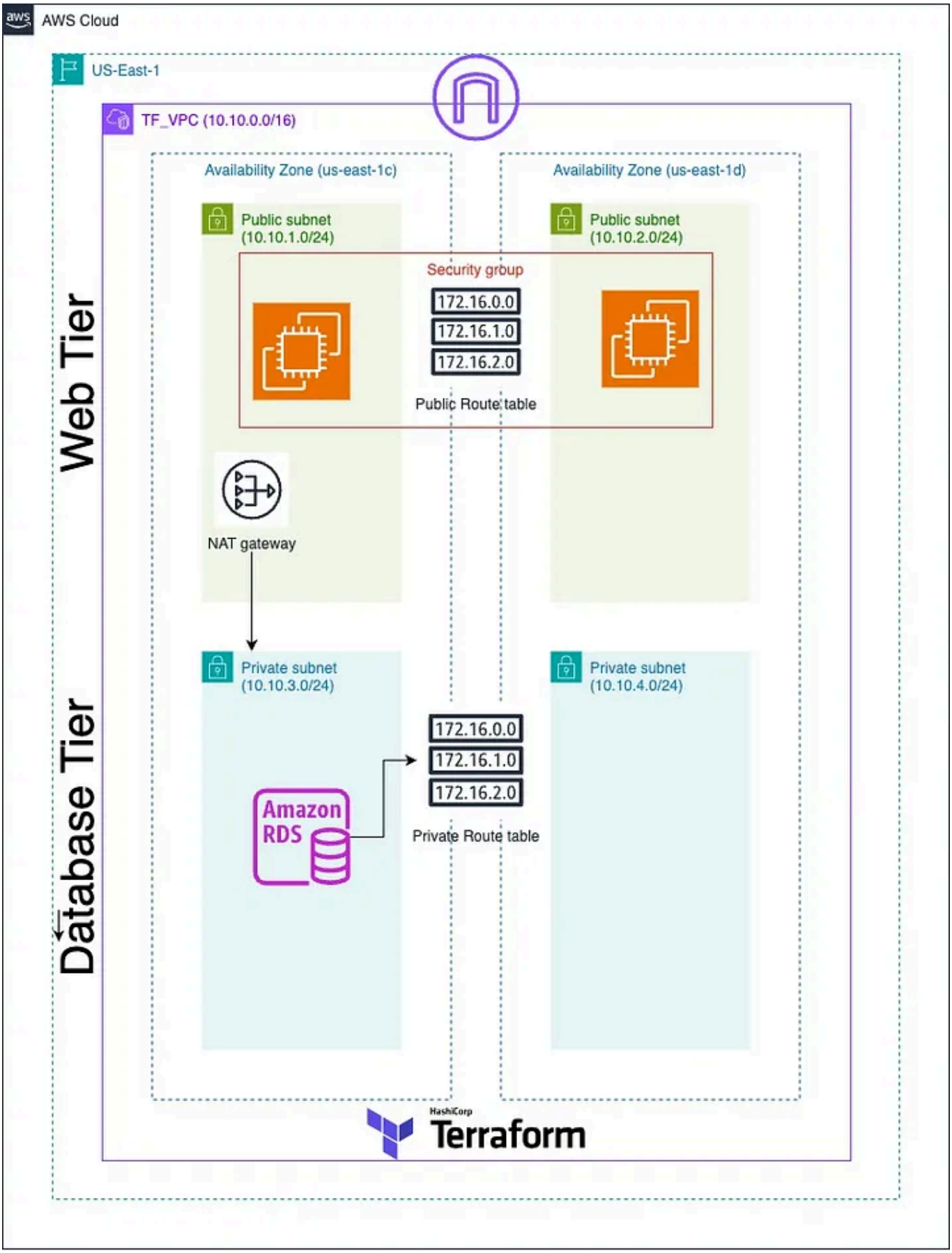
Deploying a 2-tier Architecture with Terraform

Nife Sofowoke · Follow

13 min read · Jul 10, 2024

13

> *Using Terraform Cloud and the Terraform CLI, I deployed a highly-available 2-tier architecture consisting of a Web tier and a Database tier in an AWS environment.*


AWS Cloud
US-East-1
TF_VPC (10.10.0.0/16)
Availability Zone (us-east-1c)
Availability Zone (us-east-1d)
Public subnet (10.10.1.0/24)
Public subnet (10.10.2.0/24)
Security group
172.16.0.0
172.16.1.0
172.16.2.0
Public Route table
NAT gateway
Private subnet (10.10.3.0/24)
Private subnet (10.10.4.0/24)
172.16.0.0
172.16.1.0
172.16.2.0
Private Route table
Amazon RDS
Web Tier
Database Tier
HashiCorp
Terraform

Building from my previous Terraform project, this project goes a step further to leverage Terraform Cloud as a backend management tool for Infrastructure as Code (IaC) deployment.

Terraform Cloud provides a remote workspace that is isolated and contains the Terraform configurations that are applied to an Infrastructure as Code deployment. Each workspace stores the terraform state file and variables and allows different cloud environments (e.g., development, staging, production) to be managed separately.

Amongst other things, Terraform Cloud provides the following features**:**

- **State Management:** This allows each workspace to have its state file, ensuring changes in one workspace do not affect another.
- **Variable Management:** This enables workspace-specific variables to be defined and managed, allowing different configurations for each environment.
- **Run Management:** Terraform plans and applies are executed within the workspace, with logs and outputs stored for review.
- **Collaboration:** Terraform Cloud allows multiple team members to work in the same workspace, with role-based access controls to manage permissions.

You can learn more about Terraform Clouds from the official terraform documentation.

Objectives

1. Create and deploy a 2-tier architecture with a web and database tier using Terraform IaC. Must contain:

- A custom VPC
- 2 Public subnets for the Web Server Tier
- 2 Private subnets for the Database Tier
- Appropriate route tables
- An EC2 instance with a web server in each public subnet
- 1 RDS MySQL instance in the private subnets
- Security groups configured for both the web servers and RDS instance

2. Use Terraform Cloud with a CLI-driven workflow to manage and validate the deployment.

Resources in the Infrastructure

To build and deploy a highly-available 2-tier architecture with a web and database tier in AWS, the following resources need to be included in the infrastructure:

1. VPC
2. Public Subnets
3. Private Subnets
4. Internet Gateway
5. Elastic IP
6. NAT Gateway

7. Public Route Table
8. Private Route Table
9. Route table associations for both the public and private subnets
10. Web tier / EC2 instance security group
11. EC2 instances with user data script to install a web server (Apache in this case)
12. RDS instance Security Group
13. Subnet Group for RDS instance
14. RDS instance

Prerequisites

1. An AWS account. You can sign up for free here.
2. A source code editor to build and edit your infrastructure code. I used Visual Studio Code.

installation tutorial here

5. A Terraform Cloud account. Sign up for free here
6. Knowledge and understanding of basic terraform commands; `terraform fmt` , `terraform validate` , `terraform plan` , `terraform apply` , `terraform destroy`

> ***All the Terraform configuration files I created for this project can be found in a Github repo in my GitHub account here.***

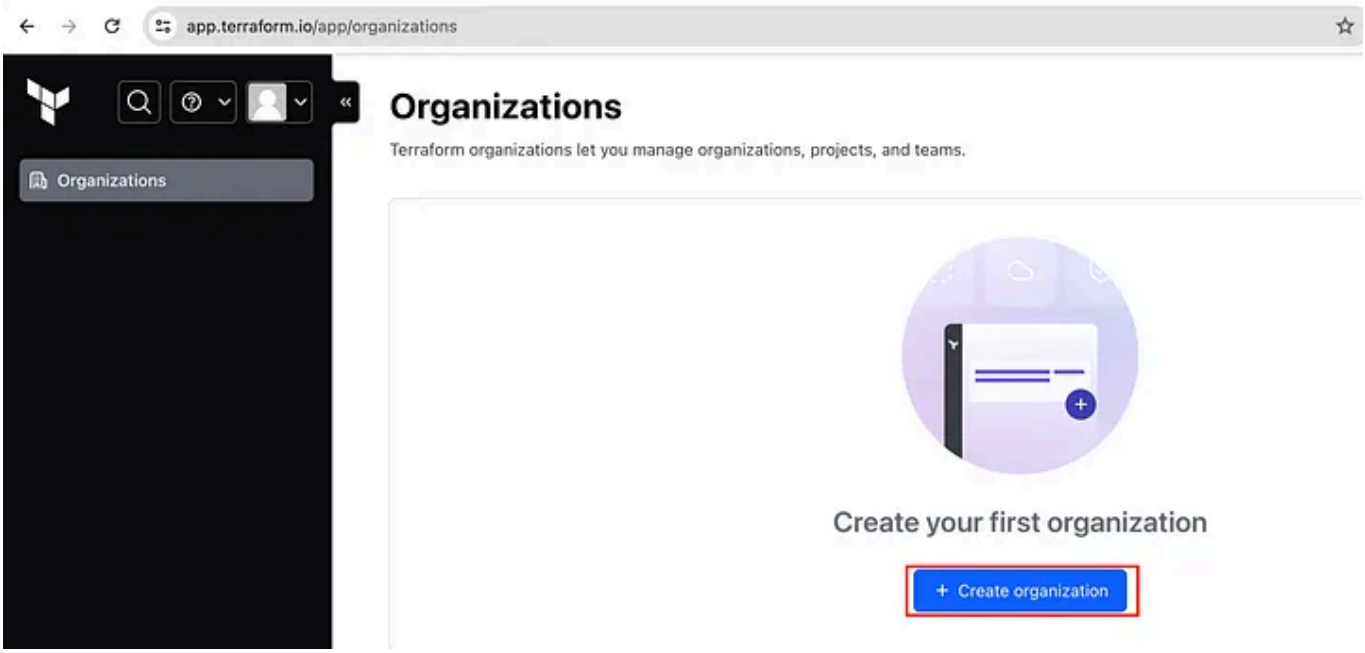
GitHub - nifss-sofowoke/terraform-2-tier-infra-project

