TravelGo – A Cloud-Based Travel Booking System

Project Description:

TravelGo is an integrated, full-stack travel booking web application that enables users to effortlessly plan and reserve transportation and accommodation across buses, trains, flights, and hotels through a single platform. Developed using Flask for backend operations and hosted on Amazon EC2, TravelGo ensures reliability, scalability, and responsiveness even under heavy traffic. Data management is handled through AWS DynamoDB, allowing for secure and efficient storage of user profiles and booking records.

The application supports secure user registration and login, interactive search for travel options, and seamless booking workflows. It also incorporates AWS Simple Notification Service (SNS) to send real-time email notifications for booking confirmations and cancellations, ensuring users remain updated throughout their travel planning.

The platform features a clean, responsive user interface that adapts to different devices. Notable features include dynamic seat selection for bus bookings, hotel filtering by preferences (such as budget or luxury), and a centralized dashboard to track and manage all bookings.

Scenario 1: Unified Booking Across All Travel Modes

TravelGo simplifies multi-modal travel planning by allowing users to book buses, trains, flights, and hotels all from one place. For example, a traveler booking a trip from Hyderabad to Bangalore can log in, explore all available transport modes, and confirm their preferred options. The Flask backend handles user queries and real-time data fetching, while AWS EC2 hosting ensures high availability and responsiveness during peak usage like weekends or holiday seasons.

Scenario 2: Instant Booking Confirmation via AWS SNS

Upon completing any booking—whether it's a hotel reservation or a flight ticket—TravelGo triggers an email notification using AWS SNS. For instance, if a user books a hotel in Chennai, they immediately receive an email confirmation with all relevant booking details. This is managed via Flask backend logic, which interacts with DynamoDB to store the booking and then calls SNS to notify the user. Admins or service providers can also receive alerts, facilitating real-time awareness and efficient customer support.

Scenario 3: Personalized Dashboard for Booking Management

TravelGo includes a personalized dashboard where users can view, manage, and cancel bookings. Bookings are categorized (bus, train, flight, hotel) and displayed with dates, prices, and status updates. Flask dynamically pulls this data from DynamoDB and renders it using responsive HTML/CSS templates. Users can easily track their travel history, upcoming plans, and process cancellations with a single click—ensuring a seamless and user-friendly experience on both desktop and mobile devices.

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 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 5.2: Attach Policies

6. **EC2 Instance Setup**

   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**

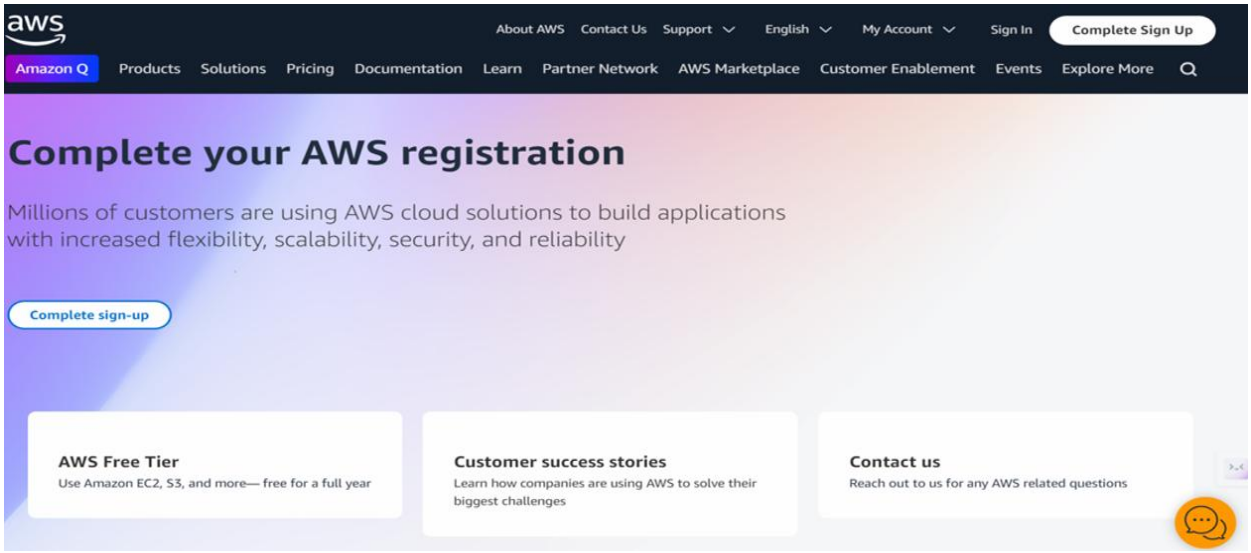
   Activity 7.1: Upload Flask Files
   Activity 7.2: Run the Flask App

8. **Testing and Deployment**

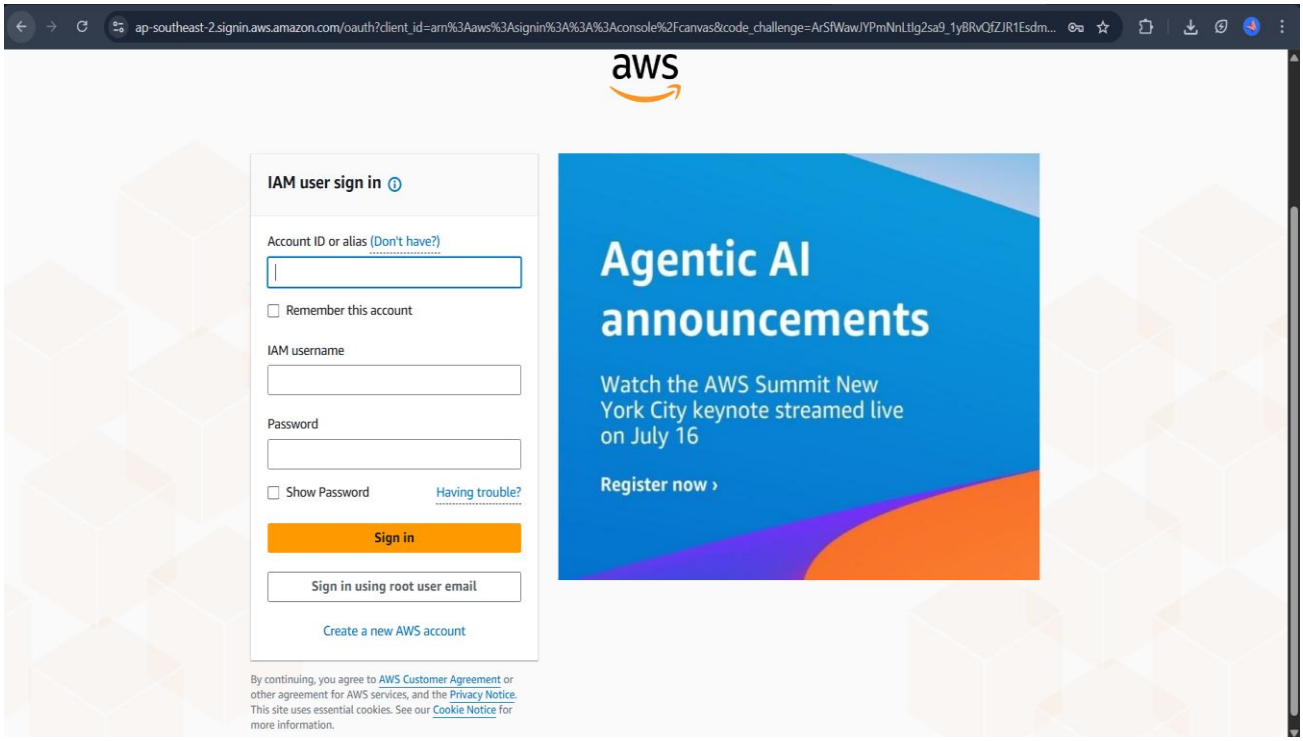
   Activity 8.1: Conduct functional testing to verify user registration, login, book requests, and notifications.

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.

