TravelGo – Your Cloud-Powered Travel Companion

Project Description:

TravelGo is a comprehensive, full-stack travel booking platform built to revolutionize the way users plan and manage their travel. Designed with a modern, intuitive user interface and powered by robust cloud infrastructure, TravelGo enables seamless reservations for buses, trains, flights, and hotels—all from a single, unified dashboard

Developed using Flask as the backend framework, TravelGo is hosted on Amazon EC2, ensuring scalability and high availability. It leverages Amazon DynamoDB for high-performance, NoSQL data storage, making it ideal for handling user profiles, travel listings, and booking records.

Scenario 1: Multi-Mode, End-to-End Booking Experience

Imagine a traveler planning a business trip from **Hyderabad to Bangalore**. With TravelGo:

They log into the platform and enter travel details. The system presents a curated list of **bus**, **train**, and **flight** options with dynamic pricing and availability. After selecting a train and a hotel near their destination, the user confirms both bookings in a single session.

Scenario 2: Automated, Real-Time Booking Confirmations

A student booking a hotel in Chennai for exam preparation completes the reservation within seconds. Immediately:

> TravelGo stores the booking information securely in DynamoDB. AWS SNS sends a customized confirmation email with check-in details and cancellation policy. Simultaneously, hotel staff receive a booking alert, ensuring rooms are reserved accordingly—minimizing booking errors and improving service reliability. AWS SNS sends a customized confirmation email with check-in details and cancellation policy. Simultaneously, hotel staff receive a booking alert, ensuring rooms are reserved accordingly—minimizing booking errors and improving service reliability.

Scenario 3: Centralized Dashboard with Travel Summary

A regular traveler logs in to check upcoming plans. In their dashboard:

> They see all their bookings—like a flight to Delhi and a hotel in Connaught Place—clearly displayed. Each entry includes booking ID, price, travel date, status (confirmed/cancelled), and a cancel or reschedule button. Each entry include booking ID, price, travel date, status (confirmed/cancelled), and a cancel or reschedule button. booking ID, price, travel date, status (confirmed/cancelled), and a cancel or reschedule button. Flask retrieves the data from DynamoDB and renders it using Jinja templates, ensuring fast, dynamic loading and real-time data updates**.**

Project WorkFlow:

1. AWS Account Setup and Login

Activity 1.1: Set up an AWS account if not already done.

Activity 1.2: Log in to the AWS Management Console

2. DynamoDB Database Creation and Setup

Activity 2.1: Create a DynamoDB Table.

Activity 2.2: Configure Attributes for User Data and Book Requests.

3. SNS Notification Setup Activity

Activity 3.1: Create SNS topics for book request notifications**.**

Activity 3.2: Subscribe users and library staff to SNS email notifications**.**

4. Backend Development and Application Setup

Activity 4.1: Develop the Backend Using Flask.

Activity 4.2: Integrate AWS Services Using boto3.

5. IAM Role Setup

Activity 5.1: Create IAM Role Activity

Activity 5.2: Attach Policies

6. EC2 Instance Setup Activity

Activity 6.1: Launch an EC2 instance to host the Flask application**.**

Activity 6.2: Configure security groups for HTTP, and SSH access.

7. Deployment on EC2

Activity 7.1: Upload Flask Files

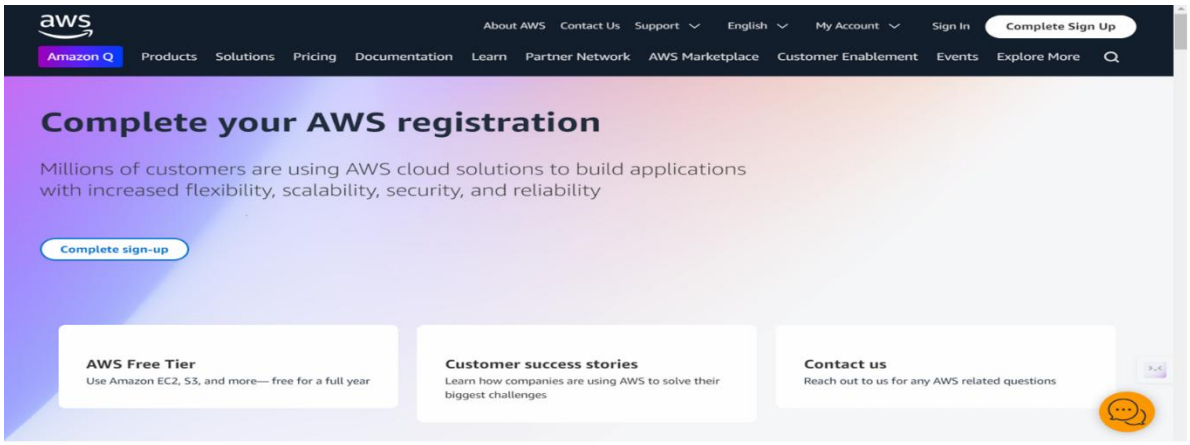
Activity 7.2: Run the Flask App

8. Testing and Deployment

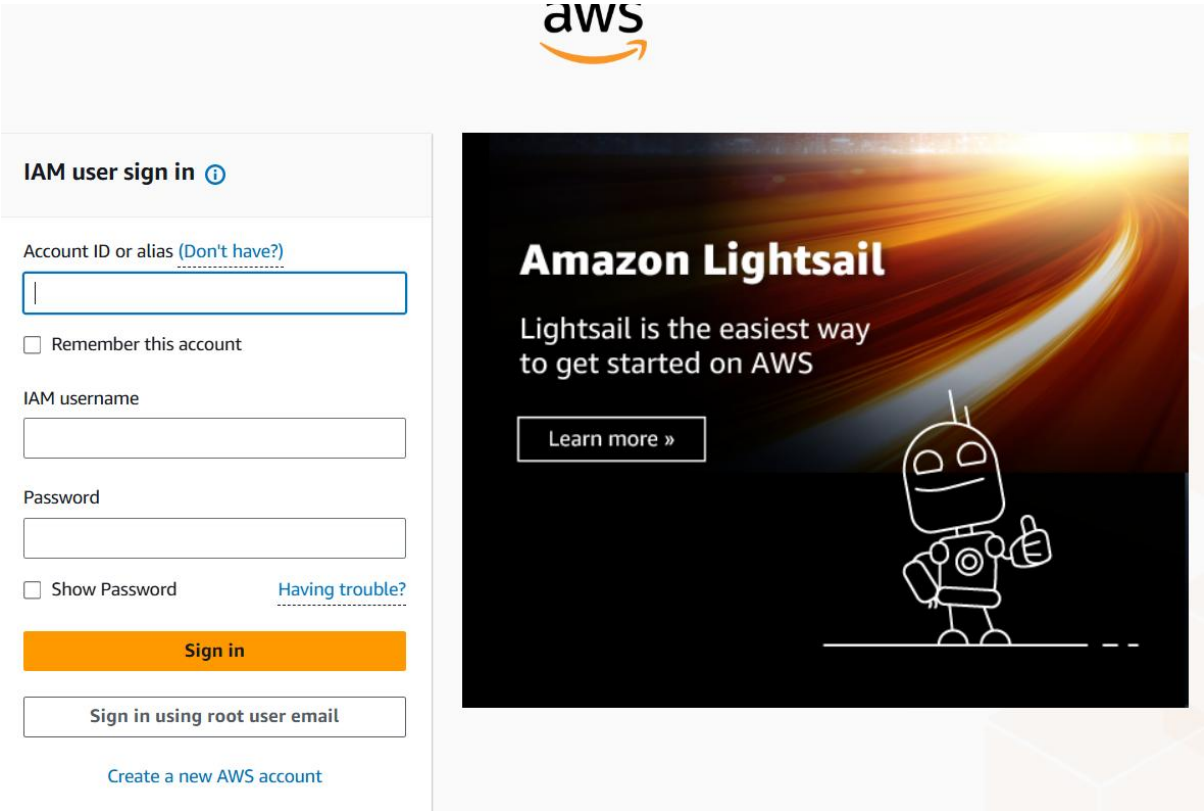
Activity 8.1: Conduct functional testing to verify user registration, login, book requests, and notifications.

Milestone 1: AWS Account Setup and Login

- **Activity 1.1: Set up an AWS account if not already done.**
  - Sign up for an AWS account and configure billing settings.

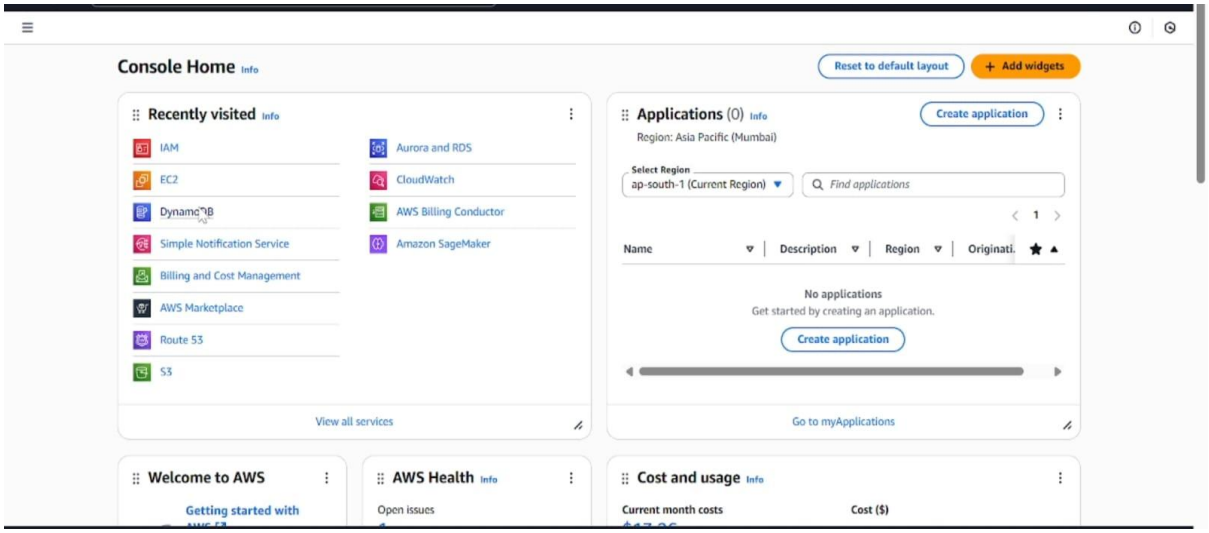


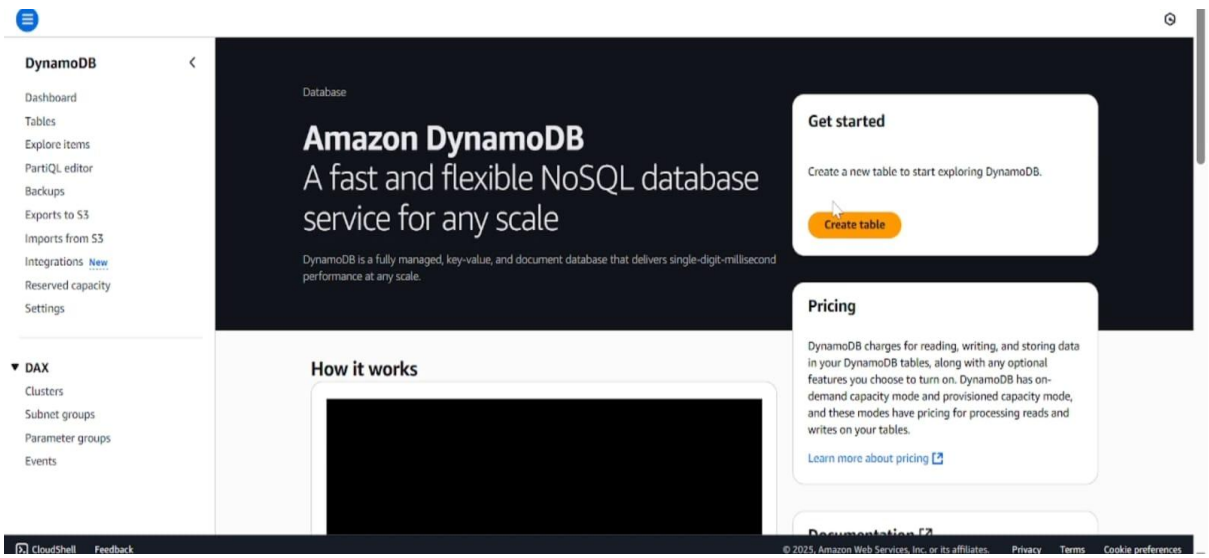
- **Activity 1.2: Log in to the AWS Management Console**
  - After setting up your account, log in to the [AWS Management Console]

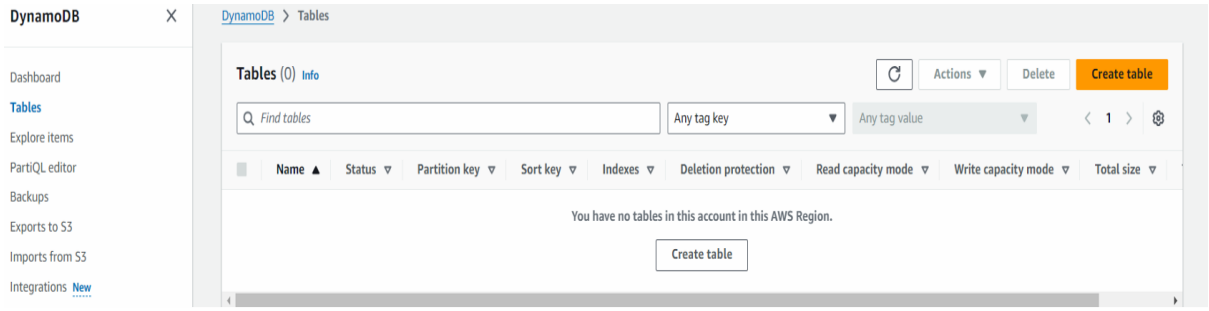


Milestone 2: DynamoDB Database Creation and Setup

- **Activity 2.1:Navigate to the DynamoDB**
  - In the AWS Console, navigate to DynamoDB and click on create tables