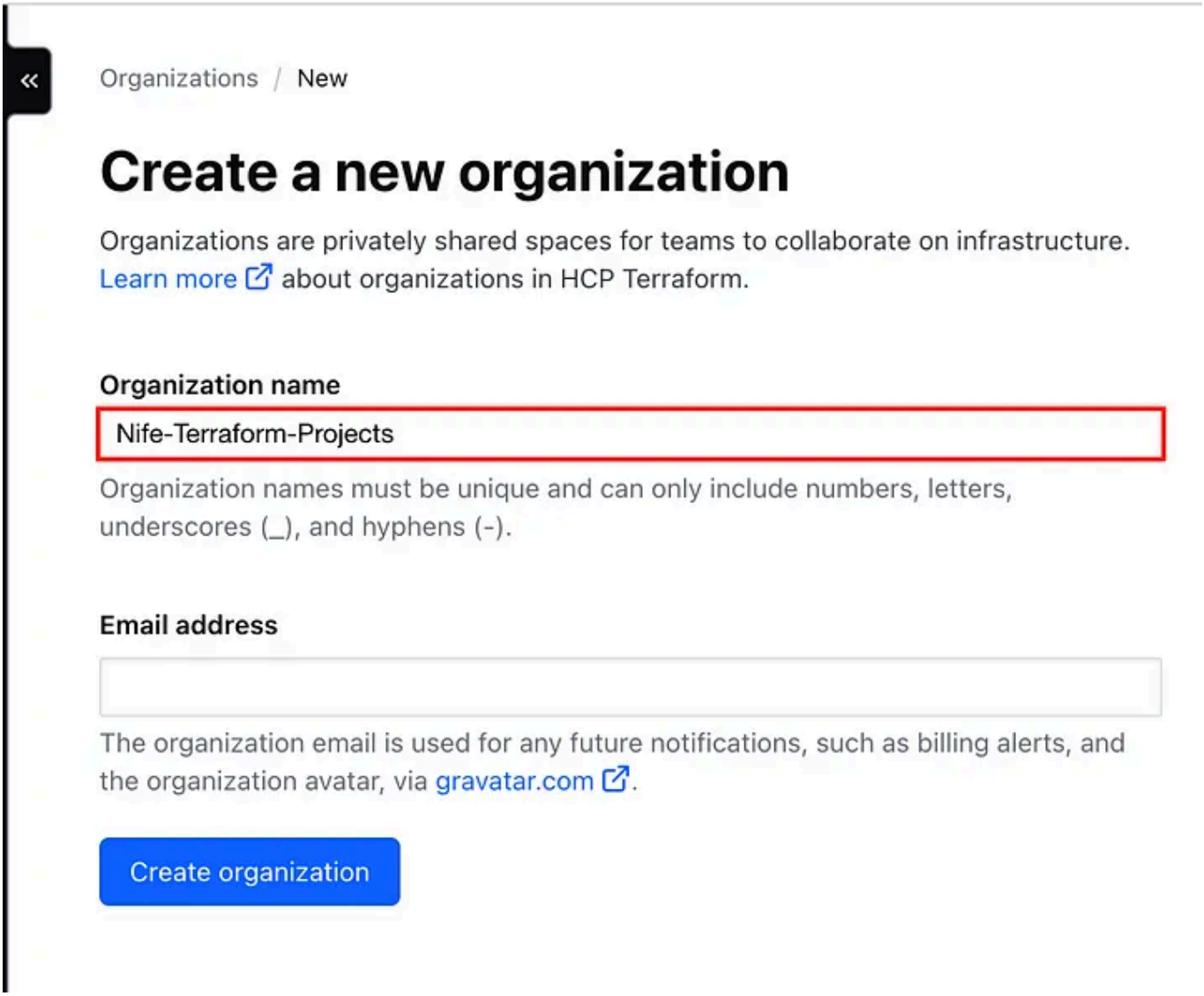
Contribute to nifss-sofowoke/terraform-2-tier-infra-project development by creating an account on GitHub.

github.com

Section 1: Terraform Cloud

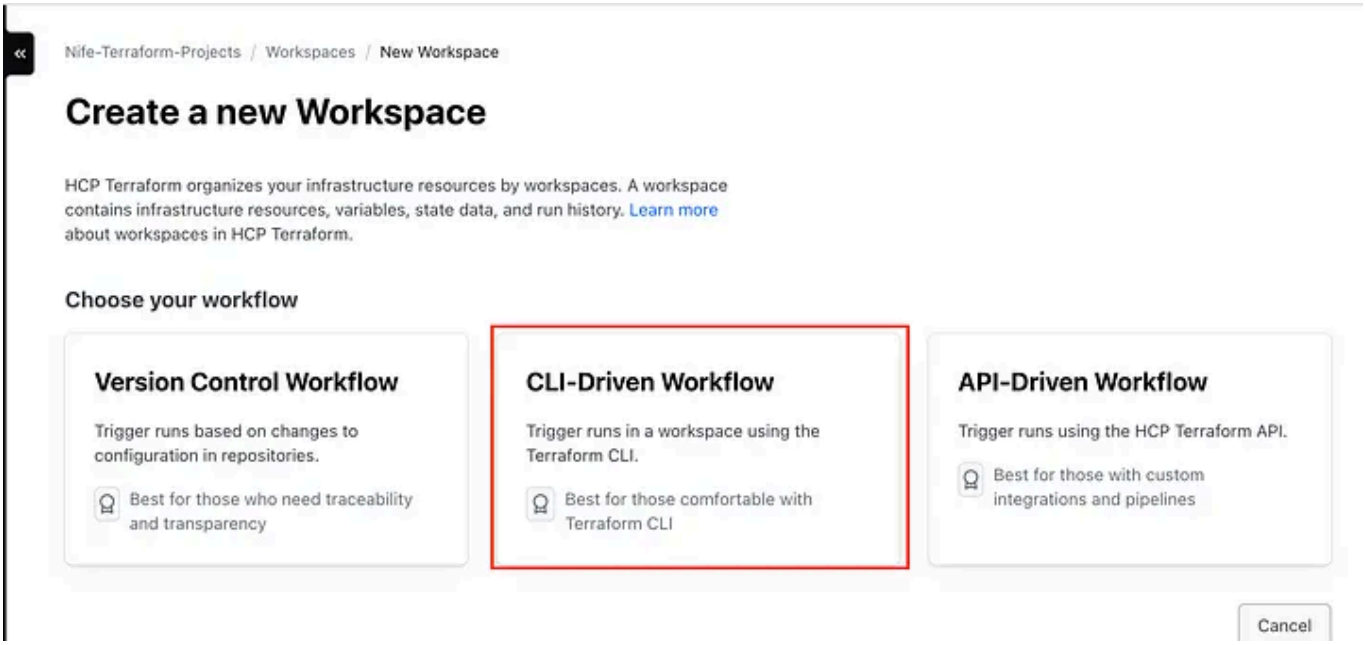
- Sign into your Terraform Cloud account
- Create a new Terraform organization



