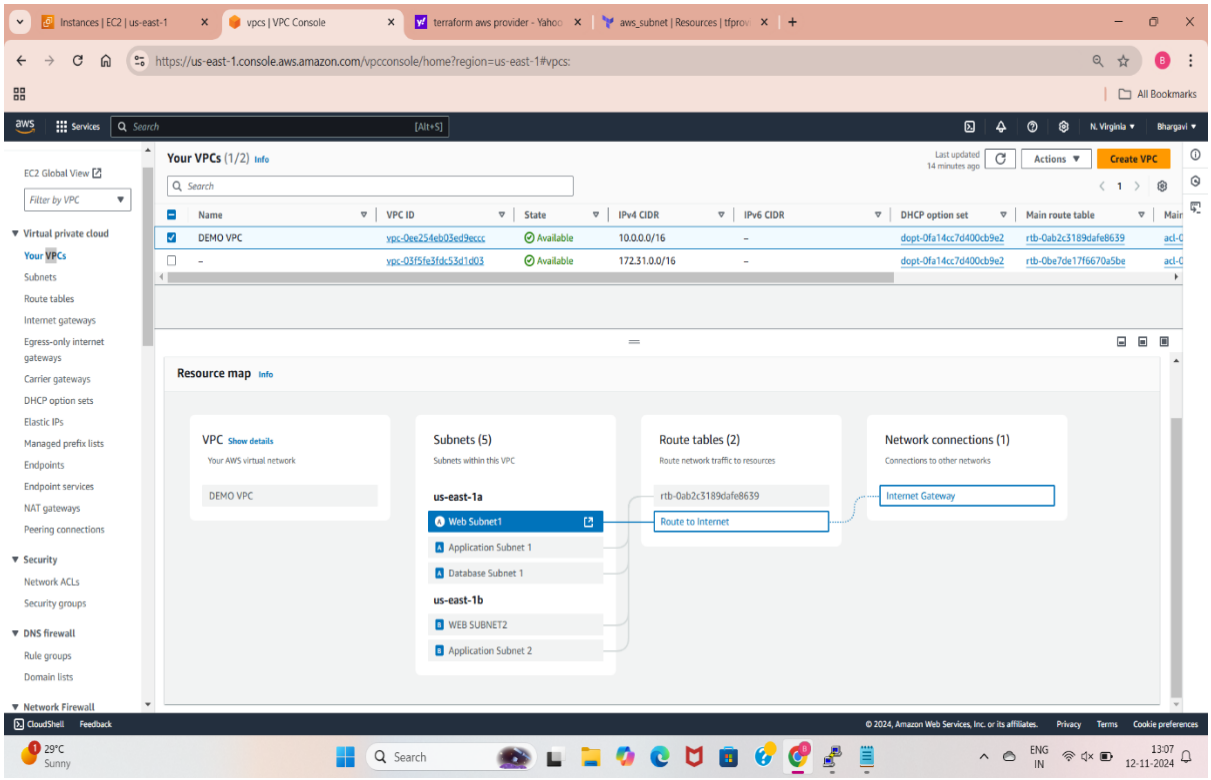
Here I have executed the command terraform apply to create a route table and associate it with the vpc.



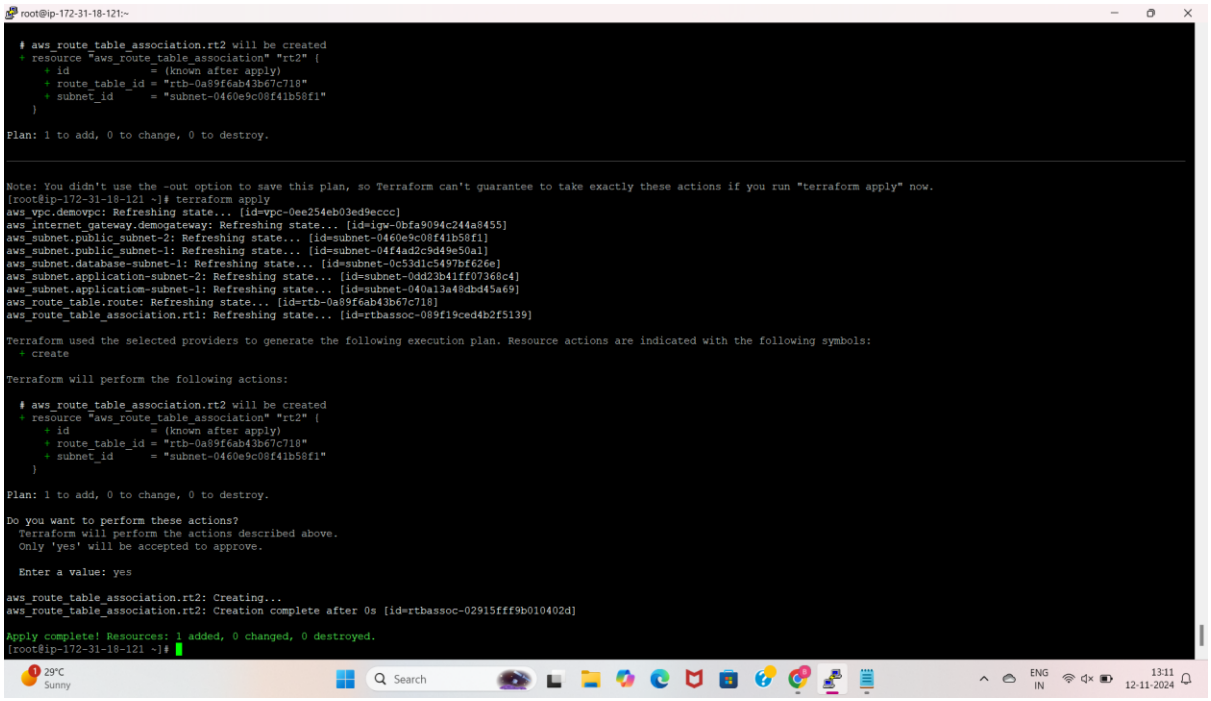
This show that there is no error in the configurations to create the route table.



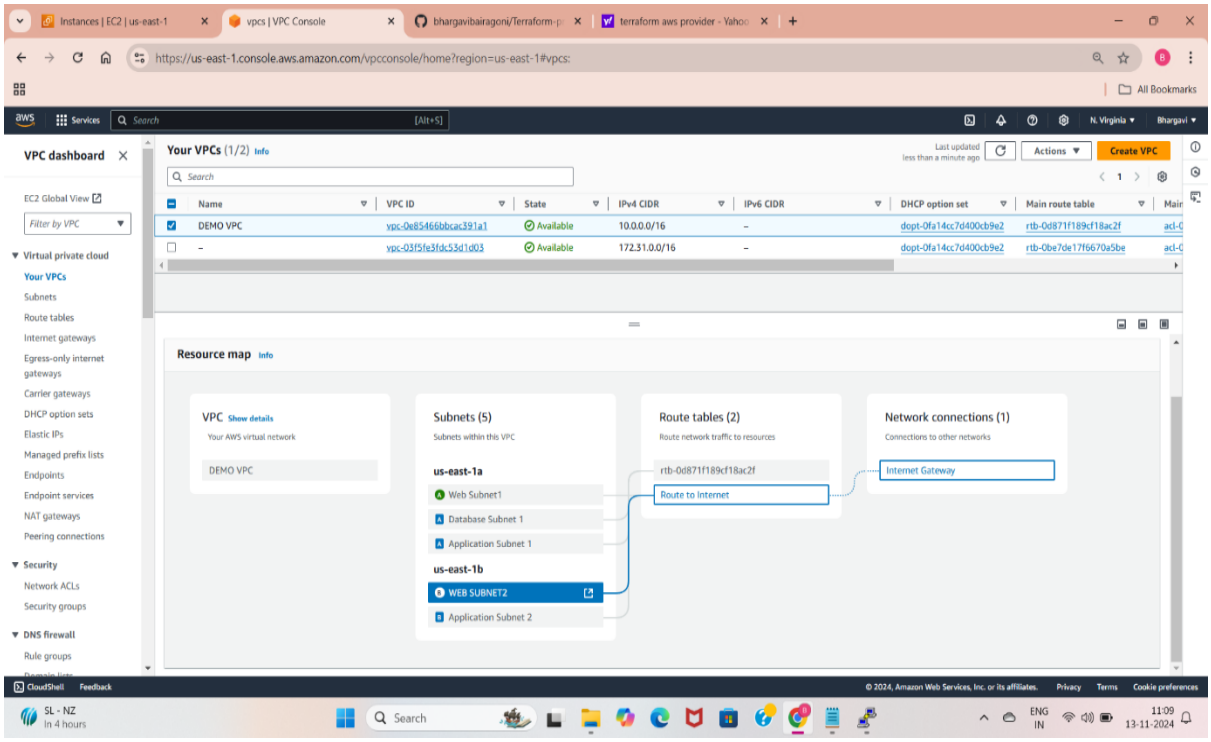
Route table is created to allow the traffic from the internet gateway to subnet.



We can the route table is associated with the internet gateway and web subnet1 successfully.



Creating the second subnet to associate it with the other web subnet.



Here second route table is associated with the internet gateway and web subnet2.

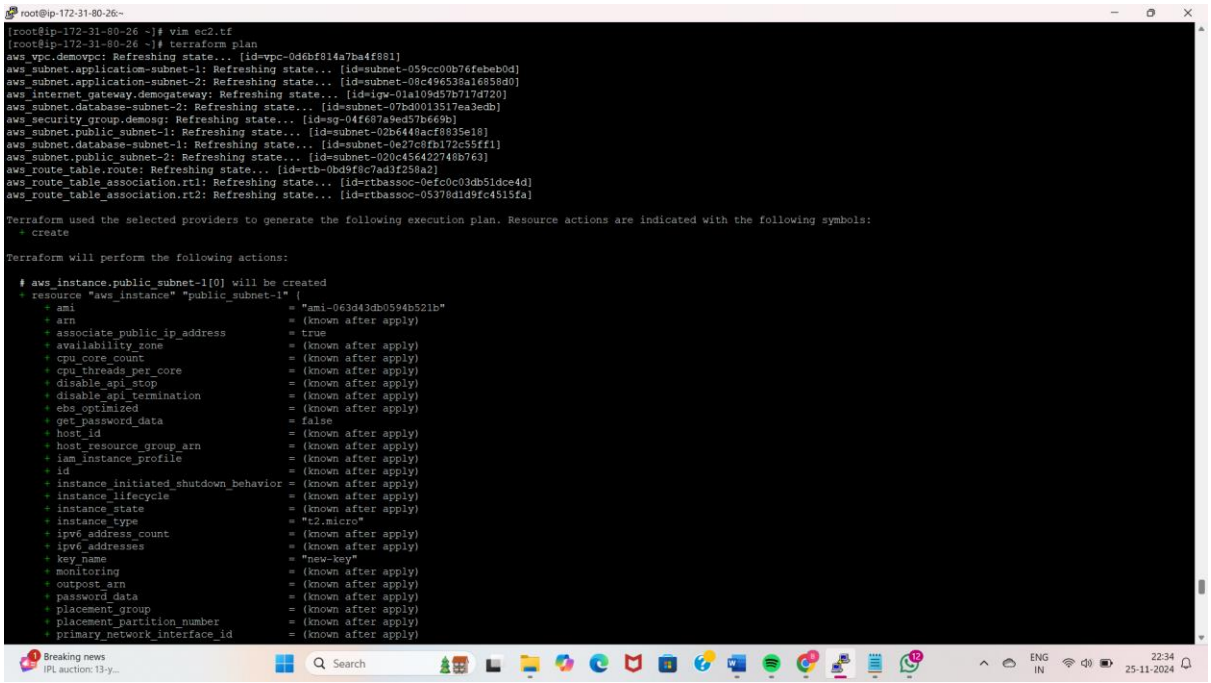
STEP 5: CREATE THE EC2 INSTANCES

To create ec2 instance we need to specify the instance type, key pair and AMI in the file which is having the configurations.