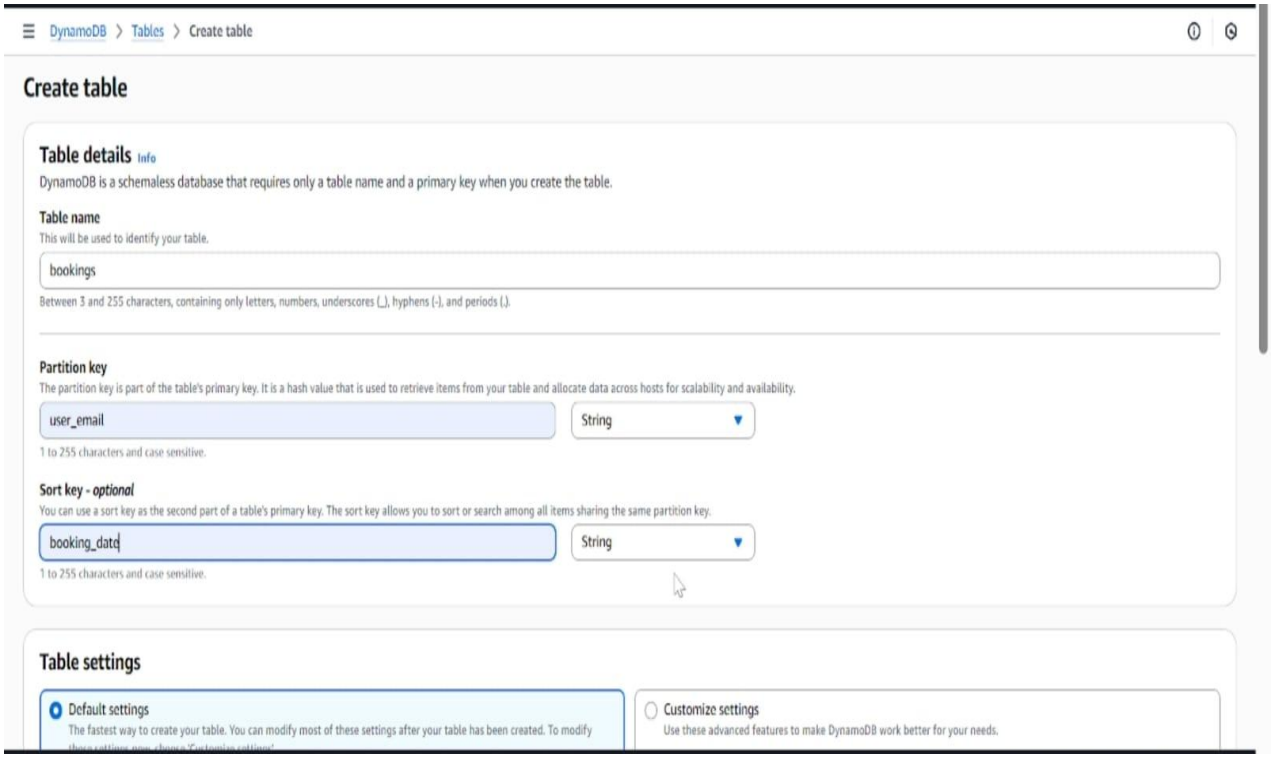


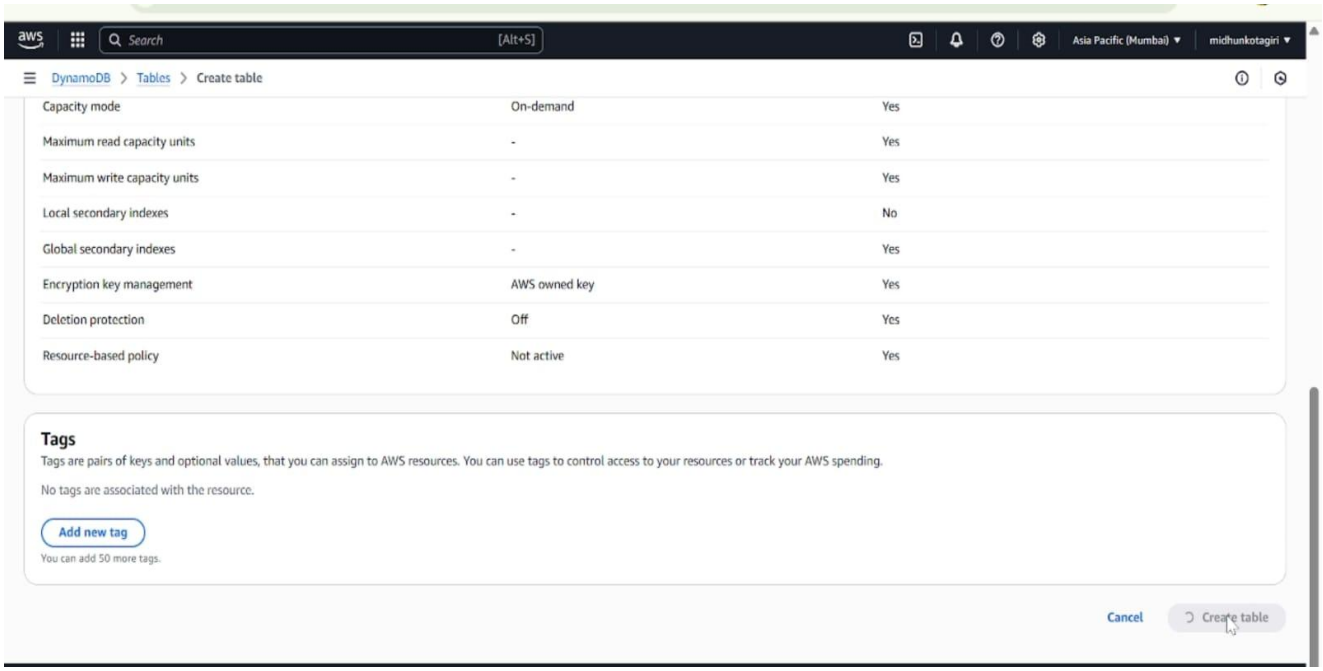




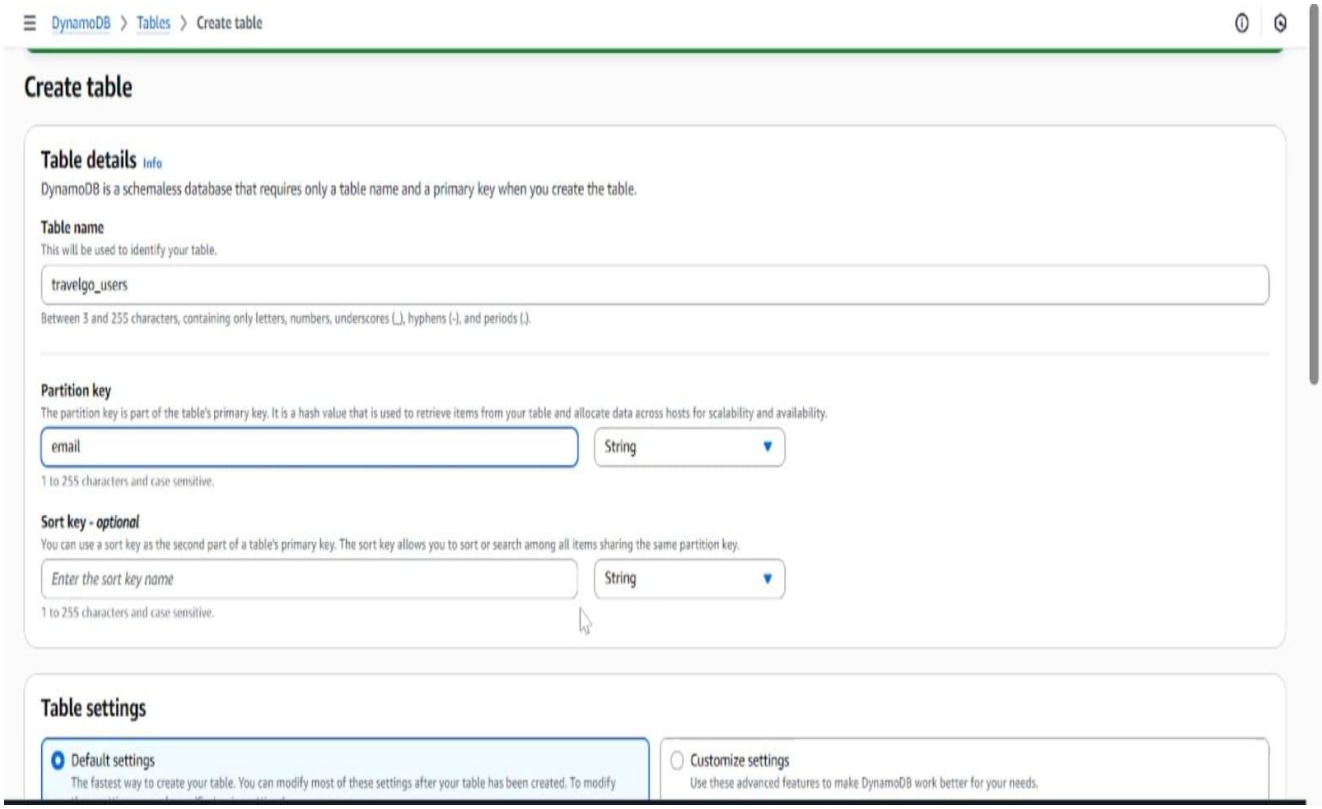
● Activity 2.2:Create a DynamoDB table for storing registration details and book requests.

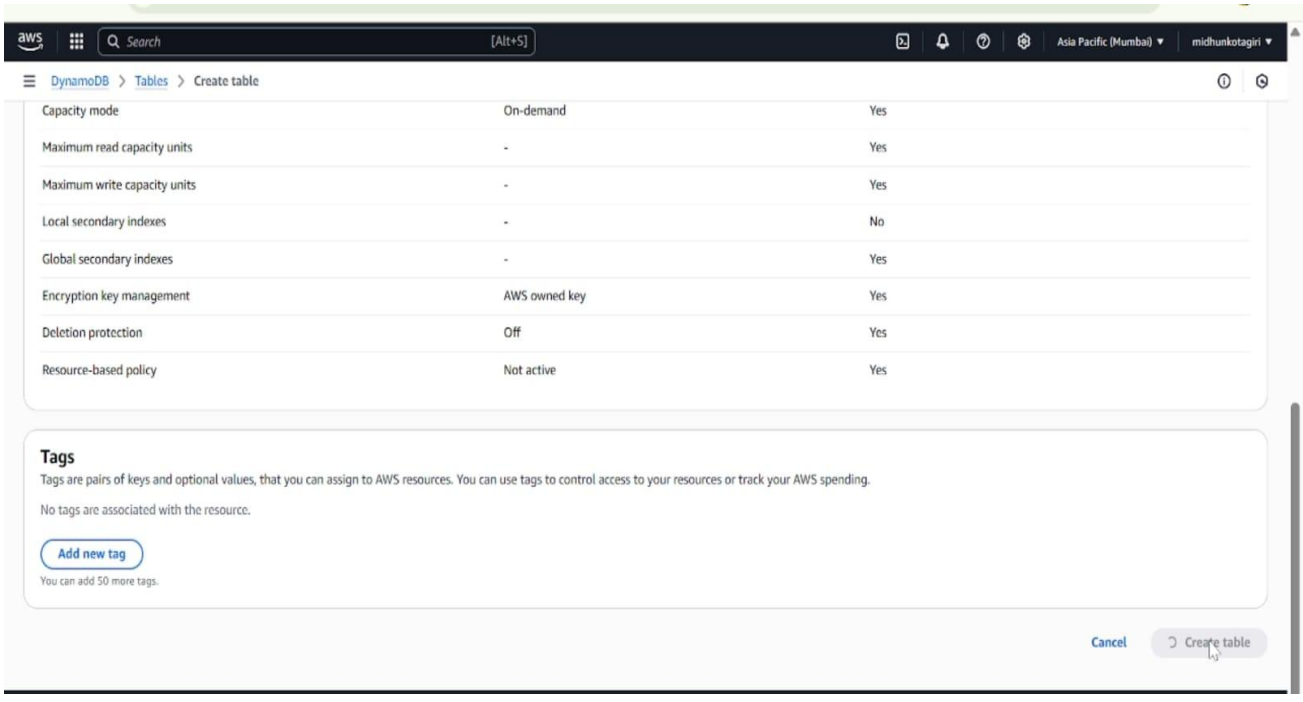
○ Create Users table with partition key “Email” with type String and click on create tables.





○ Follow the same steps to create a requests table with Email as the primary key for book requests data.

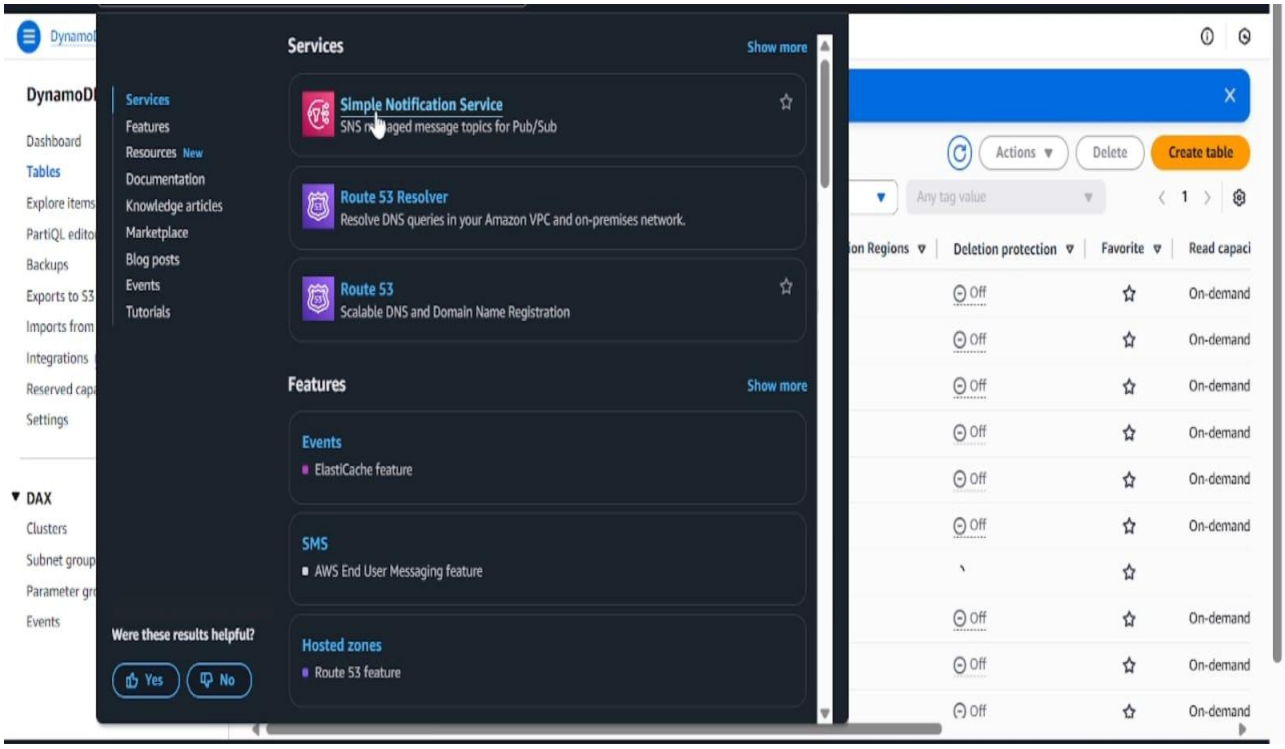


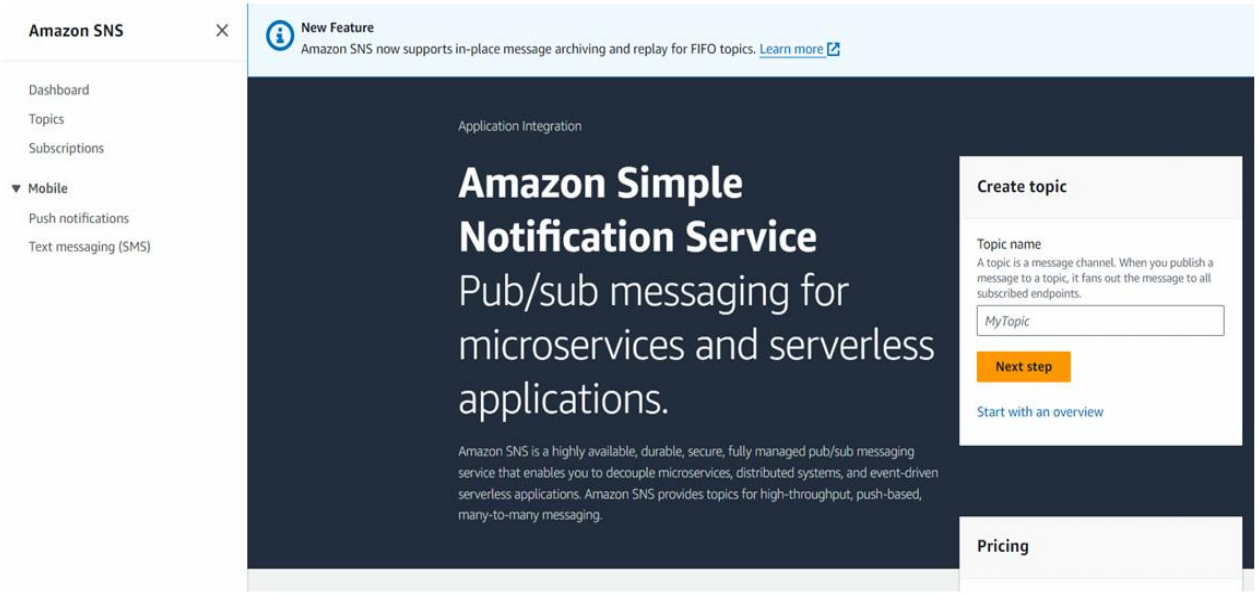


Milestone 3: SNS Notification Setup

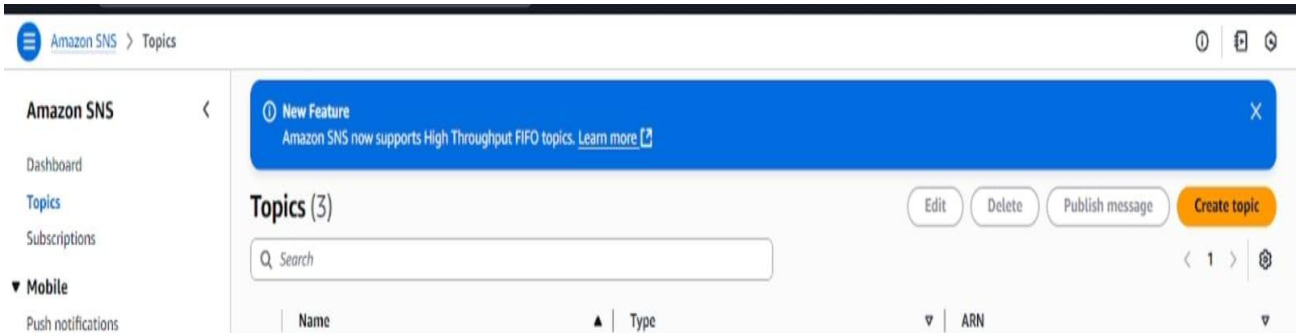
- **Activity 3.1: Create SNS topics for sending email notifications to users and library staff.**