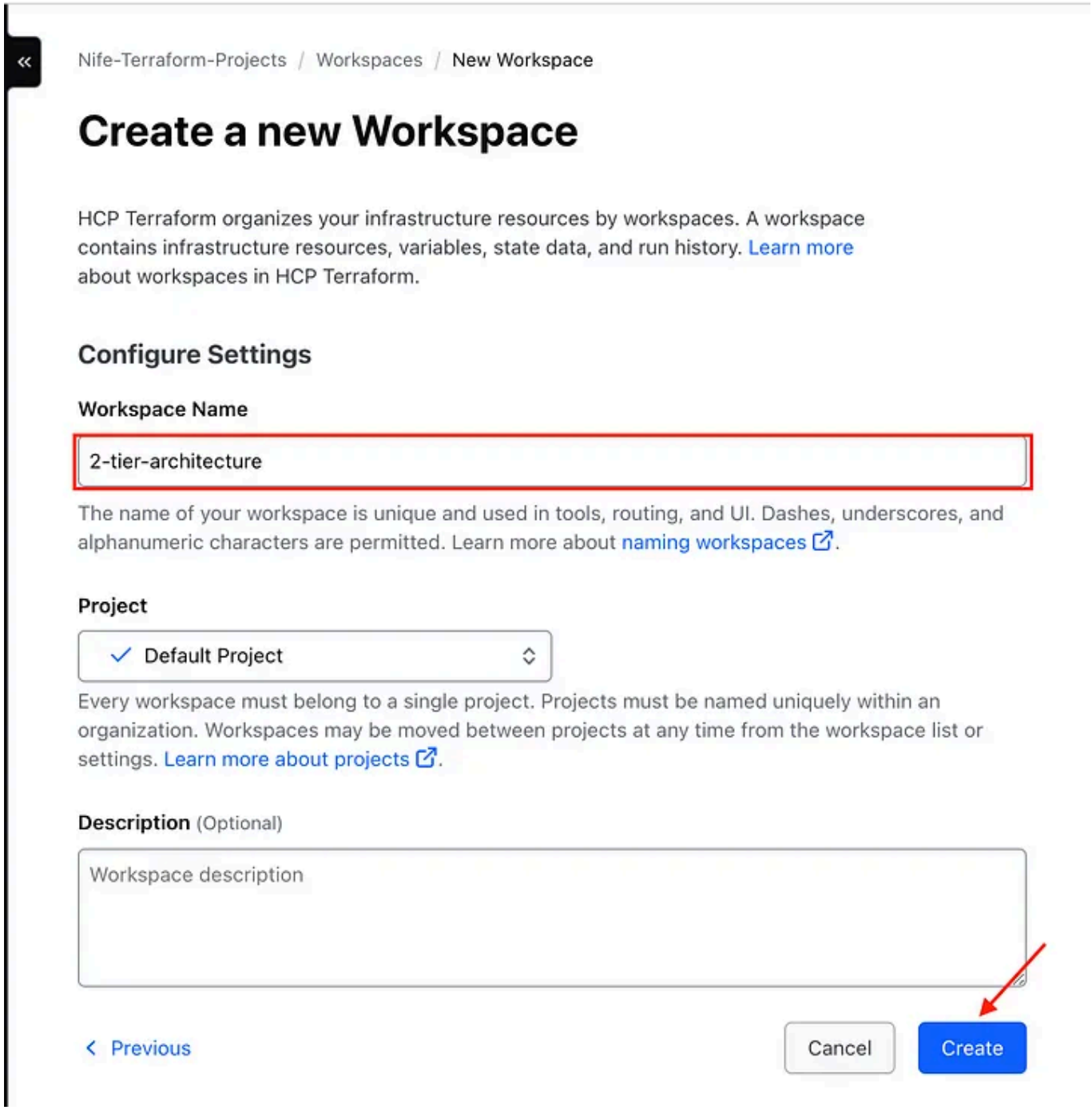


terraform organization

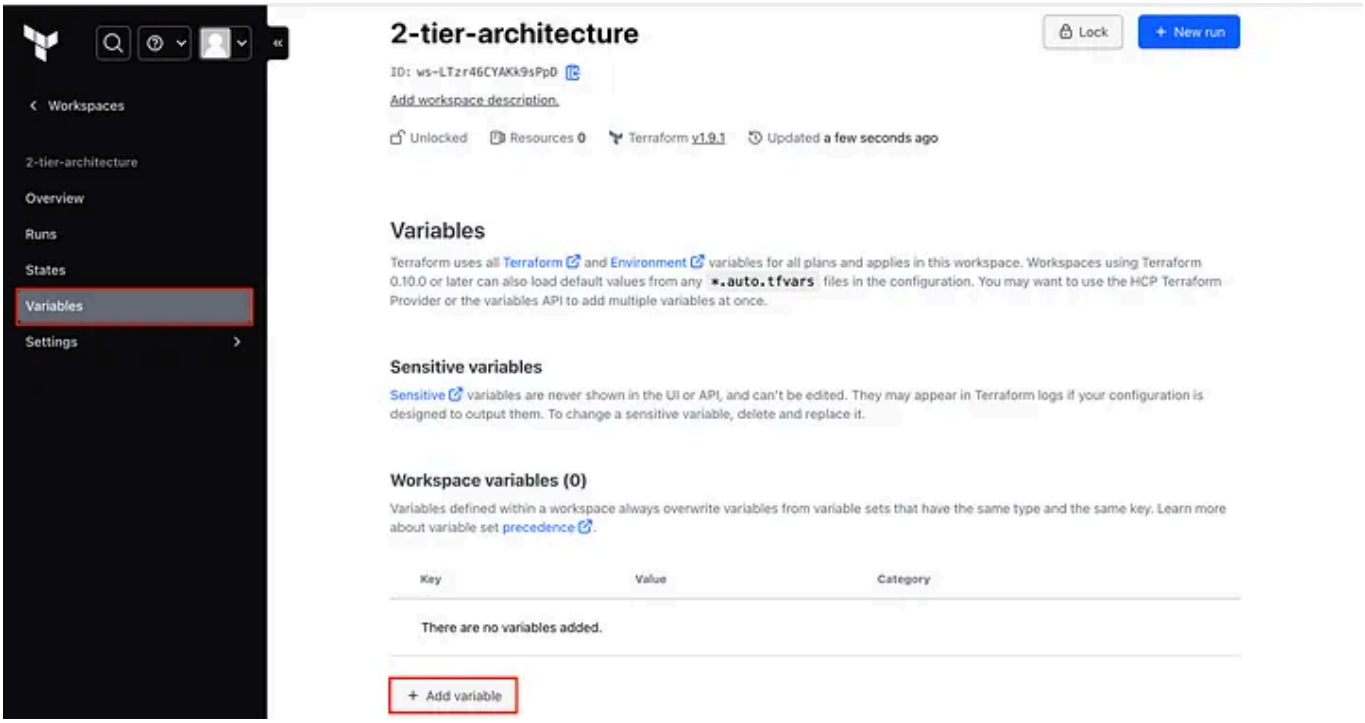
- Next, create a new workspace in that organization

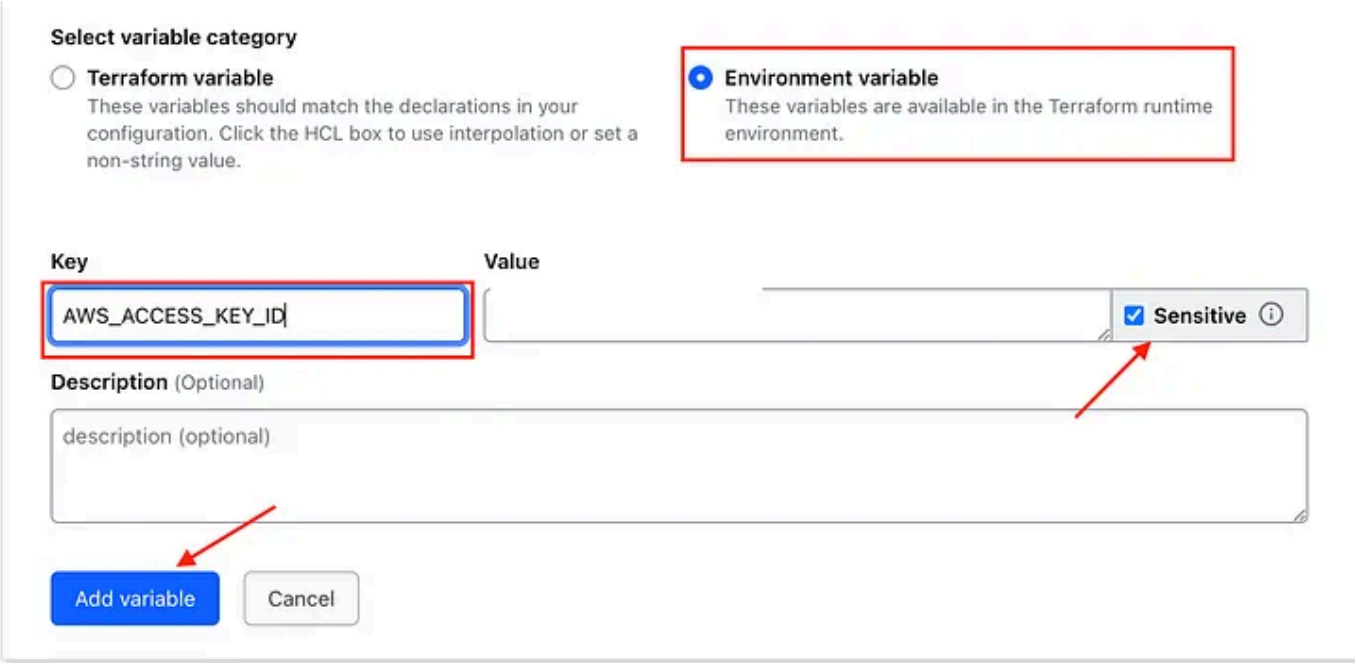


We're using a "CLI-Driven Workflow" because the Terraform CLI and its commands will be used to manage the infrastructure configuration, deployment, and state. We'll also login to Terraform cloud through the CLI to trigger changes in the workspace.



- Connect your AWS account to Terraform by adding your Access and Secret Access Keys to the workspace as Environment variables using the **“Variables”** tab. This gives Terraform access to your AWS environment which allows it to interact with the account and make changes by creating and destroying resources.





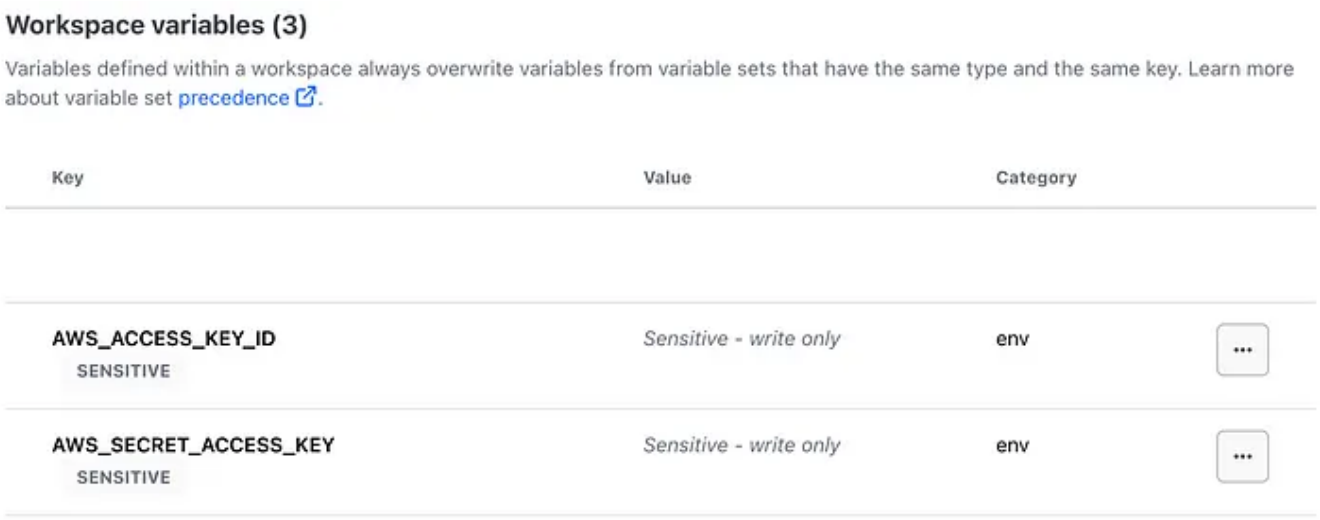
To add your access key id and secret access key like above:

- **Variable category:** Environment variable
- **Key:** AWS_ACCESS_KEY_ID
- **Value:** [your access key id]

- **Mark as sensitive**, **“add variable”** and repeat for the secret access key.

You should have something like the image below.