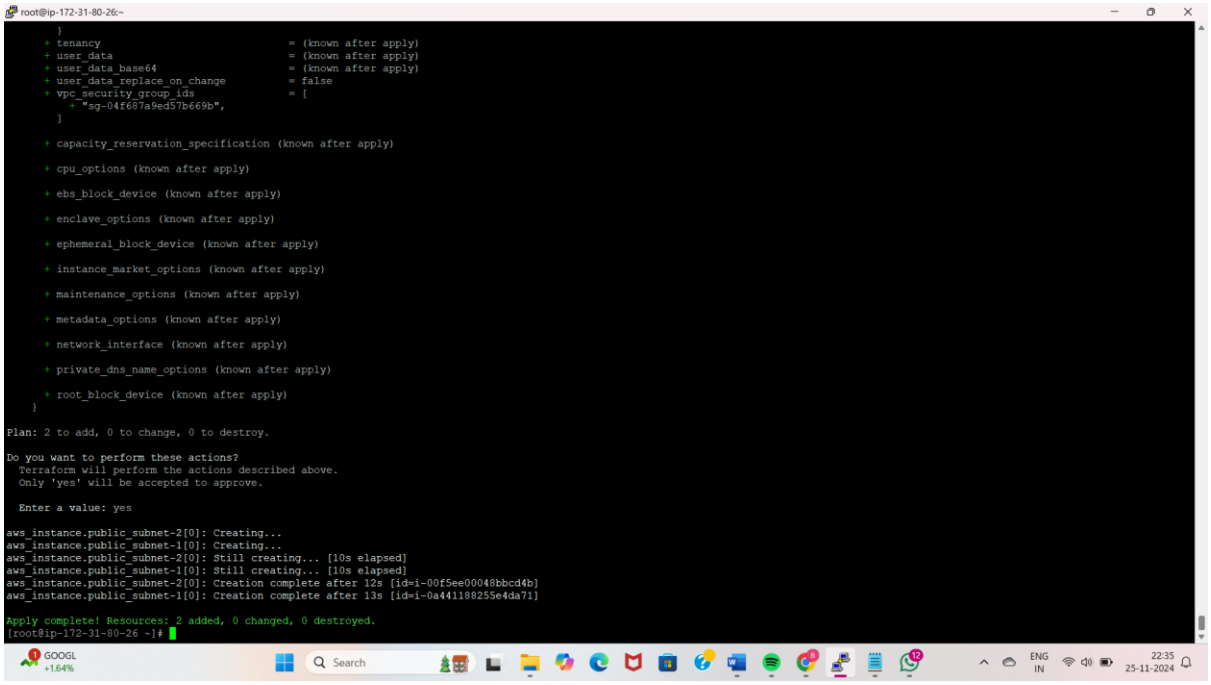
root@ip-172-31-88-64:~

```
resource "aws_instance" "public_subnet-1" {
ami="ami-063d43db0594b521b"
instance_type="t2.micro"
count=1
key_name="new-key"
vpc_security_group_ids=["${aws_security_group.demosg.id}"]
subnet_id="${aws_subnet.public_subnet-1.id}"
associate_public_ip_address=true
user_data="${file("data.sh")}"
tags={
Name="My public Instance 1 "
}
}

resource "aws_instance" "public_subnet-2" {
ami="ami-063d43db0594b521b"
instance_type="t2.micro"
count=1
key_name="new-key"
vpc_security_group_ids=["${aws_security_group.demosg.id}"]
subnet_id="${aws_subnet.public_subnet-2.id}"
associate_public_ip_address=true
user_data="${file("data_d.sh")}"
tags={
Name="My public Instance 2 "
}
}
```

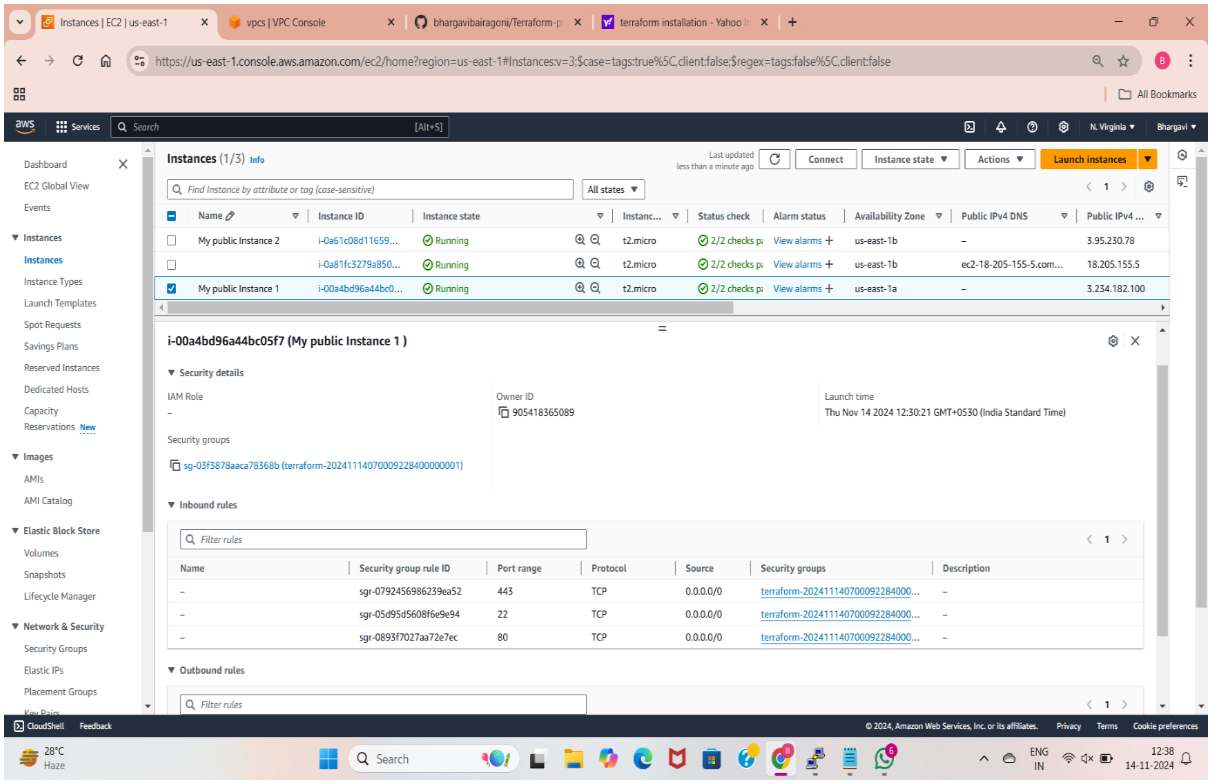
This terraform code creates two Amazon EC2 instances, each in a different public subnet. These two instances are configured with a public IP address and uses a single security group for network access.



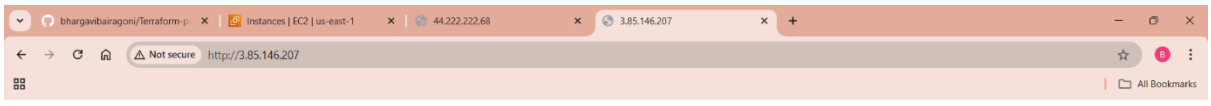
The plan specifies that an EC2 instance resource (public_subnet-1, public_subnet-2) will be created.



The terraform apply command is executed to create the infrastructure defined in the Terraform configuration (ec2.tf).



The console displays the list of EC2 instances currently running in the AWS environment. These two ec2 instances are created with the terraform code.



I have used user data in the terraform code, so it is hosting a application in apache server. I am able access this application using above created instance public IP.

STEP 6: CREATE A FILE FOR SECURITY GROUP FOR THE FRONT-END-TIER

To create any type of ec2 instance we need to create a security group to access those instances. A Security Group acts as a virtual firewall that controls inbound and outbound traffic for your EC2 instances. Here am creating a security group with the name of web_sg.tf which allows HTTP, HTTPS, and SSH. These ports helps us to connect with the terminals, access any applications through browsers.

root@ip-172-31-80-26:~

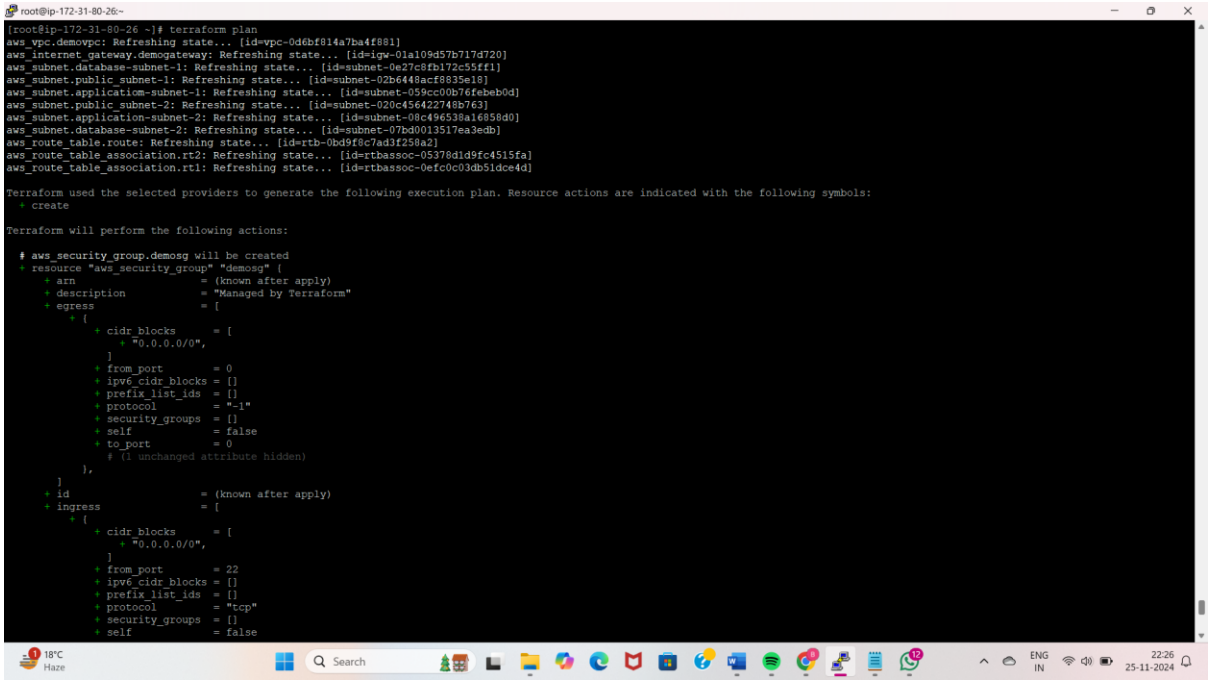
```
resource "aws_security_group" "demosg" {
vpc_id="${aws_vpc.demovpc.id}"

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"]
}

ingress{
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"]
}
tags={
Name="WEB SG"
}
}
```

This picture shows the inbound and outbound rules allowed in the security group for frontend tier.