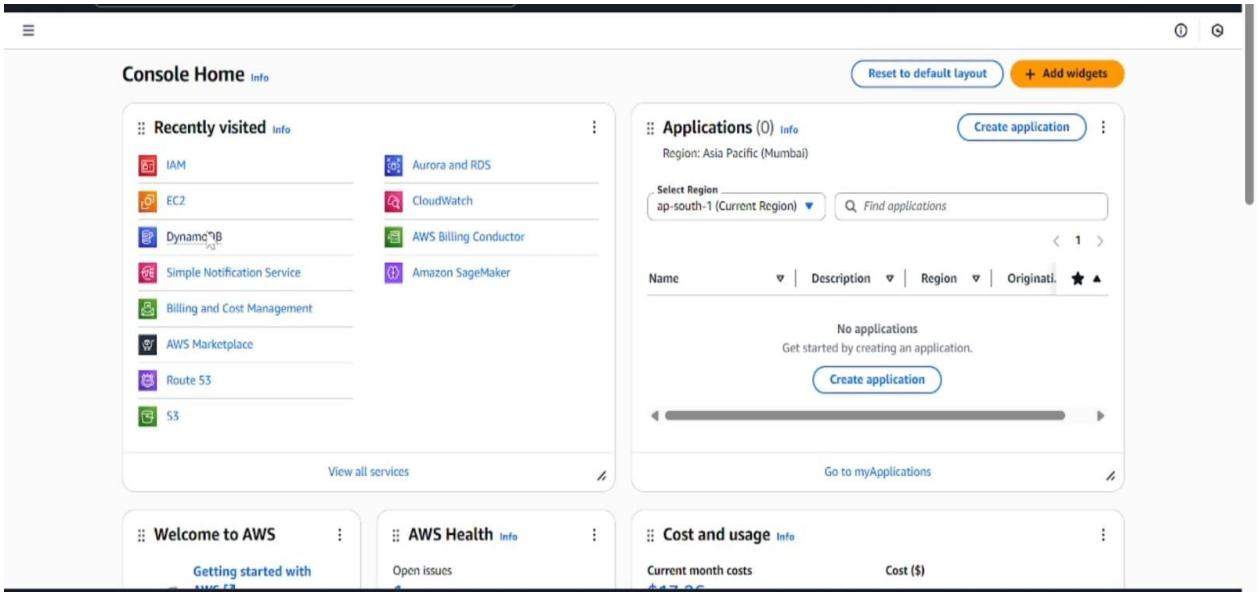


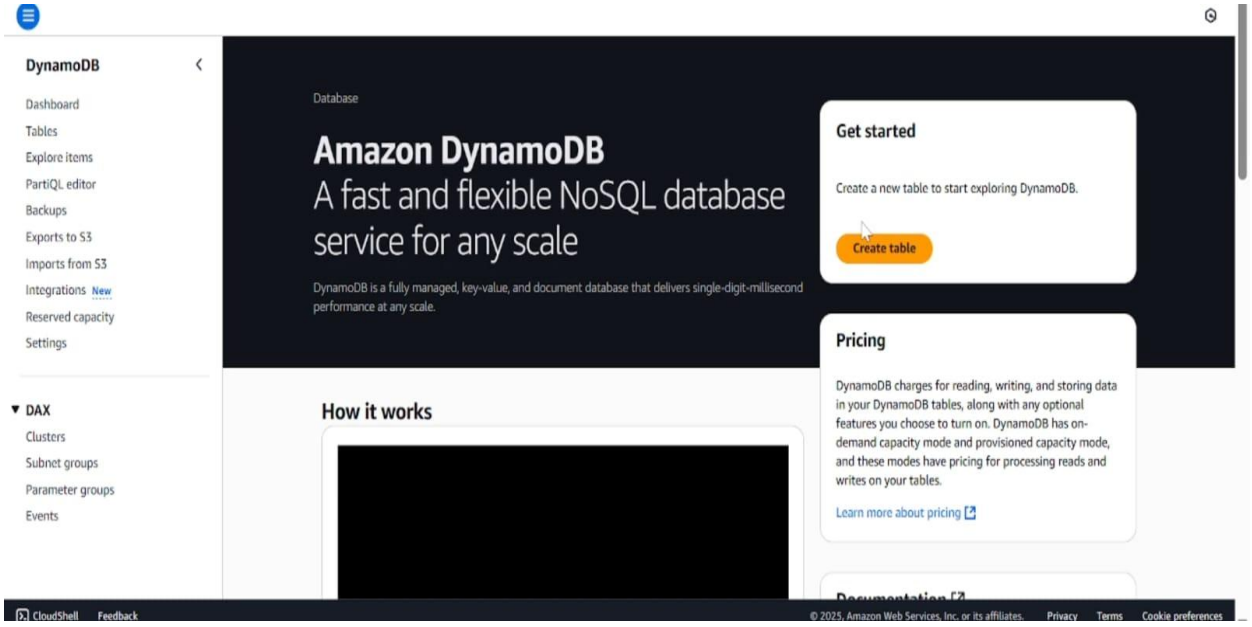
- **Activity 1.2: Log in to the AWS Management Console**
    - After setting up your account, log in to the [AWS Management Console]



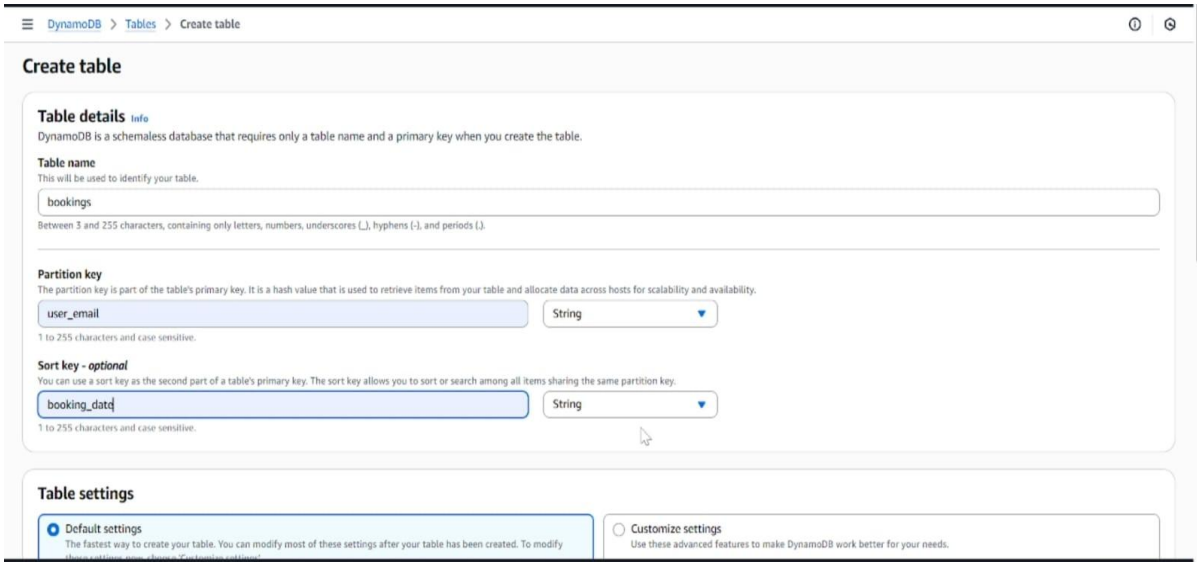
Milestone 2: DynamoDB Database Creation and Setup

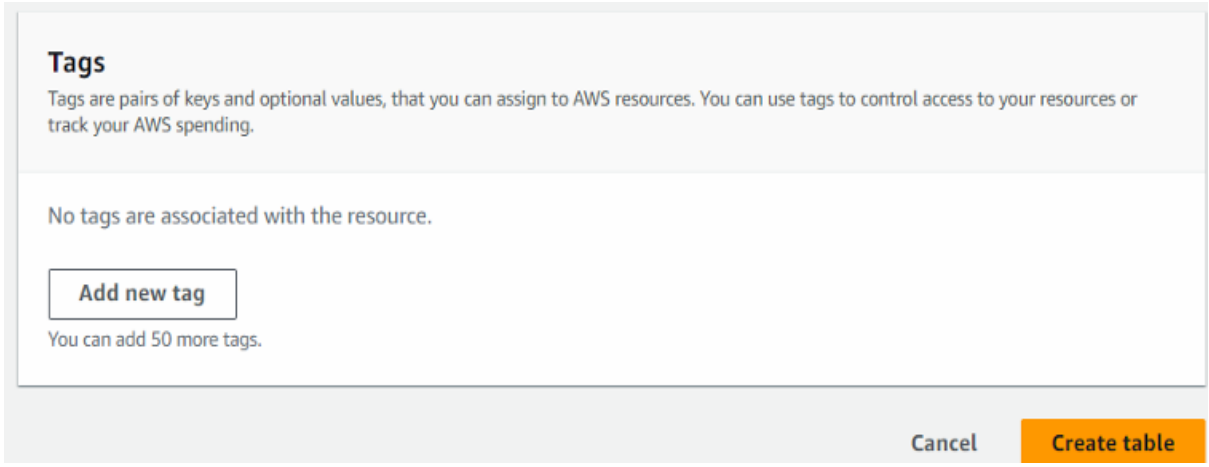
- **Activity 2.1:Navigate to the DynamoDB**
    - In the AWS Console, navigate to DynamoDB and click on create tables.



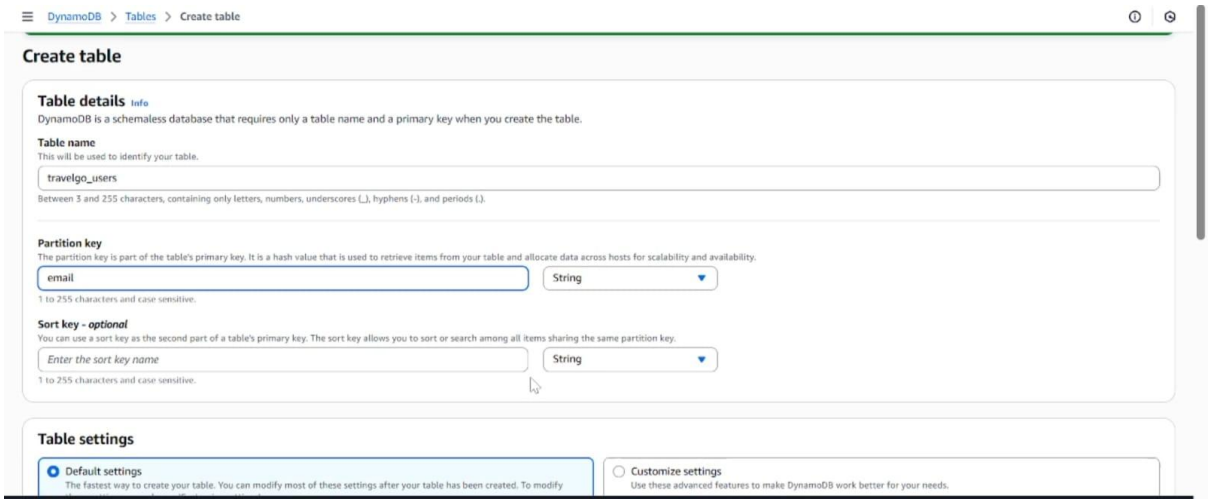


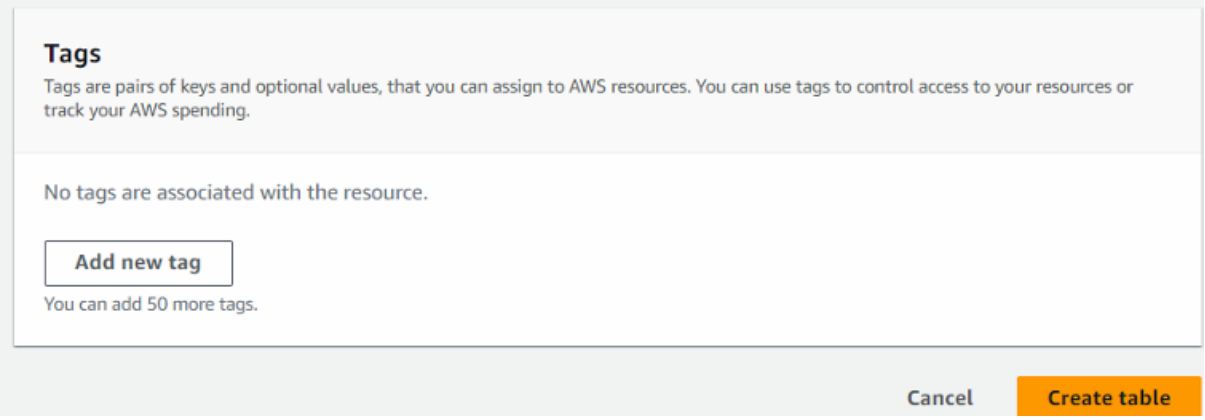
- **Activity 2.2: Create a DynamoDB table for storing registration details and book requests.**
    - Create Users table with partition key “Email” with type String and click on create tables.