> *Adding AWS credentials as environment variables in Terraform cloud, and marking them as sensitive prevents us from hardcoding these values into Terraform configuration files which is a security best practice.*



Note that you can add more variables to the workspace variables if you’d like. I didn’t use any variables in this project so I did not feel the need to add any apart from my AWS credentials.

That is all the setup I did for Terraform Cloud, the next section is all about the Terraform configuration files.

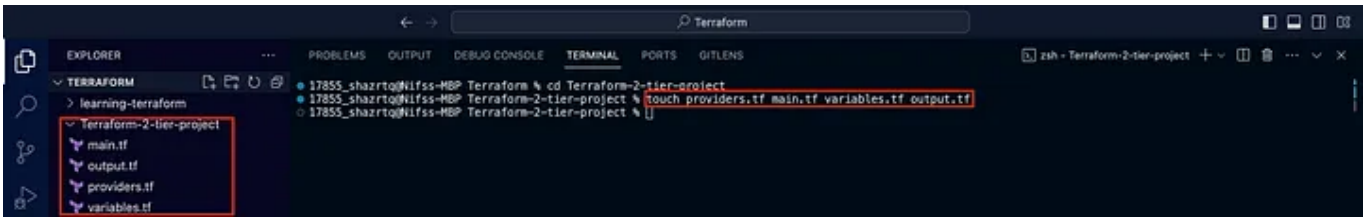
Section 2: Terraform Configuration Files

For this project, I used only 3 terraform configuration files: `providers.tf` `main.tf` and `outputs.tf`

- In VSCode in a new terminal window, create a new folder for the project

```
mmkdir <directory_name>
```

- Navigate to the newly created directory and create the configuration files.



Providers.tf

In the providers.tf file:

- create and configure the remote backend. Because we're using Terraform cloud to store the state, we'll need to configure the Terraform environment we previously set up using the code below:

```
terraform {
  backend "remote" {
    hostname     = "app.terraform.io"
    organization = "Nife-Terraform-Projects"
    workspaces {
      name = "2-tier-architecture"
    }
  }
```

By doing the above, we'll be able to use the CLI-driven workflow.

- add a `required providers` block (in this case it is AWS)

```
#required providers block
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "5.55.0"
    }
  }
}
```

- configure the AWS provider

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

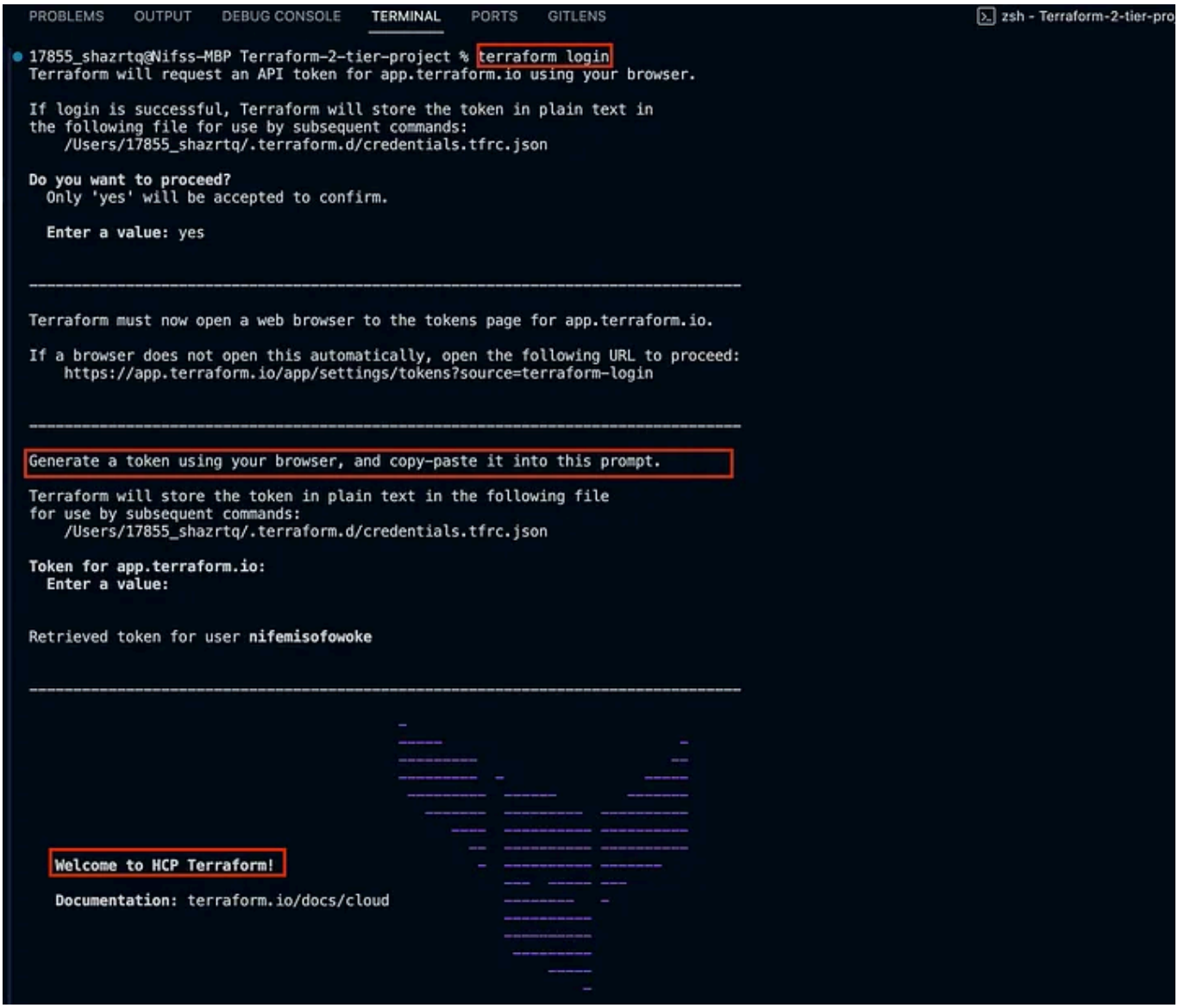
When put together, the `providers.tf` file should look like this

> *Run* `terraform fmt` *and* `terraform validate` *to make sure the file is properly formatted and is syntactically valid.*

Terraform Login

- In the CLI, log in to Terraform using the `terraform login` command.

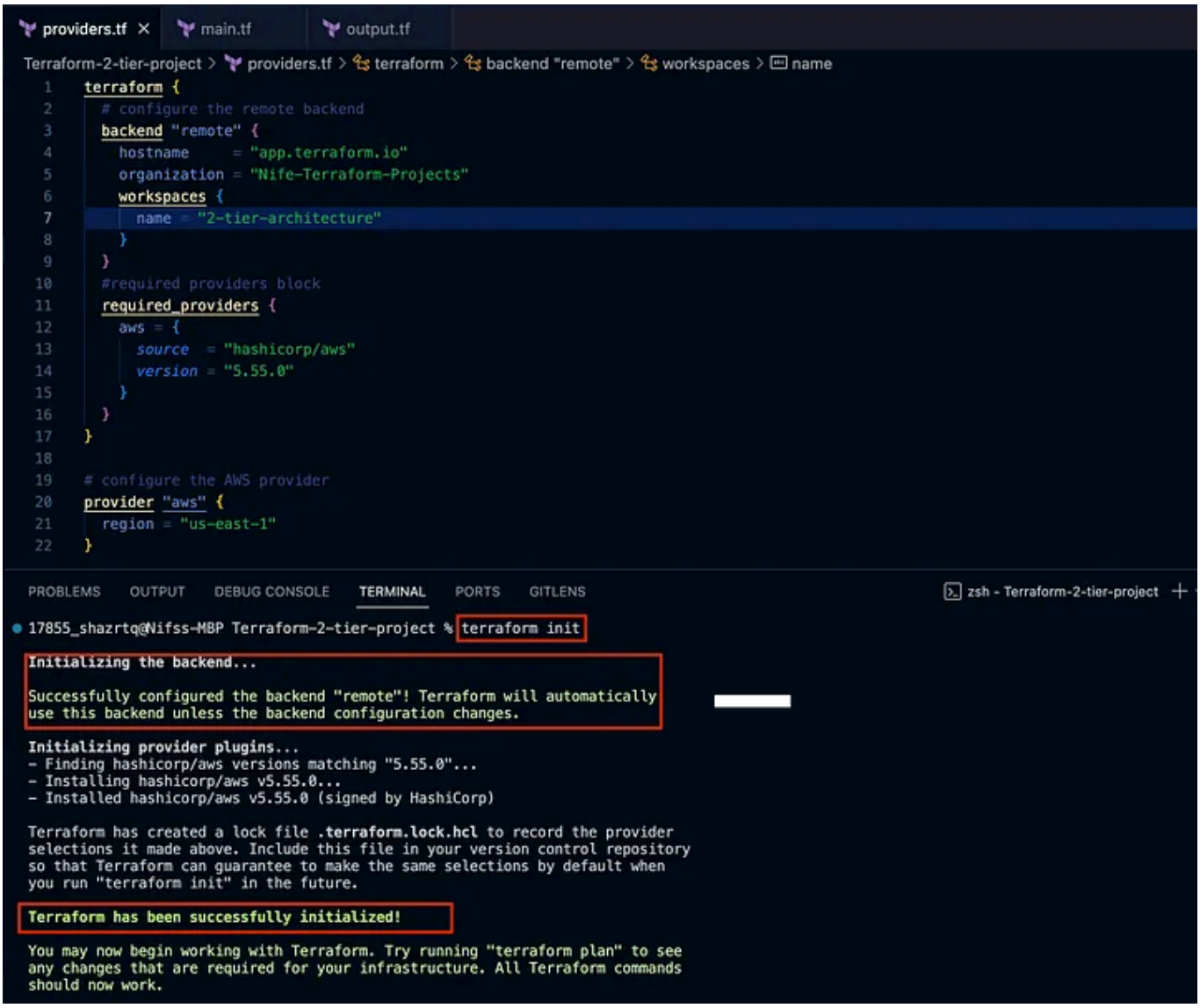
- You will be prompted to enter your Terraform cloud credentials and a token will be generated in your browser that will be used to authenticate the login as seen below:



terraform login

Initialize `providers.tf` file

- In the CLI, run the `terraform init` command to initialize the remote backend and download the required plugins.