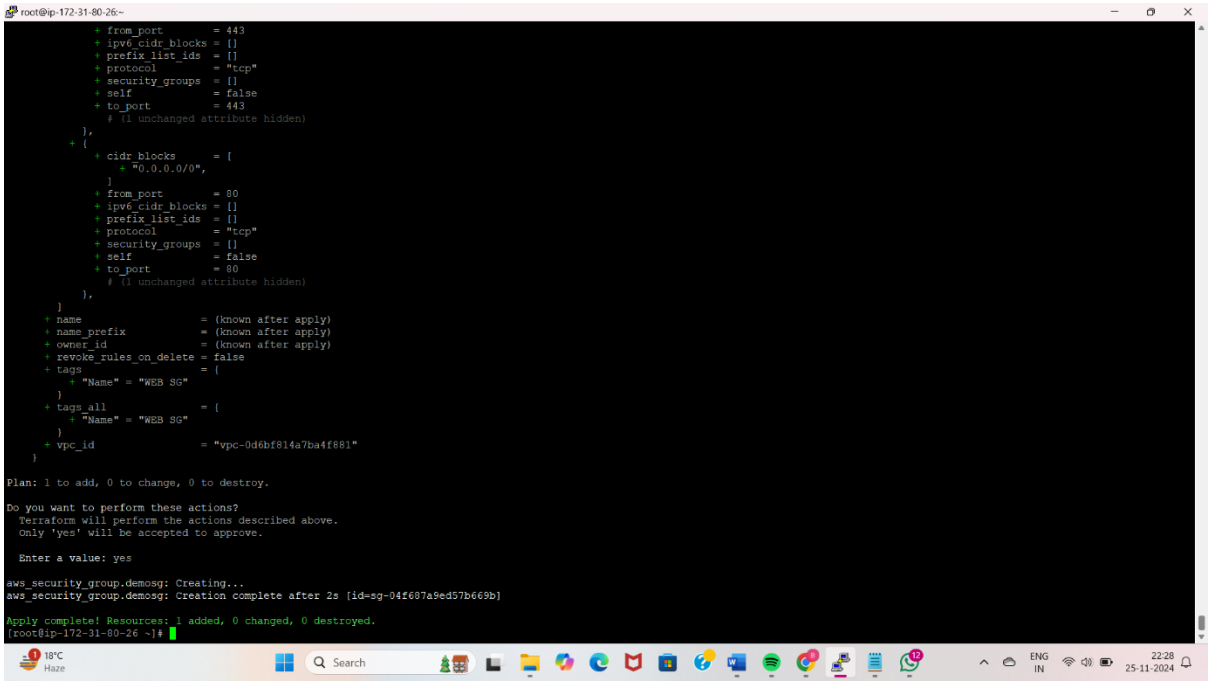
22- Allows secure shell (SSH) access to the EC2 instance.

443- This port is essential for secure communications between clients (e.g., web browsers) and the server.

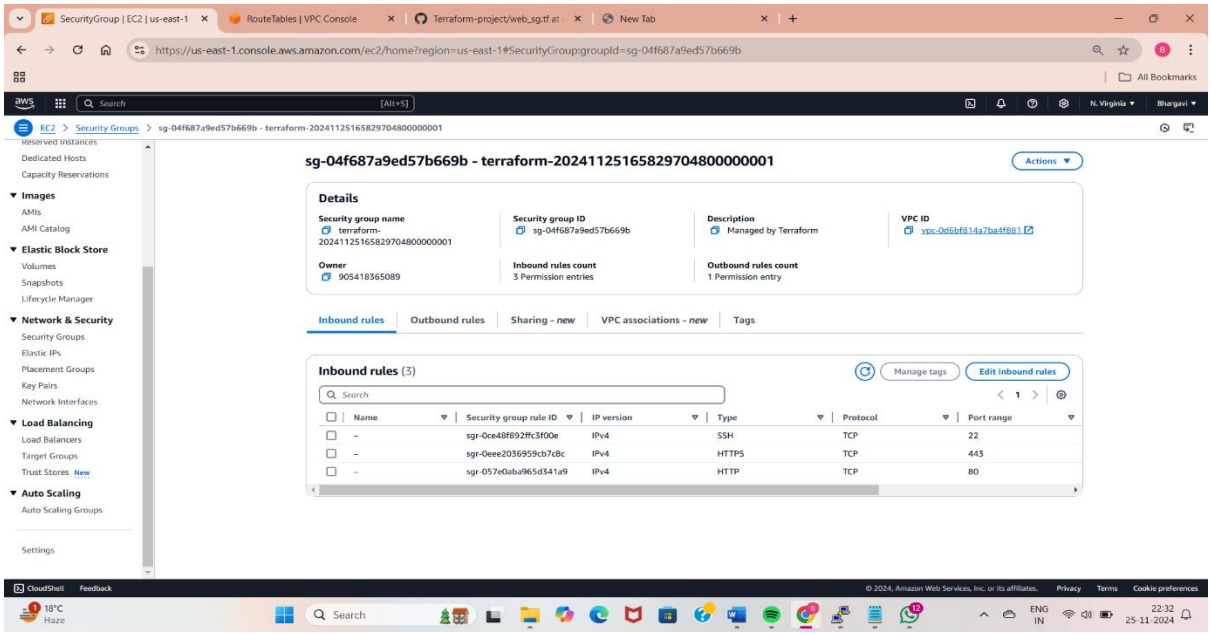
80- Typically used for public websites where secure encryption (HTTPS) is not required.



This plan specifies that an security group resource for instances public_subnet-1, public_subnet-2 will be created.



The terraform apply command is executed to create the infrastructure defined in the Terraform configuration for creating a security group(web_sg.tf).



The aws console shows that the security group for frontend tier is created with specified configurations.

STEP 7: CREATE A FILE FOR SECURITY GROUP FOR THE DATABASE TIER

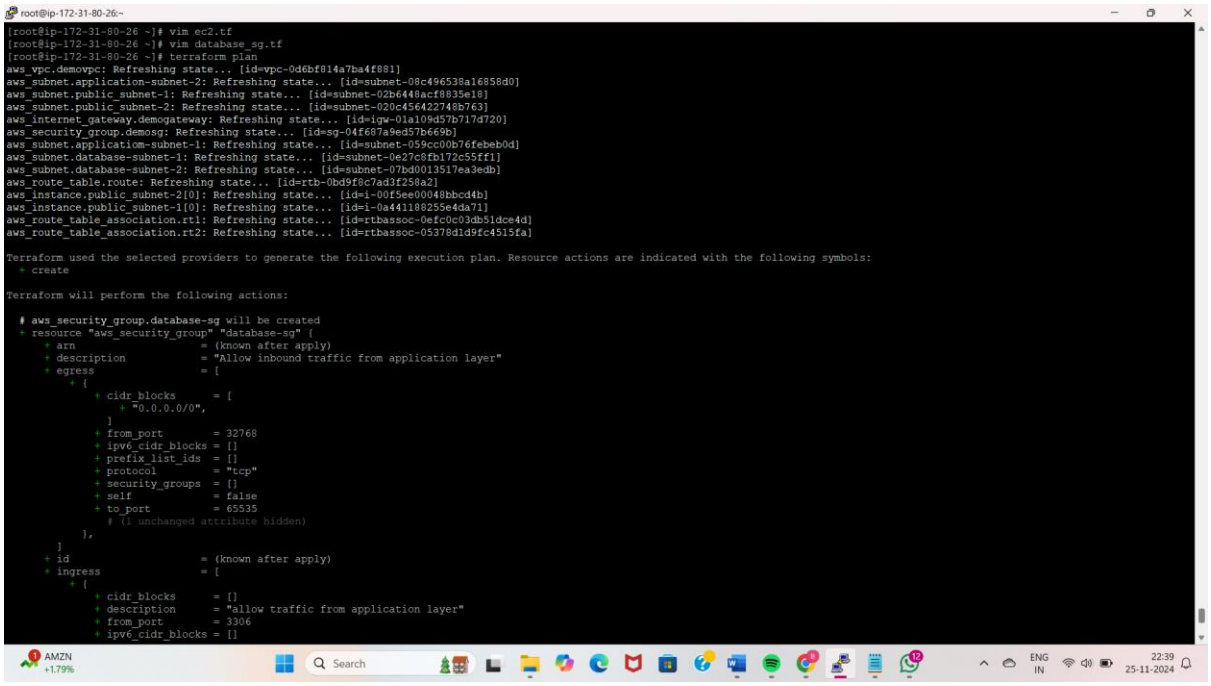
To create a Security Group for the Database Tier in Terraform we need to allow mysql port with the port number 3306. Here am creating a file with the name of database_sg.tf for database tier configurations.

root@ip-172-31-80-26:~

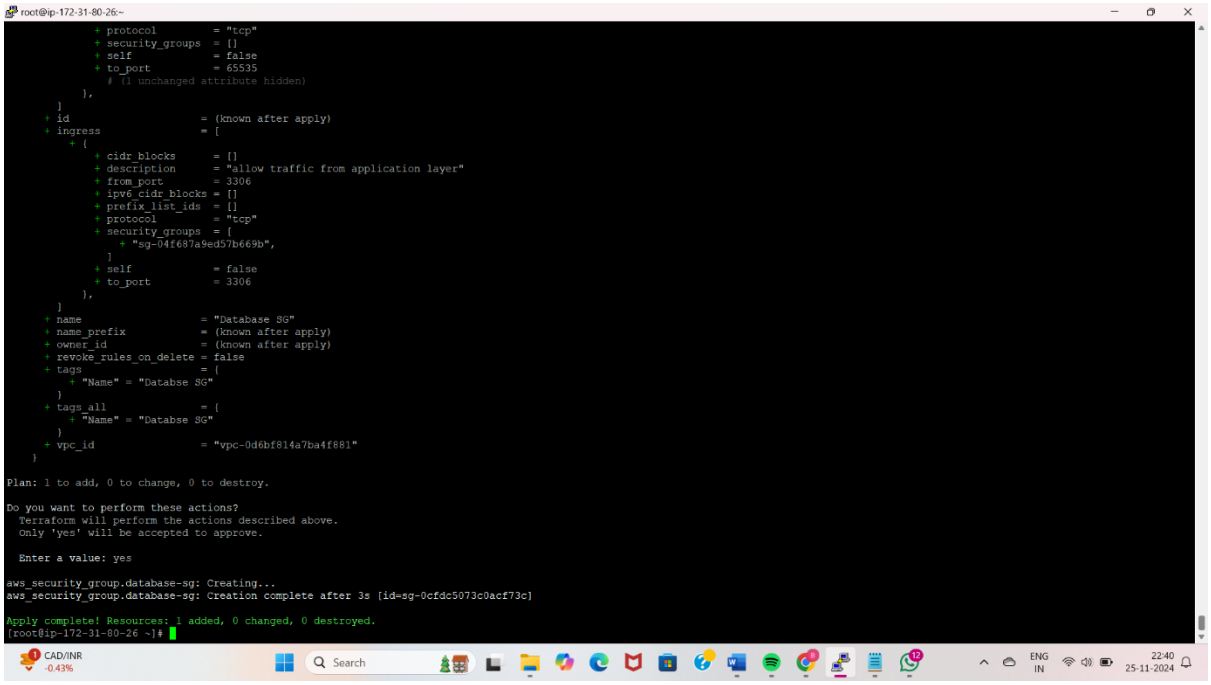
```
resource "aws_security_group" "database-sg" {
name="Database SG"
description="Allow inbound traffic from application layer"
vpc_id=aws_vpc.demovpc.id
ingress{
description="allow traffic from application layer"
from_port=3306
to_port=3306
protocol="tcp"
security_groups=[aws_security_group.demosg.id]
}
egress {
  from_port=32768
to_port=65535
protocol="tcp"
cidr_blocks=["0.0.0.0/0"]
}
tags={
Name="Databse SG"
}
}
```