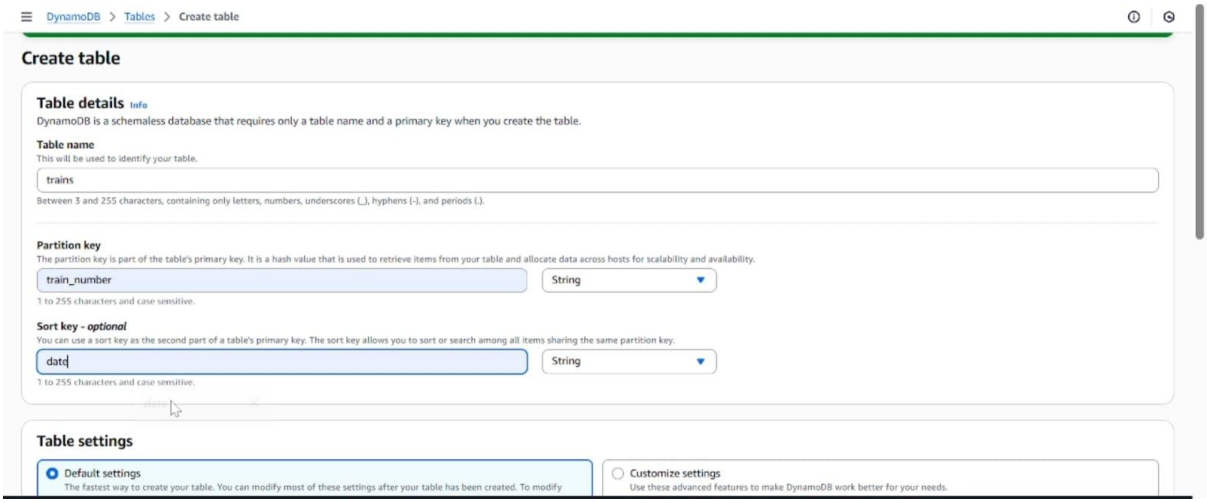


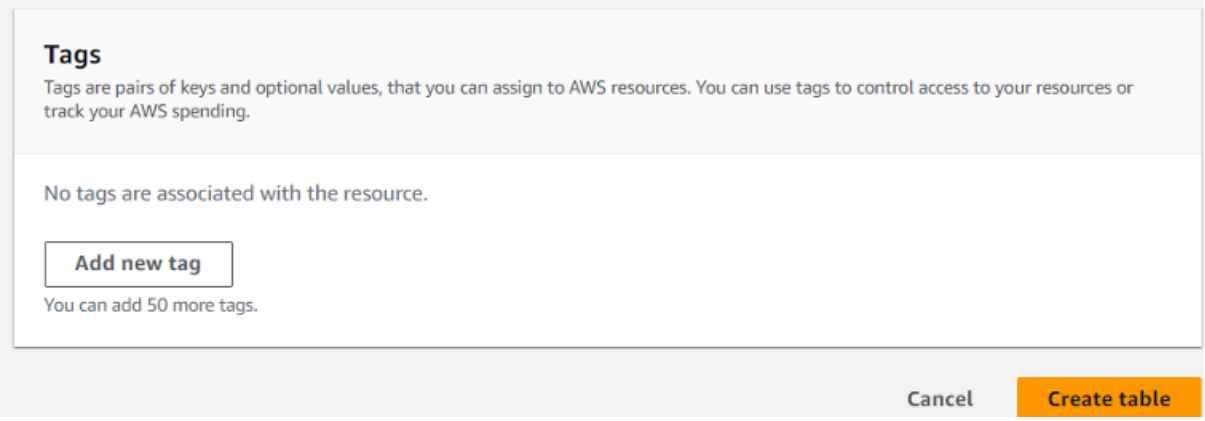
○ Follow the same steps to create a travelgo_users table with Email as the primary key.





○ Follow the same steps to create a trains table with train_number as the partition key.

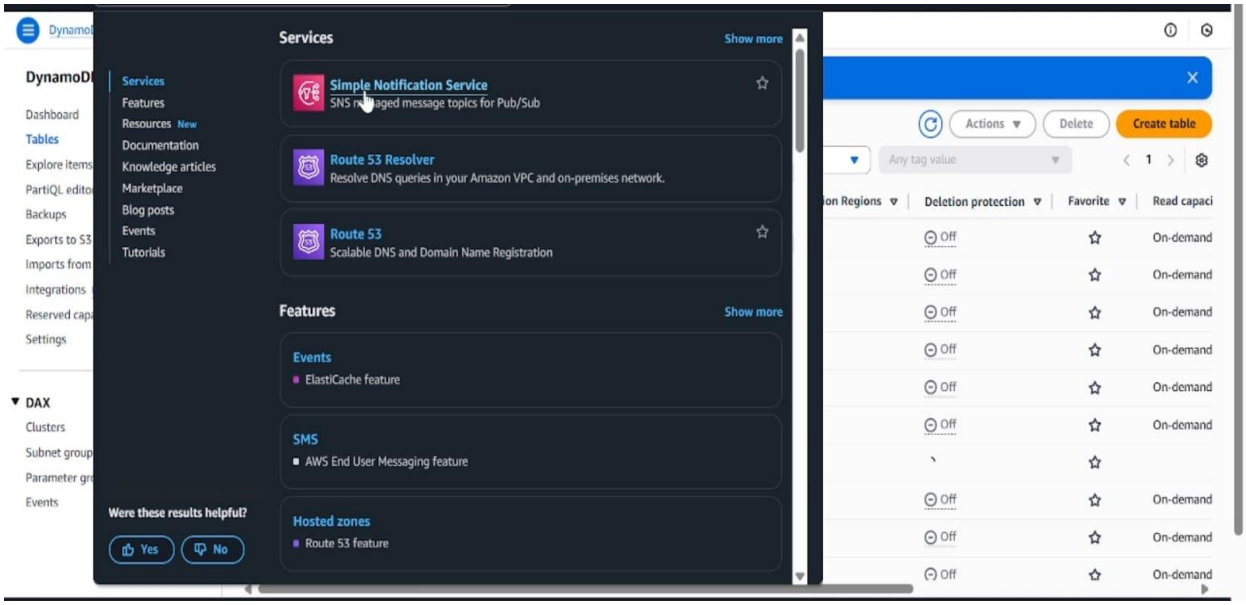


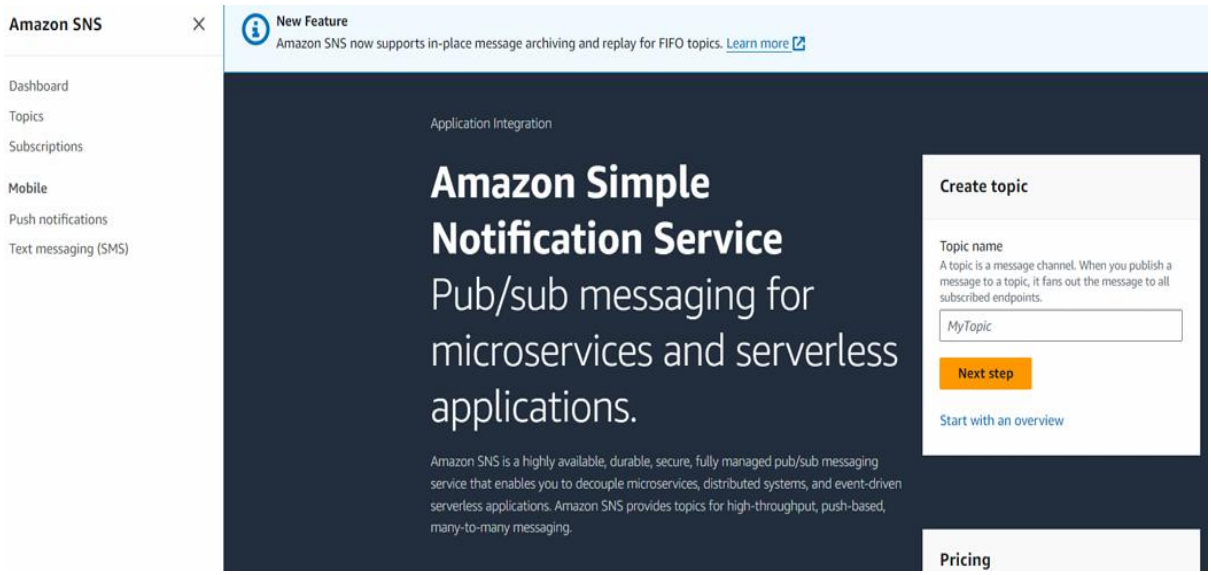


Milestone 3: SNS Notification Setup

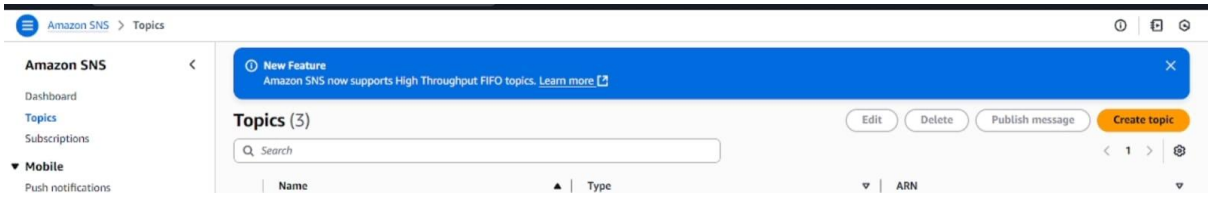
● **Activity 3.1: Create SNS topics for sending email notifications to users and library staff.**

