Multi-Tier Architecture in AWS Using VPC, EC2, ELB and RDS (Mini Project)

Name – Mithiran R

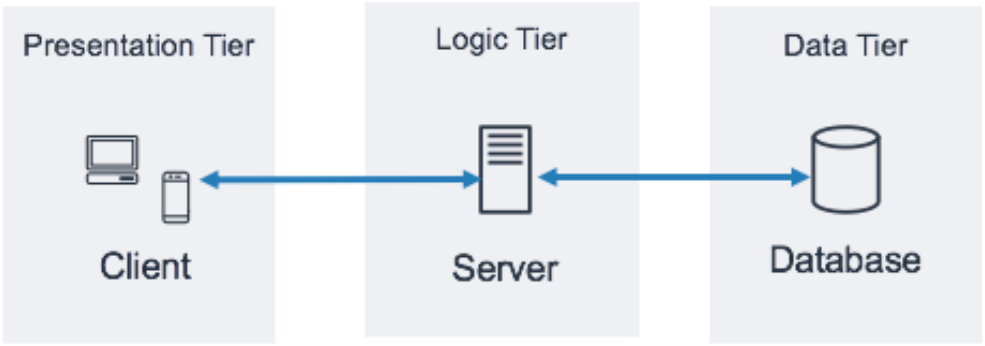
DATE – 18/08/2025

INDEX

INTRODUCTION

Multi-tier (or n-tier) architecture divides an application into separate logical/physical layers (tiers), each with a specific responsibility. That separation improves maintainability, scalability, security and allows independent evolution of each tier.

DIAGRAM



THREE-TIERS

The Common 3-Tier Architecture

- Presentation Tier (Client/UI Layer)
  - What the user interacts with (web browser, mobile app, or GUI).
  - Example: HTML/CSS/JS (React, Angular), mobile app.
- Application Tier (Logic/Server Layer)
  - The “brain” of the app where business logic lives.
  - Processes user requests, talks to the database, applies rules.
  - Example: Flask, Django, Node.js, Java Spring.
- Data Tier (Database Layer)
  - Where data is stored and managed.
  - Example: MySQL, PostgreSQL, MongoDB, AWS RDS.

STEPS INVOLVED

1. Create the VPC with their Components.

   Subnets

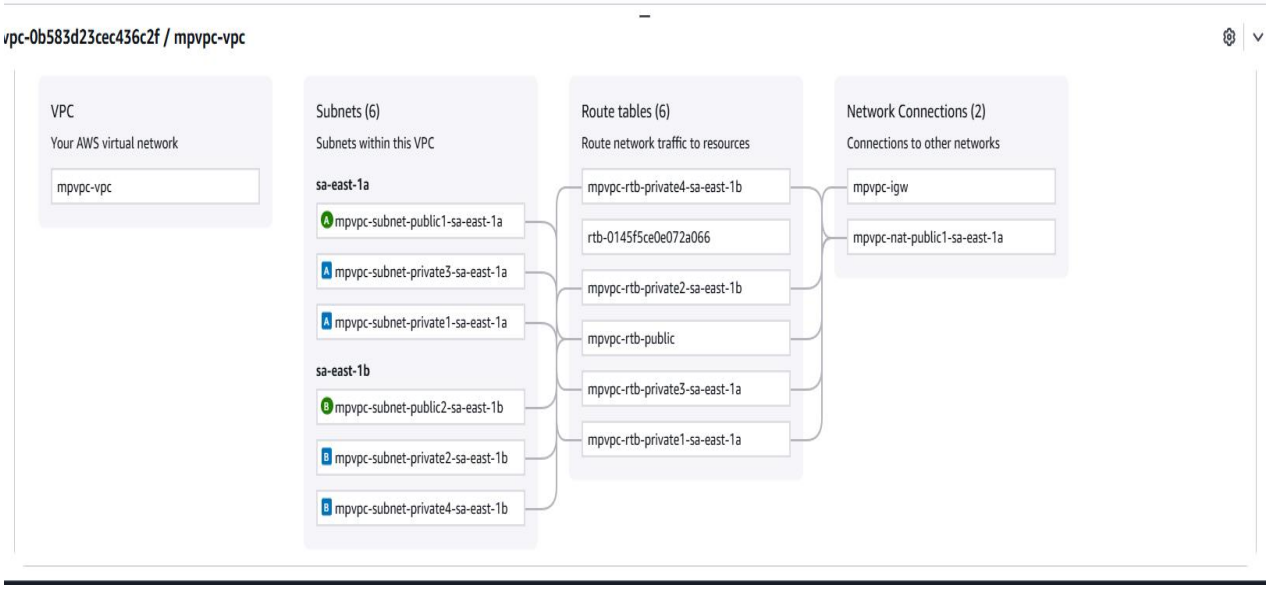
   - 2 Public Subnets.
   - 4 Private Subnets.

   Route tables

   - 1 Route Table for public subnets.
   - 1 Route Table for Private subnets.