○ In the AWS Console, search for SNS and navigate to the SNS Dashboard**.**

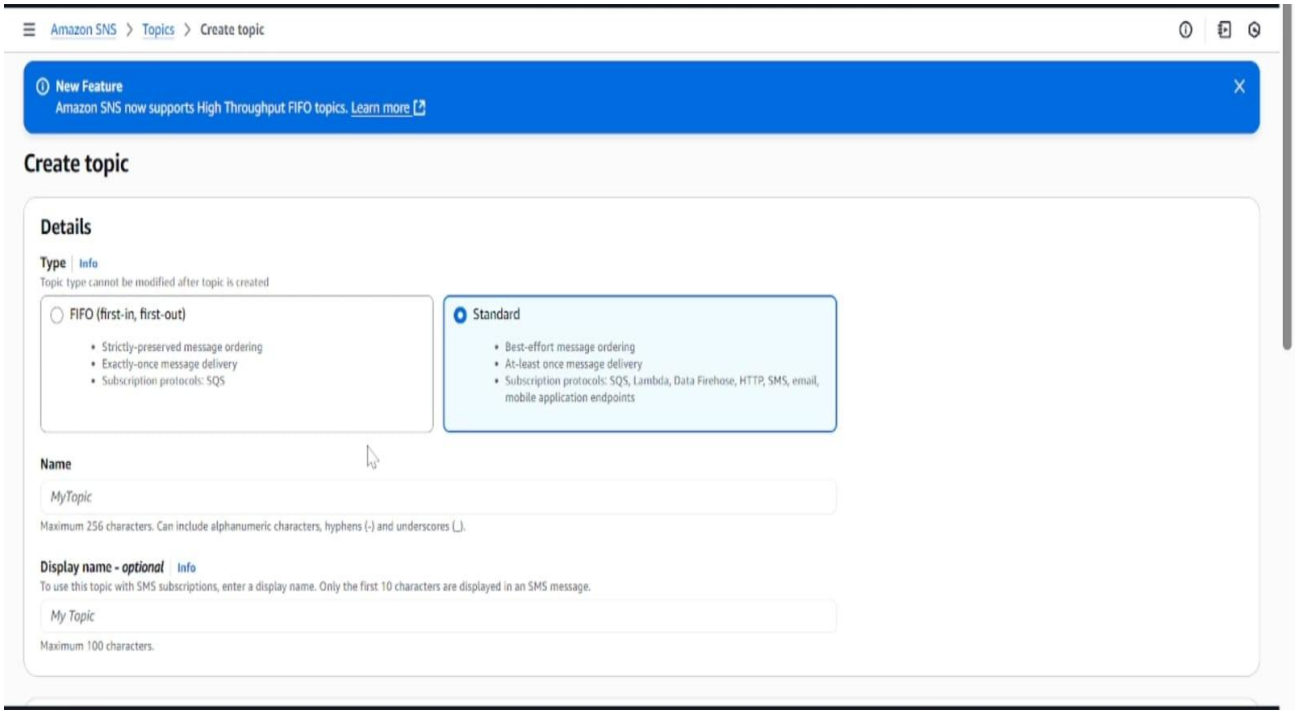


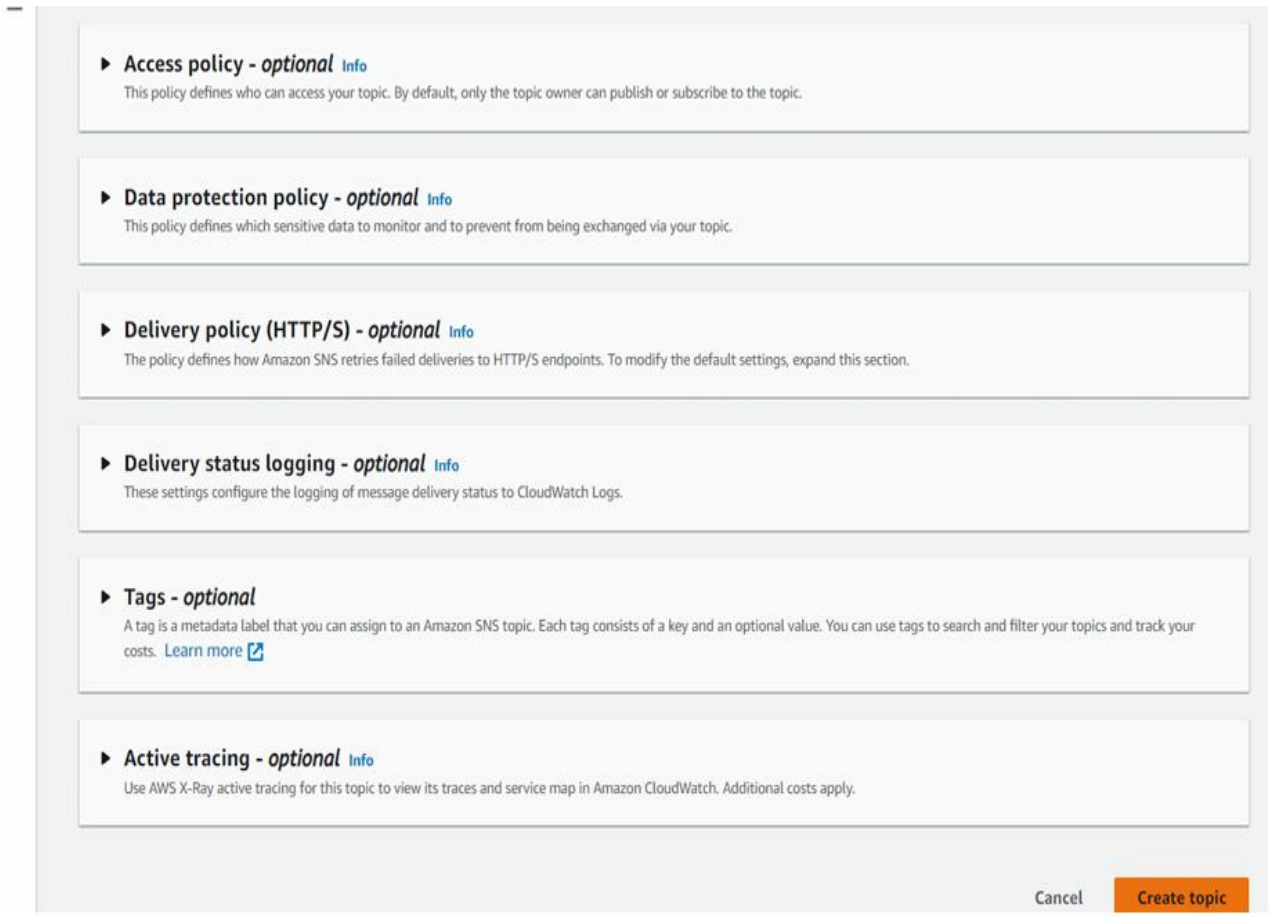


○ Click on **Create Topic** and choose a name for the topic**.**

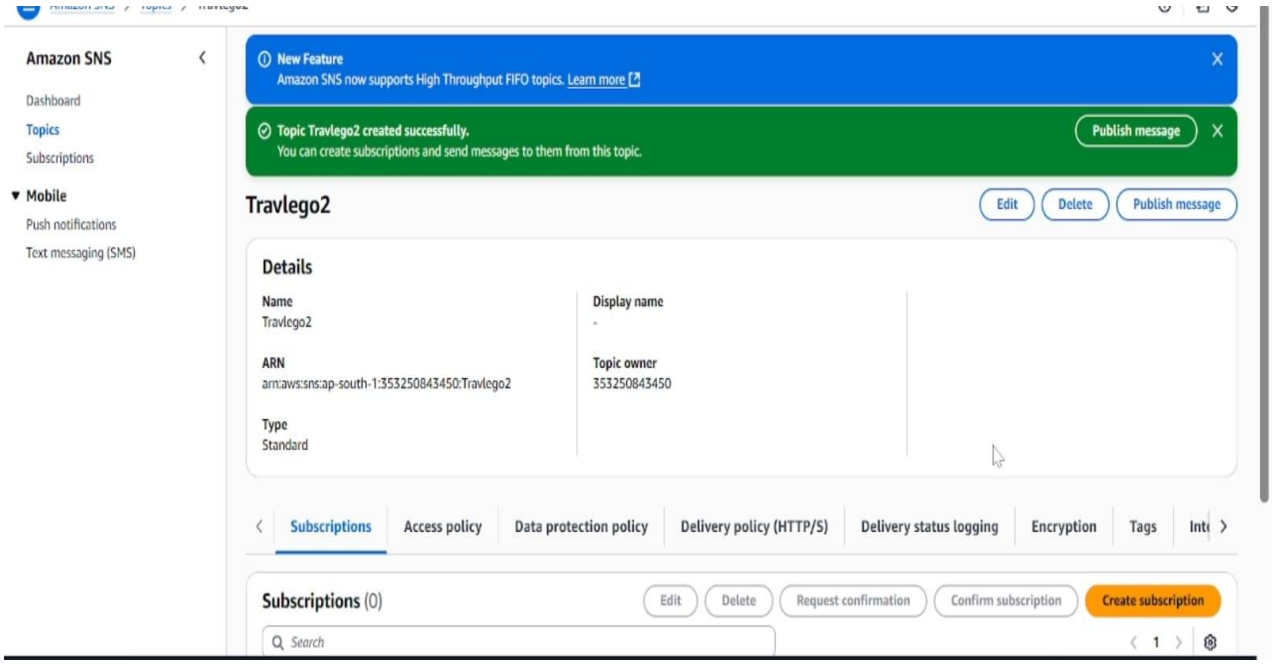


○ Choose **Standard type** for general notification use cases and Click on Create Topic.



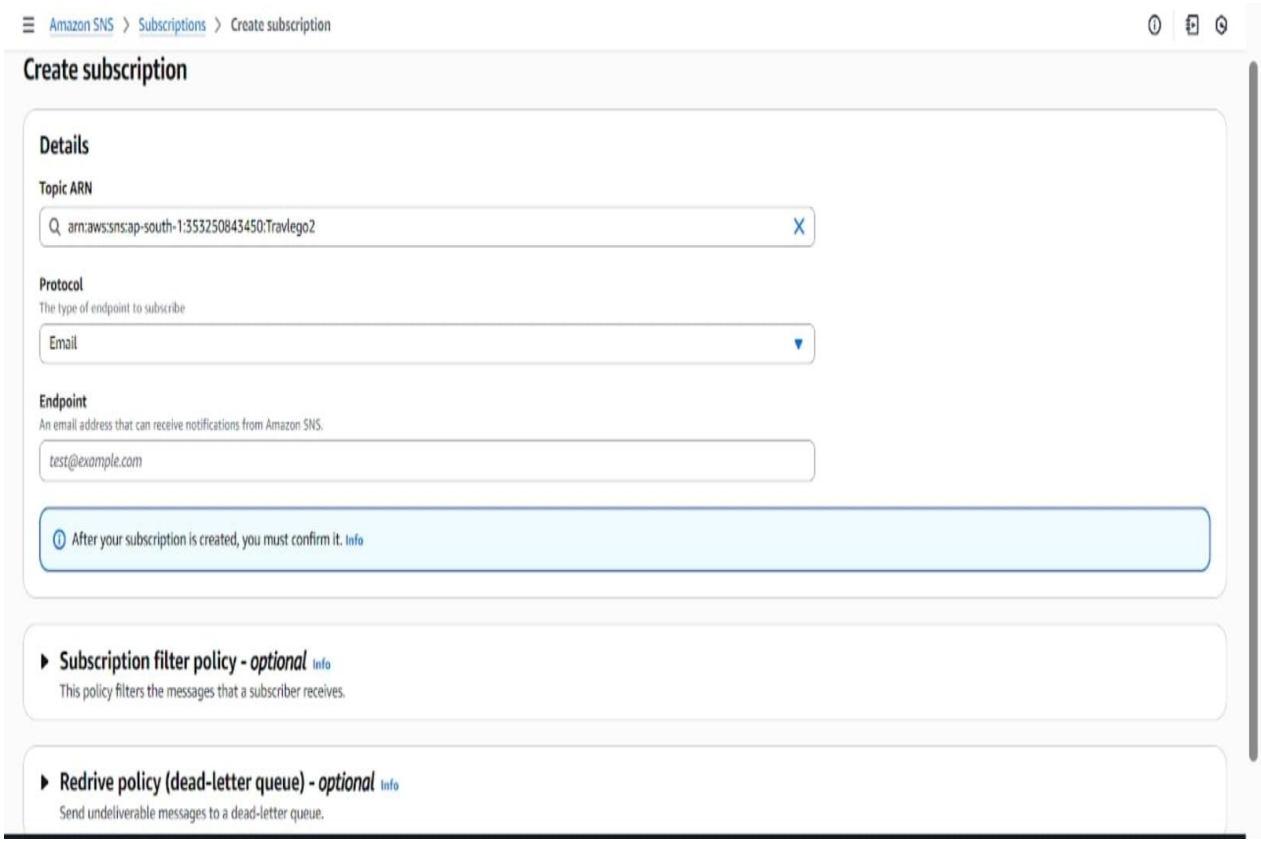


○ Configure the **SNS topic** and note down the Topic ARN



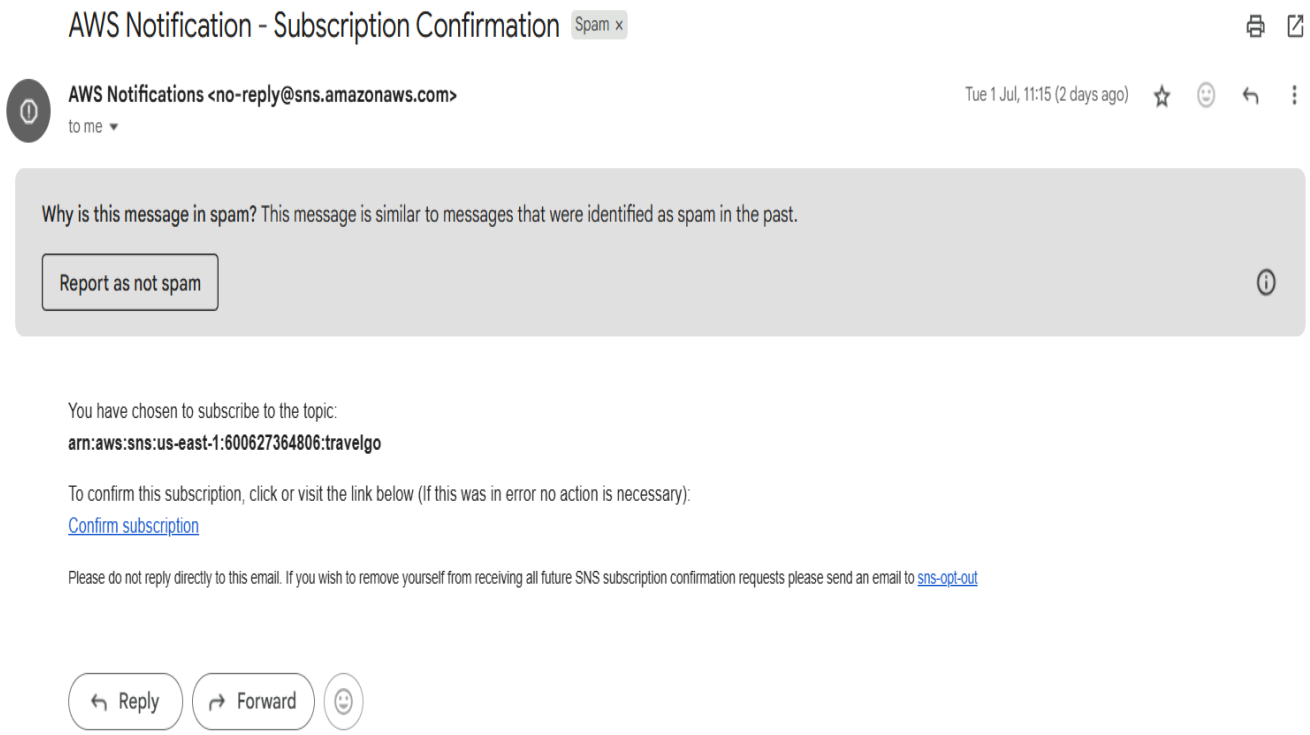
● Activity 3.2: Subscribe users and staff to relevant SNS topics to receive real-time notifications when a book request is made.

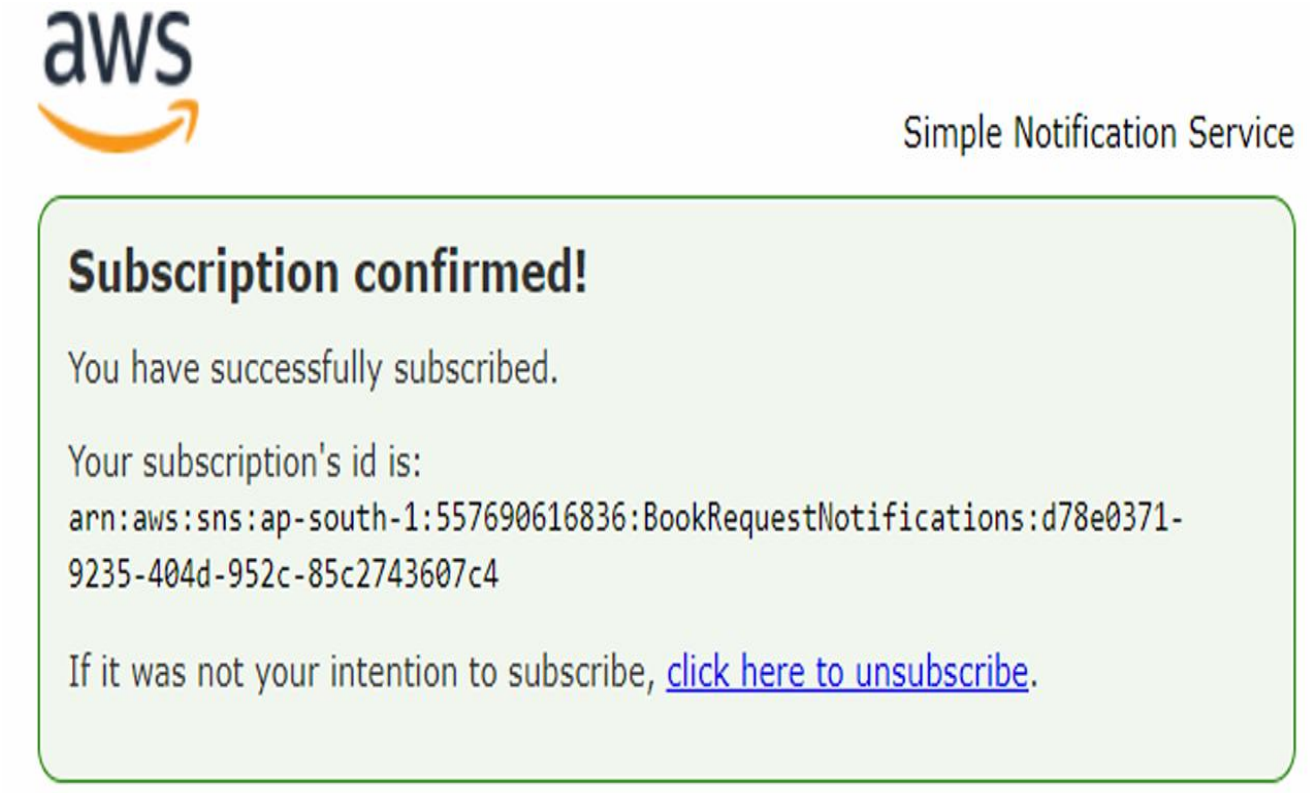
○ Subscribe users (or admin staff) to this topic via Email. When a book request is made, notifications will be sent to the subscribed emails.



○ Subscribe users (or admin staff) to this topic via Email. When a book request is made, notifications will be sent to the subscribed emails.

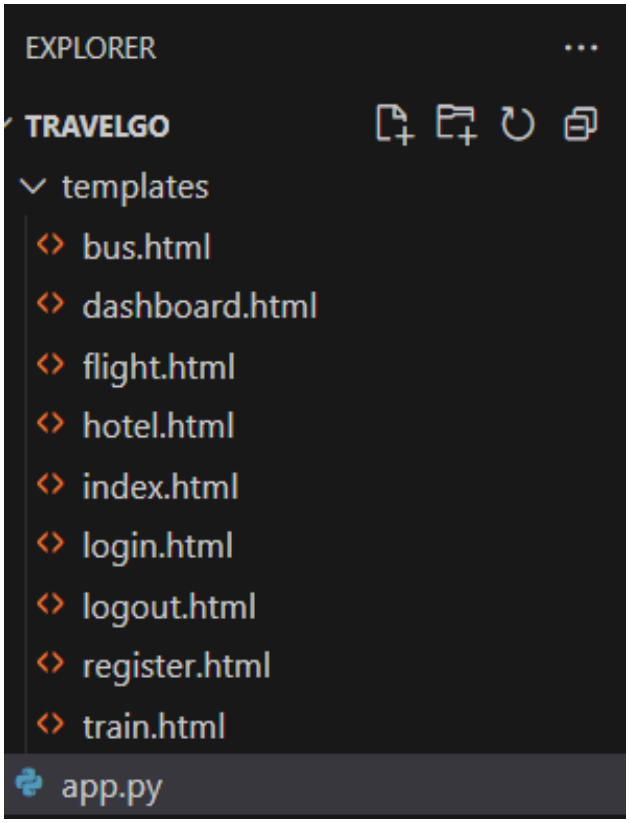
○ Navigate to the subscribed Email account and Click on the confirm subscription in the AWS Notification- Subscription Confirmation mail.





Milestone 4:Backend Development and Application Setup

- **Activity 4.1: Develop the backend using Flask**
  - File Explorer Structure



Description: set up the INSTANT LIBRARY project with an app.py file, a static/ folder for assets, and a templates/ directory containing all required HTML pages like home, login, register, subject-specific pages (e.g., computer_science.html, data_science.html), and utility pages (e.g., request-form.html, statistics.html).

Description of the code :

- **Flask App Initialization**

```
from flask import Flask, render_template, request, redirect, url_for, session, jsonify, flash
import boto3
from boto3.dynamodb.conditions import Key, Attr
from werkzeug.security import generate_password_hash, check_password_hash
from datetime import datetime
from decimal import Decimal
```

Description: import essential libraries including Flask utilities for routing, Boto3 for DynamoDB operations, SMTP and email modules for sending mails, and Bcrypt for password hashing and verification

```
app = Flask(__name__)
```

Description: initialize the Flask application instance using Flask(__name__) to start building the web app.

- **Dynamodb Setup:**

```
users_table = dynamodb.Table('travelgo_users')
trains_table = dynamodb.Table('trains') # Note: This table is declared but not used in the provided routes.
bookings_table = dynamodb.Table('bookings')
```

Description: initialize the DynamoDB resource for the ap-south-1 region and set up access to the Users and Requests tables for storing user details and book requests.

- **SNS Connection:**

```
SNS_TOPIC_ARN = 'arn:aws:sns:ap-south-1:353250843450:TravelGoBookingTopic'  # Replace with actu

# Function to send SNS notifications
# This function is duplicated in the original code, removing the duplicate.
def send_sns_notification(subject, message):
    try:
        sns_client.publish(
            TopicArn=SNS_TOPIC_ARN,
            Subject=subject,
            Message=message
        )
    except Exception as e:
        print(f"SNS Error: Could not send notification - {e}")
        # Optionally, flash an error message to the user or log it more robustly.
```

Description: Configure **SNS** to send notifications when a book request is submitted. Paste your stored ARN link in the sns_topic_arn space, along with the region_name where the SNS topic is created. Also, specify the chosen email service in SMTP_SERVER (e.g., Gmail, Yahoo, etc.) and enter the subscribed email in the SENDER_EMAIL section. Create an 'App password' for the email ID and store it in the SENDER_PASSWORD section.

