Three-Tier Web Application Deployment on AWS with Terraform and Jenkins

Project overrview

This project provides a detailed explanation of a **three-tier web application** deployed on AWS. The application is containerized using Docker and consists of a **React frontend**, a **Node.js backend**, and a **MySQL database**. The infrastructure is provisioned using **Terraform** and supports automatic scaling through **Auto Scaling Groups** (ASGs) and load balancing using **Application Load Balancers** (ALBs). Continuous deployment is achieved via **Jenkins**, ensuring that the application is automatically updated with every new code change.

---

1. Introduction

This project focuses on building a **highly available** and **scalable** three-tier application hosted on **Amazon Web Services (AWS)**. It leverages **Terraform** for infrastructure as code (IaC) and **Jenkins** for continuous integration and deployment (CI/CD). The core architecture divides the application into three layers:

- **Presentation Layer** (Frontend): Handles the user interface.
- **Application Layer** (Backend): Provides the API and business logic.
- **Database Layer**: MySQL database stores persistent data.

---

2. Project Architecture

2.1 Overview

The three-tier architecture is divided into:

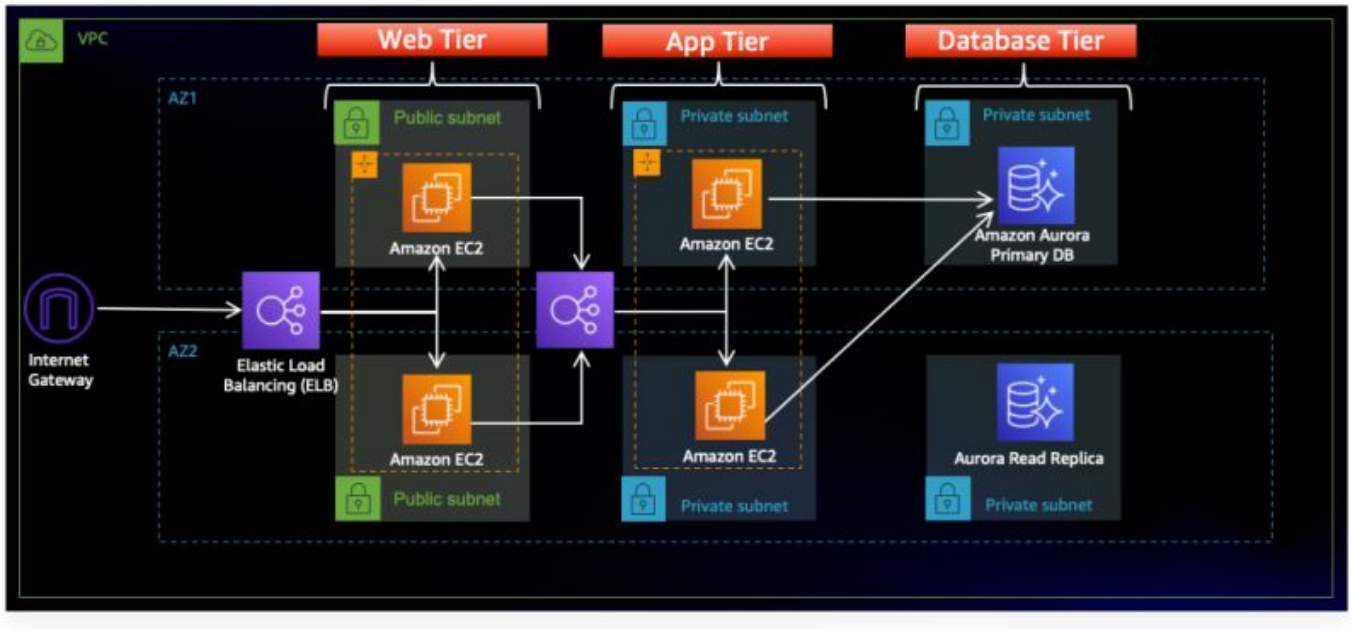
1. **Presentation Layer** (Frontend): Deployed on EC2 instances in a public subnet.
2. **Application Layer** (Backend): Deployed on EC2 instances in private subnets.
3. **Database Layer**: MySQL database running in private subnets.

Each layer is isolated in its own **security group** to enforce proper access control. The frontend is served through a **public load balancer**, while the backend communicates with the frontend via an **internal load balancer**.

2.2 High Availability & Scaling

The system achieves **high availability** through:

- **Auto Scaling Groups (ASGs)**: Dynamically adjusting instance count based on load.
- **Load Balancers (ALBs)**: Distribute traffic between healthy instances.