   Gateway

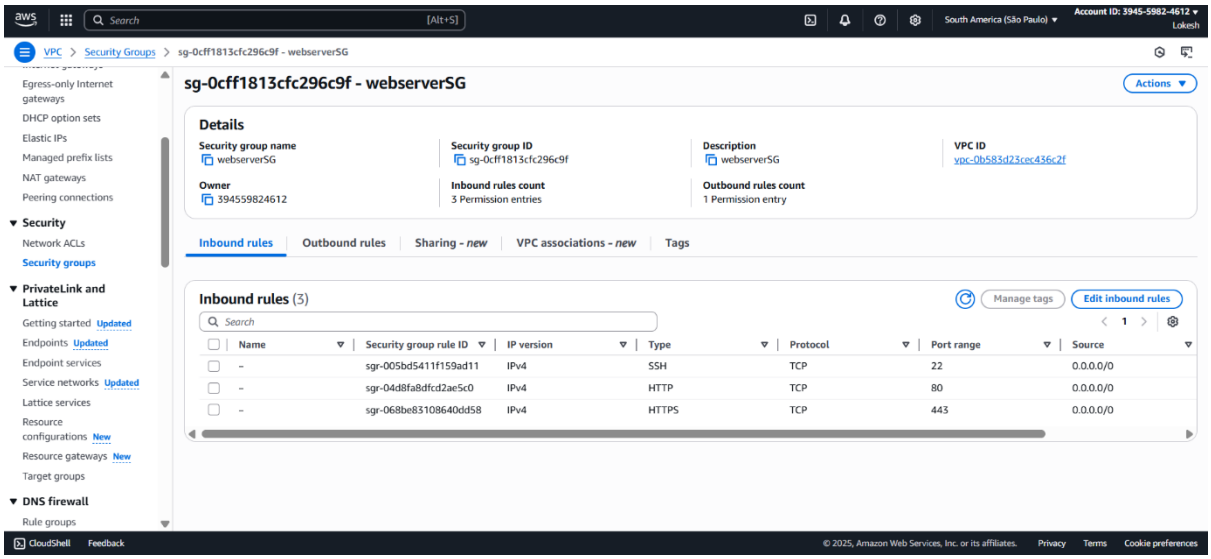
   - Internet Gateway for Public subnets.
   - NAT Gateway for Private subnets.



2. Create 2 Security Groups for Instances and 1 for database.

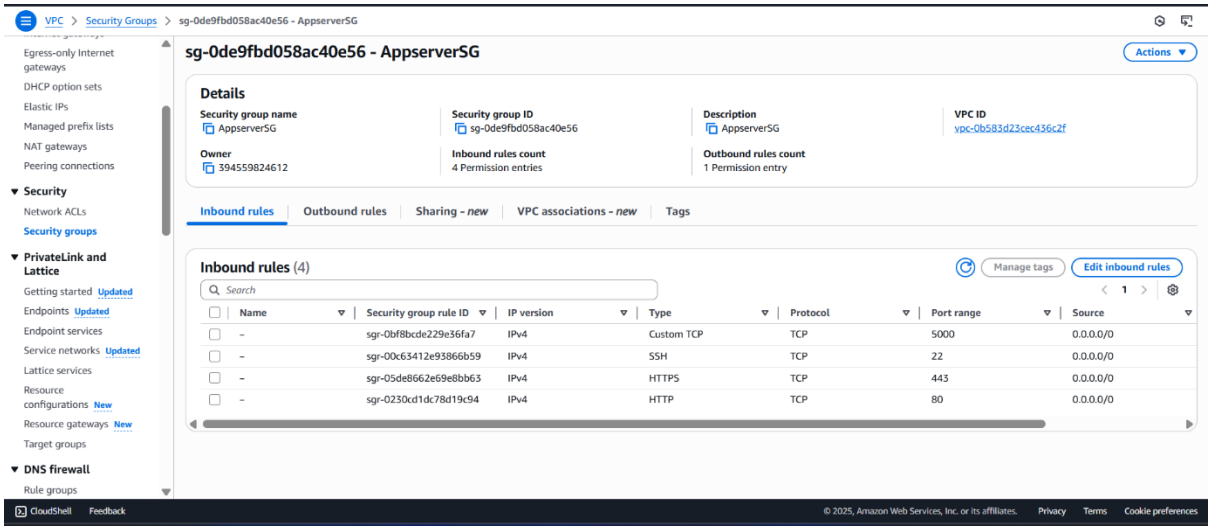
Webserver Security Groups.

- Allows SSH (ALL), HTTP (ALL), HTTPS (ALL).