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

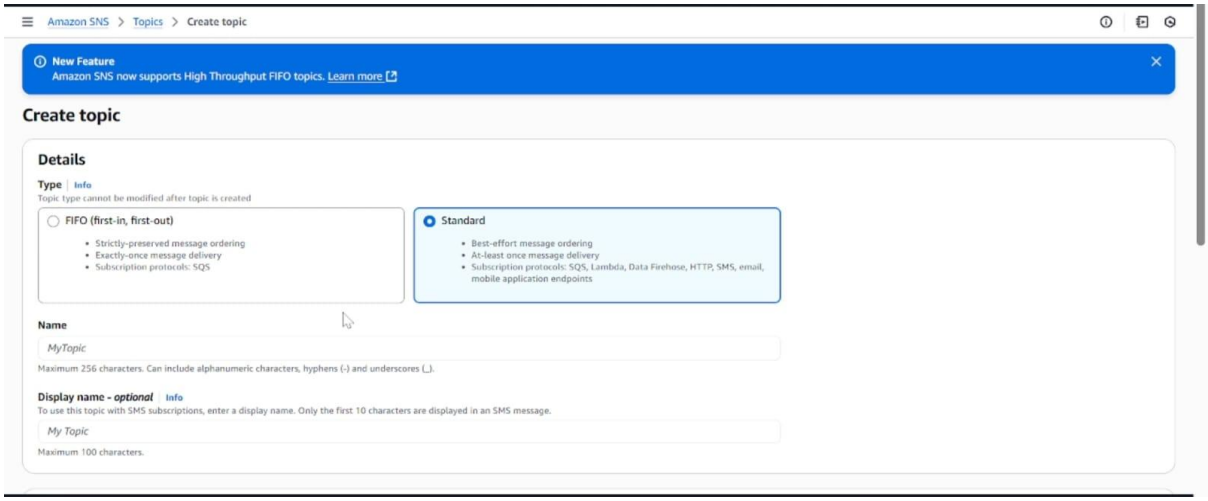




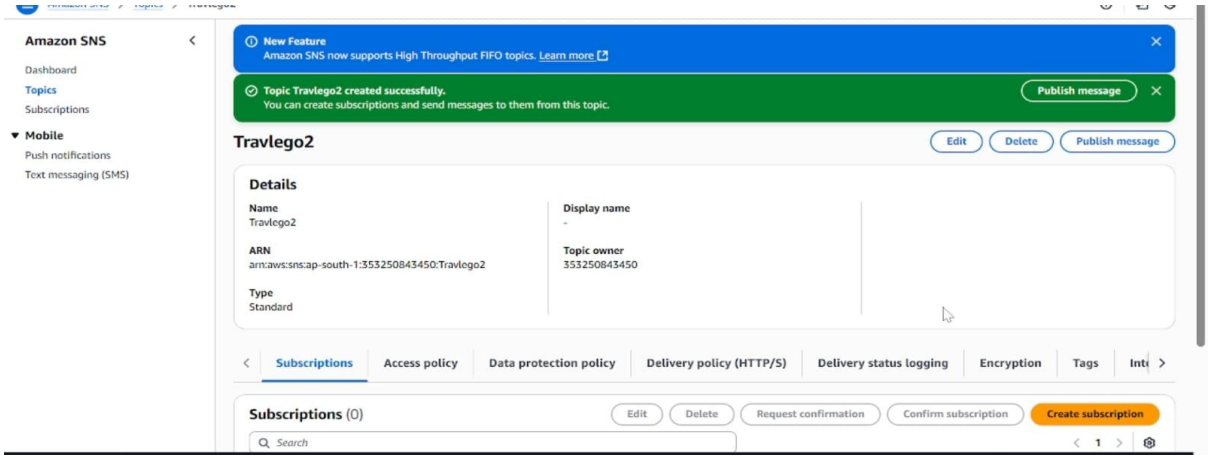
○ Click on **Create Topic** and choose a name for the topic.



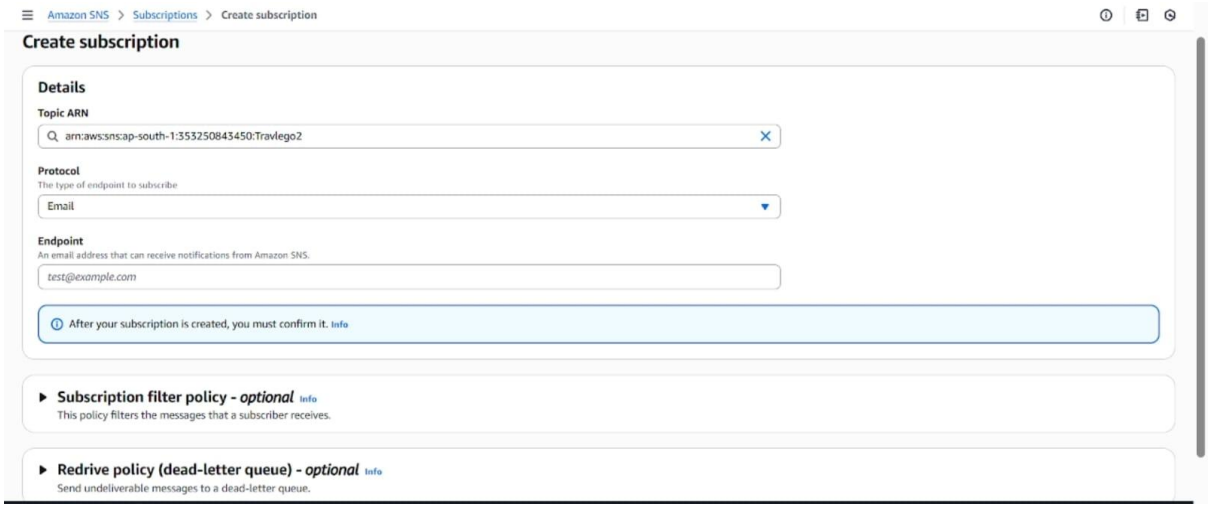
○ Choose Standard type for general notification use cases and Click on Create Topic.



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

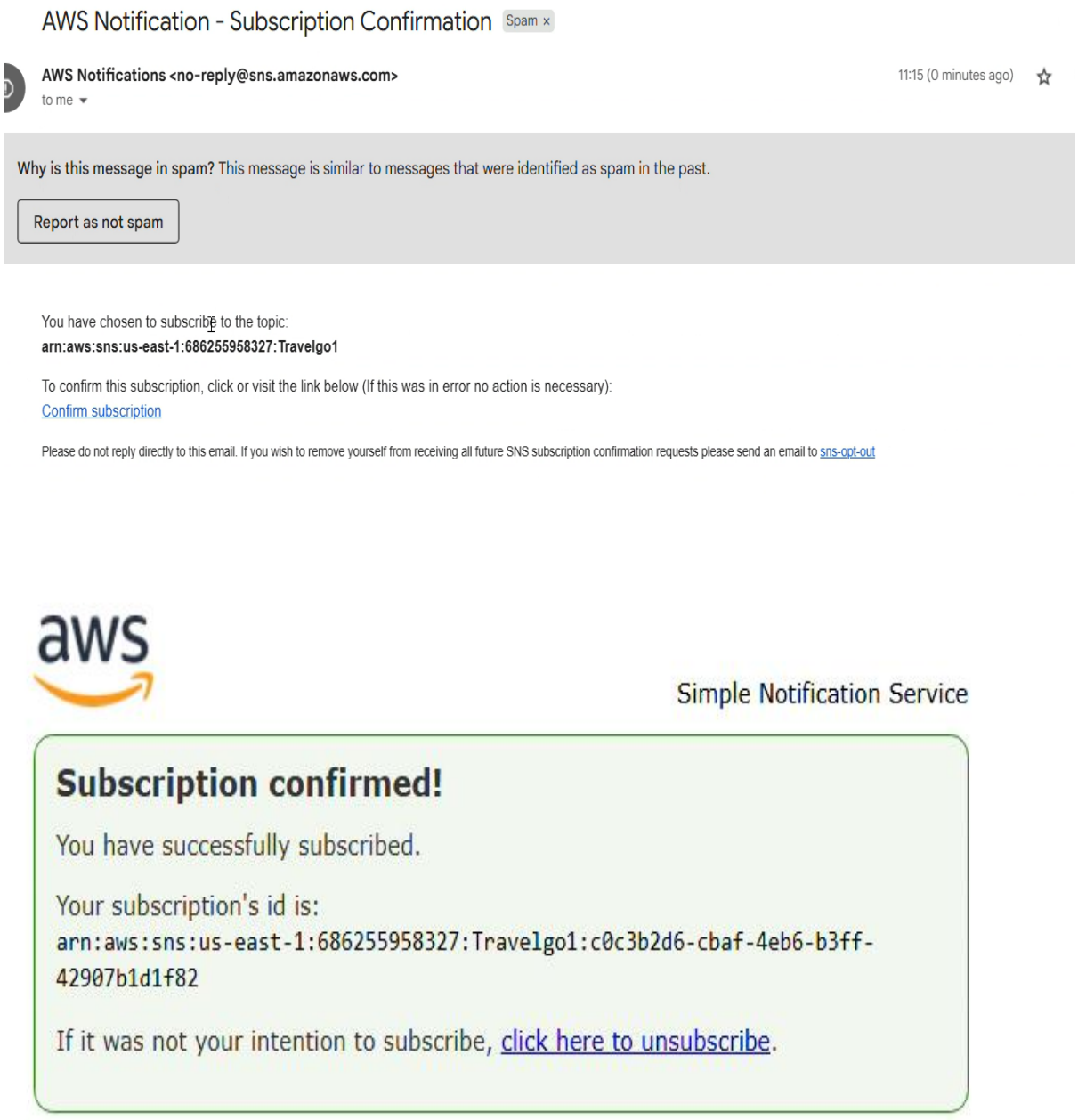


● Activity 3.2: Subscribe users and staff to relevant SNS topics to receive real-time notifications when a book request is made.



○ After subscription request for the mail confirmation.

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

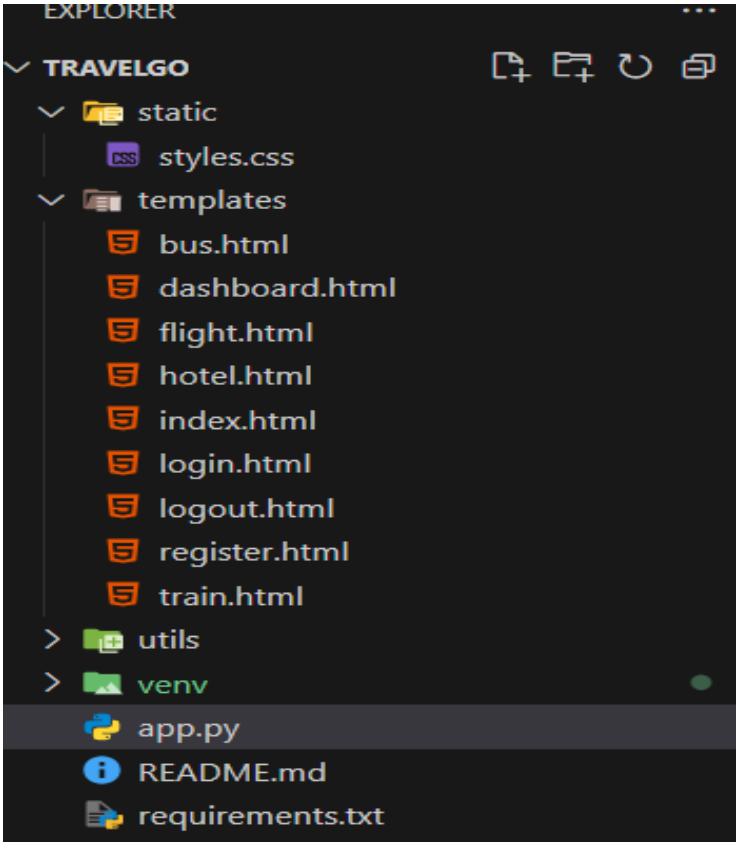


○ Successfully done with the SNS mail subscription and setup, now store the ARN link.

Milestone 4: Backend Development and Application Setup

● **Activity 4.1: Develop the backend using Flask**

○ File Explorer Structure



Description of the code :

● Flask App Initialization

```
from flask import Flask, render_template, request, redirect, url_for, ses
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
import uuid
```

Description:
Import essential libraries including Flask utilities for routing, template rendering, session handling, and request processing; Boto3 and DynamoDB conditions for performing AWS DynamoDB operations; Werkzeug security module for password hashing and verification; Datetime for handling timestamps; Decimal for precise number representation (required by DynamoDB); and UUID for generating unique booking and user identifiers.

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

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

● **Dynamodb Setup:**

```
# DynamoDB Tables
users_table = dynamodb.Table('travelgo_users')
bookings_table = dynamodb.Table('bookings')
```

Description:
Define references to DynamoDB tables used in the application.