This resource creates an AWS Security Group named database-sg, which is intended for controlling network access to resources in the Database like MySQL or similar database. The security group is

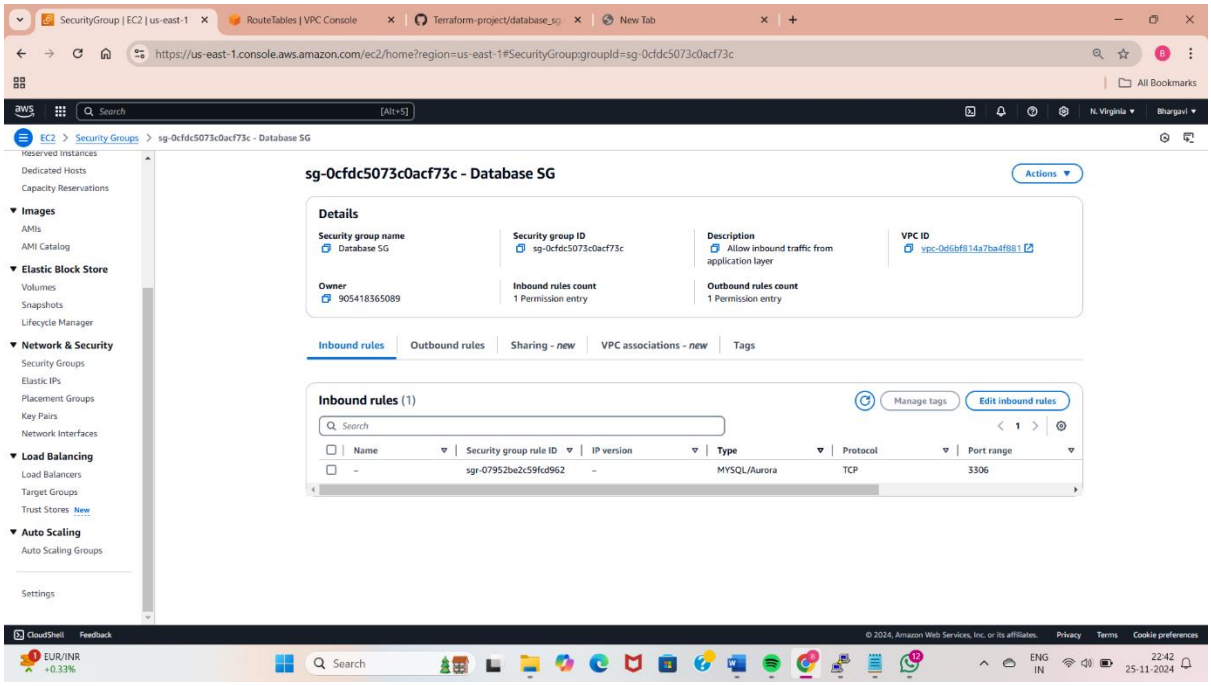
associated with the above created VPC. This security group is specifically designed to allow database instances to receive traffic from application servers with the demosg security group on port 3306. It is also configured to allow outbound communication over a range of 32768–65535.



The plan for creating security group specifies the configurations of database security group.



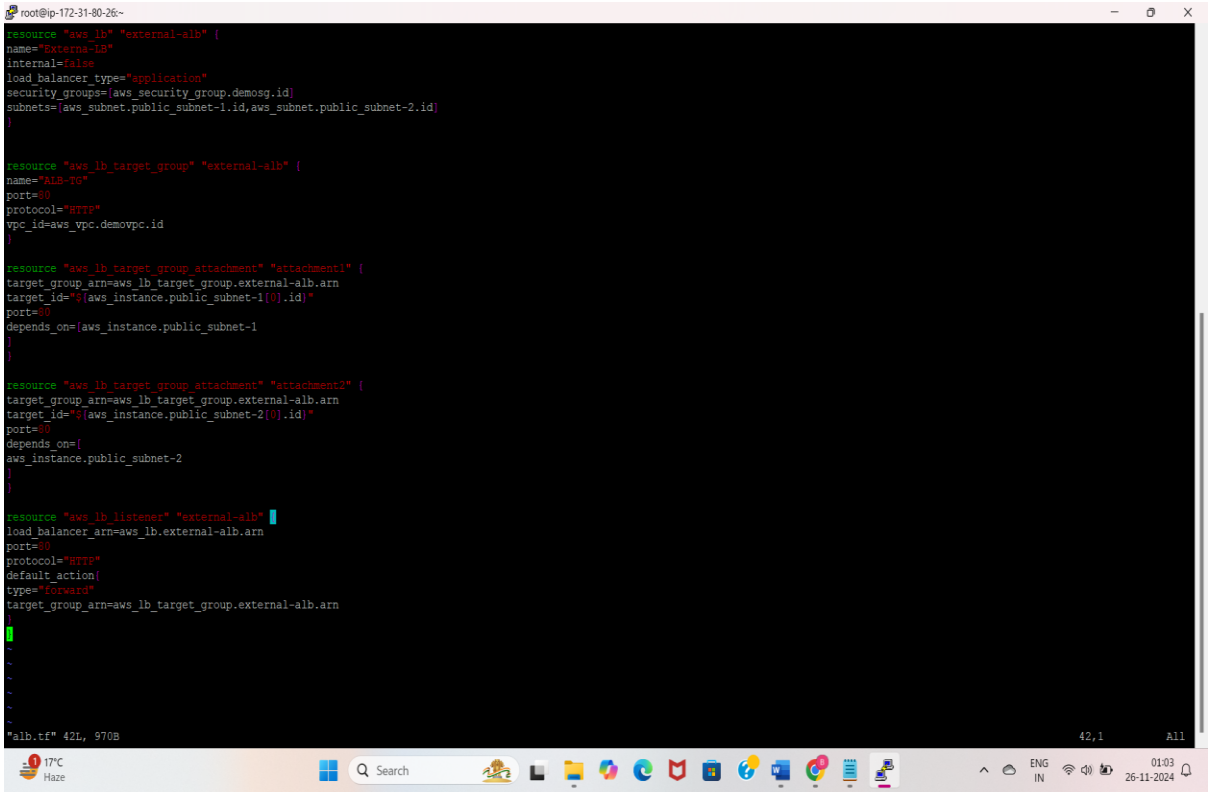
The terraform apply command is executed to create the infrastructure defined in the Terraform code which creats a security group for database tier(database_sg.tf).



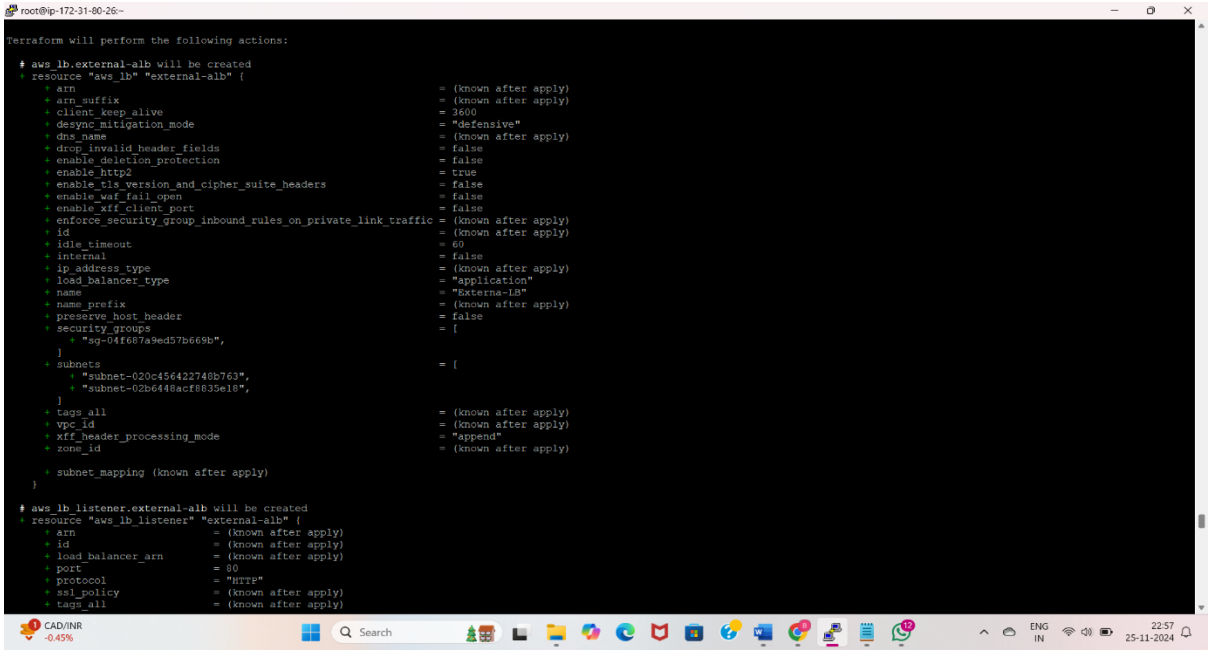
We can see in AWS management console that database security group is created with mentioned configurations.