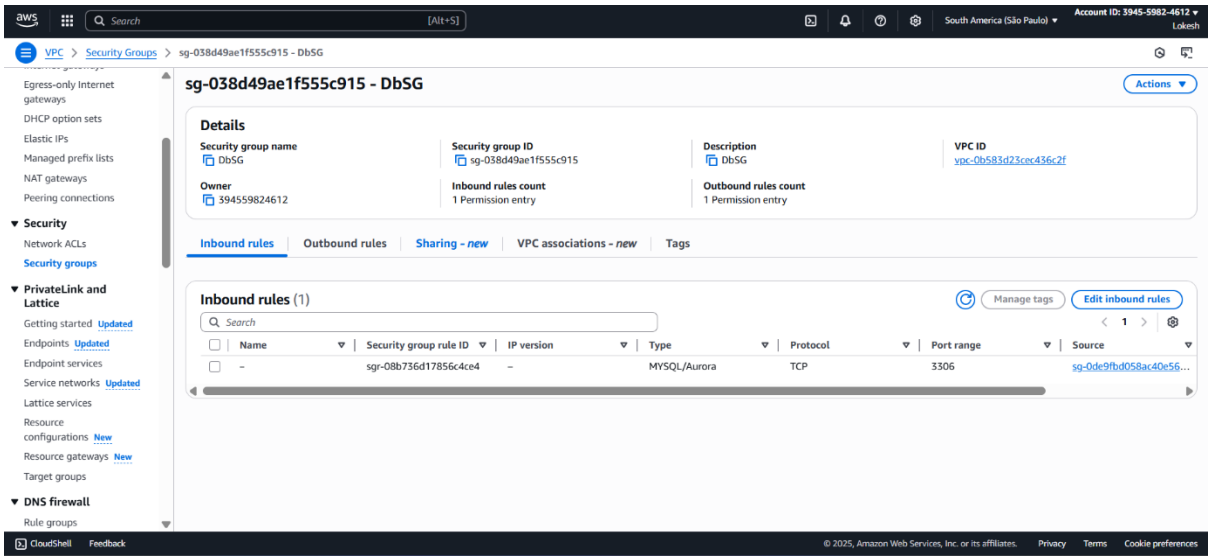
Appserver Security Groups.

- Allows CustomTCP 5000 (ALL), SSH (ALL), HTTP (ALL), HTTPS (ALL).



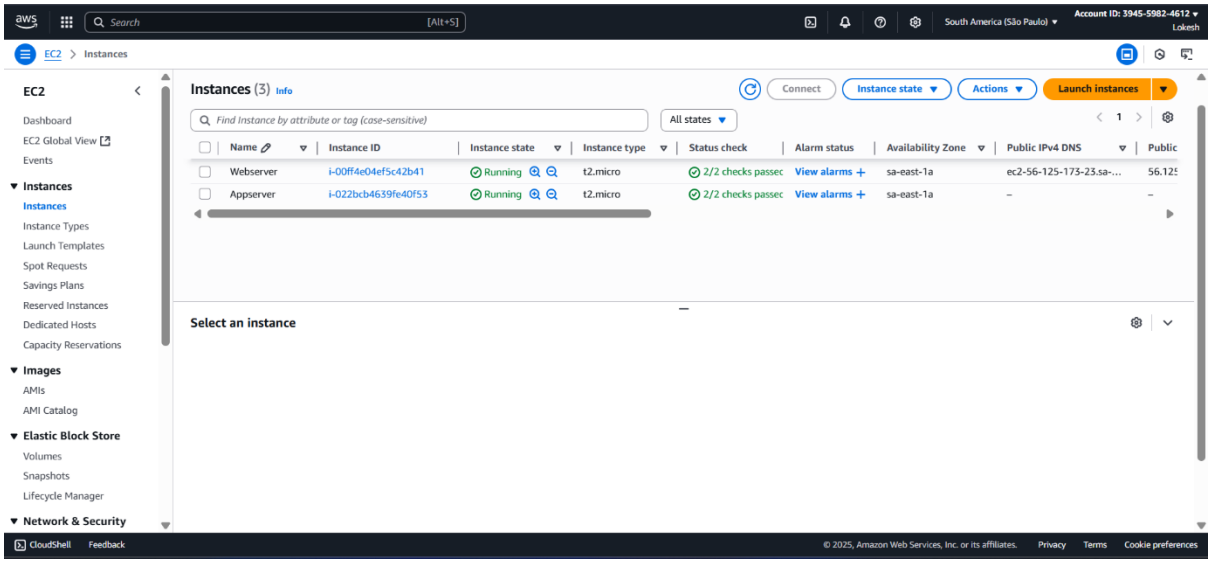
Database Security Groups.

- Allows 3306 from Appserver-SG.