● Routes for Web Pages

● Home Route:

```
# Routes
@app.route('/')
def home():
    return render_template('index.html', logged_in='email' in session)
```

Description: Define the home route (/) of the Flask application. The home() function renders the index.html template. It also checks if the user is logged in by verifying if 'email' exists in the session. If so, it passes logged_in=True to the template, allowing the frontend to dynamically adjust the UI (e.g., show or hide login/register buttons).

● Register Route:

```
@app.route('/register', methods=['GET', 'POST'])
def register():
    if request.method == 'POST':
        email = request.form['email']
        password = request.form['password']

        # Check if user already exists
        # This uses get_item on the primary key 'email', so no GSI needed.
        existing = users_table.get_item(Key={'email': email})
        if 'Item' in existing:
            flash('Email already exists!', 'error')
            return render_template('register.html')

        # Hash password and store user
        hashed_password = generate_password_hash(password)
        users_table.put_item(Item={'email': email, 'password': hashed_password})
        flash('Registration successful! Please log in.', 'success')
        return redirect(url_for('login'))
    return render_template('register.html')
```

Description: Handles the /register route for user signup using GET and POST methods. Checks if the user already exists in the travelgo_users table and prevents duplicates. If new, hashes the password, stores the user in DynamoDB, and redirects to the login page.

```python
@app.route('/login', methods=['GET', 'POST'])
def login():
    if request.method == 'POST':
        email = request.form['email']
        password = request.form['password']

        # Retrieve user by email (primary key)
        user = users_table.get_item(Key={'email': email})

        # Authenticate user
        if 'Item' in user and check_password_hash(user['Item']['password'], password):
            session['email'] = email
            flash('Logged in successfully!', 'success')
            return redirect(url_for('dashboard'))
        else:
            flash('Invalid email or password!', 'error')
            return render_template('login.html')
    return render_template('login.html')
```

Description: Handles the /login route for user authentication via GET and POST methods. On POST, it retrieves the user from travelgo_users by email and verifies the password using check_password_hash(). If valid, it stores user details in the session and updates the login count in DynamoDB. On failure, it flashes an error message and re-renders the login page.

- **Dashboard Route:**

```python
@app.route('/dashboard')
def dashboard():
    if 'email' not in session:
        return redirect(url_for('login'))
    user_email = session['email']

    # Query bookings for the logged-in user using the primary key 'user_email'
    # No GSI is needed here as 'user_email' is likely the partition key for the bookings_table.
    response = bookings_table.query(
        KeyConditionExpression=Key('user_email').eq(user_email),
        ScanIndexForward=False # Get most recent bookings first
    )
    bookings = response.get('Items', [])

    # Convert Decimal types from DynamoDB to float for display if necessary
    for booking in bookings:
        if 'total_price' in booking:
            try:
                booking['total_price'] = float(booking['total_price'])
            except (TypeError, ValueError):
                booking['total_price'] = 0.0 # Default value if conversion fails
    return render_template('dashboard.html', username=user_email, bookings=bookings)
```

Description: Defines the /dashboard route to display the logged-in user's booking history. Checks session for a valid login, then queries the bookings table using the user's email to fetch bookings in reverse chronological order. Converts any Decimal values from DynamoDB to float for proper display in the HTML template. Renders dashboard.html, passing the user's email and their bookings for display.

- **Booking Route:**

```python
# Booking Routes
@app.route('/bus')
def bus_booking():
    if 'email' not in session:
        flash('Please login to book tickets.', 'error')
        return redirect(url_for('login'))
    return render_template('bus.html')

@app.route('/train')
def train_booking():
    if 'email' not in session:
        flash('Please login to book tickets.', 'error')
        return redirect(url_for('login'))
    return render_template('train.html')

@app.route('/flight')
def flight_booking():
    if 'email' not in session:
        flash('Please login to book tickets.', 'error')
        return redirect(url_for('login'))
    return render_template('flight.html')

@app.route('/hotel')
def hotel_booking():
    if 'email' not in session:
        flash('Please login to book tickets.', 'error')
        return redirect(url_for('login'))
    return render_template('hotel.html')
```

Description: Defines individual booking routes for bus, train, flight, and hotel pages. Each route first checks if the user is logged in by verifying the session; if not, it redirects to the login page with an error message. If the user is authenticated, the corresponding booking template (bus.html, train.html, etc.) is rendered.

● Deployment Code:

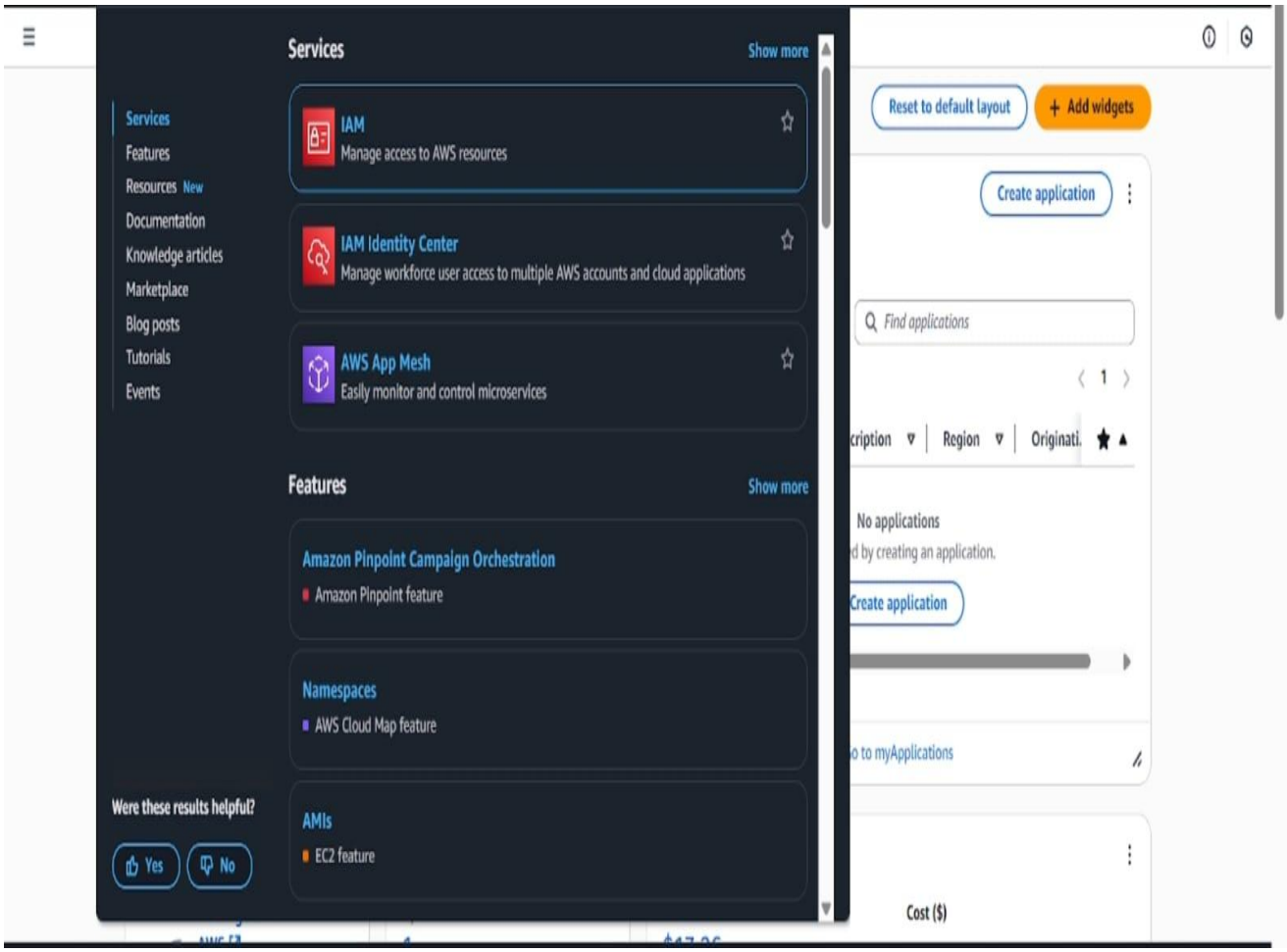
```
if __name__ == '__main__':
    # IMPORTANT: In a production environment, disable debug mode and specify a production-ready host.
    app.run(debug=True, host='0.0.0.0',port=5000)
```

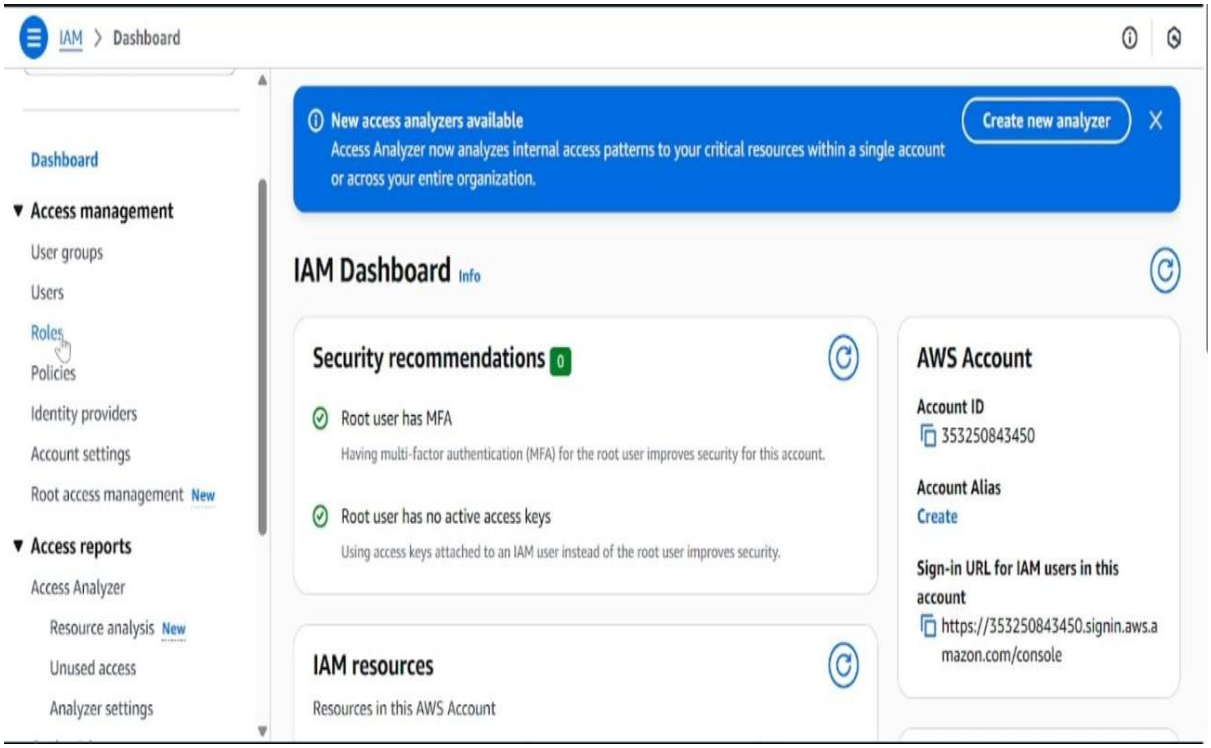
Description: Starts the Flask application when the script is run directly. Runs the app in debug mode for development, allowing live code reloading and error display. Binds the server to all IP addresses (0.0.0.0) on port 5000, making it accessible on the local network or EC2 instance.

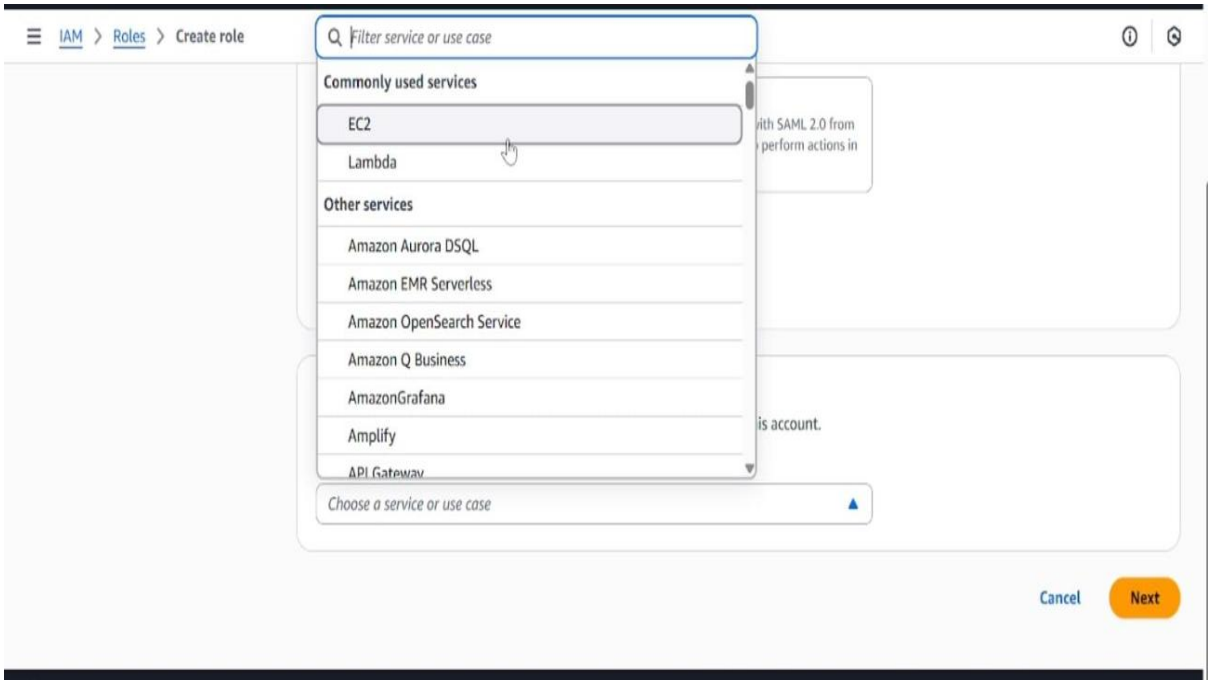
Milestone 5: IAM Role Setup

● Activity 5.1: Create IAM Role.

○ In the AWS Console, go to IAM and create a new IAM Role for EC2 to interact with DynamoDB and SNS.







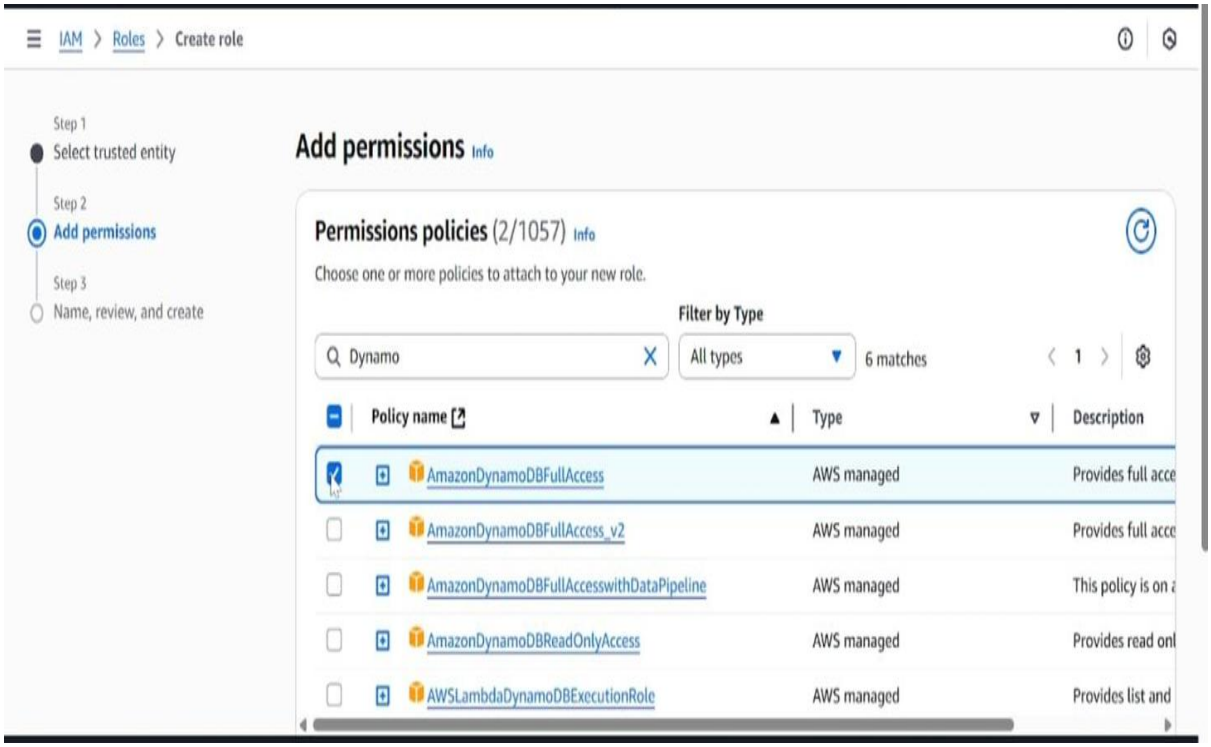
● Activity 5.2: Attach Policies.