---

3. Technology Stack

The following technologies are used in the project:

- **AWS**: For cloud infrastructure (EC2, ALB, ASG, VPC).
- **Terraform**: Infrastructure as Code (IaC) tool for AWS resource provisioning.
- **Jenkins**: Continuous integration and delivery pipeline.
- **Docker**: To containerize the application (frontend, backend, database).
- **React**: For the user-facing frontend.
- **Node.js/Express**: For the backend API.
- **MySQL**: Relational database for data persistence.

---

4. Infrastructure Setup

The project is deployed on AWS using Terraform modules to create a secure, scalable, and highly available environment. Below is an overview of the key components:

4.1 VPC & Subnets

- **VPC**: A Virtual Private Cloud (VPC) is created with multiple public and private subnets.
- **Public Subnets**: Host the frontend instances and load balancers.
- **Private Subnets**: Host backend and database instances.

4.2 Security Groups

Security groups are defined for each layer to control access between the frontend, backend, and database layers. **Bastion host** access is enabled to connect to instances in the private subnet.

4.3 Auto Scaling Groups

- **Frontend ASG**: Hosts EC2 instances running the React frontend.

- **Backend ASG**: Hosts EC2 instances running the Node.js backend.

4.4 Load Balancers

- **Public ALB**: Distributes traffic to the frontend.
- **Internal ALB**: Distributes traffic between the backend and frontend layers.

4.5 NAT Instance

The NAT instance in the public subnet provides internet access to instances in private subnets.

4.6 Route Tables

Route tables are created to manage routing between public and private subnets. Private subnets use a route table that routes traffic through the **NAT Instance**, while public subnets route traffic through the **Internet Gateway (IGW)**.

4.7 Internet Gateway (IGW)

An Internet Gateway (IGW) is attached to the VPC to allow resources in public subnets to connect to the internet.

---

5. Application Setup

The application is containerized, with each tier being hosted as a separate Docker container.

5.1 Frontend (React)

The frontend is a React application. It runs in a container on EC2 instances in the public subnet. The instances are part of the frontend ASG, which is automatically scaled based on traffic load.

5.2 Backend (Node.js)

The backend is a Node.js application using Express.js. It runs in a container within the backend ASG and is scaled based on API traffic.

5.3 Database (MySQL)

The MySQL database runs in a private subnet to maintain security. The database containers are part of the database ASG.

---

6. Continuous Deployment

6.1 Jenkins Pipeline

Jenkins is used to automate the build and deployment of the application. The pipeline performs the following steps:

1. **Build Docker Images** for frontend, backend, and database.
2. **Push Docker Images** to Docker Hub.

3. **Trigger Terraform** to provision AWS infrastructure.
4. **Deploy Containers** onto EC2 instances using **user data scripts**.

6.2 Pipeline Example

Here's a sample pipeline configuration in Jenkins:

```
pipeline {
    agent any
    stages {
        stage('Terraform Apply') {
            steps {
                sh 'terraform init'
                sh 'terraform apply -auto-approve'
            }
        }
        stage('Build Docker Images') {
            steps {
                sh 'docker build -t azhn/frontend ./frontend'
                sh 'docker build -t azhn/backend ./backend'
            }
        }
        stage('Push Docker Images') {
            steps {
                sh 'docker push azhn/frontend'
                sh 'docker push azhn/backend'
            }
        }
    }
}
```

7. Accessing the Application

After deployment, the application can be accessed through the public DNS of the frontend ALB. The backend is connected internally via the internal ALB.

- **Frontend URL: http://**
- **Backend: Communicates with the frontend internally via the ALB.**
- **Database: Accessible only by backend in the private subnet.**

8. Security Considerations

- **Private Subnets: The backend and database are isolated in private subnets for security.**
- **Security Groups: Managed to enforce access control between tiers.**
- **NAT Instance: Provides secure outbound internet access for private instances.**
- **Bastion Host: Enables secure access to private instances.**

9. Future Enhancements

- **Implement SSL for secure communication.**
- **Use Amazon RDS for better database management.**
- **Introduce monitoring and logging with CloudWatch.**

10. Summary

This project provides a comprehensive setup for deploying a three-tier web application on AWS, utilizing Terraform for infrastructure provisioning and Jenkins for continuous deployment. The architecture is highly available and scalable with the ability to handle varying traffic loads through auto-scaling and load balancing. Each tier of the application is isolated for security, and the use of Docker containers ensures a consistent and efficient deployment process.

11. Author

This project was developed by **Abdelaziz Hasan**, a DevOps Engineer with expertise in cloud infrastructure, continuous integration, and deployment pipelines.

- Application-Repo
- Infrastructure-Repo

12. Contact Information

If you have any questions, feedback, or would like to collaborate on future projects, feel free to reach out:

- **Email**: abdelazizhasantawfiq@gmail.com

You can find more about my work and connect with me through:

- LinkedIn
- GitHub