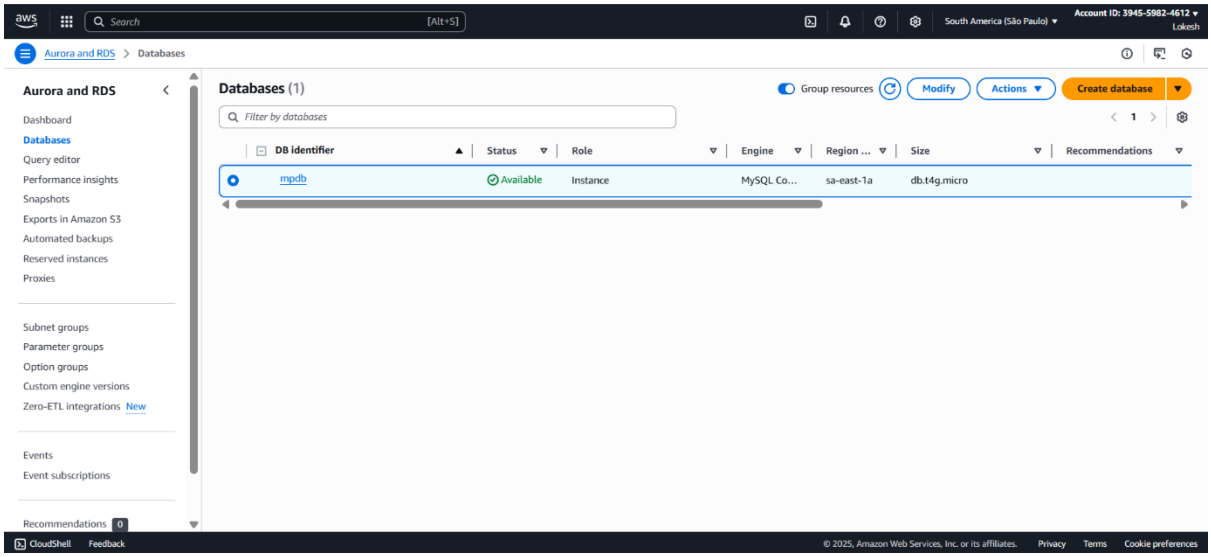
3. Create a 2 EC2 Instance.

- Create the Webserver Instance with ubuntu OS , your vpc, enable public IP Address, public subnet 1 and Webserver Security Groups.
- Create the Appserver Instance with ubuntu OS , your vpc, disable public IP Addres, private subnet 1 and Appserver Security Groups.



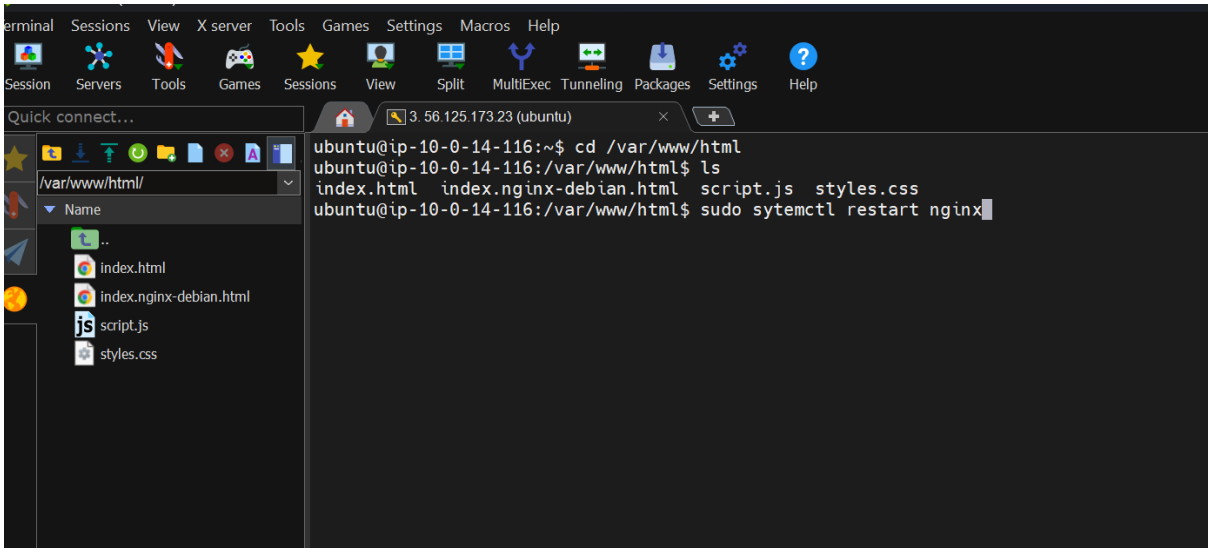
4. Create the RDS.

Create the RDS with standard, MYSQL, set name and password, connect with your instance, your VPC, your database security groups, and your subnet group.