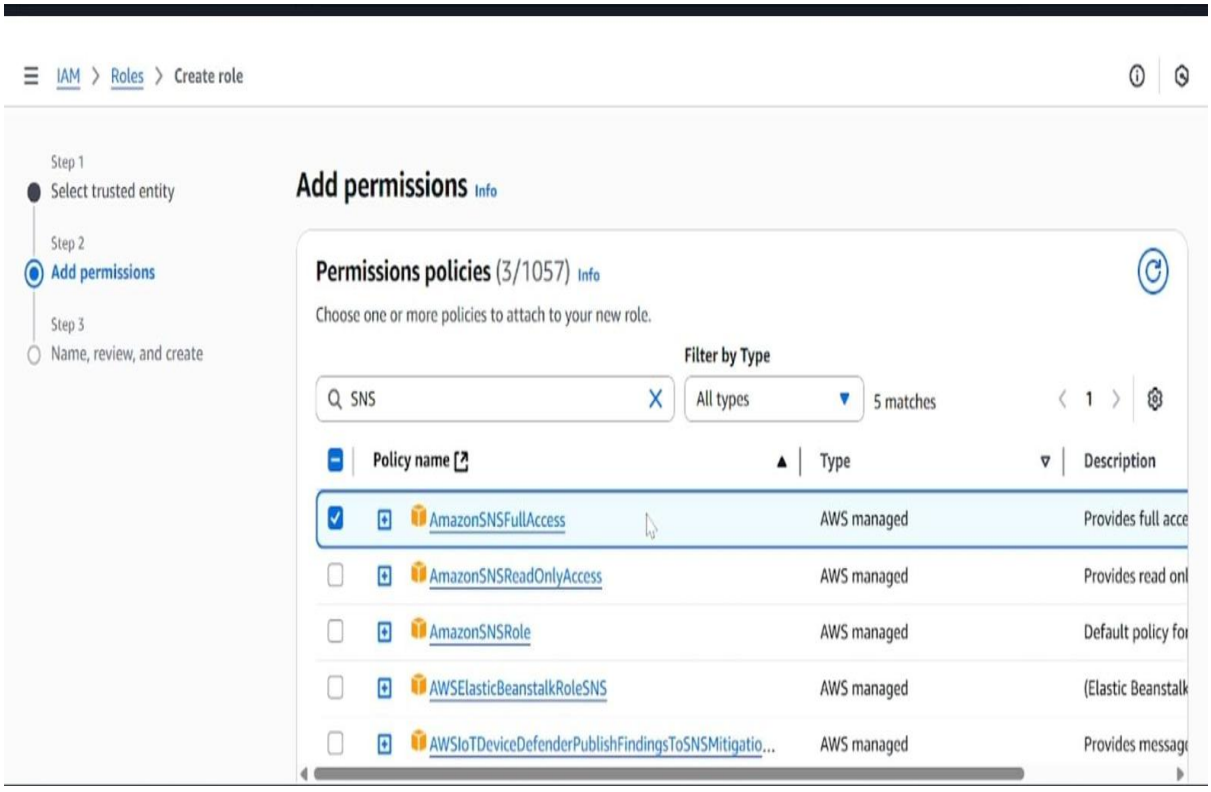
Attach the following policies to the role:

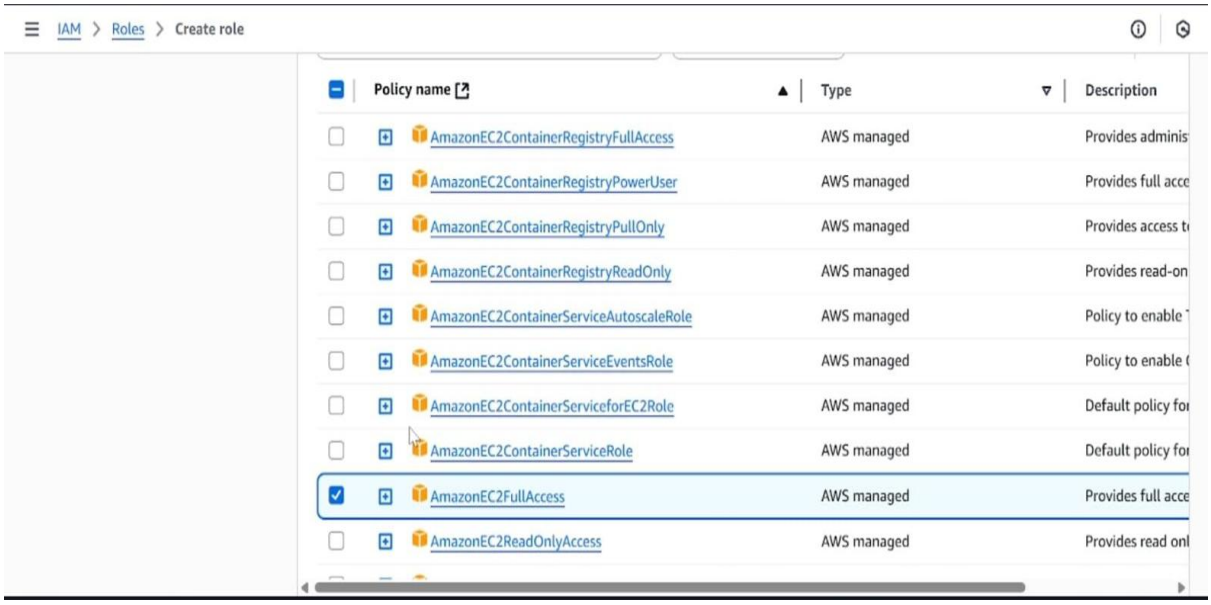
● **AmazonDynamoDBFullAccess: Allows EC2 to perform read/write operations on DynamoDB.**

● **AmazonSNSFullAccess:** Grants EC2 the ability to send notifications via SNS.

● **AmazonEC2FullAccess:** Provides full permissions to manage all EC2 resources, including instances, volumes, and networking.

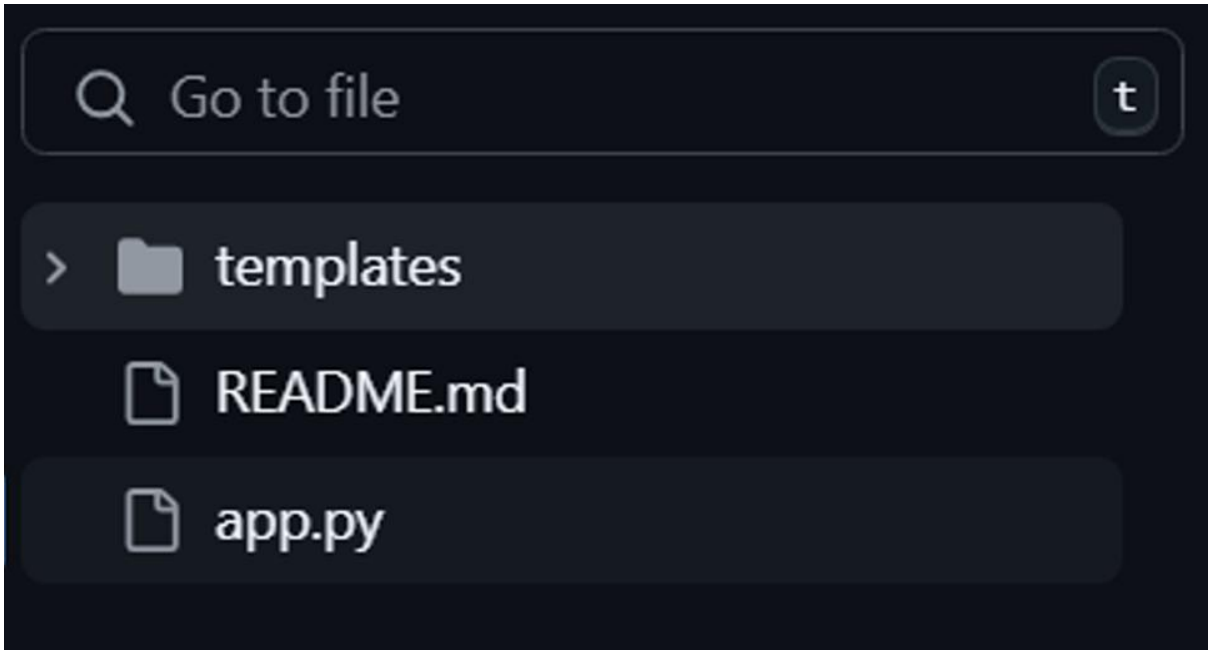


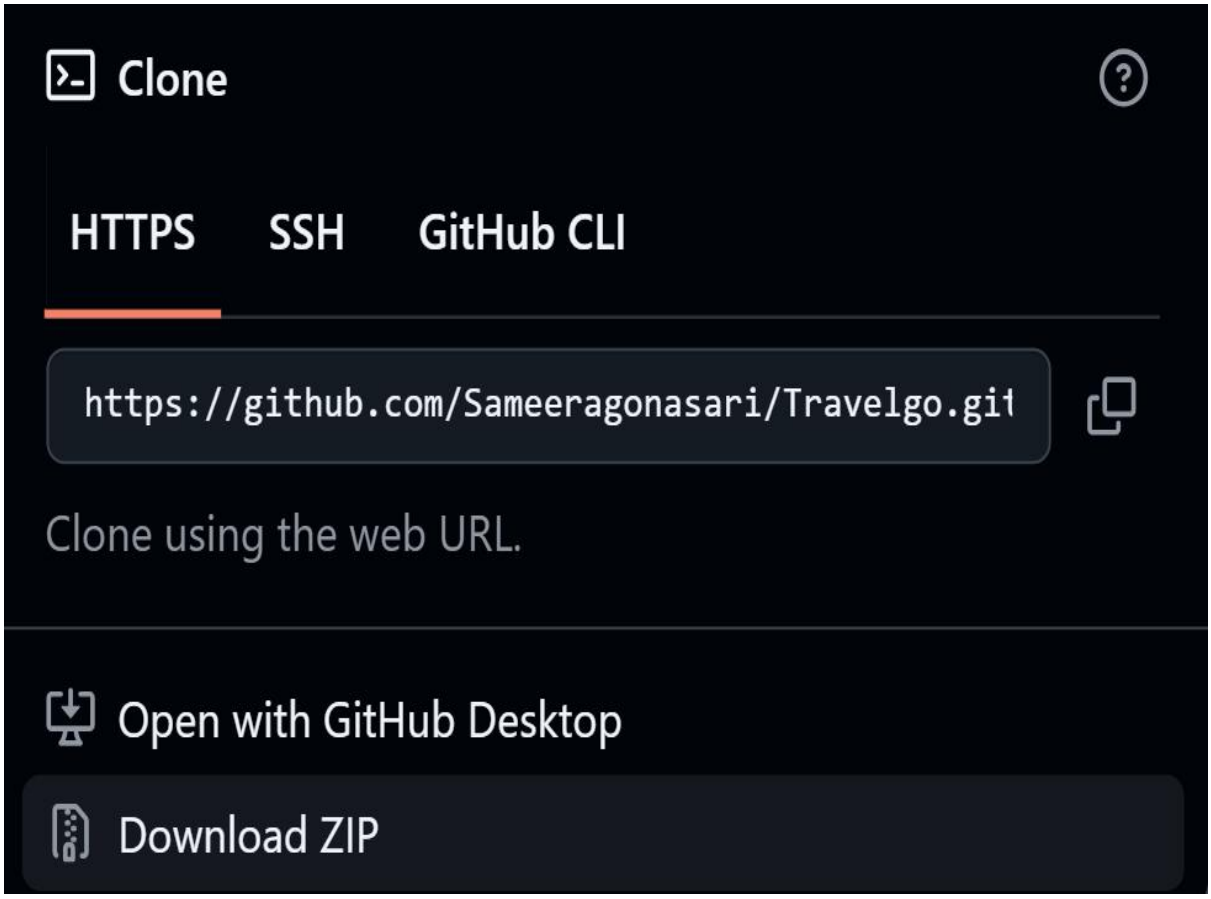




Milestone 6: EC2 Instance Setup

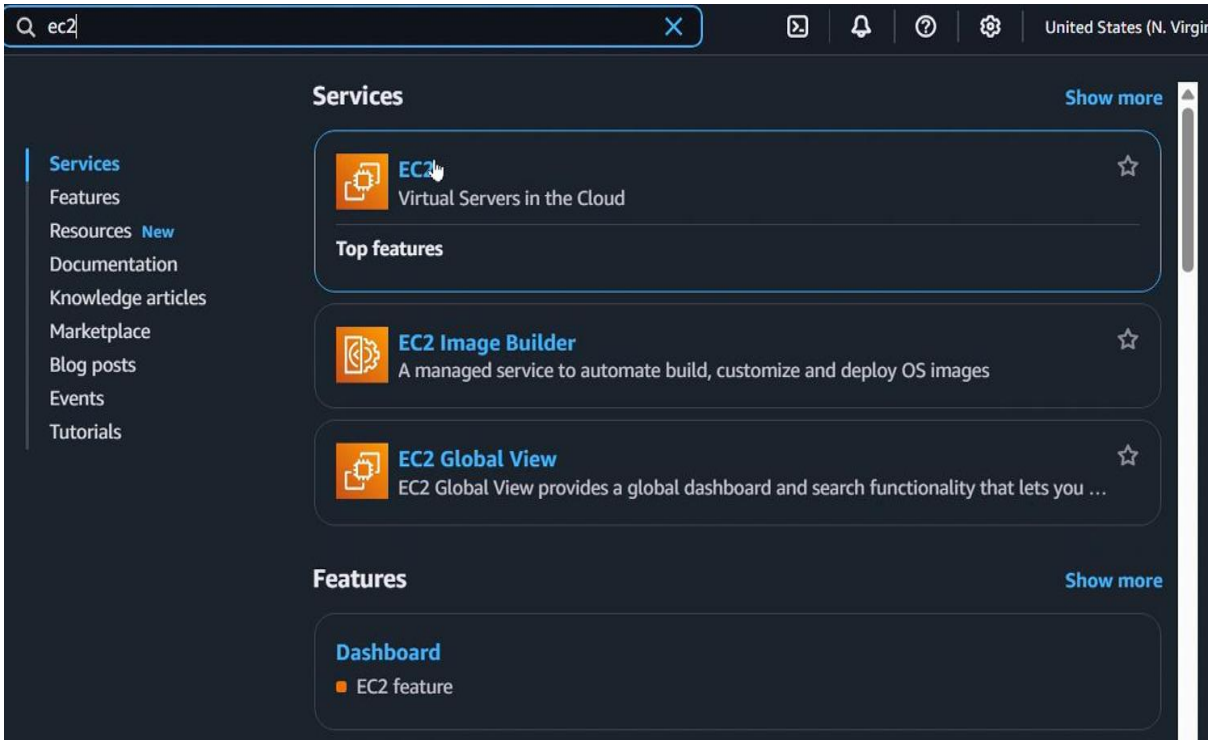
- **Note:** Load your Flask app and Html files into GitHub repository.



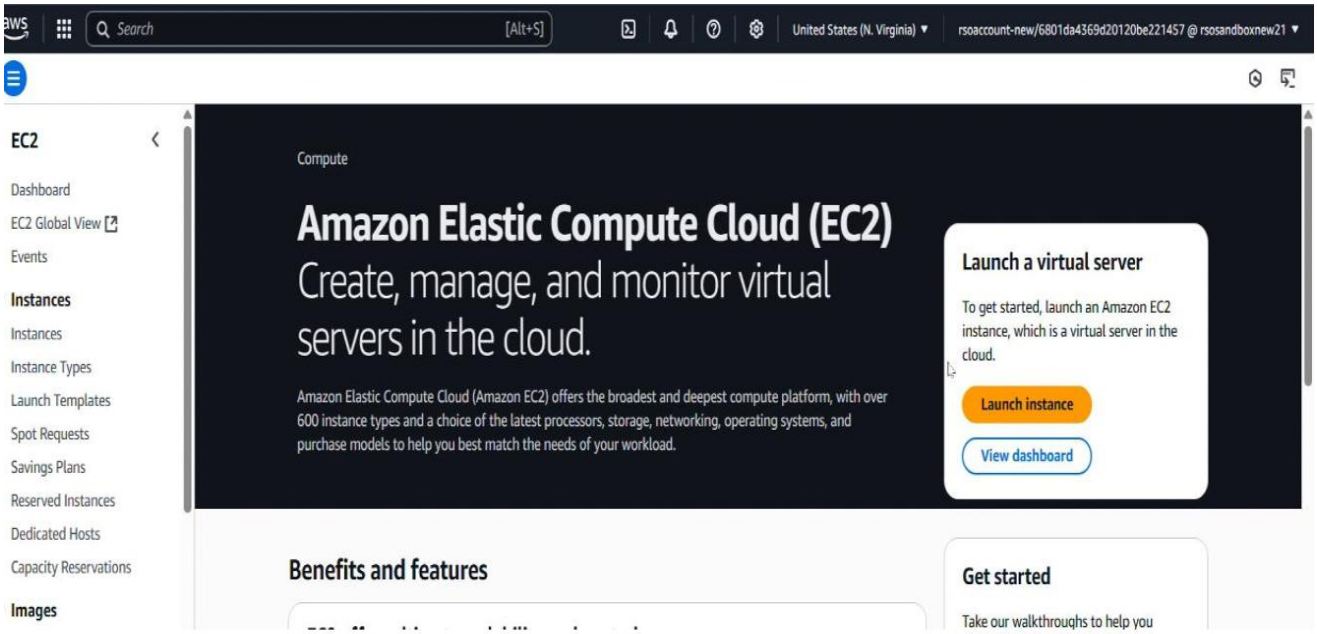


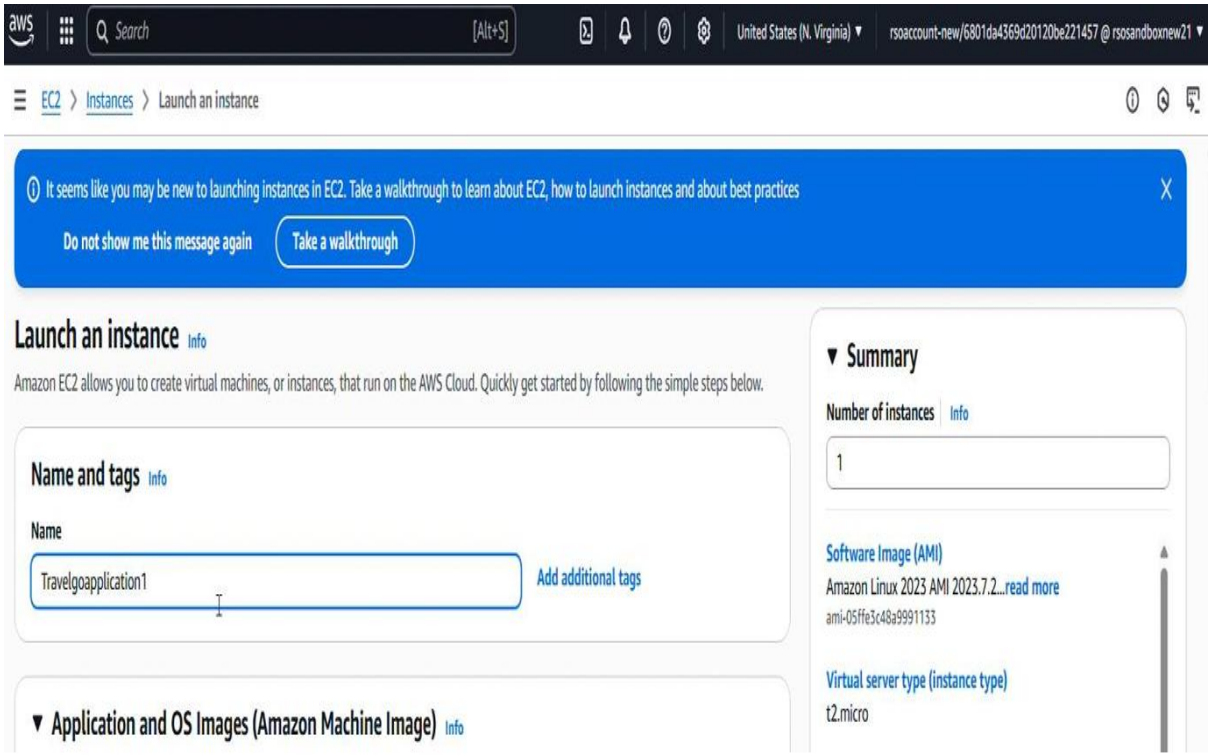
● Activity 6.1: Launch an EC2 instance to host the Flask application.

● Launch EC2 Instance ○ In the AWS Console, navigate to EC2 and launch a new instance.