STEP 8: CREATE A FILE FOR APPLICATION LOAD BALANCER

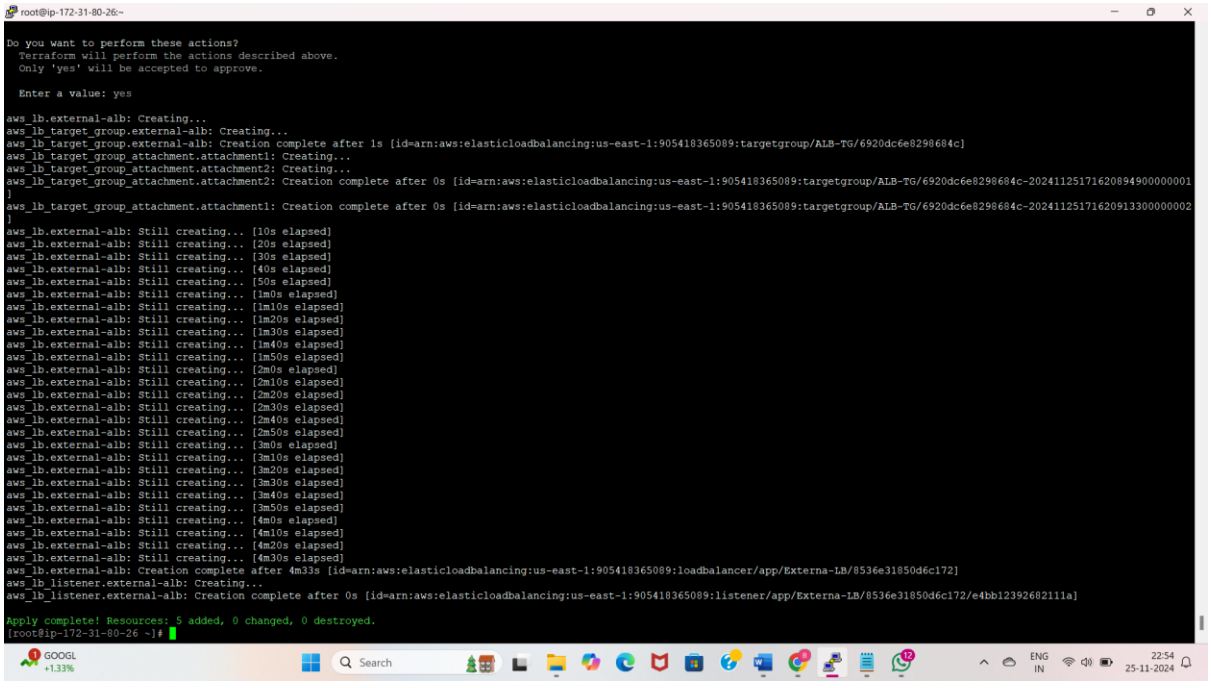
An Application Load Balancer (ALB) is a service provided by Amazon Web Services that distributes incoming traffic across multiple targets, such as EC2 instances, containers, or IP addresses, within one or more Availability Zones. To create a load balancer we need to specify configurations in a file. Here am creating a alb.tf file for load balancer and target group.



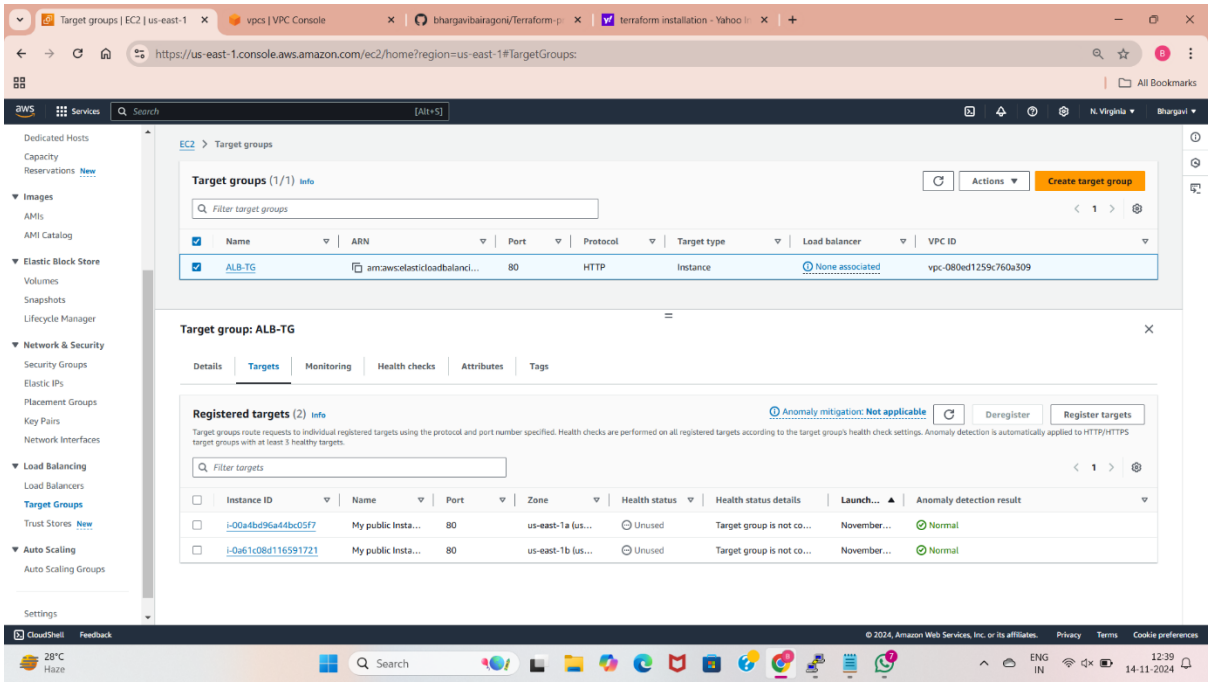
The Terraform configuration provided defines an Application Load Balancer (ALB) setup that includes the load balancer itself, target groups, target group attachments, and a listener for handling incoming HTTP traffic. Associating the load balancer with a security group, demosg, which controls inbound and outbound traffic rules for the ALB.



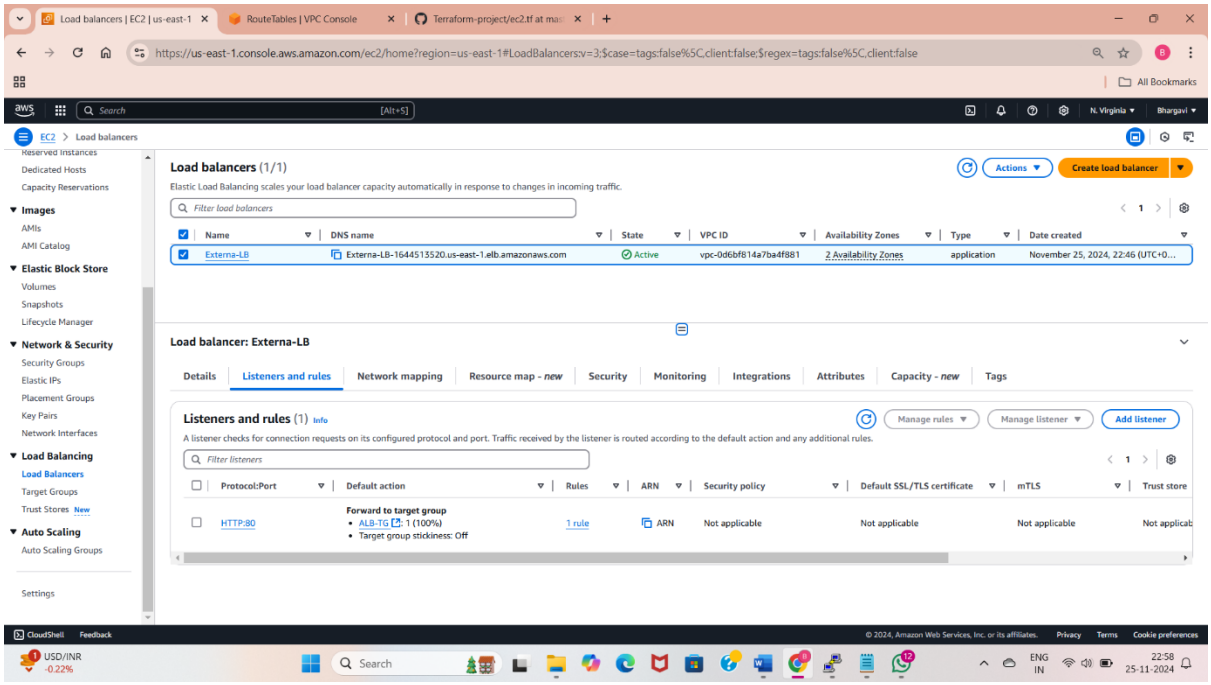
The plan for creating load balancer specifies the required configurations to distribute the load across the servers.



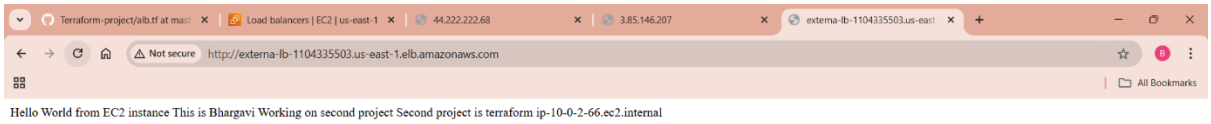
The Terraform apply command creating the load balancer and target group with two instances, including the specified configurations.



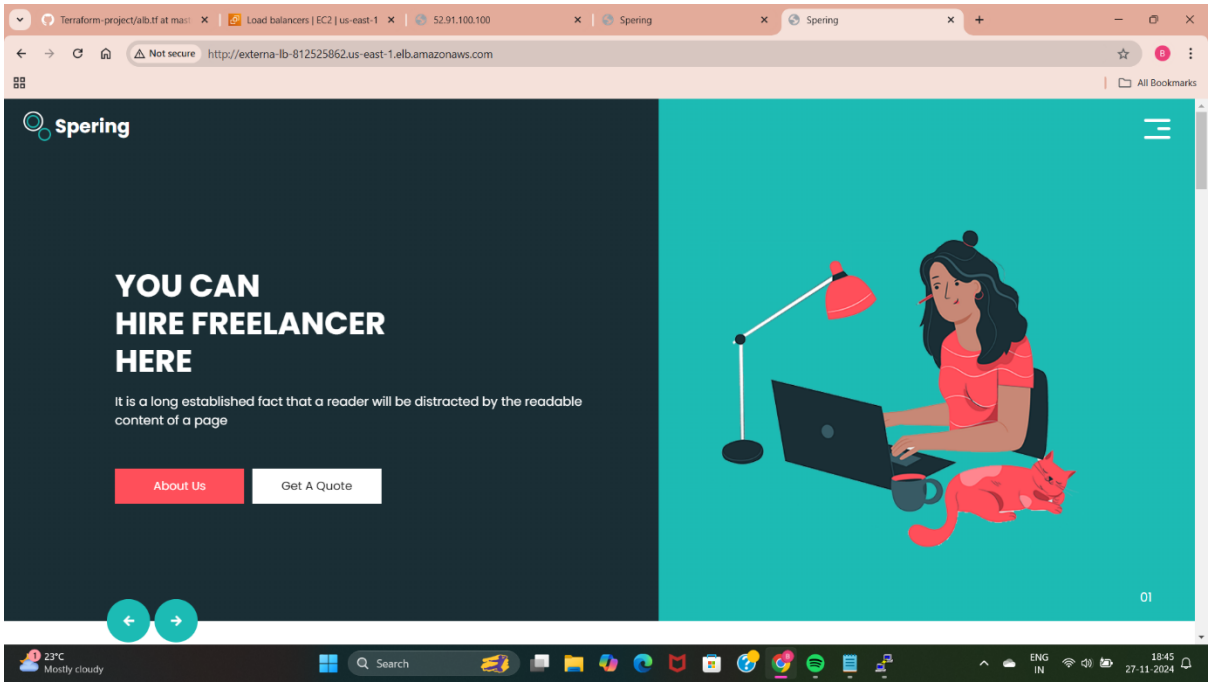
Above console shows the target group which is created with two instances and the instances are in healthy state.