- users_table connects to the travelgo_users table, which stores user registration data such as names, emails, and hashed passwords.
- bookings_table connects to the bookings table, which stores all travel booking records including buses, trains, flights, and hotels, along with user details, booking IDs, and timestamps.

● **SNS Connection**

```
SNS_TOPIC_ARN = 'arn:aws:sns:us-east-1:713881794827:travelgo:302dcf42-b7bc

# Function to send SNS notifications
def send_sns_notification(subject, message):
    try:
        sns_client.publish(
            TopicArn=SNS_TOPIC_ARN,
            Subject=subject,
            Message=message
        )
        print(f"[✓] SNS notification sent: {subject}")
    except Exception as e:
        print(f"SNS Error: Could not send notification - {e}")
```

Description:
Define the **SNS Topic ARN** used to publish notifications through **AWS Simple Notification Service (SNS)**.
The send_sns_notification() function is responsible for sending real-time **email alerts or notifications** (e.g., booking confirmations or cancellations).
It uses the sns_client.publish() method to send a message with a given subject and body to the specified SNS topic.
Error handling is included to catch and print any exceptions if the notification fails.

- **Routes for Web Pages**
    - **Home Route:**

```python
# 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:**

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

        # Check if user already exists
        existing = users_table.get_item(Key={'email': email})
