Demo Project: Automate AWS Infrastructure

This project demonstrates how to automate the provisioning of AWS infrastructure using Terraform. The infrastructure components include a VPC, Subnet, Route Table, Internet Gateway, Security Group, and an EC2 instance running a Docker container. The Terraform script will also deploy a Docker container (e.g., running Nginx) on the EC2 instance automatically via user_data.

Step 1: Configure Terraform Variables

Create a file named `terraform.tfvars` with content similar to:

```
vpc_cidr_blocks      = "10.0.0.0/16"
subnet_cidr_block    = "10.0.10.0/24"
avail_zone           = "us-east-1a"
env_prefix           = "dev"
my_ip                = "167.57.113.78/32"
instance_type        = "t2.micro"
public_key_location  = "/home/eb/.ssh/id_rsa.pub"
private_key_location = "/home/eb/.ssh/id_rsa"
```

Step 2: Create VPC & Subnet

Your **main.tf** includes the following:

- **VPC Resource:**

```
resource "aws_vpc" "myapp-vpc" {
  cidr_block = var.vpc_cidr_blocks
  tags = {
    Name = "${var.env_prefix}-vpc"
  }
}
```

- **Subnet Resource:**

```
hcl
CopyEdit
resource "aws_subnet" "myapp-subnet-1" {
  vpc_id            = aws_vpc.myapp-vpc.id
  cidr_block        = var.subnet_cidr_block
  availability_zone = var.avail_zone
  tags = {
    Name = "${var.env_prefix}-subnet-1"
  }
}
```

Step 3: Create Route Table and Internet Gateway

- **Internet Gateway:**

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

- **Default Route Table:**

```
resource "aws_default_route_table" "main-rtb" {
  default_route_table_id = aws_vpc.myapp-vpc.default_route_table_id

  route {
    cidr_block = "0.0.0.0/0"
    gateway_id = aws_internet_gateway.myapp-igw.id
  }
  tags = {
    Name = "${var.env_prefix}-main-rtb"
  }
}
```

Step 4: Create Security Group

- **Default Security Group:**

```
resource "aws_default_security_group" "default-sg" {
  vpc_id = aws_vpc.myapp-vpc.id

  ingress {
    from_port   = 22
    to_port     = 22
    protocol    = "TCP"
    cidr_blocks = [var.my_ip]
  }

  ingress {
    from_port   = 8080
    to_port     = 8080
    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"]
    prefix_list_ids = []
  }

  tags = {
    Name = "${var.env_prefix}-default-sg"
  }
}
```

Step 5: Obtain the Latest Amazon Linux 2 AMI

- **Data Source for AMI:**

```
data "aws_ami" "latest-amazon-linux-image" {
  most_recent = true
  owners      = ["amazon"]
  filter {
    name   = "name"
    values = ["Deep Learning Base AMI (Amazon Linux 2) Version 57.9"]
  }
}

output "aws-ami_id" {
  value = data.aws_ami.latest-amazon-linux-image.id
}
```

Step 6: Create SSH Key Pair

- **Key Pair Resource:**

```
resource "aws_key_pair" "ssh-key" {
  key_name   = "server-key"
  public_key = file(var.public_key_location)
}
```

Step 7: Create the EC2 Instance

- **EC2 Instance Resource:**

```
resource "aws_instance" "myapp-server" {
  ami                    = data.aws_ami.latest-amazon-linux-image.id
  instance_type          = var.instance_type
  subnet_id              = aws_subnet.myapp-subnet-1.id
  vpc_security_group_ids = [aws_default_security_group.default-sg.id]
  availability_zone      = var.avail_zone
  associate_public_ip_address = true
  key_name               = aws_key_pair.ssh-key.key_name

  user_data = file("entry-script.sh")
  user_data_replace_on_change = true

  connection {
    type        = "ssh"
    host        = self.public_ip
    user        = "ec2-user"
    private_key = file(var.private_key_location)
  }

  provisioner "file" {
    source      = "entry-script.sh"
    destination = "/home/ec2-user/entry-script-on-ec2.sh"
  }

  provisioner "remote-exec" {
    script = "entry-script.sh"
  }

  provisioner "local-exec" {
    command = "echo ${self.public_ip} > output.txt"
  }

  tags = {
    Name = "${var.env_prefix}-server"
  }
}

output "ec-public_ip" {
  value = aws_instance.myapp-server.public_ip
```

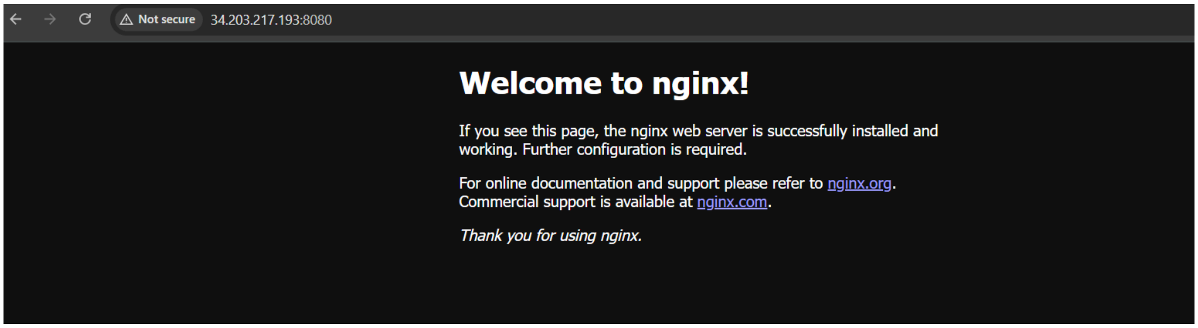
- **Entry Script (entry-script.sh):**

  Create a file named **entry-script.sh** with the following content:

  ```
  #!/bin/bash
  sudo yum update -y
  sudo yum install -y docker
  sudo systemctl start docker
  sudo usermod -aG docker ec2-user
  docker run -p 8080:80 nginx
  ```

Step 8: Deploy the Infrastructure

1. **Preview the Terraform Plan:** `terraform plan`
2. **Apply the Configuration:** `terraform apply -auto-approve`
3. **SSH into the Instance:**
   - Since `~/.ssh/id_rsa` is the default location we can also do: `ssh ec2-user@<public ip>`
4. **Verify the Docker Container:** `docker ps`
   - Confirm that the Nginx container is running.
5. **Access the Application:**
   - Open your browser and navigate to: `http://<public-ip>:8080`



Welcome to nginx!

If you see this page, the nginx web server is successfully installed and working. Further configuration is required.

For online documentation and support please refer to nginx.org.
Commercial support is available at nginx.com.

Thank you for using nginx.

Step 9: Provisioners Branch (Feature: feature/provisioners)

In case you want to add additional provisioning steps (such as a custom script execution) you can create a separate feature branch called **feature/provisioners**. Provisioners are used as a last resort and are not recommended for primary configuration, but can be useful for post-deployment tasks.

1. **Create the Feature Branch:** `git checkout -b feature/provisioners`
2. **Add/Update Provisioner Blocks in Your Terraform Code:**
   - For example, you can add a remote-exec provisioner to execute additional configuration commands:

```
provisioner "remote-exec" {
  inline = [
    "echo 'Additional configuration or scripts can be executed here'",
    "sudo docker ps"
  ]
}
```

Step 10: Clean Up

- **Destroy the Infrastructure:** `terraform destroy -auto-approve`