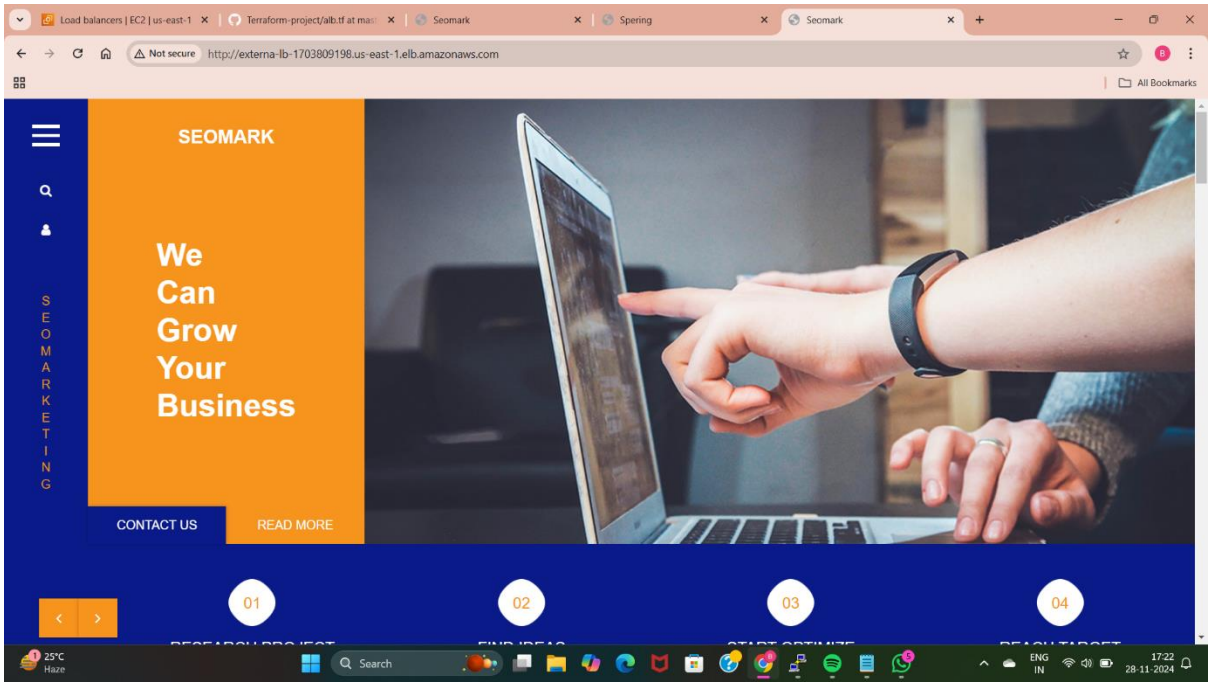
This is the load balancer which is created with the name of External-LB, we can also observe the DNS name which helps us to access any application which is used in the registered instances.



When I am trying to access the DNS of the load balancer am able to access the application which is hosted while we creating the instance using(data.sh)user data.



When I am trying to access the above created DNS name of load balancer am able to access the spering website which is stored in one of the ec2 instances.



When I am trying to increase the load then the traffic is distributed to other instance which is having other application with the name of Browny.

STEP 9: CREATE A FILE FOR THE RDS INSTANCE

Amazon RDS (Relational Database Service) is a managed database service provided by AWS that makes it easier to set up, operate, and scale relational databases in the cloud. RDS handles routine database tasks such as provisioning, patching, backup, recovery, and scaling, allowing developers and businesses to focus on application development rather than managing the underlying infrastructure.

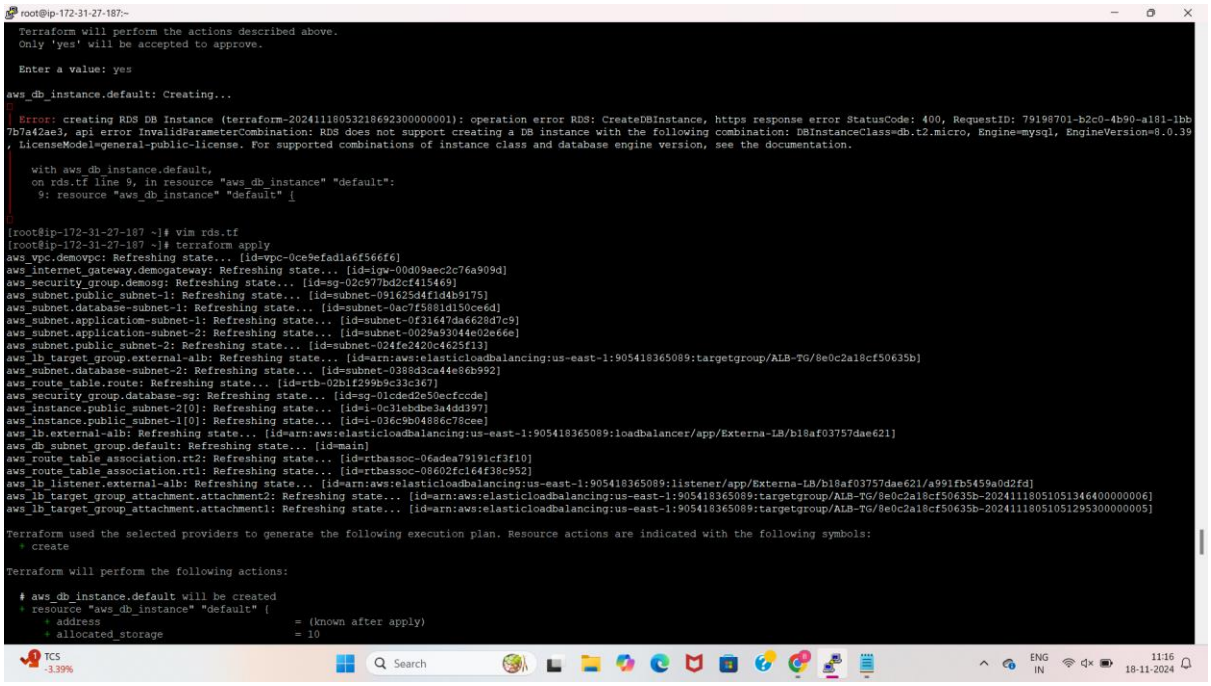
To create rds we need to take any database engine like mysql, oracle etc, for this we need to create a file so that database is automatically created when we run the file. Here am creating a file with the name of rds.tf.

root@ip-172-31-80-26:~

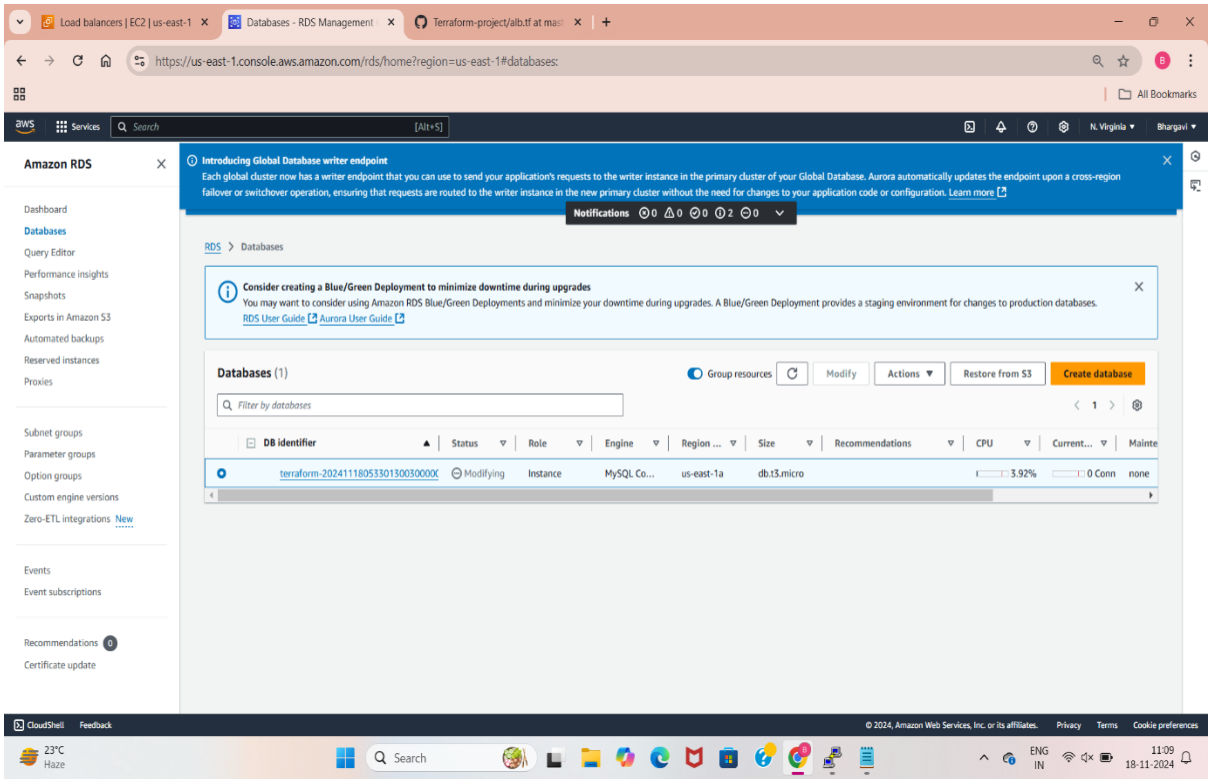
```
resource "aws_db_subnet_group" "default" {
name="main"
subnet_ids=[aws_subnet.database-subnet-1.id, aws_subnet.database-subnet-2.id]
tags={
Name="My DB subnet group"
}
}

resource "aws_db_instance" "default" {
allocated_storage=10
db_subnet_group_name=aws_db_subnet_group.default.id
engine="MySQL"
engine_version="8.0.39"
instance_class="db.t3.micro"
multi_az=true
db_name="mydb"
username="username"
password="password"
skip_final_snapshot=true
vpc_security_group_ids=[aws_security_group.database-sg.id]
}
~
```

This Terraform configuration creates an RDS instance for a MySQL database in a specific subnet group within a VPC. This ensures that the database is launched within specified subnets for better control and availability. The aws_db_instance creates an RDS instance with MySQL as the database engine.