        if 'Item' in existing:
            flash('User already exists!', 'error')
            return render_template('register.html')

        # Hash password and store user
        hashed_password = generate_password_hash(password)
        user_data = {
            'email': email,
            'name': name,
            'password': hashed_password,
            'logins': 0
        }

        users_table.put_item(Item=user_data)
        print(f"[✓] Registered new user: {email}")

        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.

● **login Route:**

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

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

        if 'Item' in user_response:
            user = user_response['Item']
            if check_password_hash(user['password'], password):
                session['email'] = email
                session['user_name'] = user.get('name', email)

                # Update login count
                users_table.update_item(
                    Key={'email': email},
                    UpdateExpression='ADD logins :val',
                    ExpressionAttributeValues={':val': 1}
                )

                flash('Login successful! Welcome back.', 'success')
                return redirect(url_for('dashboard'))

        flash('Invalid email or password. Please try again.', '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
    try:
        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
        for booking in bookings:
            for key, value in booking.items():
                if isinstance(value, Decimal):
                    booking[key] = float(value)

    except Exception as e:
        print(f"Error fetching bookings: {e}")
        bookings = []

    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:**

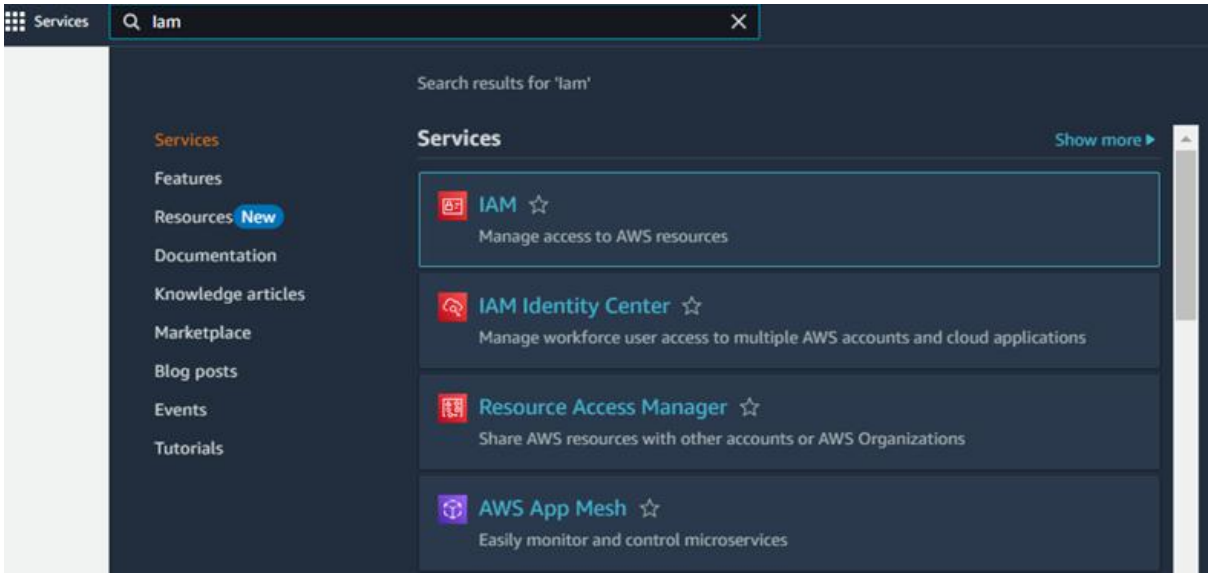
```python
if __name__ == '__main__':
    app.run(debug=True, host='0.0.0.0', port=5000)
```

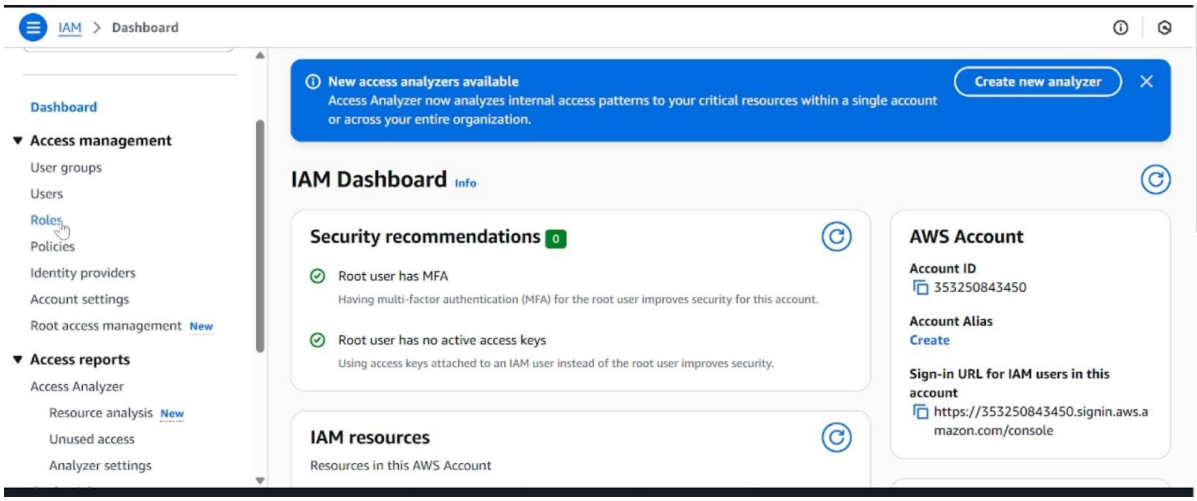
Description:
Starts the Flask application when the script is run directly.
Runs the app in debug mode for development, allowing live code reloading and error display.
Binds the server to all IP addresses (0.0.0.0) on port 5000, making it accessible on the local network or EC2 instance.

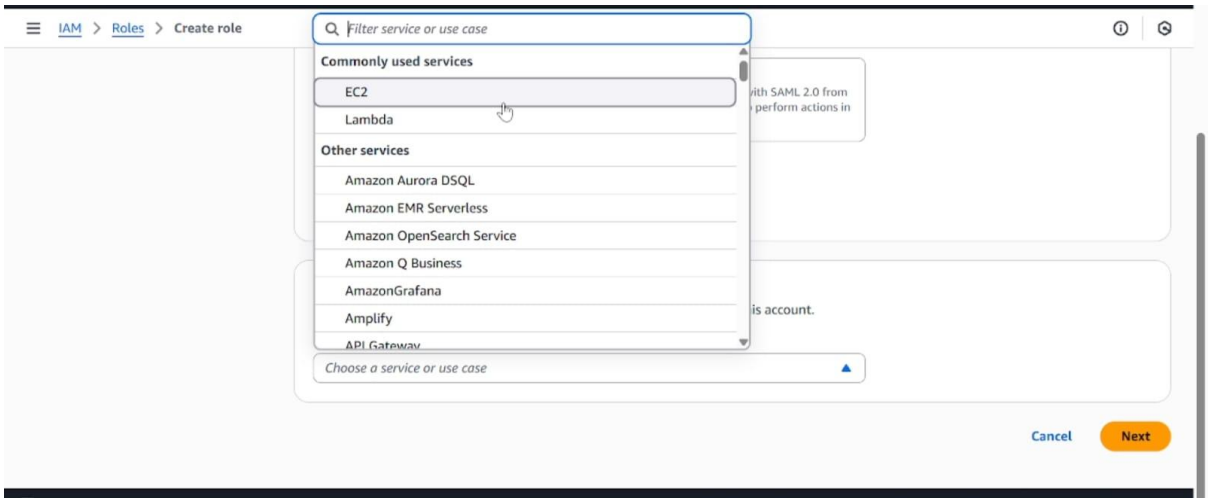
Milestone 5: IAM Role Setup

- **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.







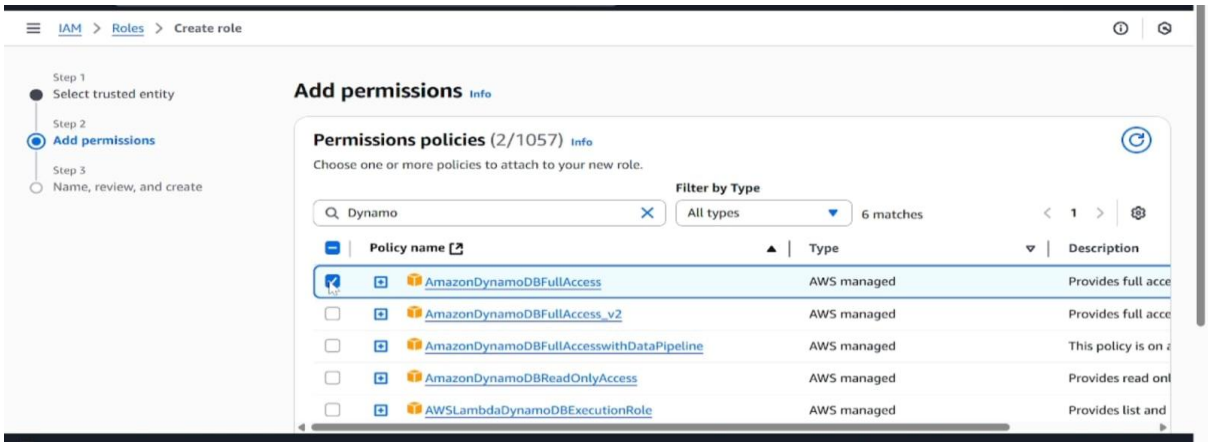
● **Activity 5.2: Attach Policies.**

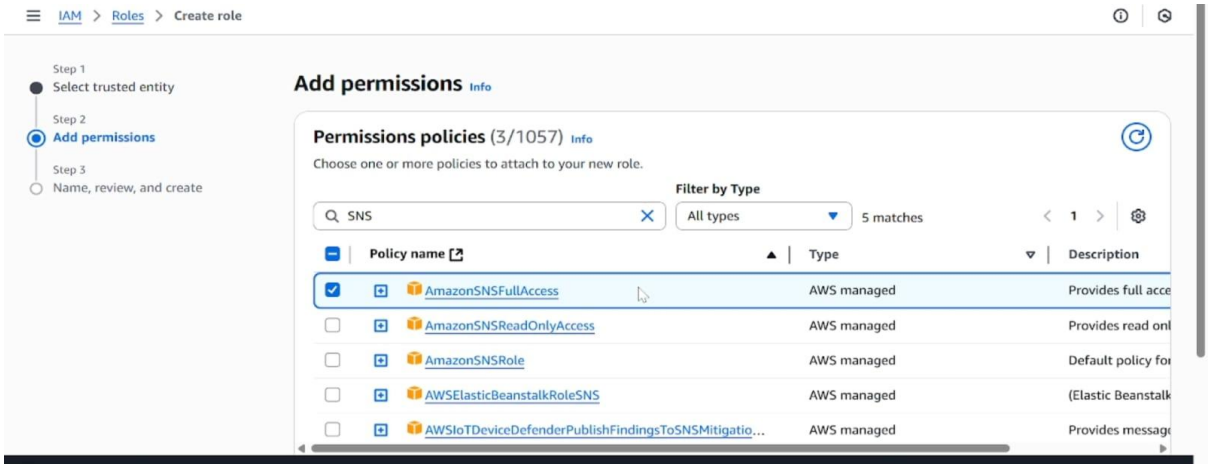
Attach the following policies to the role:

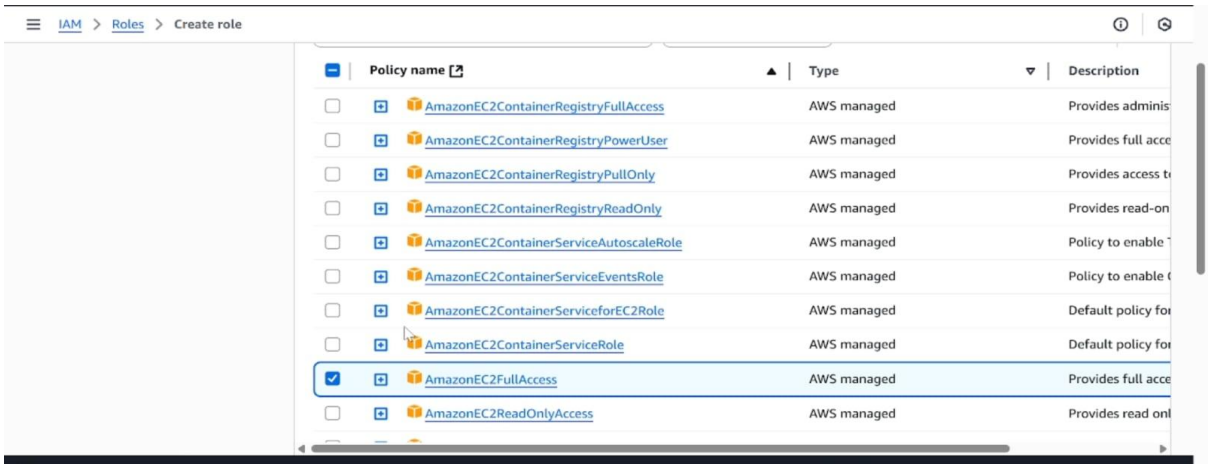
● **AmazonDynamoDBFullAccess:** Allows EC2 to perform read/write operations on DynamoDB.

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

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

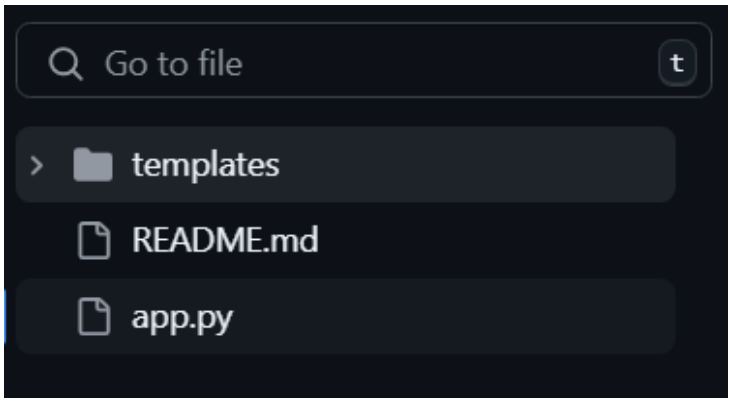


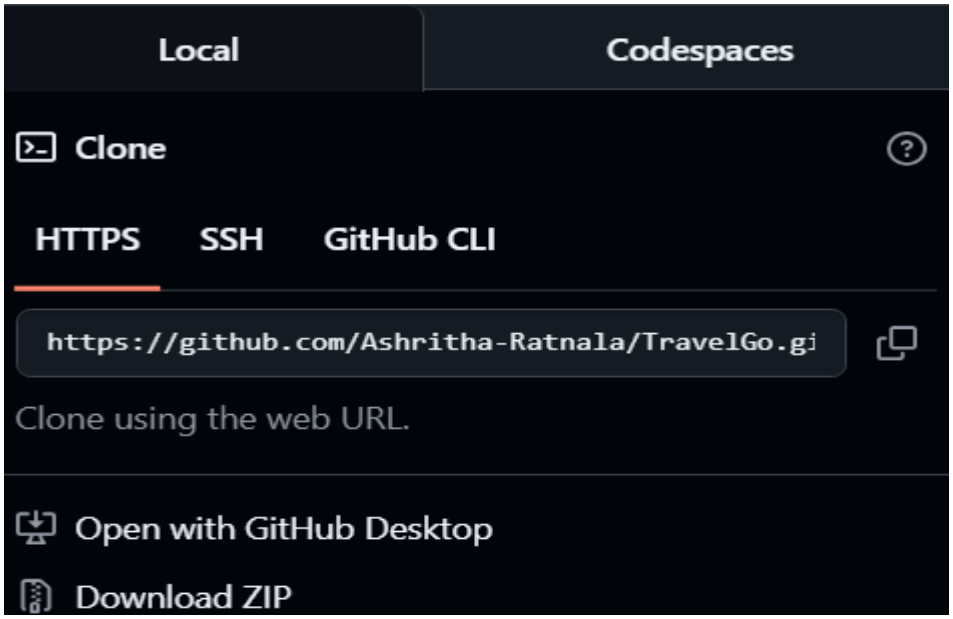




Milestone 6: EC2 Instance Setup

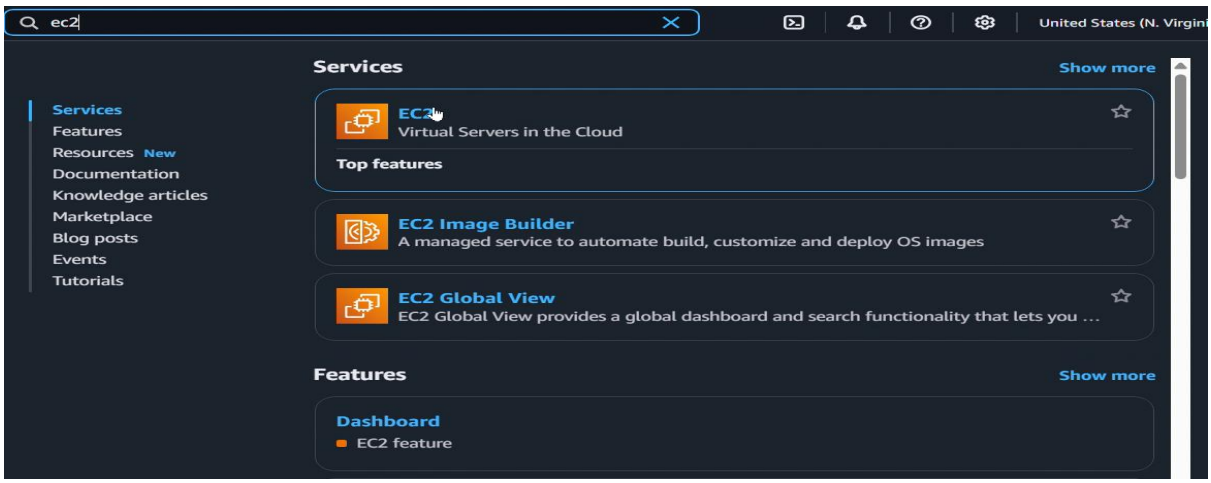
- Note: Load your Flask app and Html files into GitHub repository.



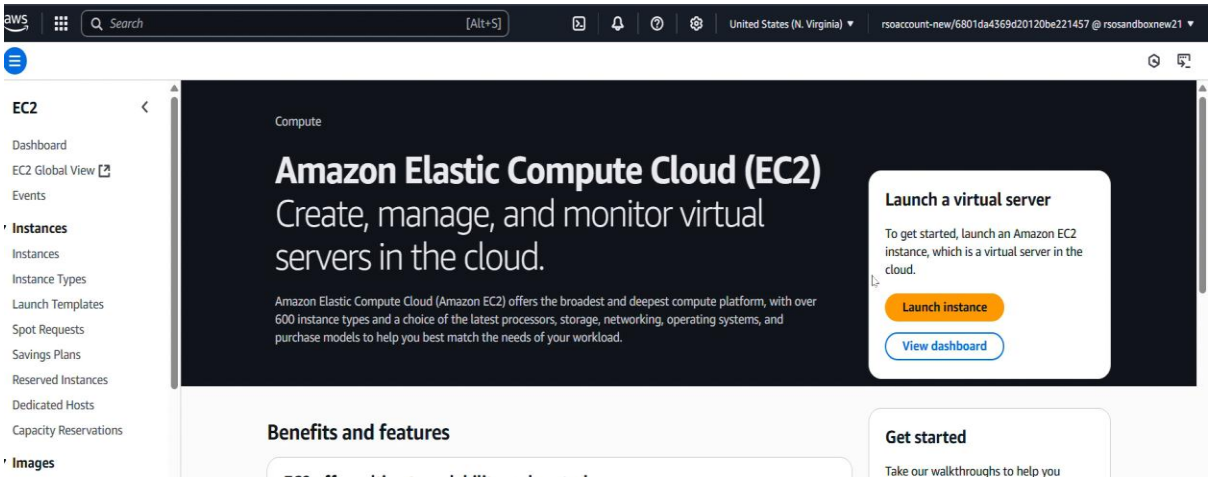


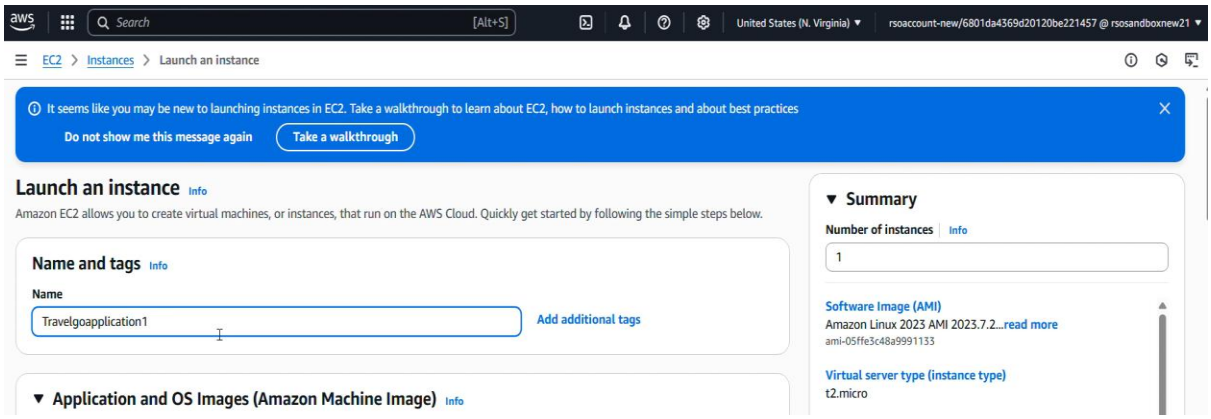