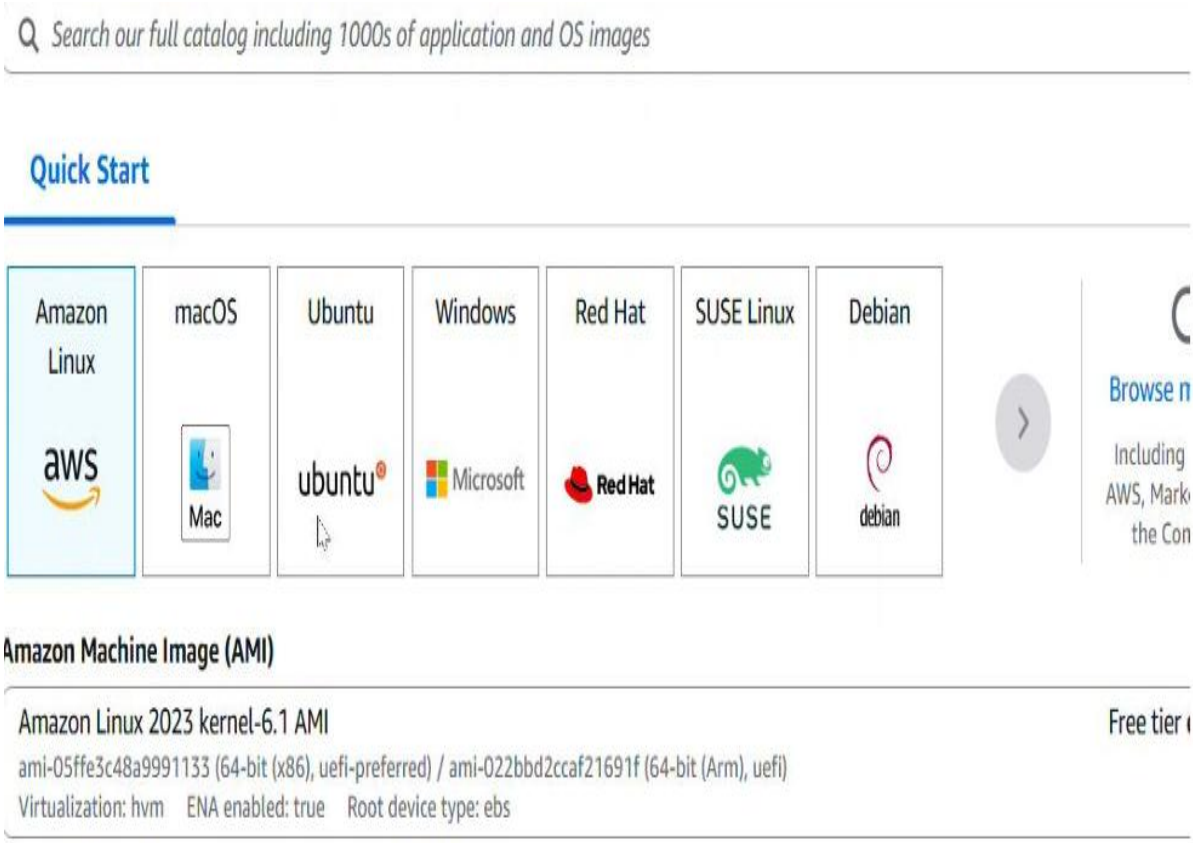


● Click on **Launch instance** to launch EC2 instance.





● Choose Amazon Linux 2 or Ubuntu as the AMI and t2.micro as the instance type (free-tier eligible).



- Create and download the key pair for Server access.

Create key pair

Key pair name

Key pairs allow you to connect to your instance securely.

travel-go-application-1

The name can include up to 255 ASCII characters. It can't include leading or trailing spaces.

Key pair type

- RSA
  RSA encrypted private and public key pair
- ED25519
  ED25519 encrypted private and public key pair

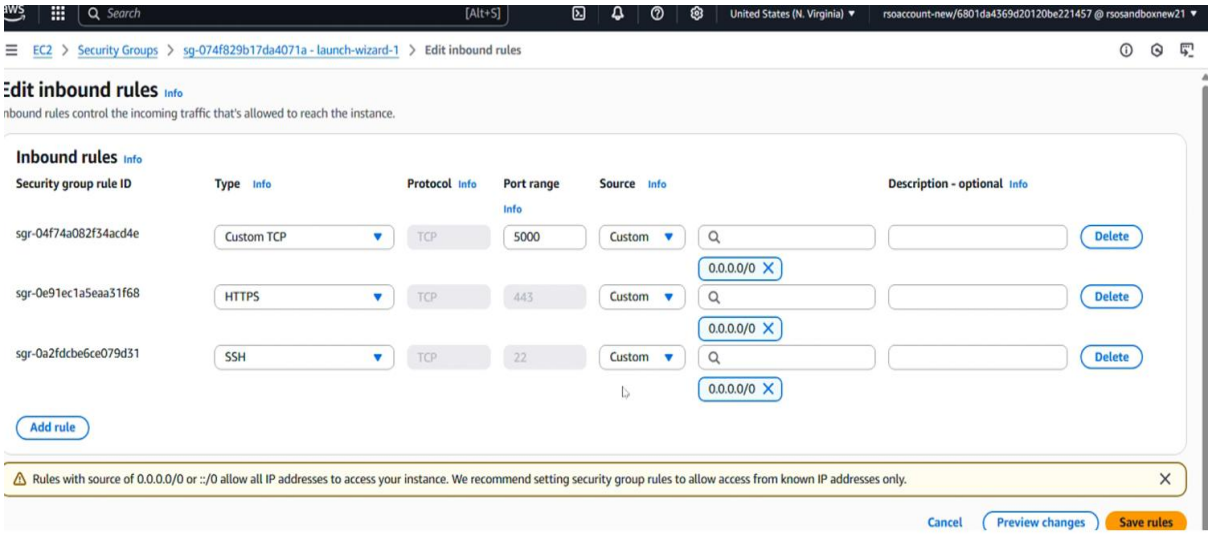
Private key file format

- .pem
  For use with OpenSSH
- .ppk
  For use with PuTTY

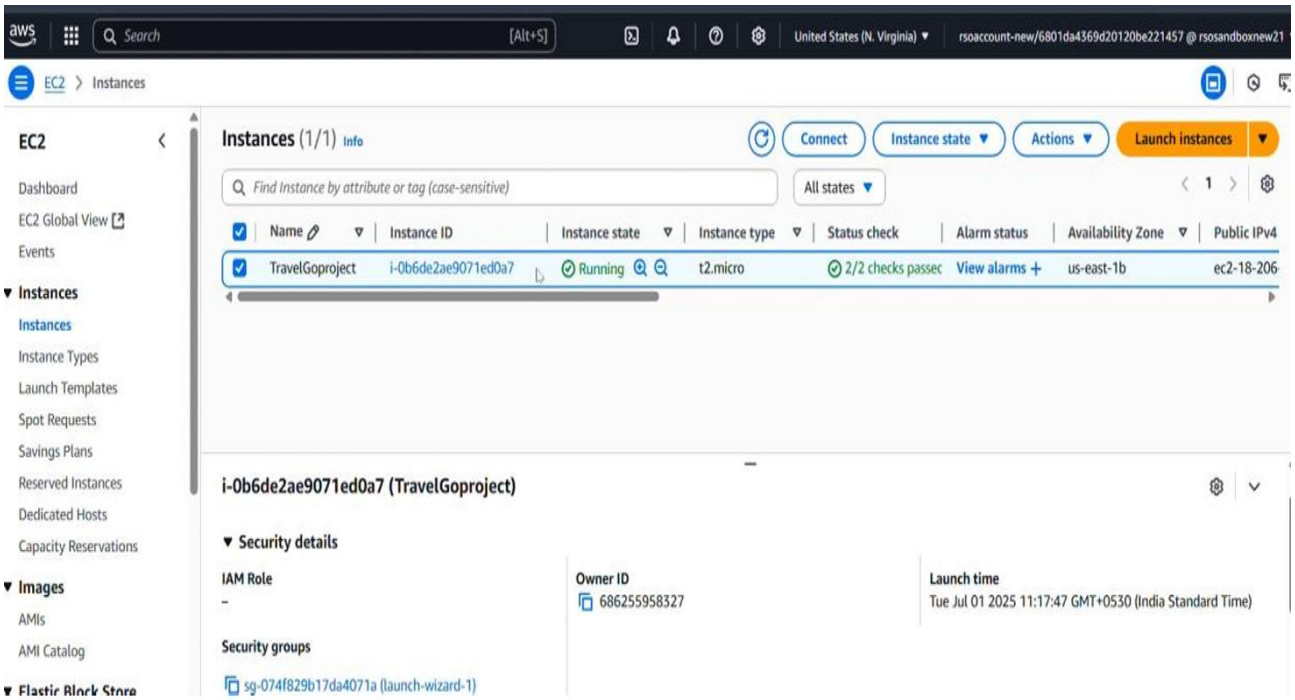
When prompted, store the private key in a secure and accessible location on your computer. **You will need it later to connect to your instance.** Learn more

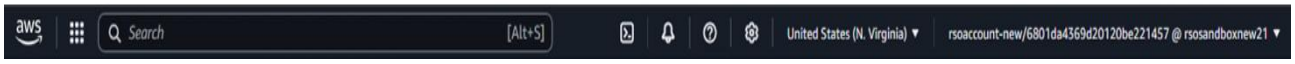
- **Activity 6.2:Configure security groups for HTTP, and SSH access.**

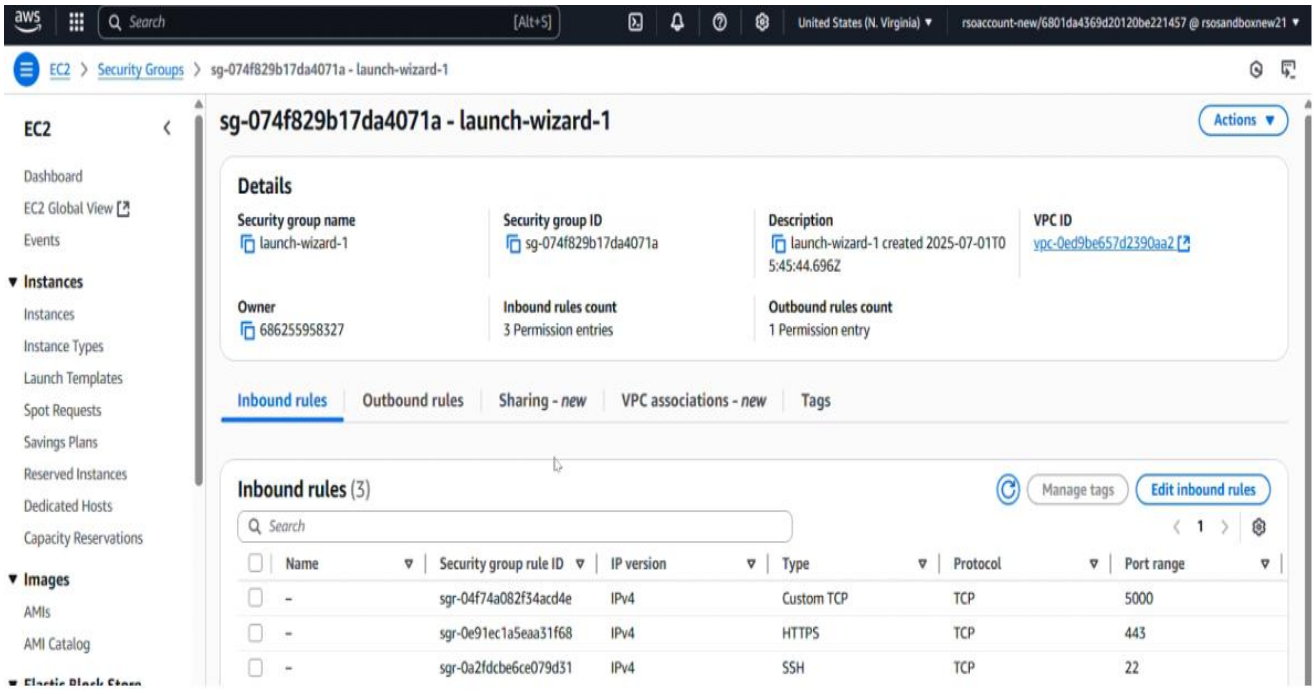
Edit inbound rules:

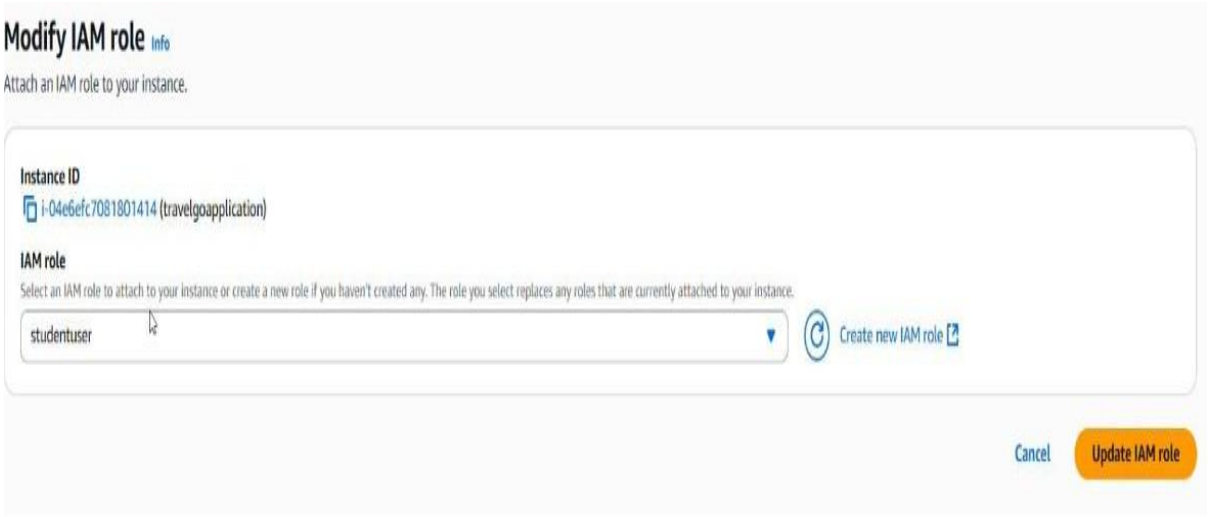


● To connect to EC2 using EC2 Instance Connect, start by ensuring that an IAM role is attached to your EC2 instance. You can do this by selecting your instance, clicking on Actions, then navigating to Security and selecting Modify IAM Role to attach the appropriate role. After the IAM role is connected, navigate to the EC2 section in the AWS Management Console. Select the EC2 instance you wish to connect to. At the top of the EC2 Dashboard, click the Connect button. From the connection methods presented, choose EC2 Instance Connect. Finally, click Connect again, and a new browser-based terminal will open, allowing you to access your EC2 instance directly from your browser.

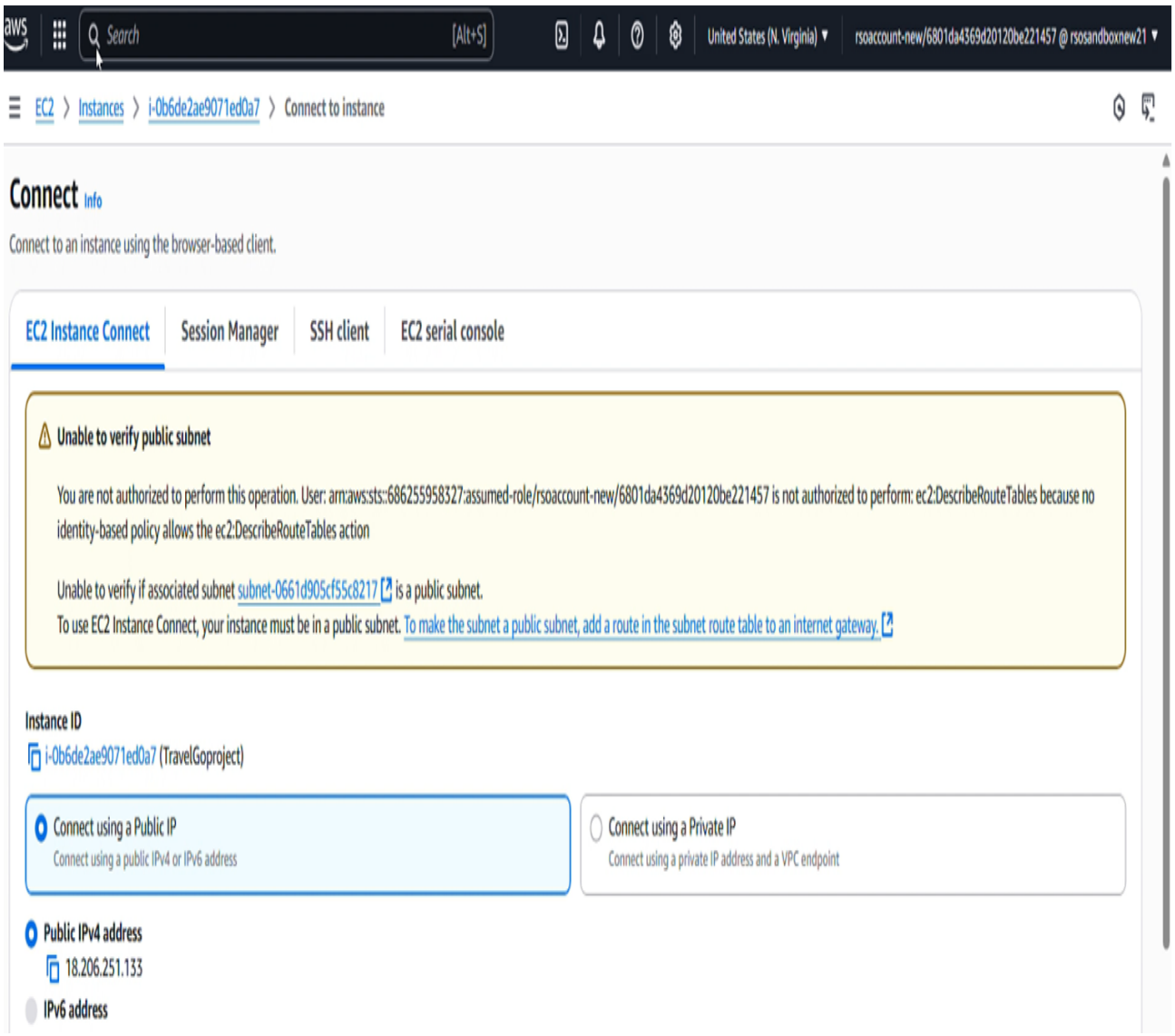








- Now connect the EC2 with the files



Milestone 7: Deployment on EC2

Activity 7.1: Install Software on the EC2 Instance Install Python3, Flask, and Git:

On Amazon Linux 2:

Sudo yum install git -y

sudo yum install python3 -y

sudo yum install python3-pip -y

Pip install flask

Pip install boto3

Activity 7.2: Clone Your Flask Project from GitHub

Clone your project repository from GitHub into the EC2 instance using Git.

Run: 'git clone https://github.com/your-github-username/your-repository-name.git**'**

Note: change your-github-username and your-repository-name with your credentials here: 'git clone https://github.com/AlekhyaPenubakula/InstantLibrary.git'

- This will download your project to the EC2 instance.