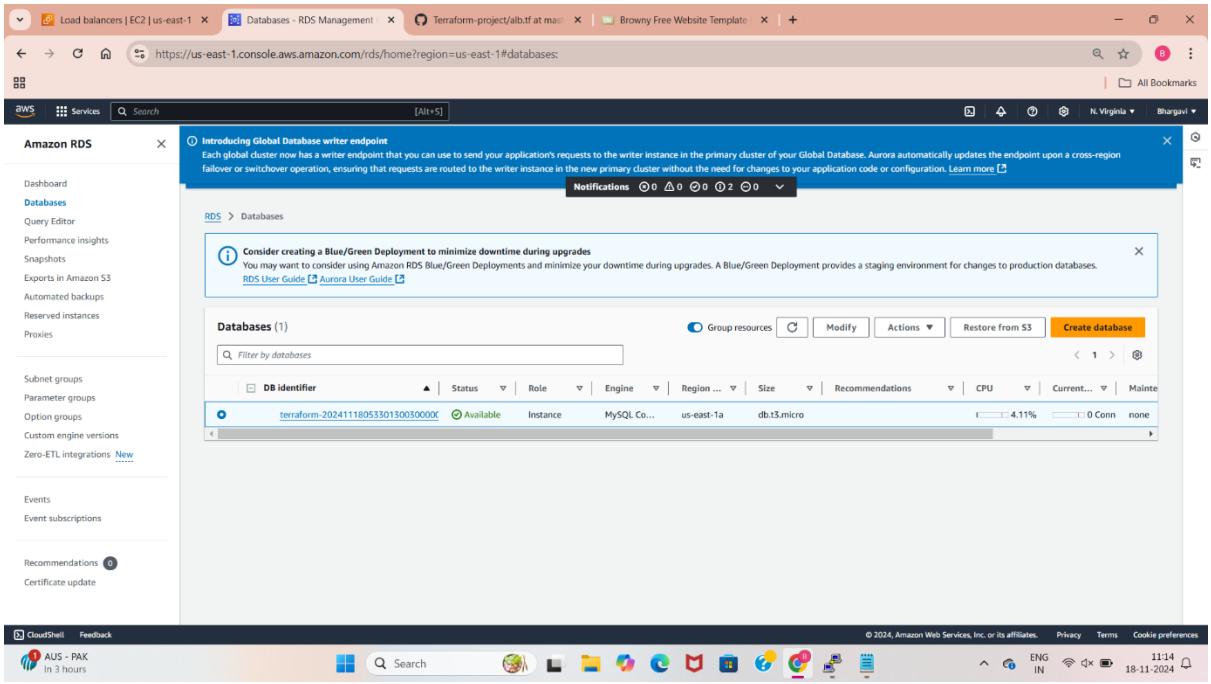
Here, am using 8.0.39 MySQL version, db.t3.micro instance, which is cost-effective and suitable for development and testing purposes. Username and password define the credentials for the database admin user. Snapshot prevent the data loss when(we set it to true) the data is deleted unintendly.



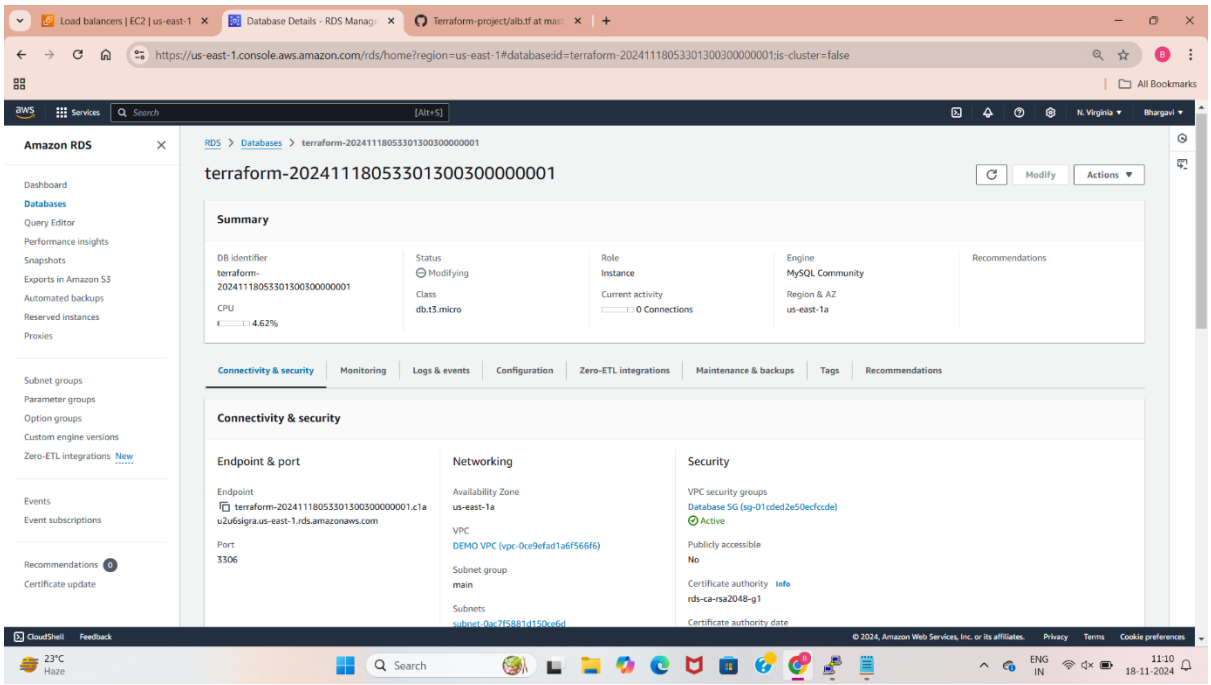
The plan for creating RDS with the specified configuration(rds.tf) is successful.



We can see the RDS database is creating in aws console. It takes some time to create and make the database to available state.



Here the database is in available state, so we can get endpoint to access this database.



Using the endpoint of this database we can connect our mysql engine to instance to create any database and store the database.

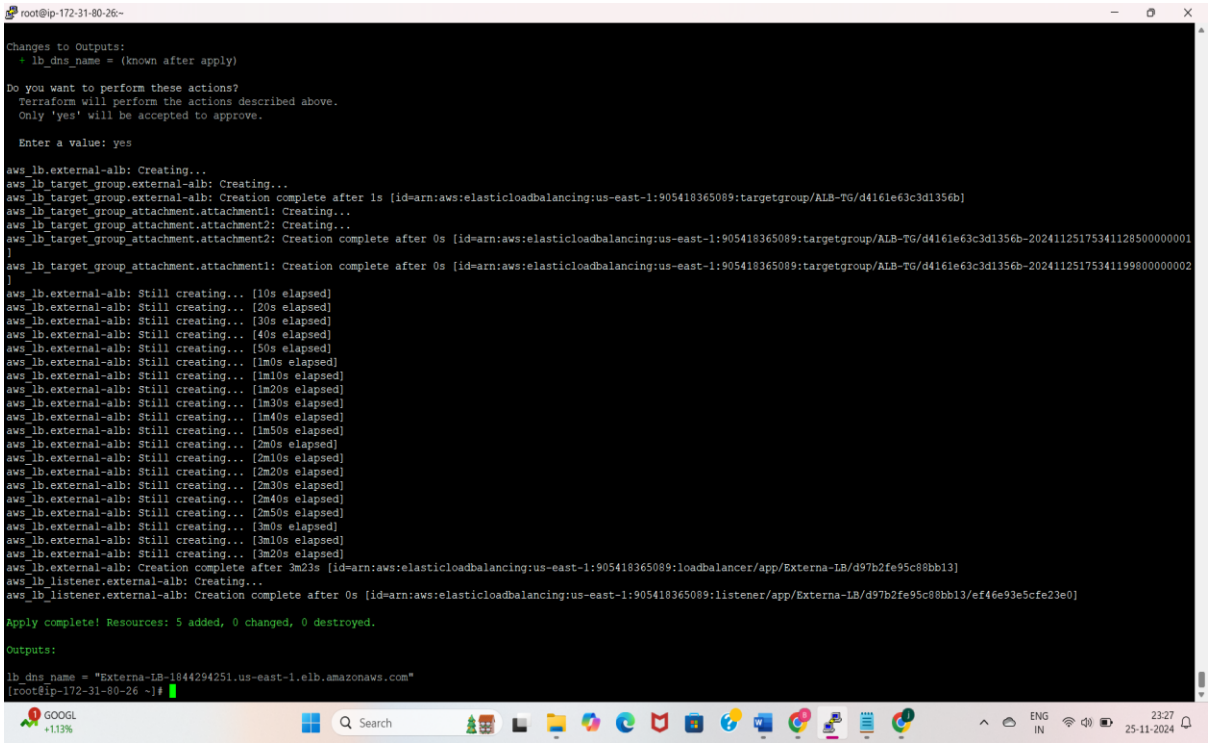
STEP 10: CREATE A FILE FOR OUTPUT

This Terraform output block is used to display the DNS name of the load balancer created in the Terraform configuration. To create the DNS name we need to create a file with the configurations. Am creating a outputs.tf file for DNS name.

root@ip-172-31-80-26:~

```
output "lb_dns_name" {
description ="The DNS Name of the Load Balancer"
value=aws_lb.external-alb.dns_name
}
~
```

Outputs are often used to display useful information, such as resource IDs, URLs(DNS).



Here we can see the DNS for load balancer “Ecterna-LB-1844294251.us-eat-1.elb.amazonaws.com”. If we use this we can access the applications which are in the registered instances.

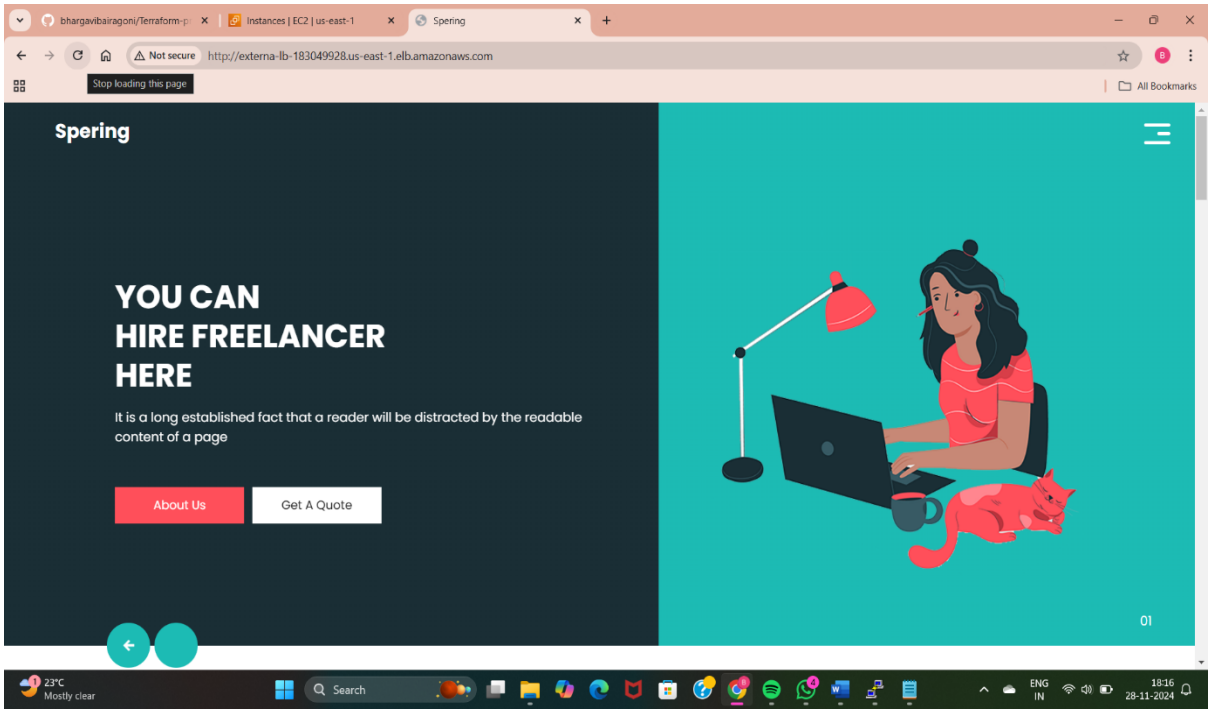
```
Apply complete! Resources: 20 added, 0 changed, 0 destroyed.

Outputs:

lb_dns_name = "Externa-LB-235412804.us-east-1.elb.amazonaws.com"
[root@ip-172-31-28-241 ~]#
```

This is the DNS name of the load balancer.