Now we can proceed to creating the remaining configuration files.

Main.tf

We'll create all the resources for the 2-tier infrastructure using the respective resource blocks for each resource.

To create AWS resources in Terraform, I recommend using the AWS Provider documentation in the Terraform registry to build out your configurations for the resource block. This also includes identifying parts of the configurations that are optional or required. It's a great resource.

1. VPC

- Create a VPC with CIDR `10.10.0.0/16` using the Terraform resource name `aws_vpc`
- Use the resource block below:

```
resource "aws_vpc" "TF_VPC" {
  cidr_block = "10.10.0.0/16"

  tags = {
    Name = "2-tier-VPC"
  }
}
```

2. Public Subnets

- Create 2 public subnets with 2 different availability zones in the VPC created above
- Public subnet 1: `10.10.1.0/24` & Public subnet 2: `10.10.2.0/24`
- Terraform resource name: `aws_subnet`

```
# Public Subnet 1
resource "aws_subnet" "Public1" {
  vpc_id                  = aws_vpc.TF_VPC.id
  cidr_block              = "10.10.1.0/24"
  availability_zone       = "us-east-1c"
  map_public_ip_on_launch = true

  tags = {
    Name = "Public Subnet 1"
  }
```

```
}

# Public Subnet 2
resource "aws_subnet" "Public2" {
  vpc_id                  = aws_vpc.TF_VPC.id
  cidr_block              = "10.10.2.0/24"
  availability_zone       = "us-east-1d"
  map_public_ip_on_launch = true

  tags = {
    Name = "Public Subnet 2"
  }
}
```

3. Private Subnets

- Create 2 private subnets with 2 different availability zones in the VPC created above.
- Private subnet 1: `10.10.3.0/24` & Private subnet 2: `10.10.4.0/24`
- Terraform resource name: `aws_subnet`

```
# Private Subnet 1
resource "aws_subnet" "Private1" {
  vpc_id                  = aws_vpc.TF_VPC.id
  cidr_block              = "10.10.3.0/24"
  availability_zone       = "us-east-1c"
  map_public_ip_on_launch = false

  tags = {
    Name = "Private Subnet 1"
  }
}

# Private Subnet 2
resource "aws_subnet" "Private2" {
  vpc_id                  = aws_vpc.TF_VPC.id
  cidr_block              = "10.10.4.0/24"
  availability_zone       = "us-east-1d"
```

```
  map_public_ip_on_launch = false

  tags = {
    Name = "Private Subnet 2"
  }
}
```

4. Internet Gateway

- Create an Internet gateway for the VPC so the instances in the public subnet can connect to the Internet
- Terraform resource name: `aws_internet_gateway`
- Use the code below

```
resource "aws_internet_gateway" "TF_IGW" {
  vpc_id = aws_vpc.TF_VPC.id
}
```

5. Elastic IP

- Create an Elastic IP because we'll be making a public NAT gateway for the resources deployed in the private subnet
- Set an explicit dependency with the Internet Gateway using the `depends on` argument. This informs Terraform that the Elastic IP should not be created until the Internet Gateway has been deployed because traffic from the NAT gateway would be routed to the Internet Gateway.
- Terraform resource name: `aws_eip`

```
resource "aws_eip" "NAT_eip" {
  domain     = "vpc"
  depends_on = [aws_internet_gateway.TF_IGW]
}
```

6. NAT Gateway

- Create a public NAT Gateway so the RDS instance in the private subnet would be able to connect to the internet by routing traffic to the Internet gateway. However, the instance would be unable to receive unsolicited inbound connections from the internet.
- Explicitly depends on the Elastic IP
- Terraform resource name: `aws_nat_gateway`

```
resource "aws_nat_gateway" "TF_NATGW" {
  allocation_id = aws_eip.NAT_eip.id
  subnet_id     = aws_subnet.Public1.id
  depends_on    = [aws_eip.NAT_eip]
```

7. Public Route Table

- Create and configure a public route table with a default route (0.0.0.0/0) that allows all outbound traffic from the public subnet to be directed to the internet by pointing to the internet gateway.
- Terraform resource name: `aws_route_table`

```
resource "aws_route_table" "TF_Public_Route" {
  vpc_id = aws_vpc.TF_VPC.id

  route {
    cidr_block = "0.0.0.0/0"
    gateway_id = aws_internet_gateway.TF_IGW.id
  }
}
```

8. Private Route Table

- Create and configure a private route table with a default route (0.0.0.0/0) that allows all outbound traffic from the private subnet to be directed to the internet by pointing to the NAT gateway.
- Terraform resource name: `aws_route_table`

```
resource "aws_route_table" "TF_Private_Route" {
  vpc_id = aws_vpc.TF_VPC.id

  route {
    cidr_block     = "0.0.0.0/0"
    nat_gateway_id = aws_nat_gateway.TF_NATGW.id
  }
}
```

9. Public Route Subnet Association

- Assign the 2 public subnets to the public route table
- Terraform resource name: `aws_route_table_association`

```
# Public Subnet 1
resource "aws_route_table_association" "Public1" {
  subnet_id      = aws_subnet.Public1.id
  route_table_id = aws_route_table.TF_Public_Route.id
}

# Public Subnet 2
resource "aws_route_table_association" "Public2" {
  subnet_id      = aws_subnet.Public2.id
  route_table_id = aws_route_table.TF_Public_Route.id
}
```

10. Private Route Subnet Association

- Assign the 2 private subnets to the private route table
- Terraform resource name: `aws_route_table_association`

```
# Private Subnet 1
resource "aws_route_table_association" "Private1" {
  subnet_id      = aws_subnet.Private1.id
  route_table_id = aws_route_table.TF_Private_Route.id
}

# Private Subnet 2
resource "aws_route_table_association" "Private2" {
  subnet_id      = aws_subnet.Private2.id
  route_table_id = aws_route_table.TF_Private_Route.id
}
```

11. Web Server Security Group

- Create a security group for the 2 instances that'll be deployed in the public subnets

- The security group is configured to allow incoming HTTP (Port 80) and SSH (Port 22) traffic from any source IP address (0.0.0.0/0). It is also configured to allow all outbound traffic within the VPC.
- Terraform resource name: `aws_security_group`

```
resource "aws_security_group" "apache_SG" {
  name        = "apache_SG"
  description = "Allow SSH, Web traffic and all outbound traffic"
  vpc_id      = aws_vpc.TF_VPC.id

  tags = {
    Name = "apache-TF-SG"
  }

  # Create Ingress Rule to allow Web Traffic from any IP
  ingress {
    cidr_blocks = ["0.0.0.0/0"]
    from_port   = 80
    to_port     = 80
    protocol    = "tcp"
  }
  # Create Ingress Rule to allow SSH from any IP
  ingress {
    cidr_blocks = ["0.0.0.0/0"]
    from_port   = 22
    to_port     = 22
    protocol    = "tcp"
  }
  # Create Egress Rule
  egress {
    cidr_blocks = ["0.0.0.0/0"]
    from_port   = 0
    to_port     = 0
    protocol    = "-1"
  }
}
```

12. EC2 Instances

- Deploy 1 EC2 instance in each public subnet
- Bootstrap a user-data script to install and run the Apache web server on both instances
- Terraform resource name: `aws_instance`

```
# 1st server
resource "aws_instance" "apache_server1" {
  ami                         = "ami-06c68f701d8090592"
  instance_type               = "t2.micro"
  key_name                    = "Nife-LUIT-KEYS"
  vpc_security_group_ids      = [aws_security_group.apache_SG.id]
  subnet_id                   = aws_subnet.Public1.id
  associate_public_ip_address = true
  tags = {
    Name = "apache-server1"
  }

  user_data = <<-EOF
    #!/bin/bash

    # update all packages on the server
    yum update -y

    # install apache web server
    yum install httpd -y

    # start apache
    systemctl start httpd

    # enable apache to automatically start when system boots up
    systemctl enable httpd

    EOF

}

# Create 2nd server
resource "aws_instance" "apache_server2" {
  ami                         = "ami-06c68f701d8090592"
  instance_type               = "t2.micro"
  key_name                    = "Nife-LUIT-KEYS"
  vpc_security_group_ids      = [aws_security_group.apache_SG.id]
  subnet_id                   = aws_subnet.Public2.id
```

```
  associate_public_ip_address = true
  tags = {
    Name = "apache-server2"
  }

  user_data = <<-EOF
    #!/bin/bash

    # update all packages on the server
    sudo yum update -y

    # install apache web server
    sudo yum install httpd -y

    # start apache
    sudo systemctl start httpd

    # enable apache to automatically start when system boots up
    sudo systemctl enable httpd

    EOF

}
```

13. RDS Instance Security Group

- Configure a security group for the RDS instance that will be deployed in the private subnets
- Allow inbound MySQL traffic from the Web server security group on port 3306
- Allow all outbound traffic from the RDS instance
- Terraform resource name: `aws_security_group`

```
resource "aws_security_group" "RDS_SG" {
  name        = "RDS_SG"
  description = "Allows inbound MySQL traffic and allows all outbound traffic fr
```

```
  vpc_id      = aws_vpc.TF_VPC.id

  tags = {
    Name = "RDS-TF-SG"
  }

  # Create Ingress Rule to allow inbound MySQL traffic from the Web server secur
  ingress {
    security_groups = [aws_security_group.apache_SG.id]
    from_port       = 3306
    to_port         = 3306
    protocol        = "tcp"
  }
  # Create Egress Rule to allow all outbound traffic from the RDS instance
  egress {
    cidr_blocks = ["0.0.0.0/0"]
    from_port   = 0
    to_port     = 0
    protocol    = "-1"
  }
}
```