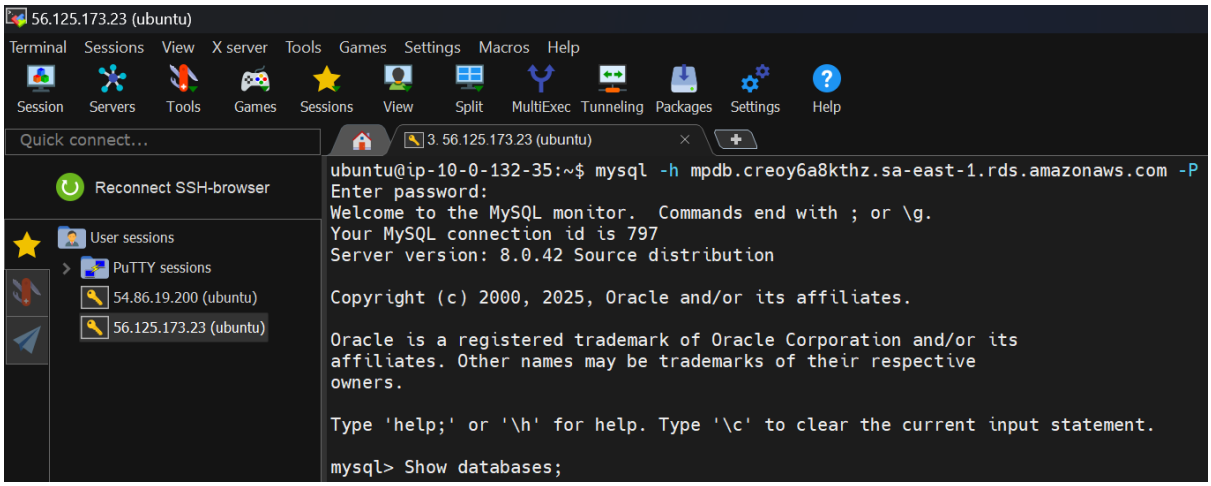
5. Setup for Webserver.

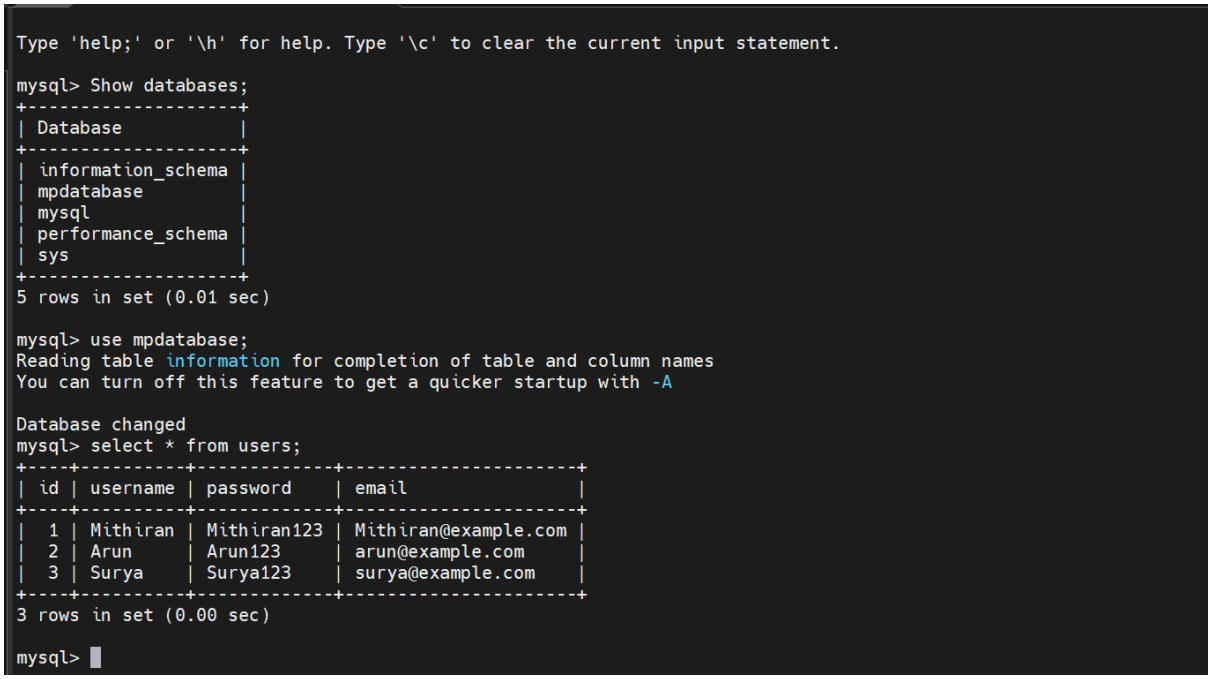
- First go and connect with the Terminal.
- And install nginx server using “sudo apt install nginx –y” command.
- Then go to cd /var/www/html.
- Now deploy your frontend packages.
- Then restart server using “sudo systemctl restart nginx” command.