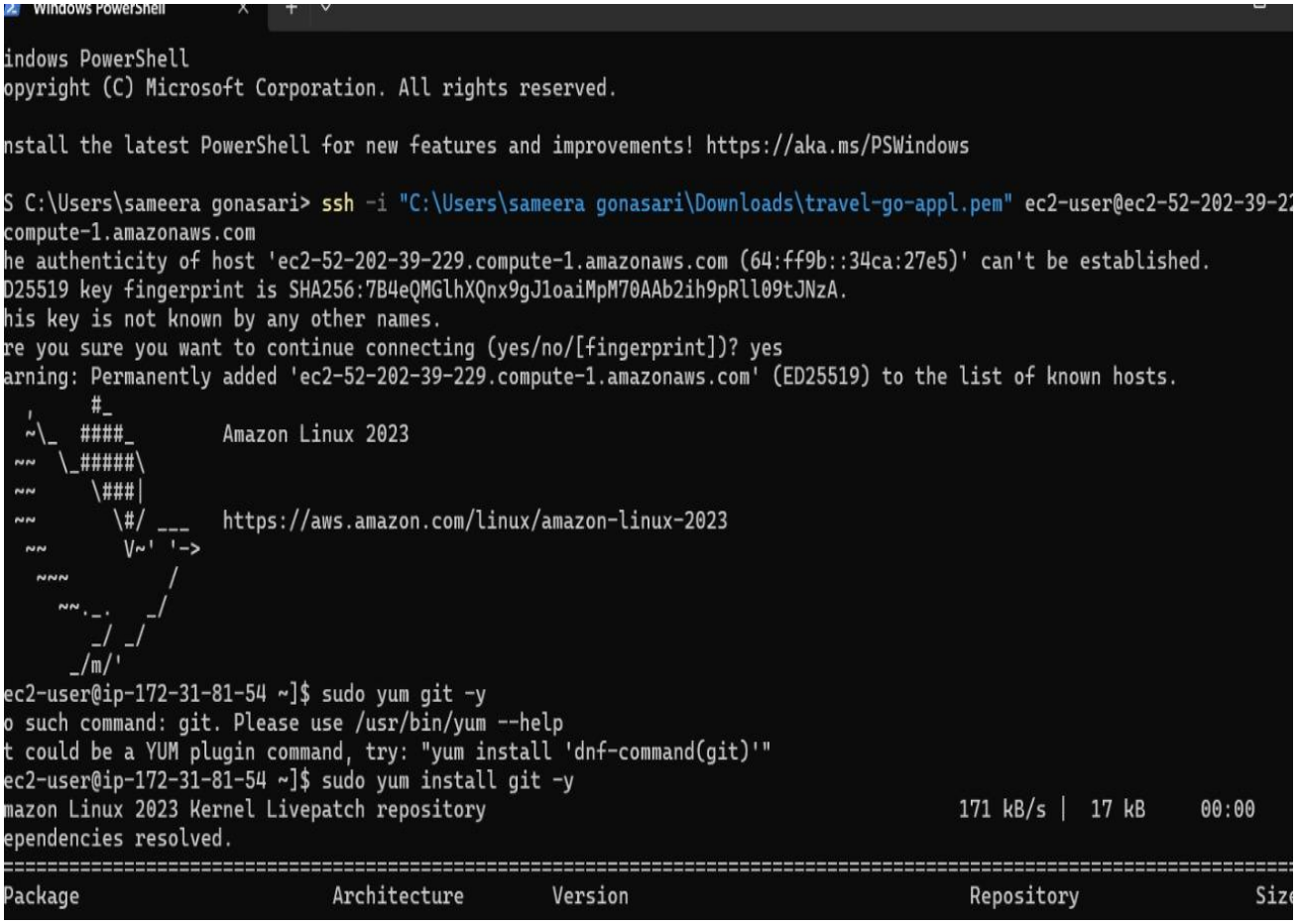
To navigate to the project directory, run the following command:

cd InstantLibrary

Once inside the project directory, configure and run the Flask application by executing the following command with elevated privileges:

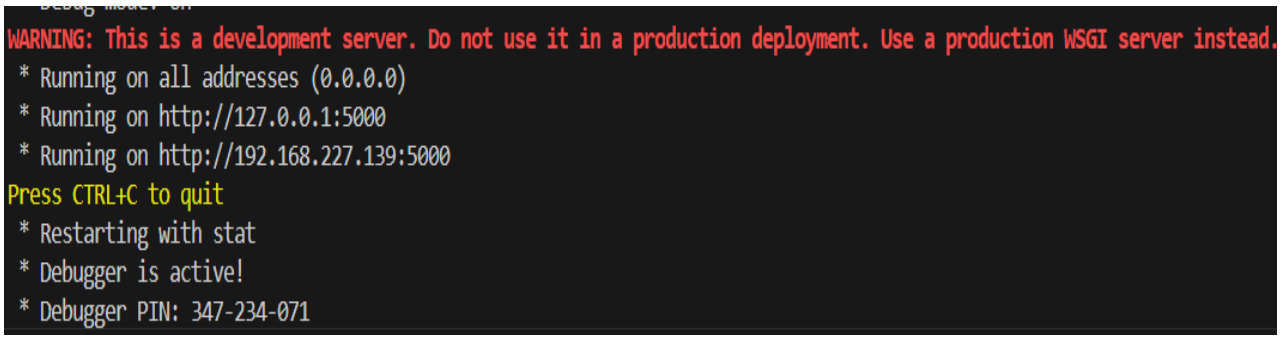
Run the Flask Application

sudo flask run --host=0.0.0.0 --port=80



Verify the Flask app is running:

http://your-ec2-public-ipo Run the Flask app on the EC2 instance



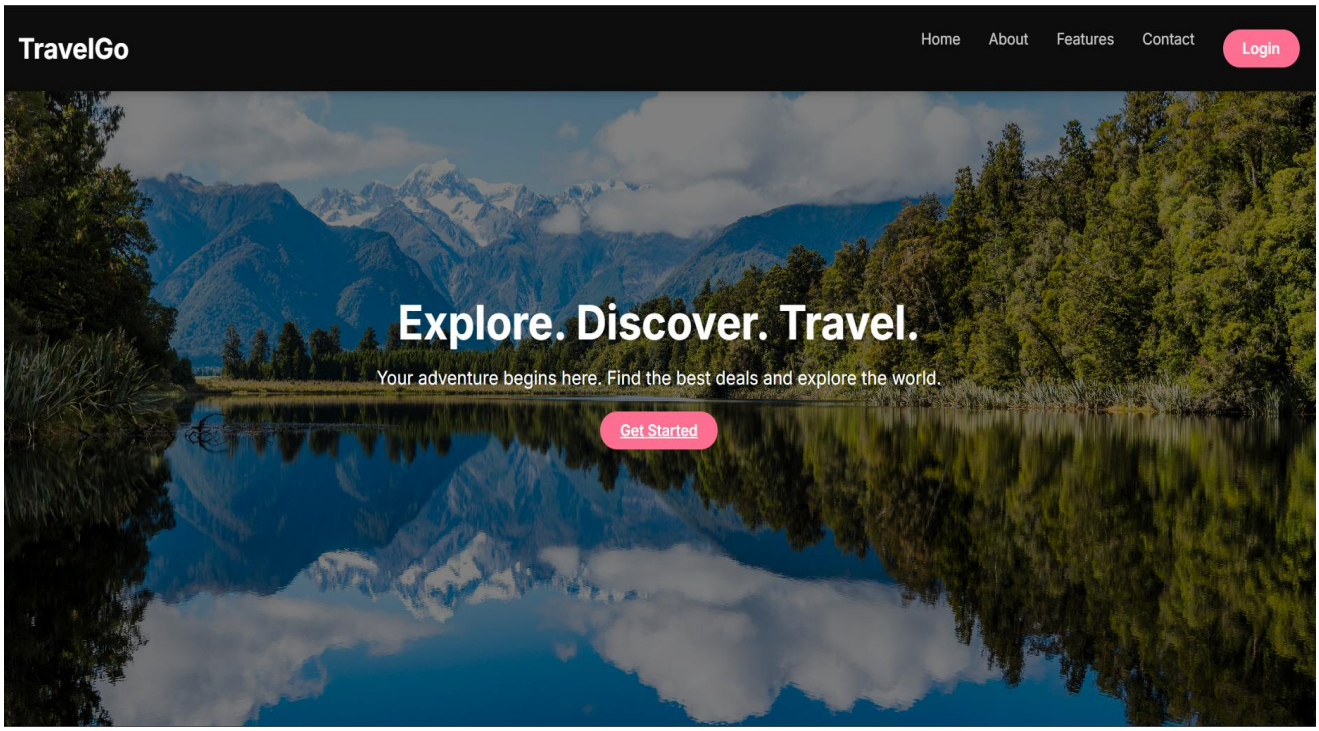
Access the website through:

PublicIPs: http://52.202.39.229:5000/

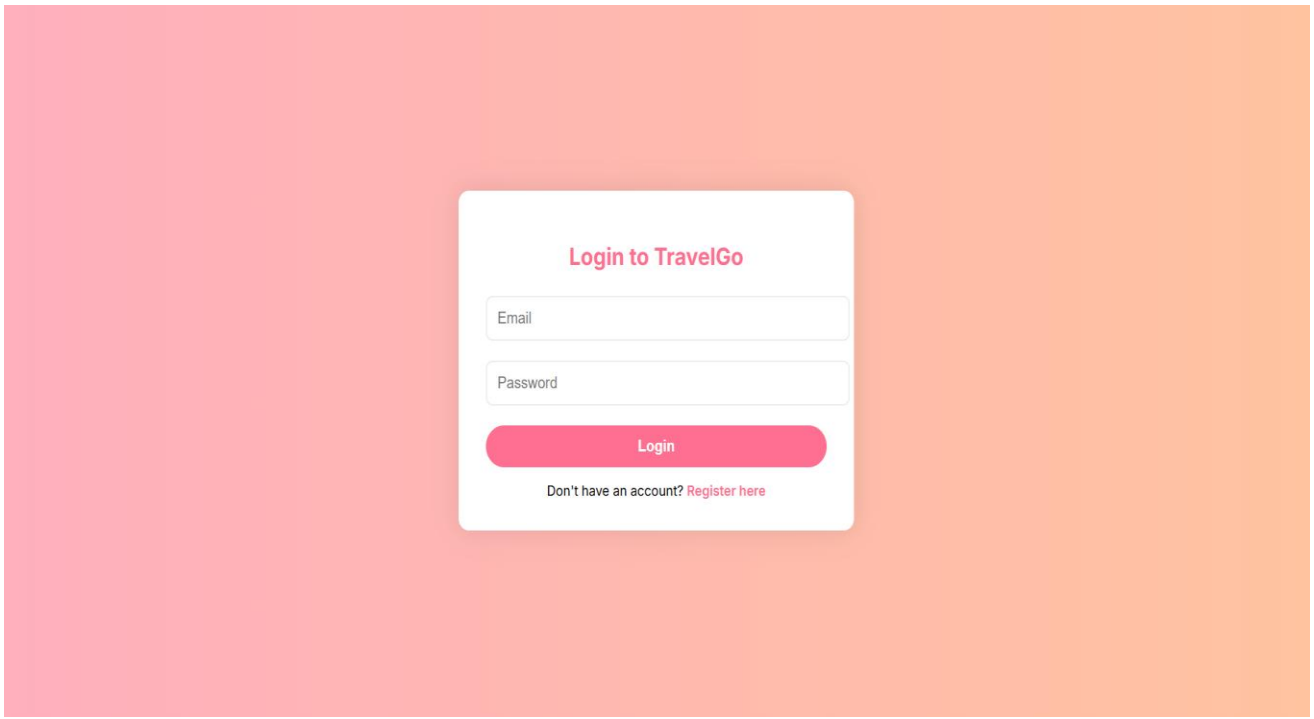
Milestone 8: Testing and Deployment

● **Activity 8.1:** Conduct functional testing to verify user registration, login, book requests, and notifications.

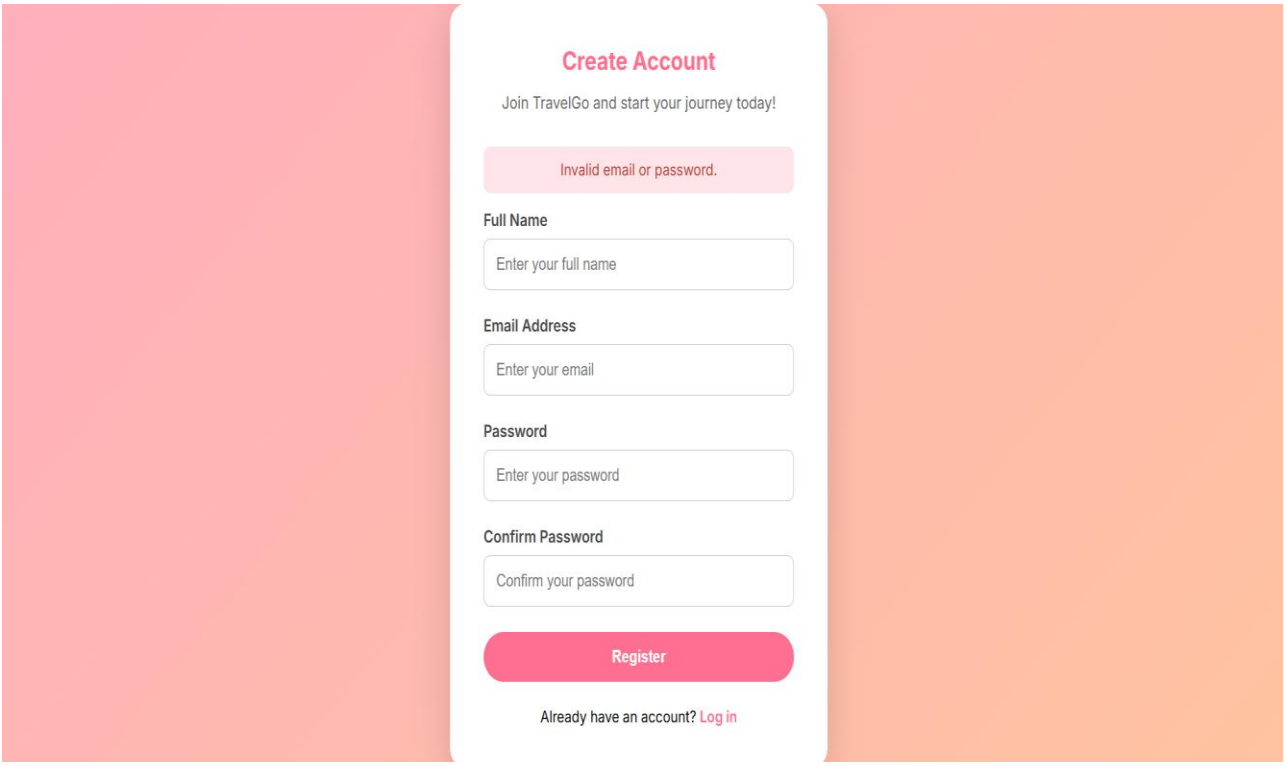
Index page:



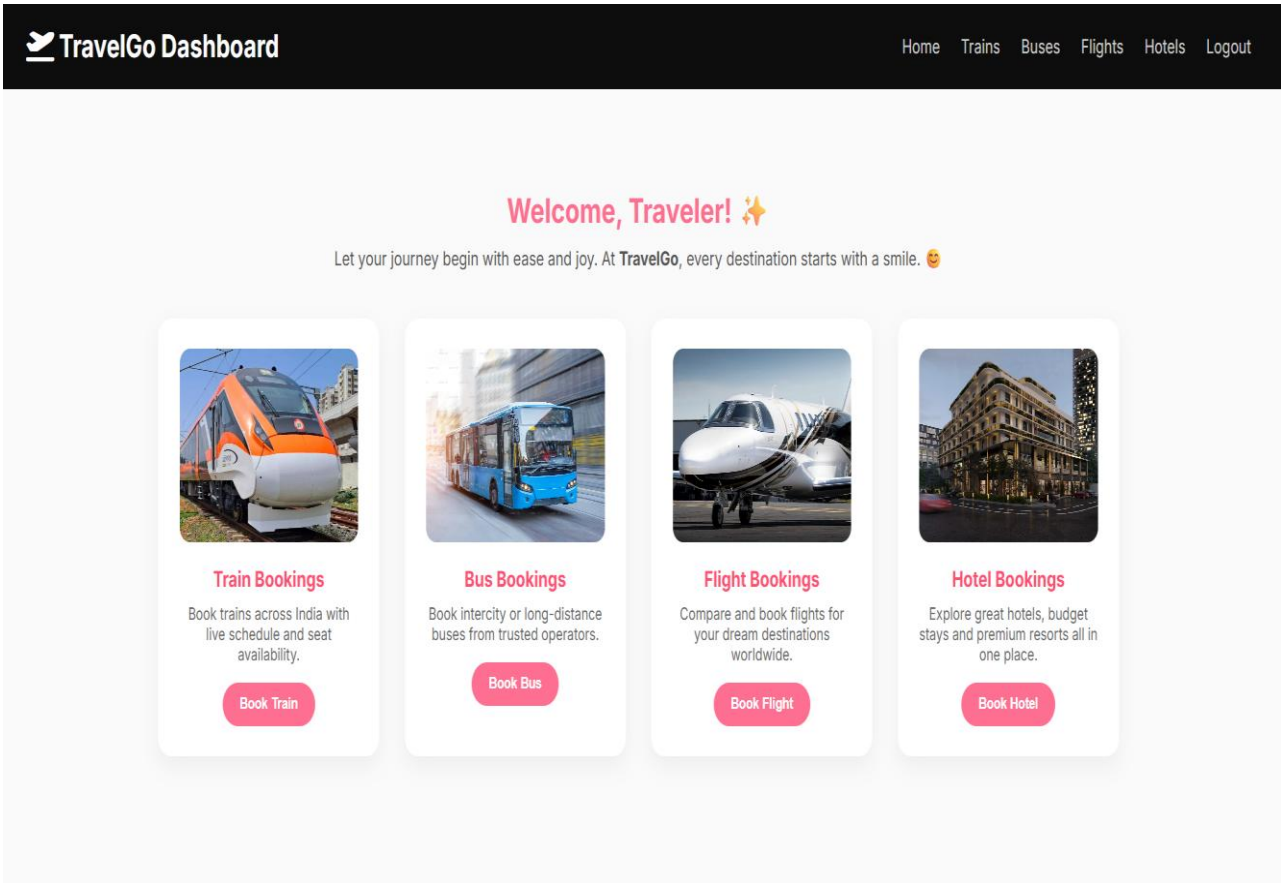
Login page:



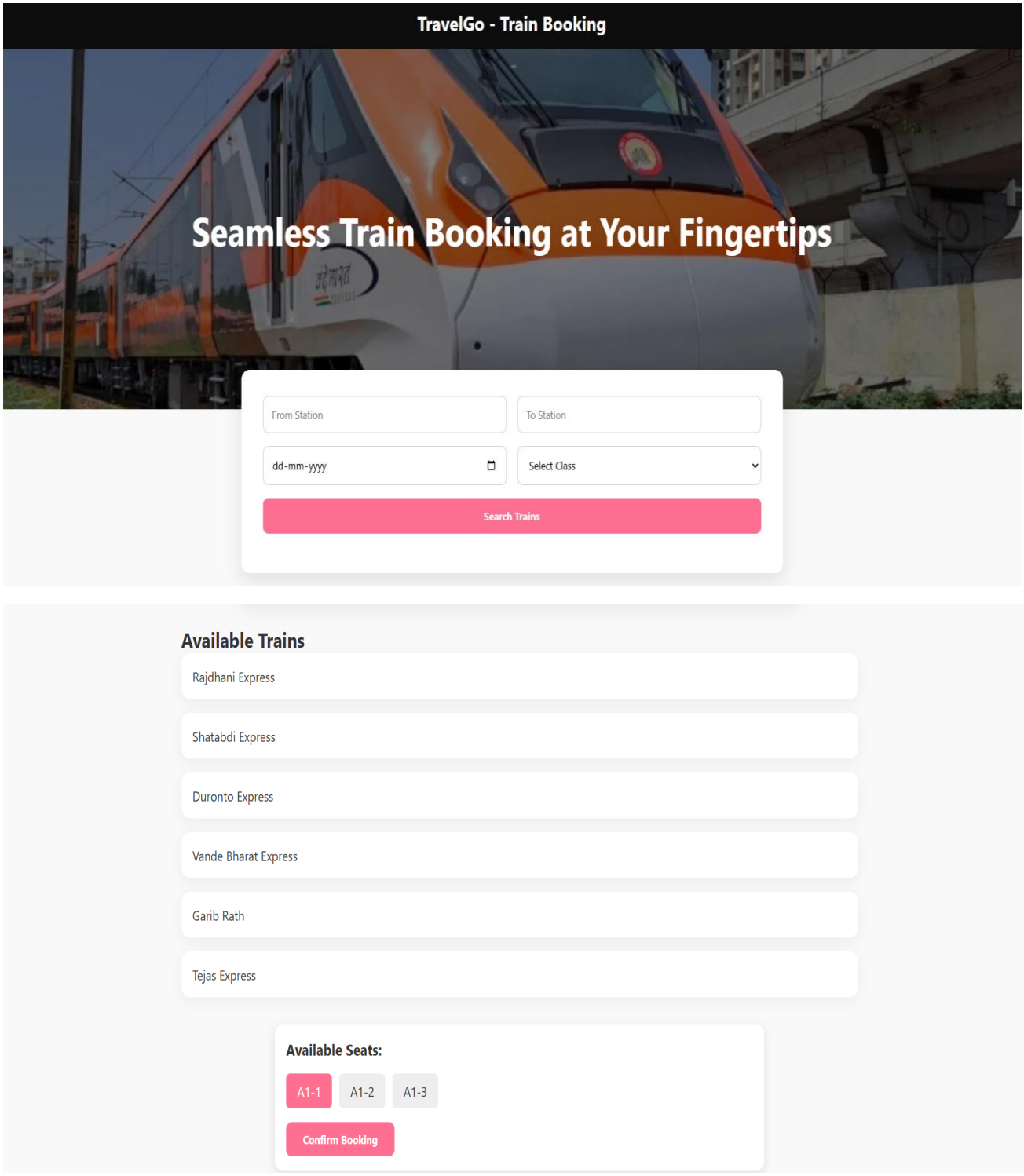
Register page:



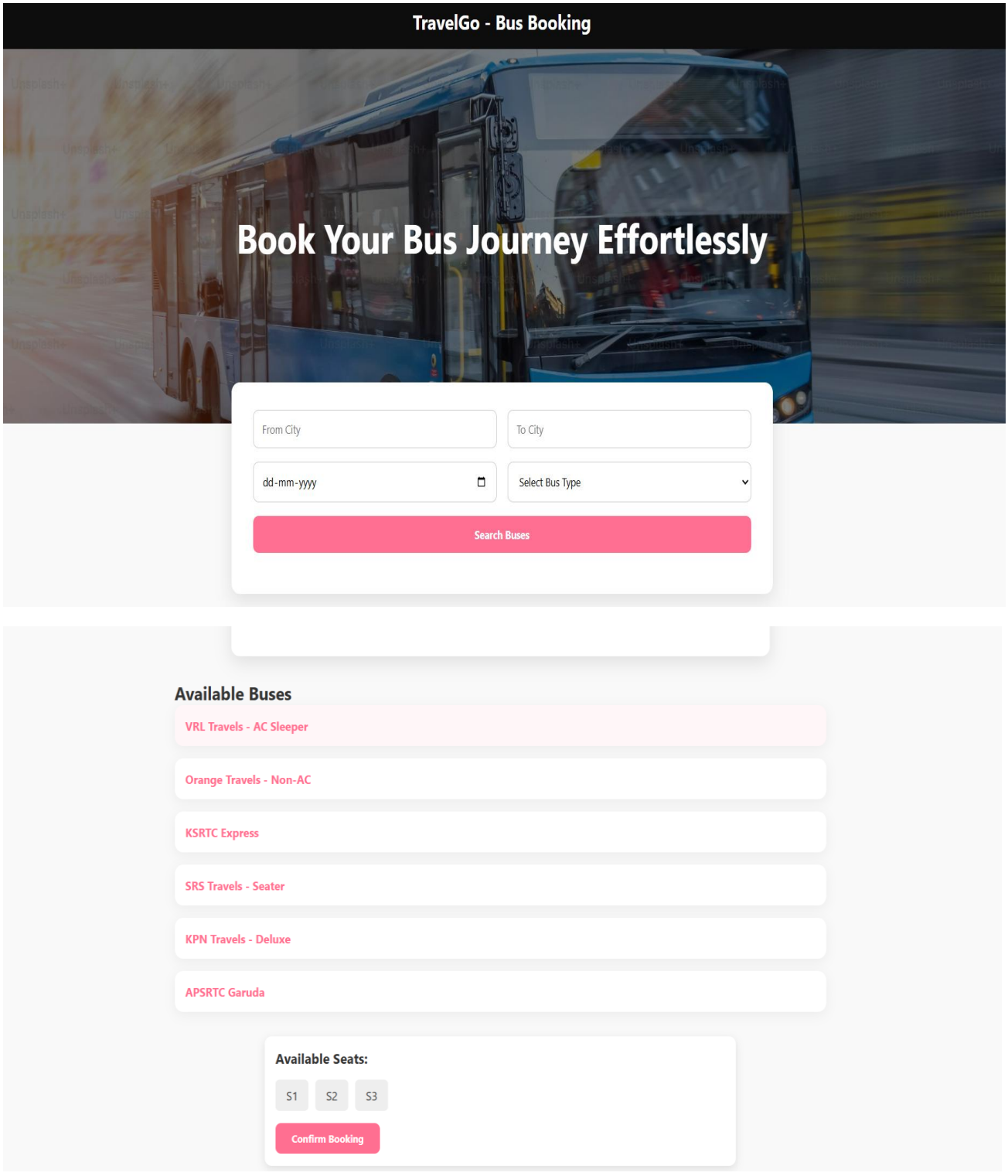
Dashboard page:



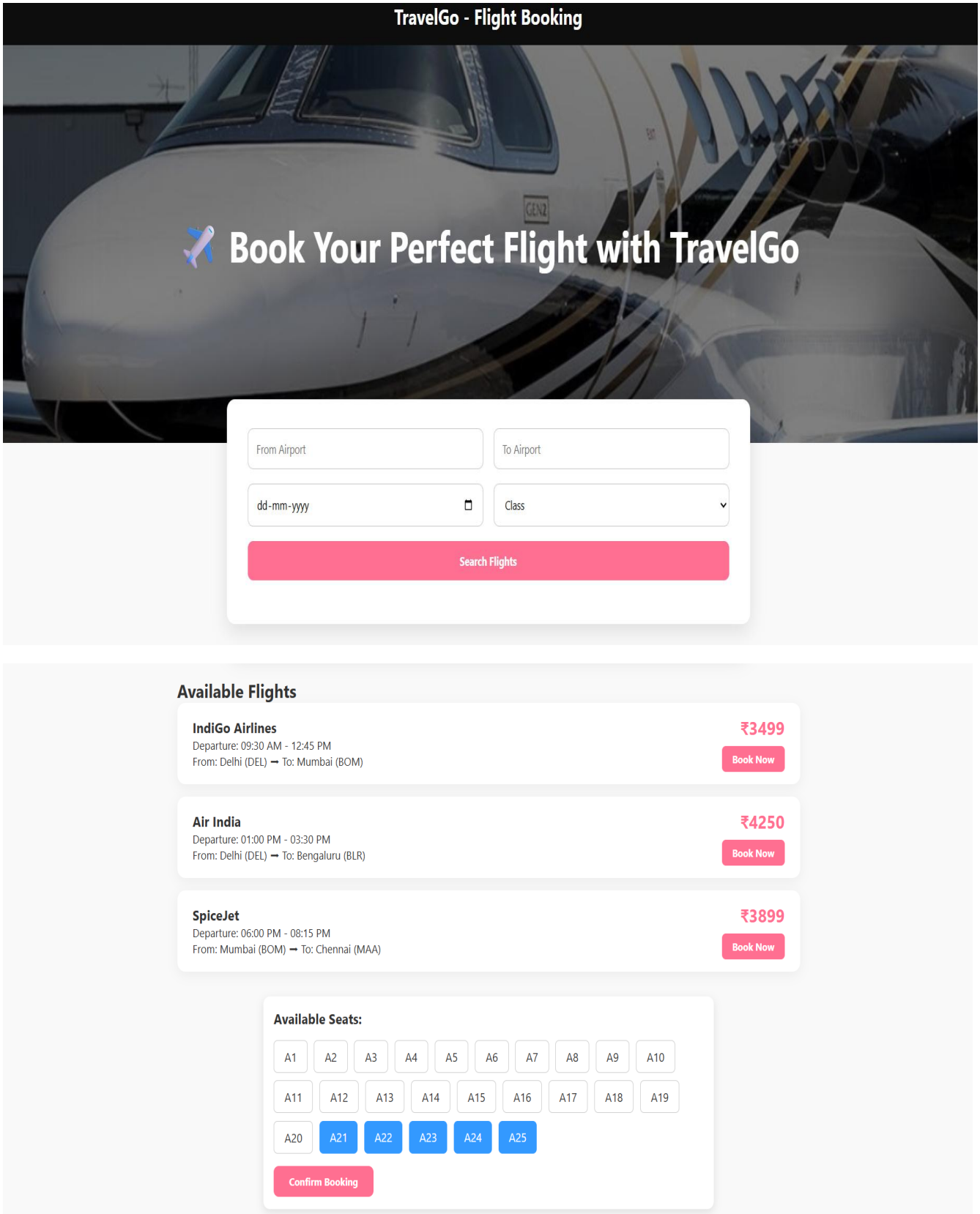
Train booking page:



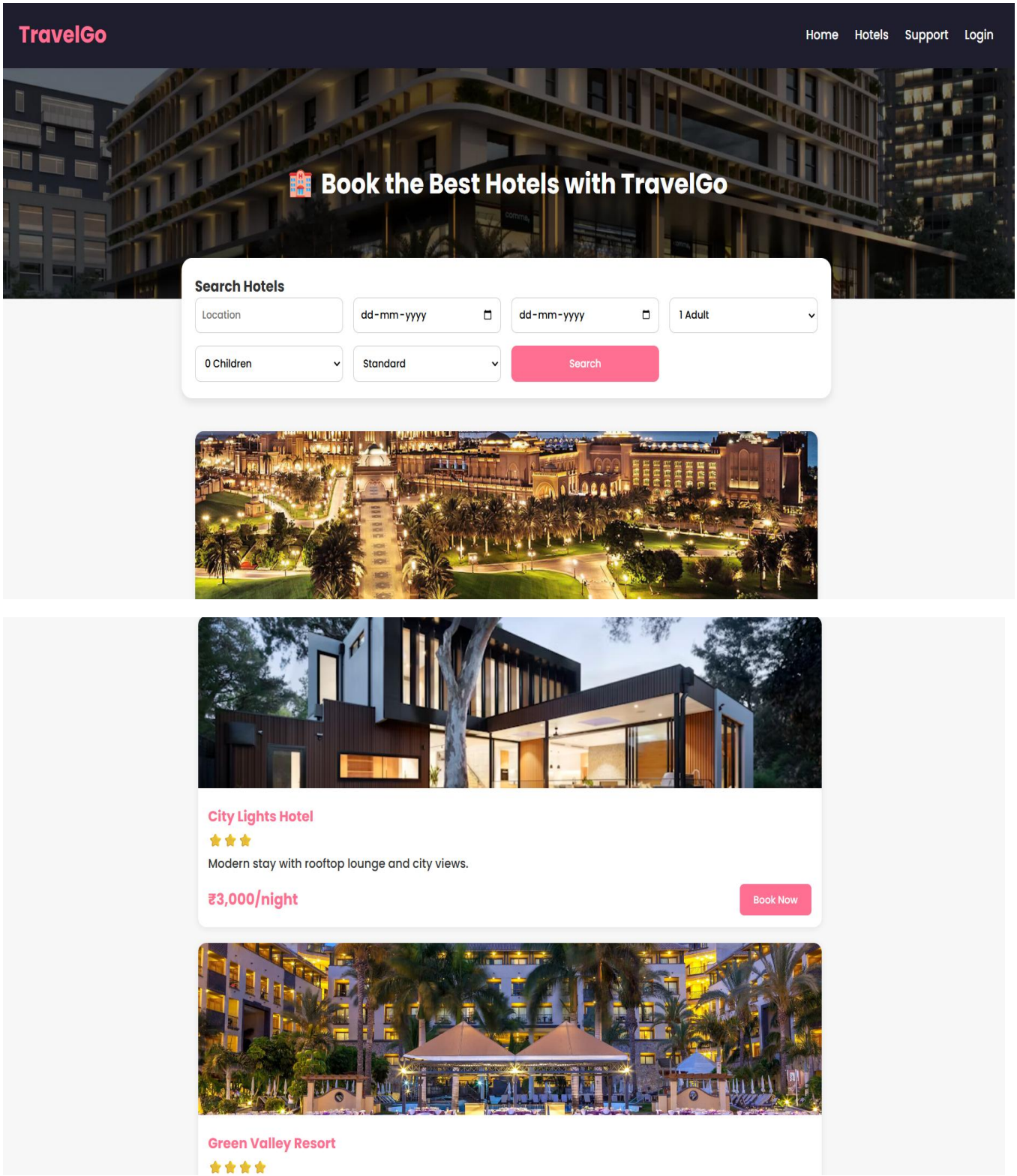
Bus booking page:

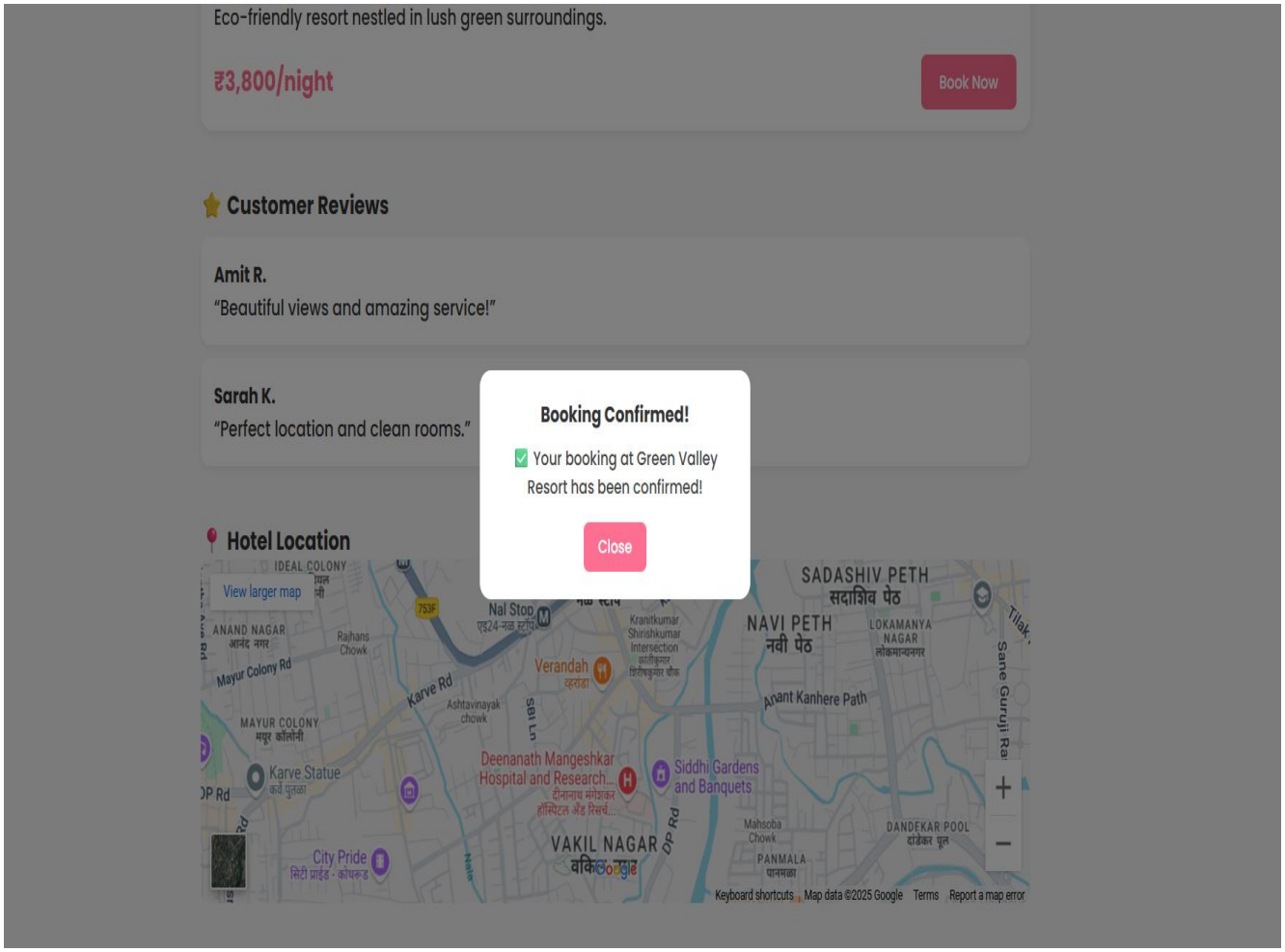


Flight booking page:



Hotel booking page:





Conclusion:

TravelGo successfully integrates modern cloud technologies to deliver a seamless, user friendly travel booking experience. By combining Flask, AWS EC2, DynamoDB, and SNS, the platform ensures real-time responsiveness, secure data handling, and instant user notifications. Its unified interface for booking buses, trains, flights, and hotels—along with dynamic dashboards and filtering options—makes it a scalable and reliable solution for modern travel needs. TravelGo not only enhances user convenience but also demonstrates the practical application of full-stack development and cloud services in solving real-world problems.