14. Subnet Group

- Create a subnet group for the database tier to launch the RDS instance
- This allows for the specification of which subnet a database instance can be launched into, providing better control over network configuration and ensuring high availability across multiple availability zones (AZs)
- The two private subnets should also be specified for the instance to be launched
- Terraform resource name: `aws_db_subnet_group`

```
resource "aws_db_subnet_group" "RDS_subnet_group" {
  name       = "rds-db"
  subnet_ids = [aws_subnet.Private1.id, aws_subnet.Private2.id]
```

```
  tags = {
    Name = "My DB subnet group"
  }
}
```

15. RDS Instance

- Deploy the RDS instance in the private subnets as specified during the configuration of the subnet group
- Terraform resource name: `aws_db_instance`

```
resource "aws_db_instance" "RDS_instance" {
  allocated_storage      = 10
  db_name                = "myrds"
  engine                 = "mysql"
  engine_version         = "8.0"
  instance_class         = "db.t3.micro"
  username               = "nife"
  password               = "Mypassword"
  parameter_group_name   = "default.mysql8.0"
  skip_final_snapshot    = true
  db_subnet_group_name   = aws_db_subnet_group.RDS_subnet_group.id
  vpc_security_group_ids = [aws_security_group.RDS_SG.id]
}
```

Putting together all the resource blocks, the `main.tf` file should look like this:

> *Run* `terraform fmt` *and* `terraform validate` *to make sure the file is properly formatted and is syntactically valid.*

Outputs.tf

Define the output values that would be needed after deploying the infrastructure:

1. Instance Public IP URL

- Will be used to access the Apache web server
- Terraform resource name: `instance_public_ip_url`

```
output "instance_public_ip_url" {
  description = "Apache Servers Public IP URL"
  value       = ["http://${aws_instance.apache_server1.public_ip}", "http://${aw
}
```

2. Instance Public DNS

- Terraform resource name: `instance_public_dns`

```
output "instance_public_dns" {
  description = "Apache Servers Public DNS"
  value       = ["http://${aws_instance.apache_server1.public_dns}", "http://${a
}
```

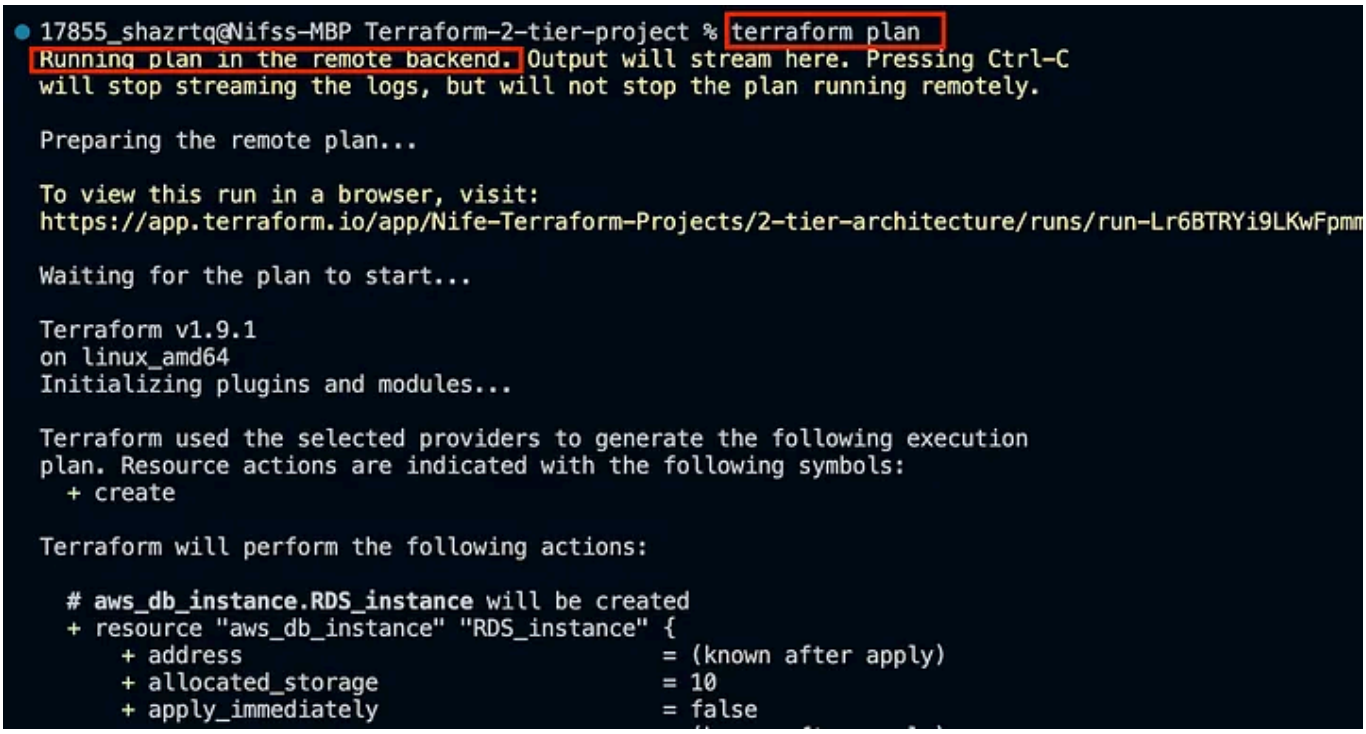
The `outputs.tf` file should look like this:

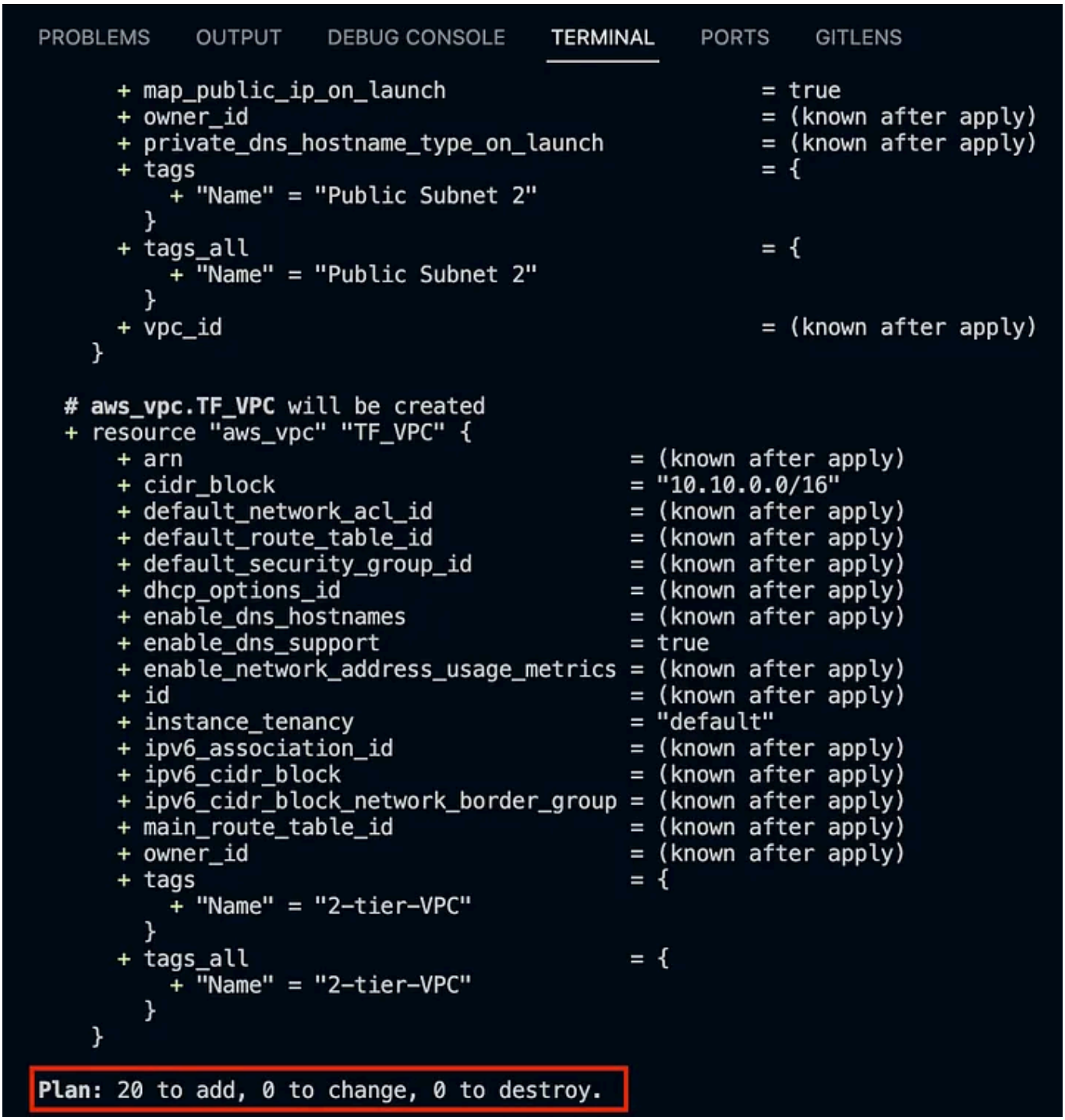
> *Run* `terraform fmt` *and* `terraform validate` *to make sure the file is properly formatted and is syntactically valid.*

After creating, validating, and initializing the configuration files needed to deploy the infrastructure we'll proceed with the remaining terraform workflow.

Terraform Plan

Run the `terraform plan` command to see the proposed changes Terraform will make to deploy the resources in the configuration file.