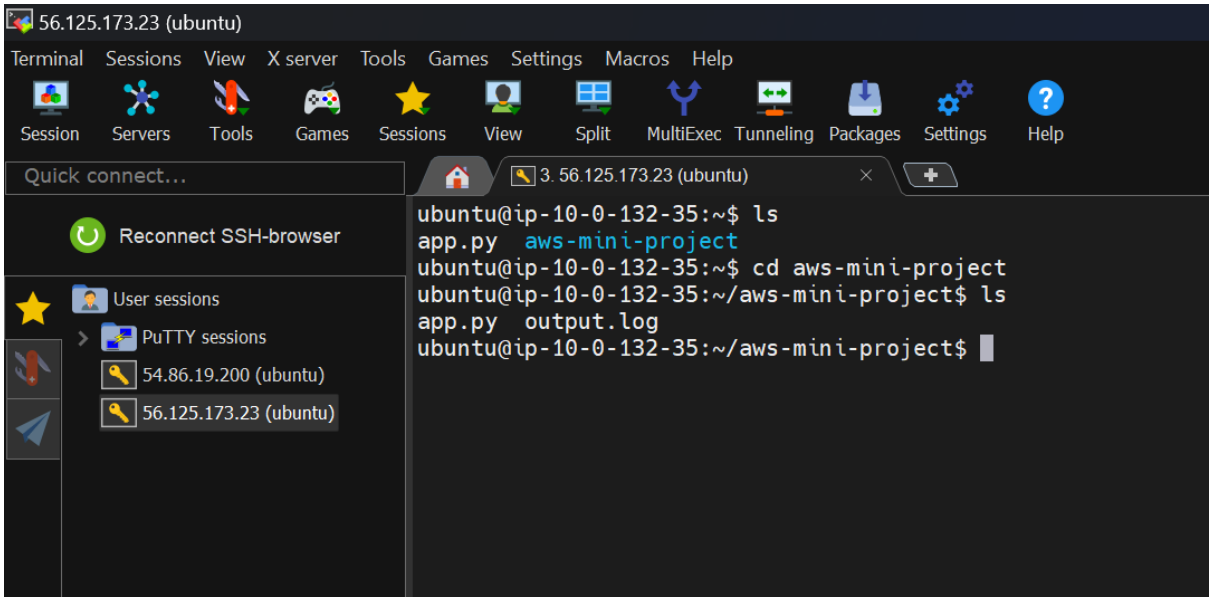
6. Setup for Appserver.
    - Connect the Appserver through via the Webserver with Appserver key.
    - Now install the MYSQl and Python using this commands.
        - sudo apt install mysql-client –y
        - sudo apt install mysql-server –y
        - sudo apt install python3 –y
        - sudo apt install python3-pip python3-venv –y
        - pip3 install flask flask-mysql-connector flask-cors --break-system-packages
    - Then connect the RDS MYSQL in this server using “mysql –h endpoint URL –P 3306 – name –p password”
    - Now it moved mysql session then create the database and use the database then create the table and insert the values using this commands.
        - create database dbname;
        - use dbname;
        - create table tablename;

- insert into table(fiels)values(values);
- select * from tablename;





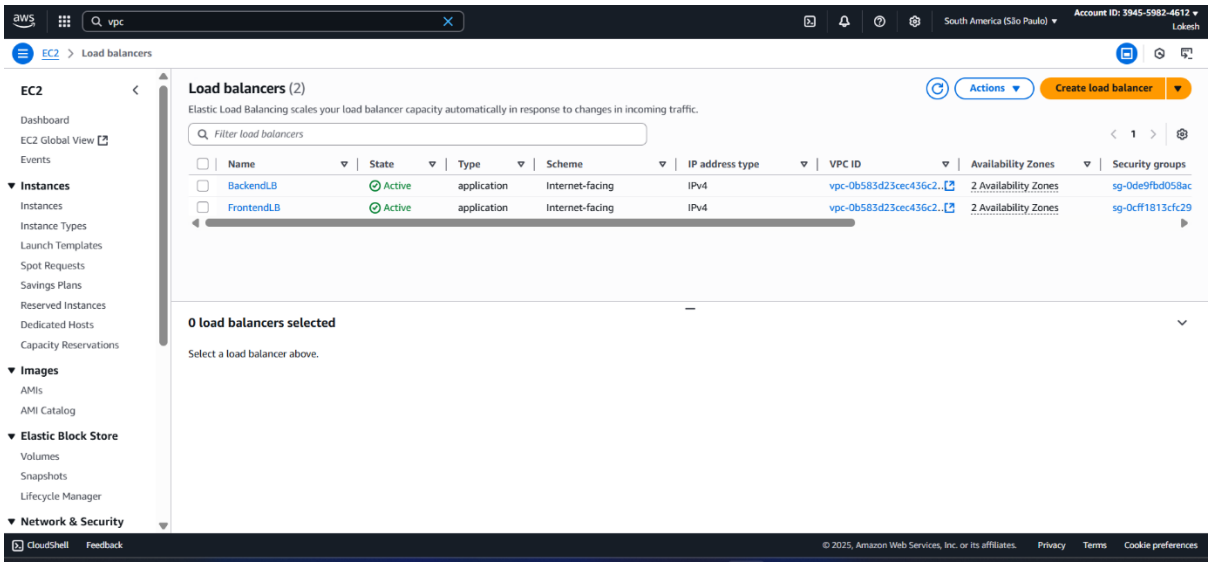
- Now come back to appserver terminal using exit.
- And create a directory then change the directory and deploy your backend package.