When am using the above DNS am getting this web application.

```
Apply complete! Resources: 20 added, 0 changed, 0 destroyed.

Outputs:

lb_dns_name = "Externa-LB-183049928.us-east-1.elb.amazonaws.com"
```

I have created new instances with the new configurations in the userdata and added the instances to the load balancer. Am getting this DNS for the load balancer.



By copying the above created load balancer DNS we can access the application .

STEP11: CREATE A FILE FOR VARIABLE

Instead of writing the same value multiple times (like an IP range or instance type), we can define it once as a variable and use multiple times. This makes our setup flexible and easier to update or use in different environments.

Here am creating a file for variables to give cidr block for vpc and subnet. The file is created with the name of vars.tf.

The vars.tf file is used to declare all the variables that can be referenced in other Terraform configuration files. This helps centralize variable definitions, making the code cleaner and easier to manage.

- ➔ We can change the CIDR blocks without modifying multiple places in your code.
- ➔ The same configuration can be reused in different environments (e.g., dev, test, prod) by changing the variable values.
- ➔ The code becomes cleaner and easier to understand.

root@ip-172-31-80-26:~

```
variable "vpc_cidr" {
default ="10.0.0.0/16"
}

variable "subnet_cidr" {
default ="10.0.1.0/24"
}

variable "subnet1_cidr" {
default="10.0.2.0/24"
}

variable "subnet2_cidr" {
default ="10.0.3.0/24"
}

variable "subnet3_cidr" {
default="10.0.4.0/24"
}

variable "subnet4_cidr" {
default="10.0.5.0/24"
}

variable "subnet5_cidr" {
default="10.0.6.0/24"
```

The vpc_cidr is a variable that holds a default CIDR block value for a VPC, subnet_cidr defines the CIDR block for a specific subnet within the VPC.

STEP12: CREATE A FILE FOR USER DATA

User data in the context of AWS EC2 instances refers to a script or set of commands provided to an instance at launch. It is used to automate tasks during the bootstrapping of the instance. This data can include shell scripts, or configuration scripts to set up and initialize software, install updates, or configure the operating system.

User data is a powerful way to automate the initialization of EC2 instances, making deployments more efficient and consistent.

root@ip-172-31-89-153:~

```
#!/bin/bash
yum update -y
yum install -y httpd
systemctl enable httpd
systemctl start httpd
echo "Hello World from EC2 instance
This is Bhargavi
Working on second project
Second project is terraform $(hostname -f)" > /var/www/html/index.html

~
~
```

This is the userdata with the simple text. This will display the text mention in the echo.

root@ip-172-31-89-117:~

```
#!/bin/bash
# Update packages and install Apache and Git
yum update -y
yum install -y httpd
systemctl start httpd
systemctl enable httpd
yum install git -y
sudo git clone https://github.com/bhargavibairagoni/spering.git
sudo mv spering/spering-html/* /var/www/html/
~
```

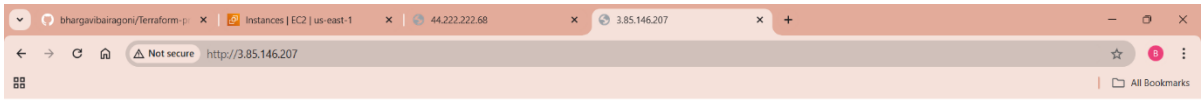
This is the script for userdata with the name of “data.sh” Here am using a github project and cloning into the userdata and moving the files of the website to the apache document root directory.

root@ip-172-31-89-117:~

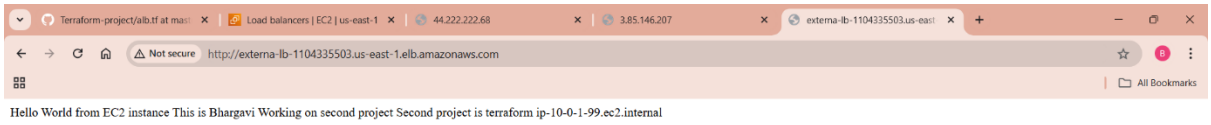
```
#!/bin/bash
# Update packages and install Apache and Git
yum update -y
yum install -y httpd
systemctl start httpd
systemctl enable httpd
yum install git -y
sudo git clone https://github.com/bhargavibairagoni/seomark.git
sudo mv seomark/seomark-html/* /var/www/html/
~
~
```

This is the other file with the name of “data_u.sh”. This clones the repository into the current working directory and files are moved to the directory served by Apache.

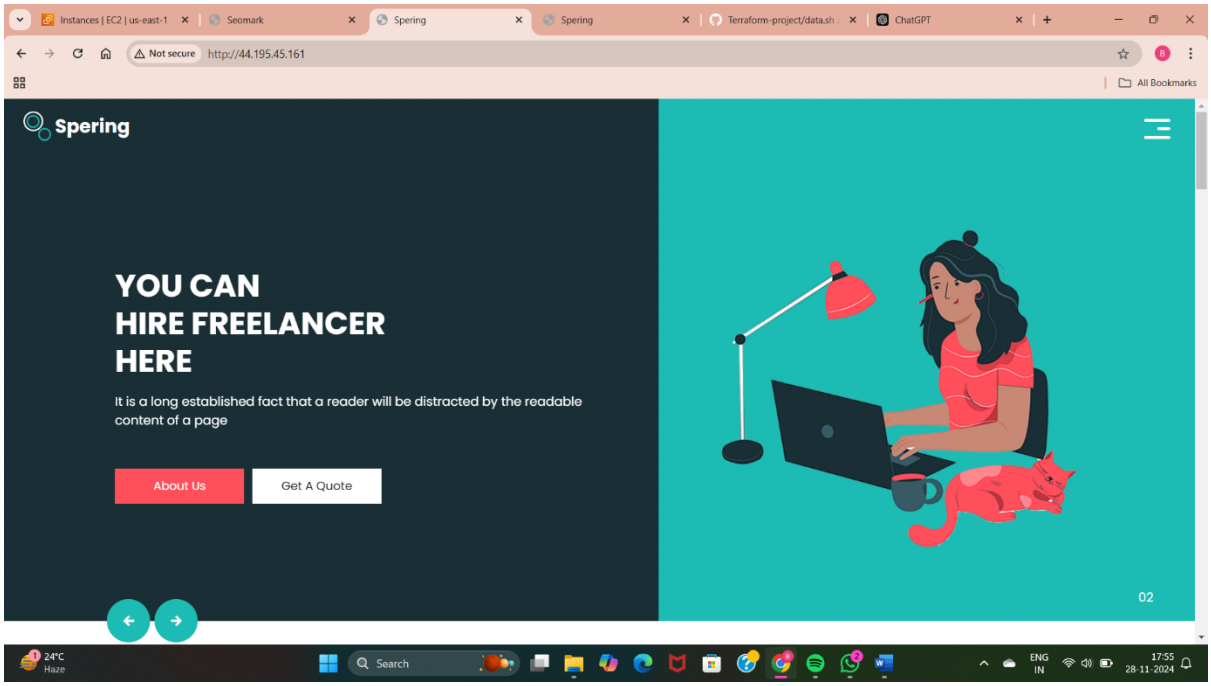
- Yum update -y : Updates the system's package repository and installs the latest updates for all installed packages.
- Installs the Apache HTTP server
- Configures Apache to start automatically whenever the instance is rebooted.
- Displays a basic welcome message by including the IP address of the instances
- Displays the websites which is cloned from the github.



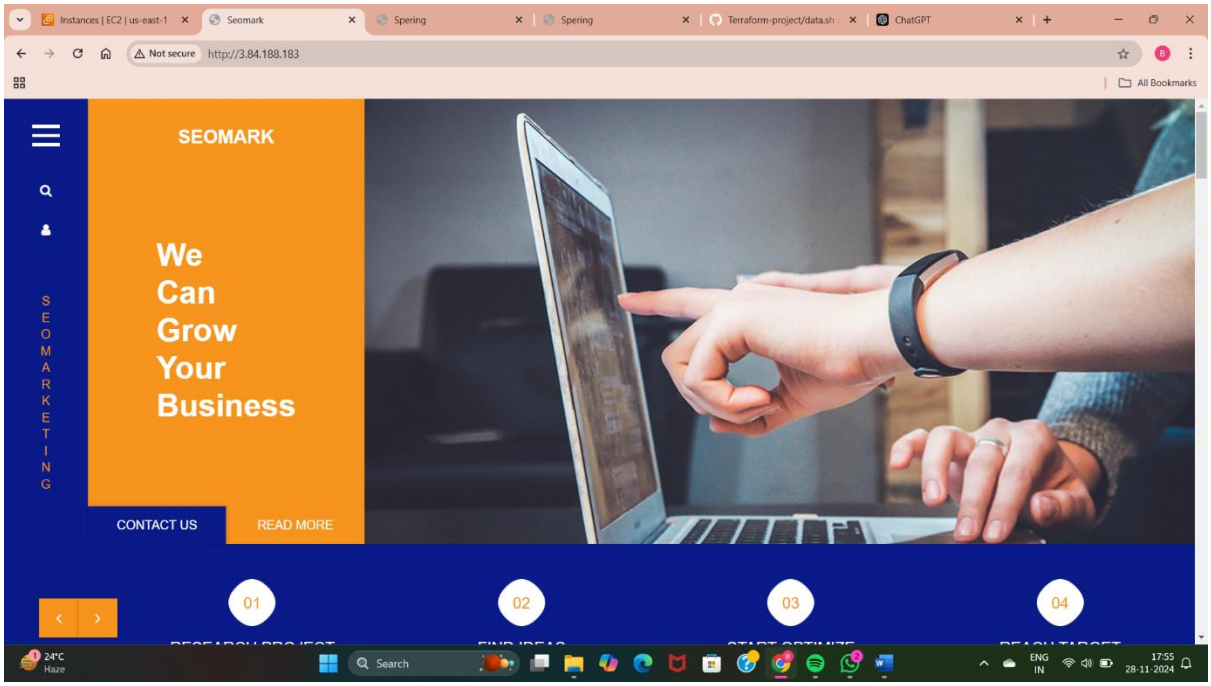
We can access the user data by using the public IP address of the instance.



This is the second instance in which I have used the user data.



This is the first website I have cloned from the github website. When I run the user data script (data.sh) one instance is created with the above website. When I taken the Public IP address and used this in the browser we can see and access the website.



This is the other application I have hosted using the userdata by specifying configurations in the userdata script(data_u.sh)