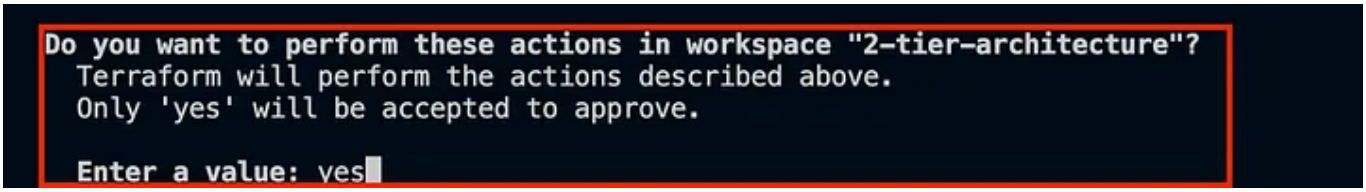


terraform plan

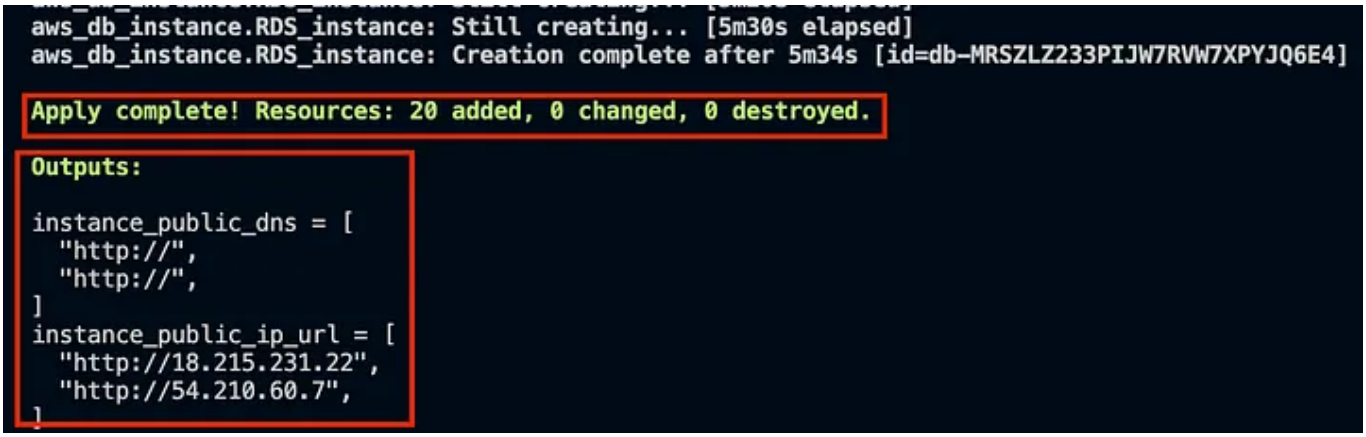
As seen above, Terraform will create **20 resources, change 0, and destroy 0.**

Terraform Apply

- Run the `terraform apply` command to execute the changes proposed in the plan



- This might take some minutes due to the RDS database being deployed (mine took about 5 minutes as seen in the image below)
- After a successful `terraform apply` the output variables declared in the `outputs.tf` file was produced in the terminal



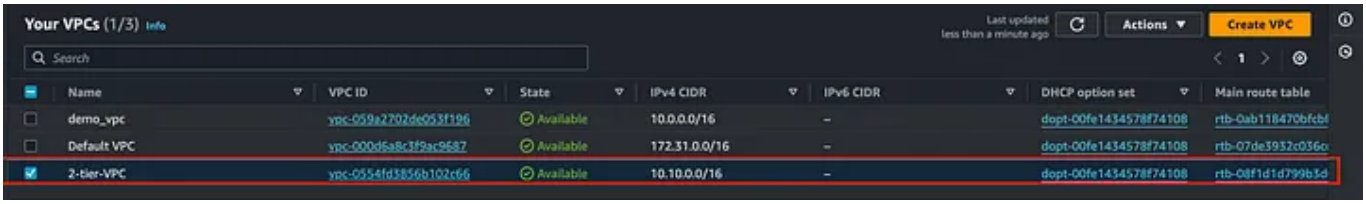
Resource Verification

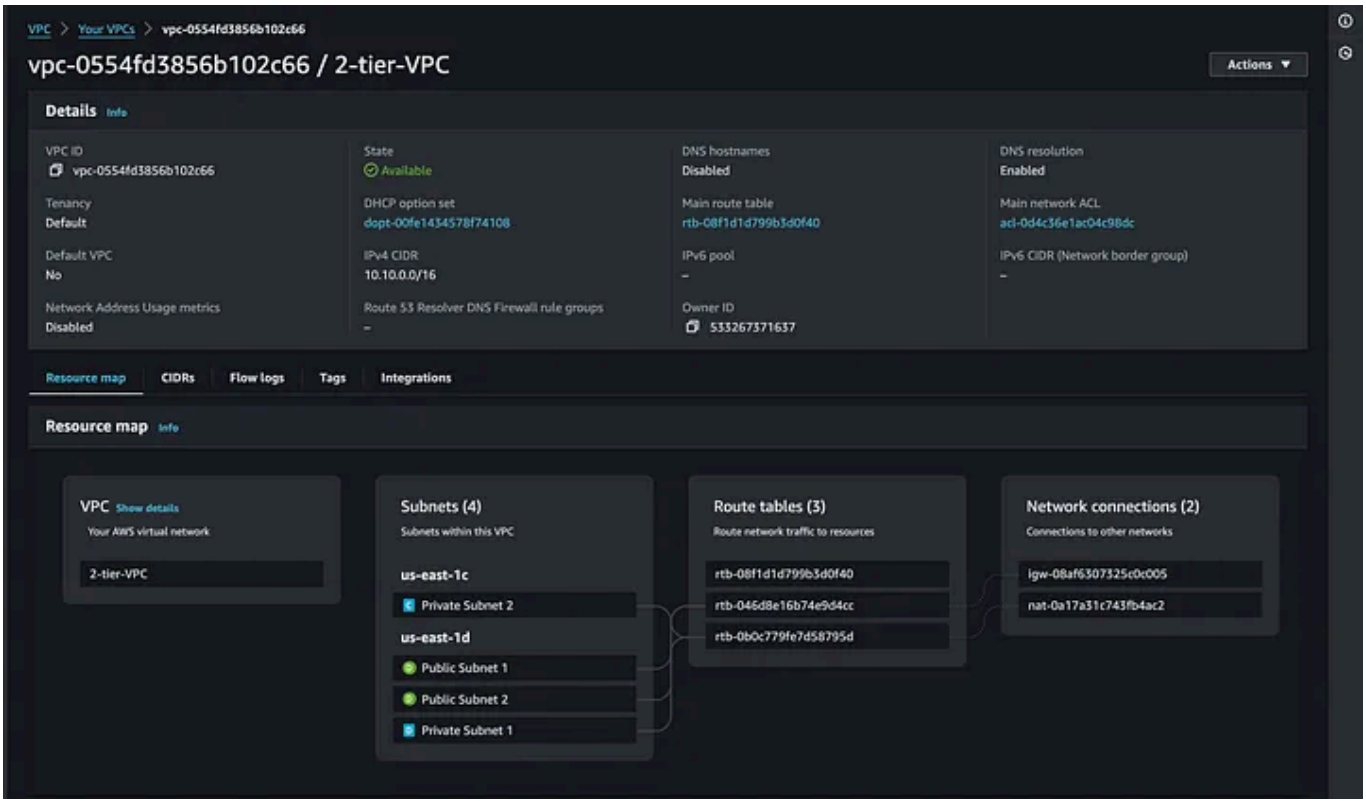
- Navigate to the AWS Console to verify that Terraform deployed the resources it was configured to create.

Verify that the:

- 2-tier custom VPC was created with its resources (2 public subnets, 2 private subnets, Elastic IP, NAT gateway, Public and Private route tables,

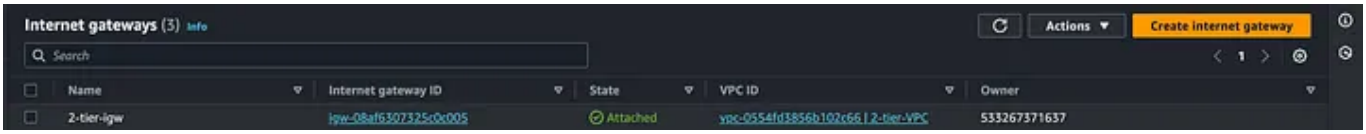
etc)