7. Create the 2 ELB.

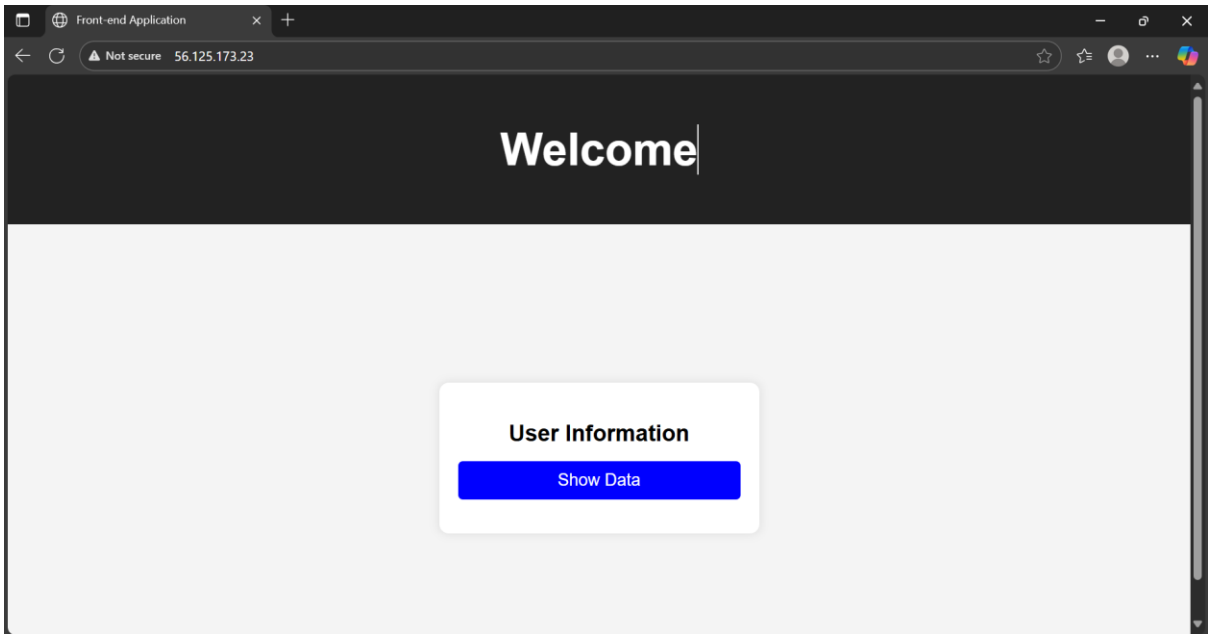
- ❖ Create Frontend LB with internet facing, you're VPC, select 2 availability zone with your public subnet, your webserver security groups and port number 80.

- ❖ Create Backend LB with internet facing, you're VPC, select 2 availability zone with your public subnet, your appserver security groups and port number 5000.

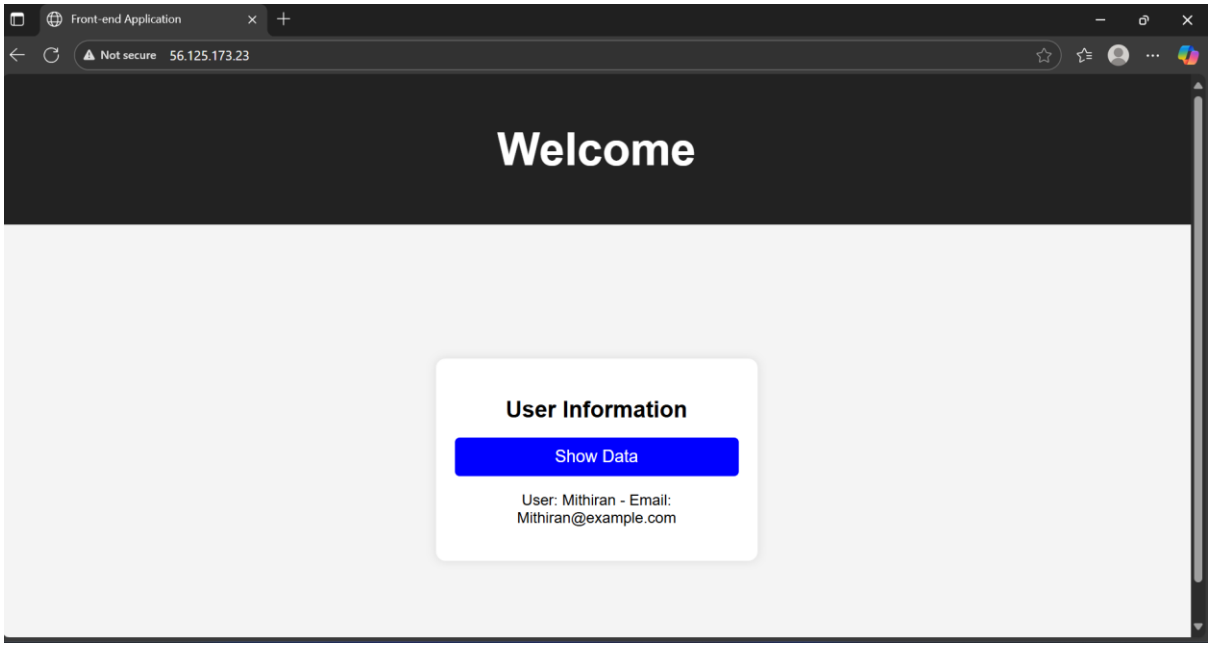


OUTPUT

Now go and copy the webserver Instance IP address and open the browser then paste IP address.



