Nadege Gaju

Lab 2: DevOps Pipeline & Deployment of Microservices on AWS

This lab is about the integration of DevOps principles with Infrastructure as Code (IaC) using Terraform to automate the deployment of a Java-based microservice to an AWS environment

Tools used

Platform: AWS, Git (for version control)

Tools: Terraform, GitHub Actions (for CI/CD), AWS CLI, Java Microservice (Spring Boot)

Microservice Architecture

The microservice we are deploying in this lab is built using Spring Boot. This microservice handles user requests and interacts with AWS services. The architecture includes:

- Backend: A Spring Boot microservice.
- Infrastructure: Provisioned on AWS using Terraform, including EC2 instances and networking resources.
- AWS Services Used:
    - EC2: Virtual machines to host the microservice.
    - IAM: Role-based access control for managing permissions.
    - Security Groups: Firewall configurations for controlling traffic.
    - VPC: Virtual Private Cloud to isolate the microservice in a secure network environment.

Terraform Configuration for Infrastructure

Defined infrastructure using Terraform code for repeatable and automated deployments.

Step 1: Installed Terraform

- Downloaded and installed Terraform on machine

Verified the installation by running:
terraform -version

Step 2: Write Terraform Configuration Files

- Created a Terraform configuration to define the required AWS infrastructure:
    - **EC2 Instance**: Defined the instance type, AMI, and key pair.
    - **Security Group**: Allowed necessary inbound/outbound traffic (HTTP).
    - **IAM Role**: Created a role for EC2 with access to necessary AWS services (S3)
    - **VPC**: Created a private network and subnets for the instance.

Terraform Code:

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

resource "aws_instance" "my_microservice_instance" {
  ami           = "ami-12345678"
  instance_type = "t2.micro"

  security_groups = "my_security_group"

  tags = {
    Name = "Microservice-EC2"
  }
}

resource "aws_security_group" "my_security_group" {
  name = "allow_http"

  ingress {
    from_port   = 80
    to_port     = 80
    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"]
 }
}
```

Step 3: Initialized and applied Terraform Configuration

terraform init

Apply the configuration to provision infrastructure:
terraform apply

Version Control Terraform Code

Stores the Terraform configuration in a version-controlled repository for collaboration and rollback purposes.

Step 1: Added Terraform Files to Git

Created a new Git repository

Added the Terraform configuration files

Step 2: Best Practices for Version Control

- **Use .gitignore**: Add a .gitignore file to exclude sensitive information
- **Commit Often**: Commit changes incrementally to track the evolution of your infrastructure.
- **Collaborate**: Share the repository with the team to enable collaborative infrastructure management.

Update CI/CD Pipeline to Integrate Terraform

Automate infrastructure provisioning by integrating Terraform into the CI/CD pipeline (GitHub Actions)

Step 1: Update GitHub Action

- Add a stage to pipeline for running Terraform commands.

Example

```
pipeline {
  agent any

  stages {
```

```
        stage('Terraform Init') {
            steps {
                sh 'terraform init'
            }
        }

        stage('Terraform Apply') {
            steps {
                sh 'terraform apply -auto-approve'
            }
        }

        stage('Deploy Microservice') {
            steps {
                sh './deploy_microservice.sh'
            }
        }
    }
}
```

Step 2: Automate the Deployment

- The pipeline now automatically provisions infrastructure using Terraform before deploying the microservice.

Deploying the Java Microservice

Deployed the microservice to the provisioned EC2 instance.

Step 1: SSH into the EC2 Instance

- After Terraform provisions the EC2 instance, use SSH to connect to it.
- Transfer your microservice JAR file to the EC2 instance.

Step 2: Run the Microservice

Installed Java on the EC2 instance:
sudo yum install java-11-openjdk

Run the Spring Boot microservice:
java -jar microservice.jar

Testing the Deployed Microservice

- Accessed the microservice using the public IP of the EC2 instance.
- Verified that the microservice is functional by making requests to its API endpoints.

Takeaways from the Lab

- **Infrastructure as Code (IaC)**: Using Terraform to manage infrastructure as code ensures repeatability and scalability of cloud resources.
- **CI/CD Automation**: Integrating Terraform into a CI/CD pipeline allows for automated infrastructure provisioning and deployment.
- **AWS Services**: We utilized AWS services such as EC2, IAM, and security groups to manage and secure the microservice environment.
- **Microservice Deployment**: Deploying a Spring Boot microservice to AWS EC2 demonstrates how cloud infrastructure can be managed alongside application deployments in a DevOps workflow.

Best Practices for Terraform and CI/CD

- **State Management**: Use Terraform remote state storage to manage state files securely.
- **Version Control**: Always version control your Terraform configuration and keep it up to date.
- **Security**: Use IAM roles and policies to ensure that services only have the permissions they need (principle of least privilege).
- **CI/CD Pipelines**: Automate infrastructure provisioning as part of the pipeline to streamline deployments.