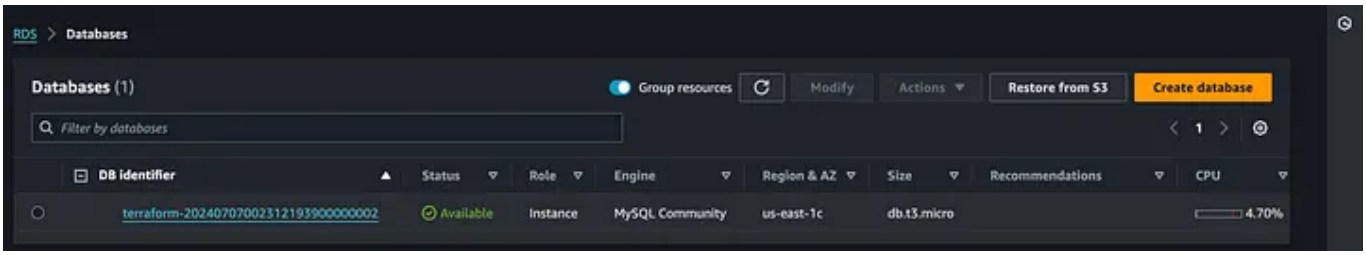
VPC Resource Map

- Internet Gateway was created



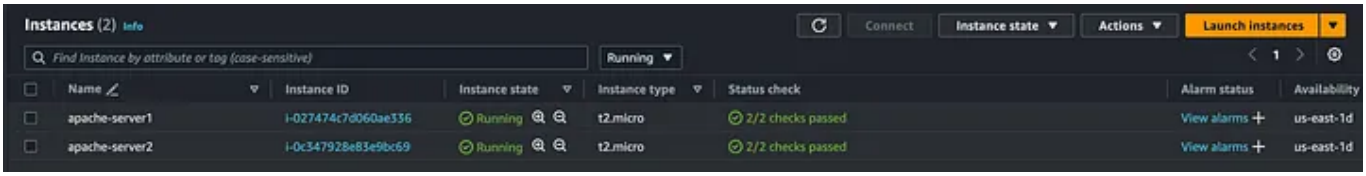
Internet gateway

- MySQL RDS database instance was deployed

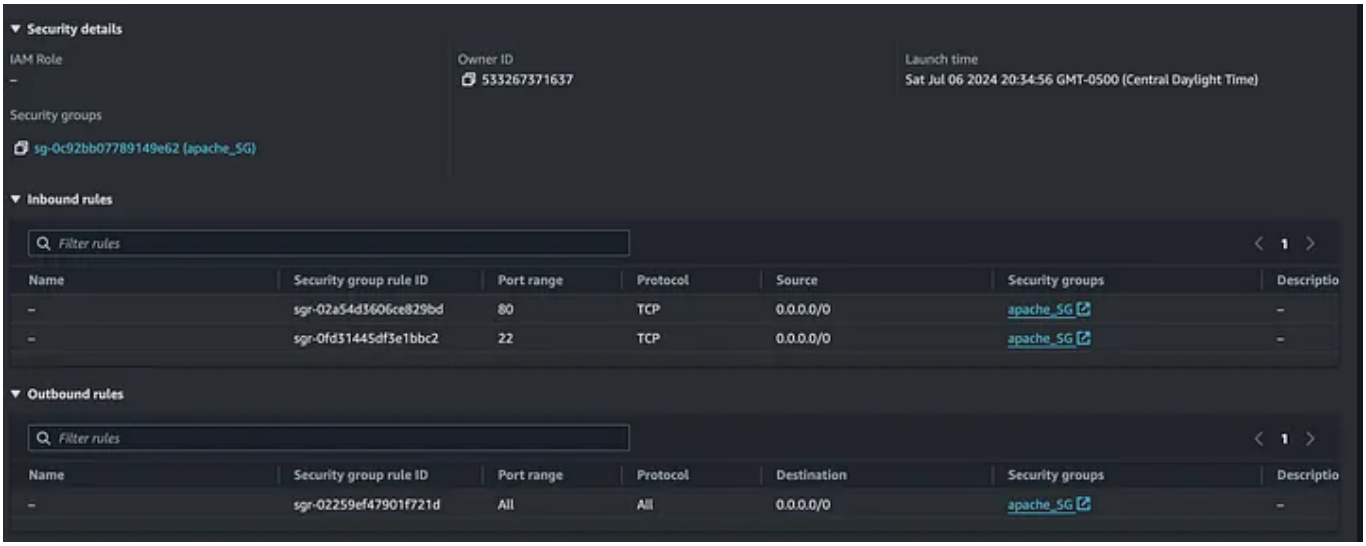


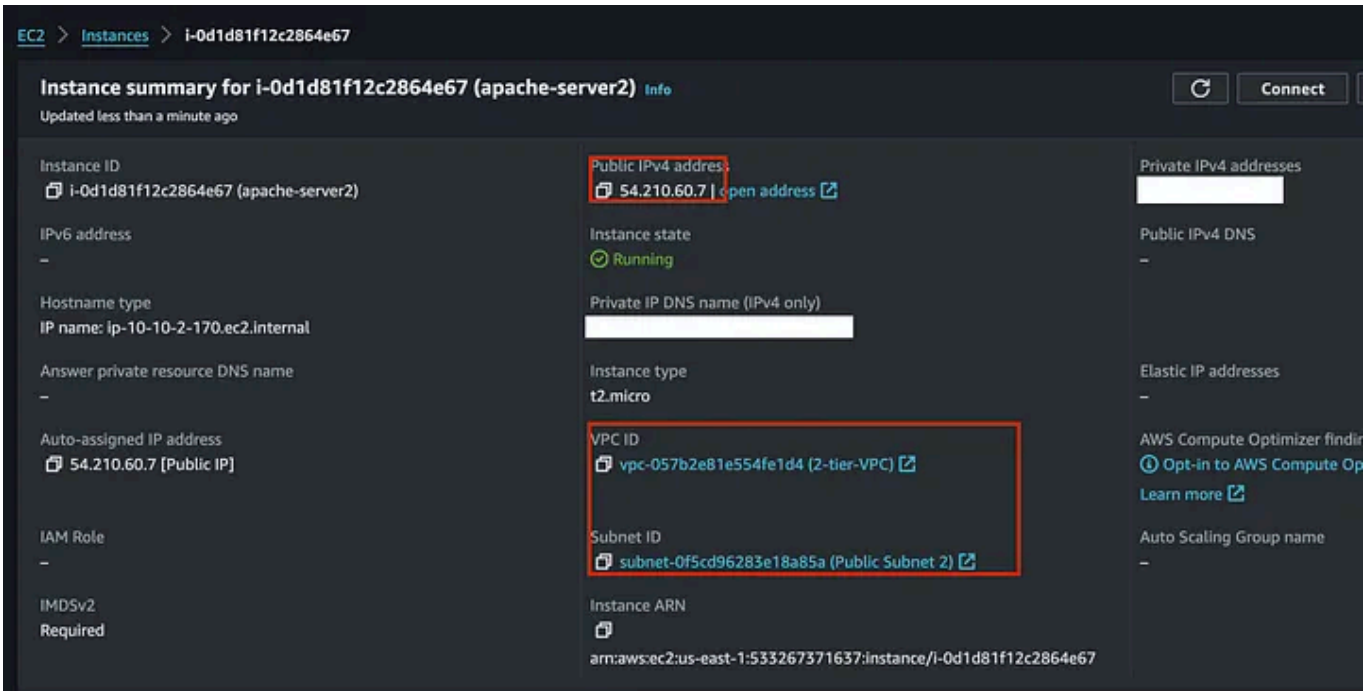
RDS instance

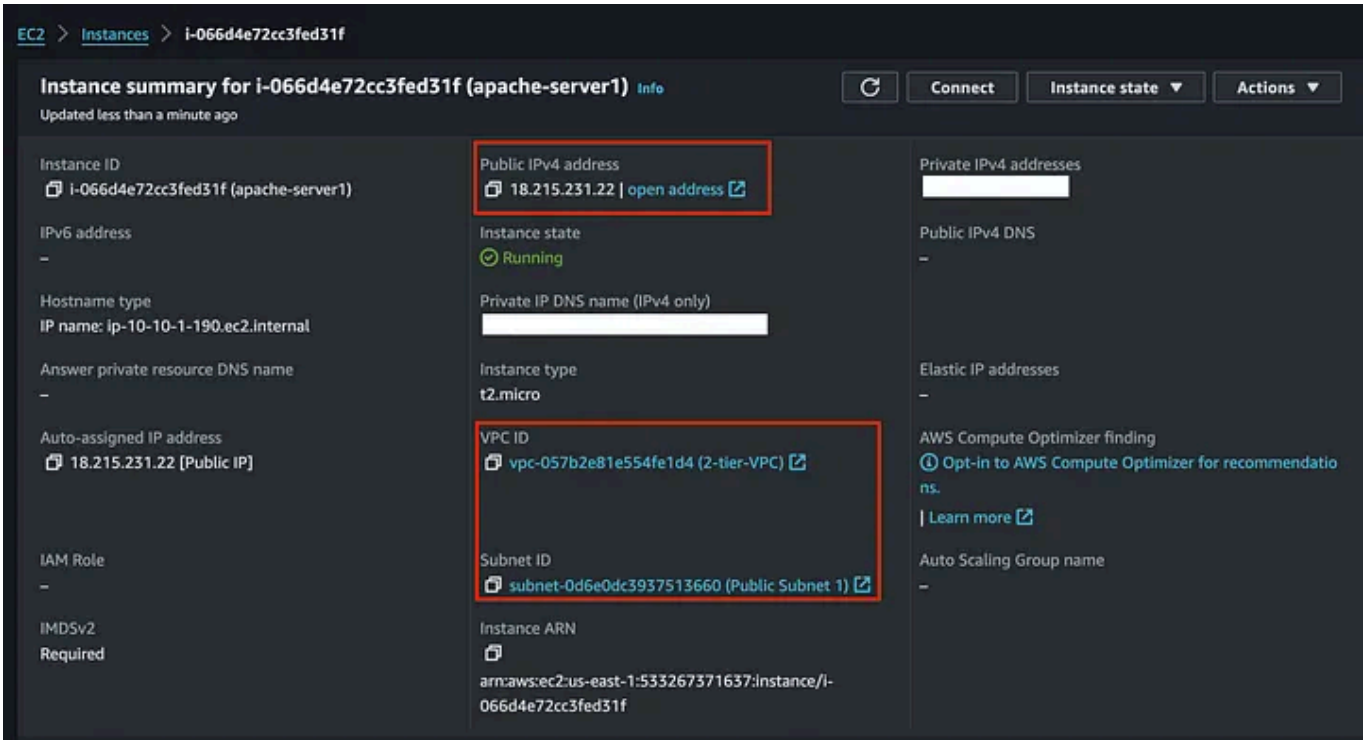
- 2 Apache web servers were deployed with their configured security groups.



running instances







Lastly, we'll verify that the Apache webservers were installed and running on the instances using the public URL outputted in the CLI after the `terraform apply` was complete.

```
Outputs:

instance_public_dns = [
  "http://",
  "http://",
]
instance_public_ip_url = [
  "http://18.215.231.22",
  "http://54.210.60.7",
]
```

Apache servers running!

Clean Up

Use the `terraform destroy` command to remove all resources and services deployed and to prevent incurring unwanted charges from the AWS Resources like the Elastic IP, Instances, etc.

```
17855_shazrtq@Nifss-MBP Terraform-2-tier-project % terraform destroy -auto-approve
Running apply in the remote backend. Output will stream here. Pressing Ctrl-C
will cancel the remote apply if it's still pending. If the apply started it
will stop streaming the logs, but will not stop the apply running remotely.
```

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

Thanks for reading! I hope you found this project valuable.

Feel free to connect with me on LinkedIn or leave any constructive feedback you have in the comments.

AWS Terraform Terraform Cloud Infrastructure As Code DevOps