Now we click Login button in this page and we will see the data of what we upload in the mysql database.



OTHER LANGUAGES INVOLVED

Frontend Packages

index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <title>Front-end Application</title>
    <link rel="stylesheet" href="styles.css">
</head>
<body>
    <div class="header">
        <h1>Welcome </h1>
    </div>

    <div class="container">
        <img src="https://greenstech.in/images/logo.png" alt="App Logo"
class="logo">

        <div class="login-box">
            <h2>Login</h2>
            <button id="loginButton">Show Data</button>
            <p id="response"></p>
        </div>
    </div>

    <script src="script.js"></script>
</body>
</html>
```

styles.css

```
body {
    font-family: Arial, sans-serif;
    background-color: #f4f4f4;
    text-align: center;
    margin: 0;
    padding: 0;
}

.header {
    background-color: #222;
    color: white;
    padding: 20px;
    font-size: 24px;
    text-align: center;
}

.container {
    display: flex;
    flex-direction: column;
    justify-content: center;
    align-items: center;
    height: 80vh;
}

.logo {
    width: 150px;
    height: auto;
    margin-bottom: 20px;
}

.login-box {
    background-color: white;
    padding: 20px;
    border-radius: 10px;
    box-shadow: 0px 0px 10px rgba(0, 0, 0, 0.1);
```

```
    width: 300px;
}

h2 {
    margin-bottom: 15px;
}

button {
    background-color: blue;
    color: white;
    border: none;
    padding: 10px 20px;
    font-size: 18px;
    cursor: pointer;
    border-radius: 5px;
    width: 100%;
}

button:hover {
    background-color: darkblue;
}
```

script.js

```
document.getElementById("loginButton").addEventListener("click",
function () {
    // Backend API URL (Make sure ALB listens on HTTP, not port 5000)
    const backendURL = "http://BackendLB-146616817.sa-east-
1.elb.amazonaws.com/login";  // Replace with actual ALB DNS

    fetch(backendURL, {
        method: "GET",
        headers: {
            "Content-Type": "application/json"
        }
    })
    .then(response => {
        if (!response.ok) {
```

```
            throw new Error("Network response was not ok " +
response.statusText);
        }
        return response.json();
    })
    .then(data => {
        if (data.username && data.email) {
            document.getElementById("response").innerText =
                `User: ${data.username} - Email: ${data.email}`;
        } else {
            document.getElementById("response").innerText = "No user data
found!";
        }
    })
    .catch(error => {
        console.error("Error fetching data:", error);
        document.getElementById("response").innerText = "Failed to load
data!";
    });
});
```

Backend Packages

app.py

```
from flask import Flask, jsonify
from flask_cors import CORS
import mysql.connector

app = Flask(__name__)
CORS(app)

# Add your Database connection details in the below three lines
db_config = {
    "host": "your database endpoint",
    "user": "admin",
    "password": "your password",
    "database": "your database"
```

```
}

@app.route('/login', methods=['GET'])
def login():
    try:
        conn = mysql.connector.connect(**db_config)
        cursor = conn.cursor(dictionary=True)
        cursor.execute("SELECT * FROM users LIMIT 1;")  # Adjust query
as needed
        user = cursor.fetchone()
        cursor.close()
        conn.close()

        if user:
            return jsonify(user)
        else:
            return jsonify({"message": "No users found"}), 404

    except Exception as e:
        return jsonify({"error": str(e)}), 500

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000, debug=True)
```

CONCLUSION

Multi-tier architecture is a way of designing applications by separating them into distinct layers — presentation (UI), application (logic), and data (database/storage). This separation improves scalability, maintainability, fault tolerance, and security.

- Presentation tier handles user interaction (browser/mobile + Nginx/ALB).
- Application tier runs business logic (Flask + Gunicorn, autoscaled).
- Data tier manages persistence (RDS MySQL/Postgres, Redis for caching).

For your Flask + MySQL on AWS project, the best approach is:

- Run Flask behind Gunicorn (via systemd) for production stability.
- Use Nginx (or ALB) as the entrypoint and load balancer.
- Keep RDS in a private subnet, with proper security groups.
- Secure everything with TLS (ACM/Certbot) and manage credentials via Secrets Manager.

☞ The essence: multi-tier architecture gives you clear separation of concerns, independent scaling per tier, and better security. It’s the backbone of almost every modern web application.

--THANK YOU--