- **Activity 6.1: Launch an EC2 instance to host the Flask application.**

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

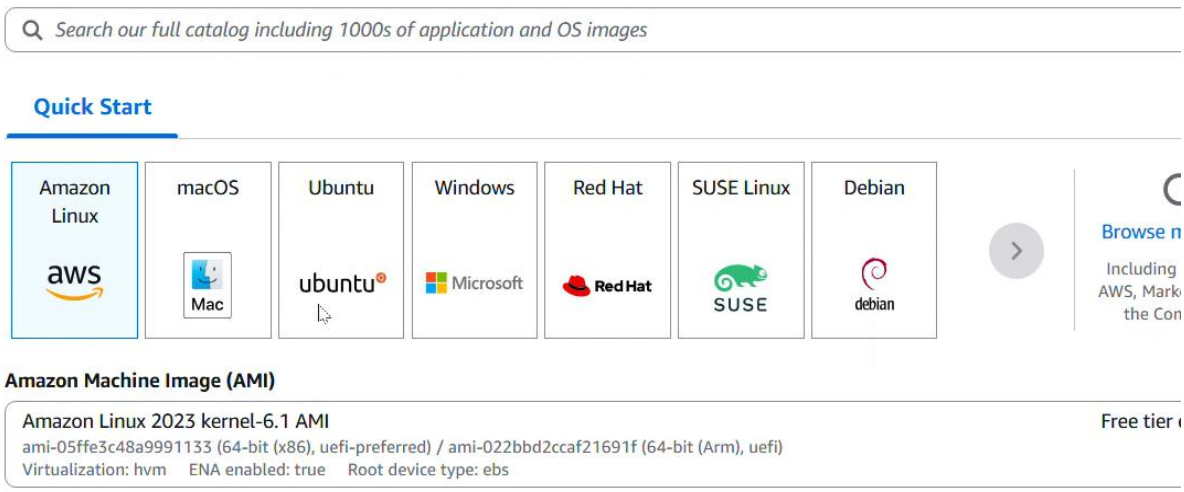


● Click on Launch instance to launch EC2 instance.

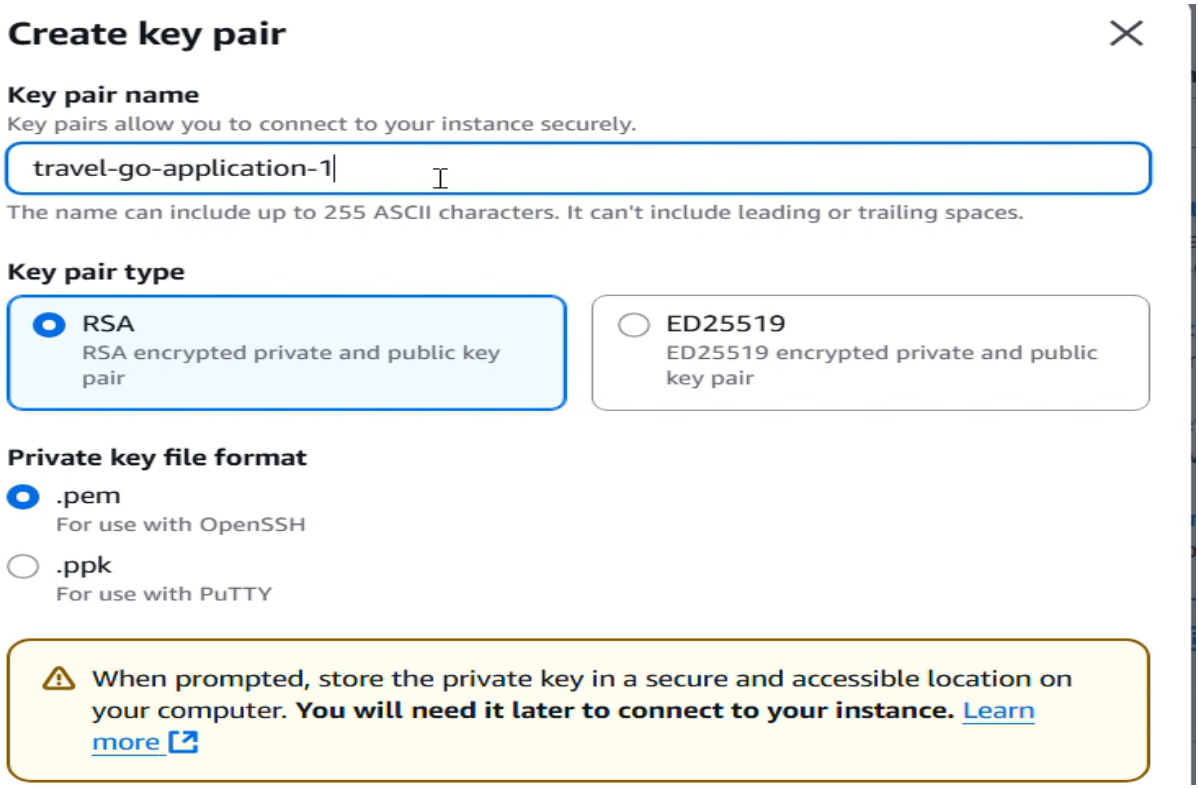




● Choose Amazon Linux 2 or Ubuntu as the AMI and t2.micro as the instance type (free-tier eligible).

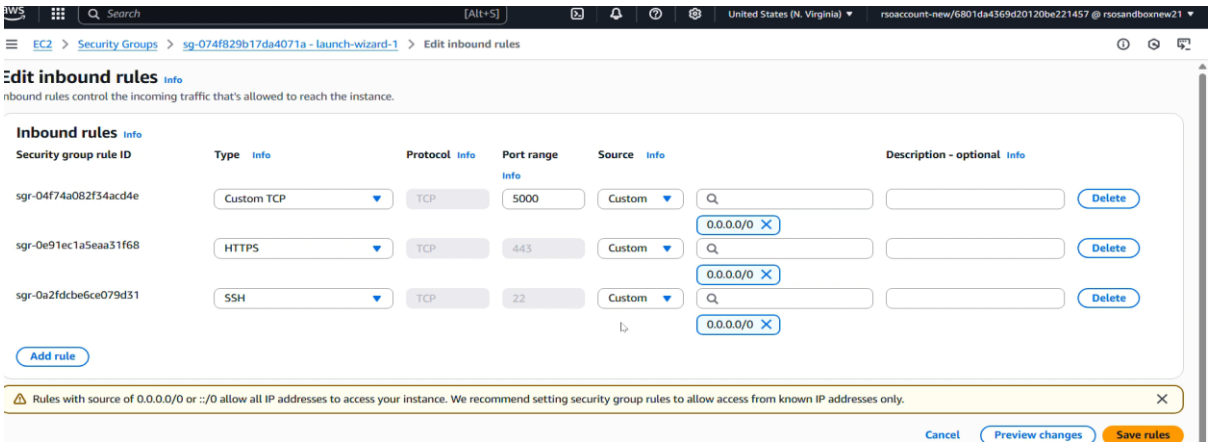


● Create and download the key pair for Server access.

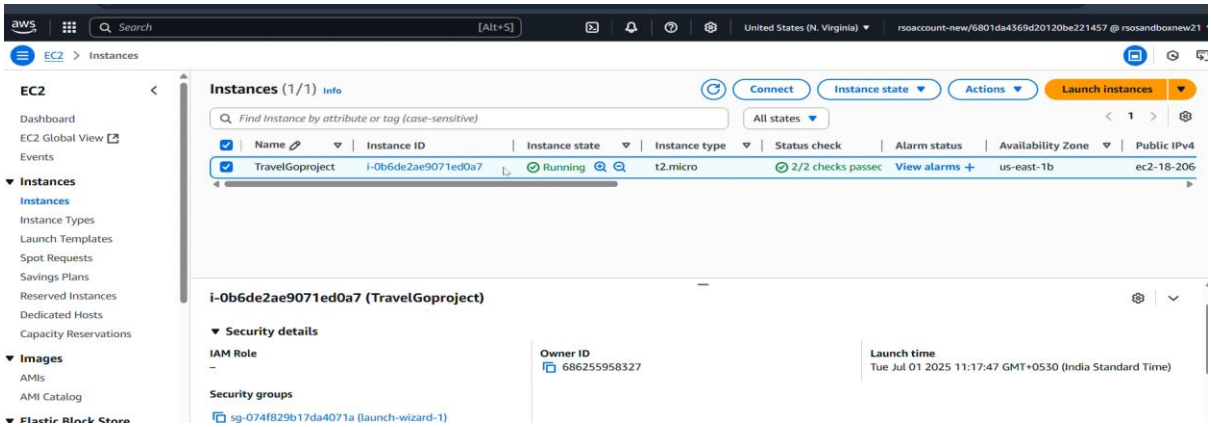


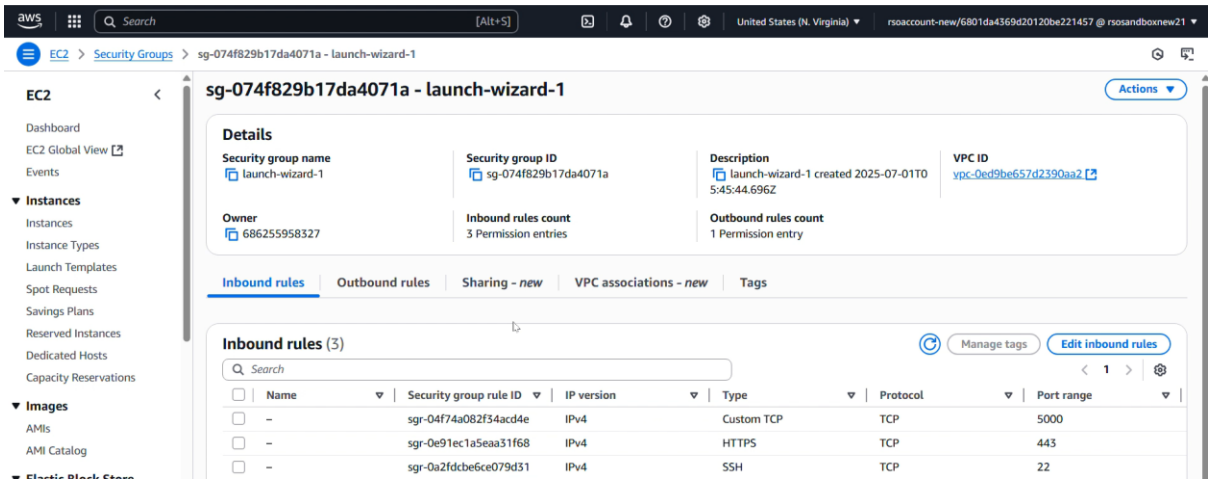
● **Activity 6.2:Configure security groups for HTTP, and SSH access.**

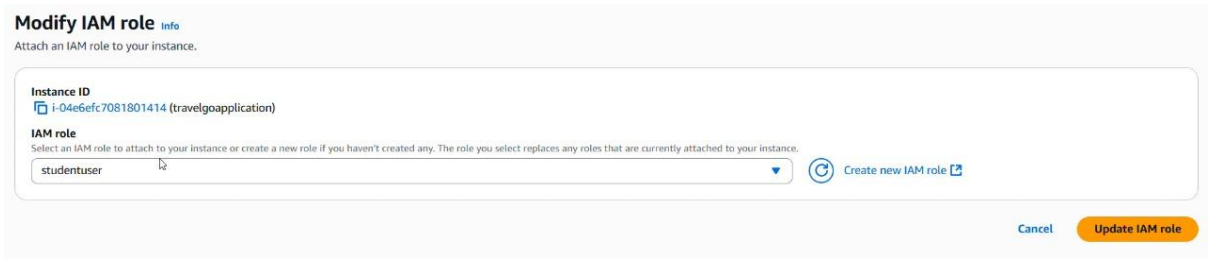
Edit inbound rules:



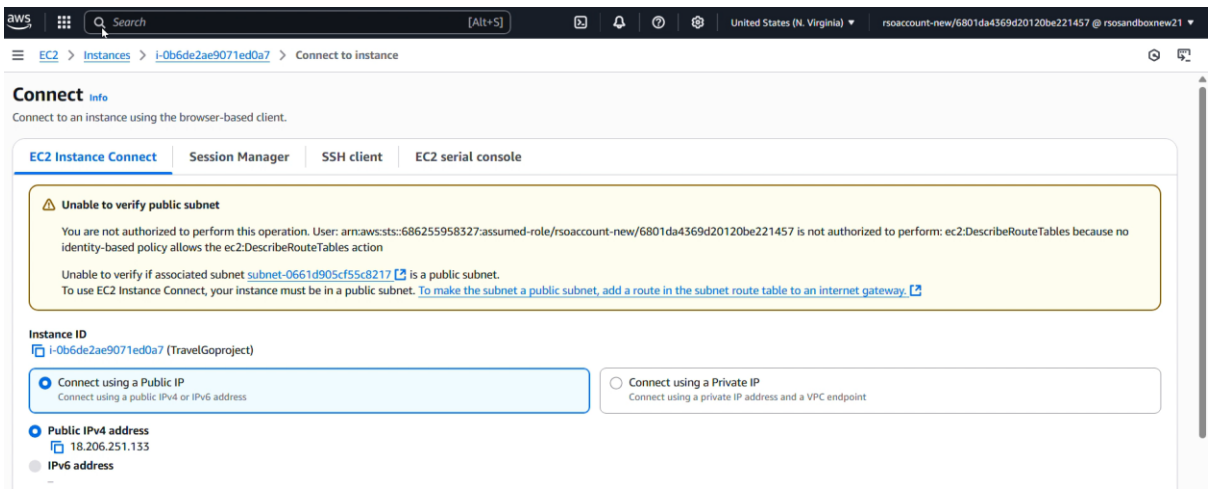
● To connect to EC2 using EC2 Instance Connect, start by ensuring that an IAM role is attached to your EC2 instance. You can do this by selecting your instance, clicking on Actions, then navigating to Security and selecting Modify IAM Role to attach the appropriate role. After the IAM role is connected, 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:

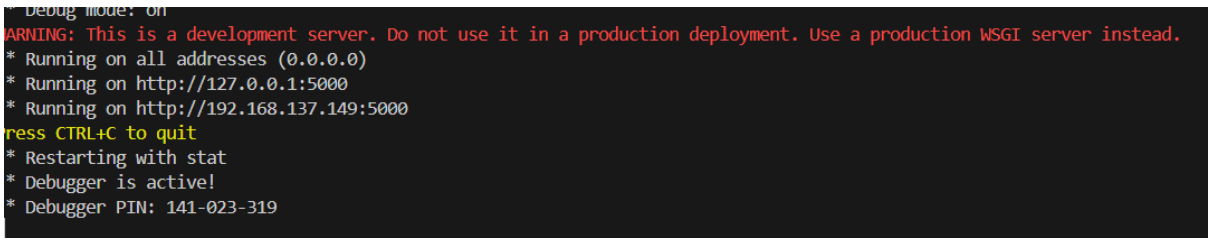
Run the Flask Application

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

Verify the Flask app is running:

http://your-ec2-public-ip

○ Run the Flask app on the EC2 instance



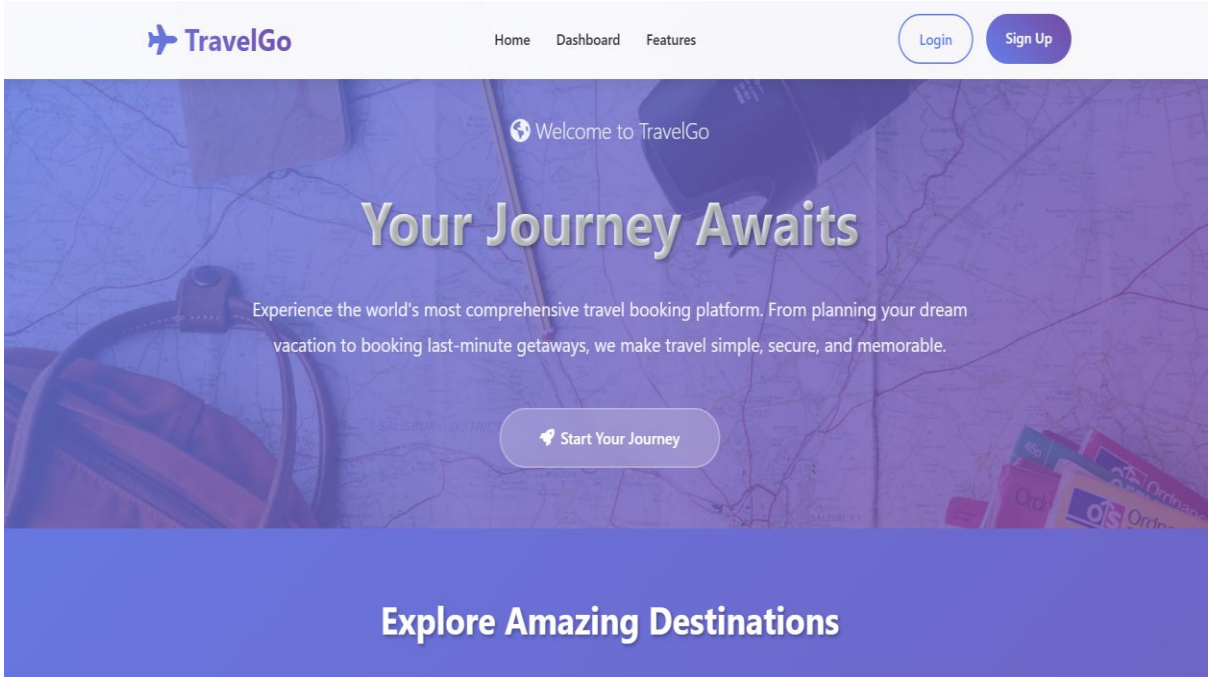
Access the website through:

PublicIPs: http://54.205.69.13:5000/

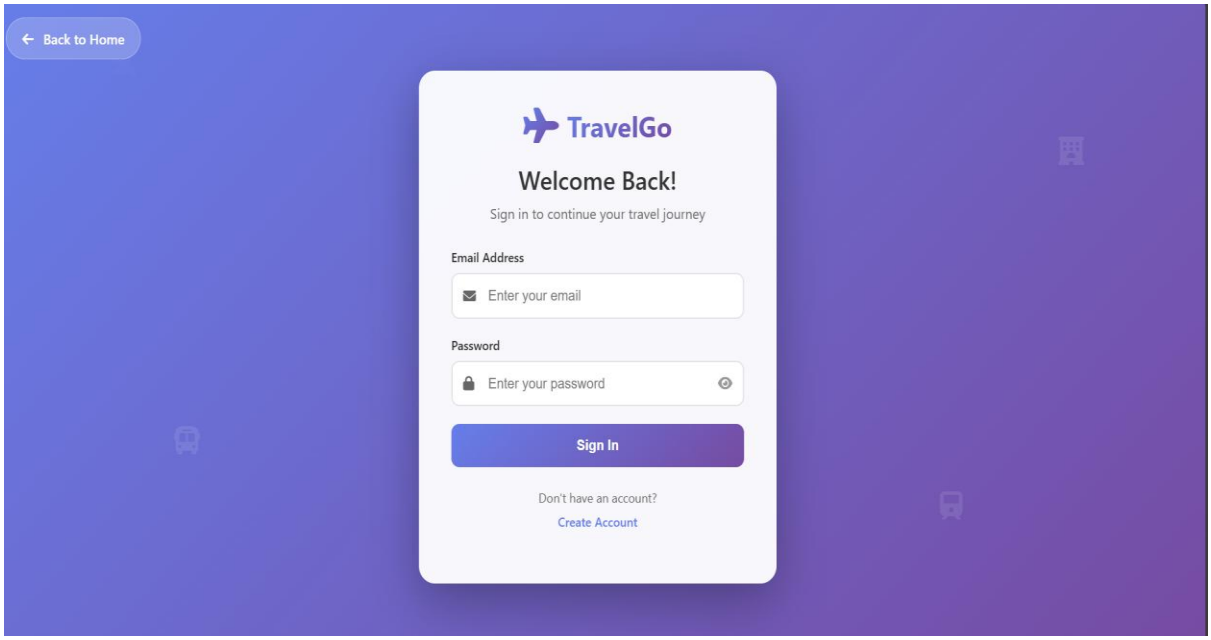
Milestone 8: Testing and Deployment

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

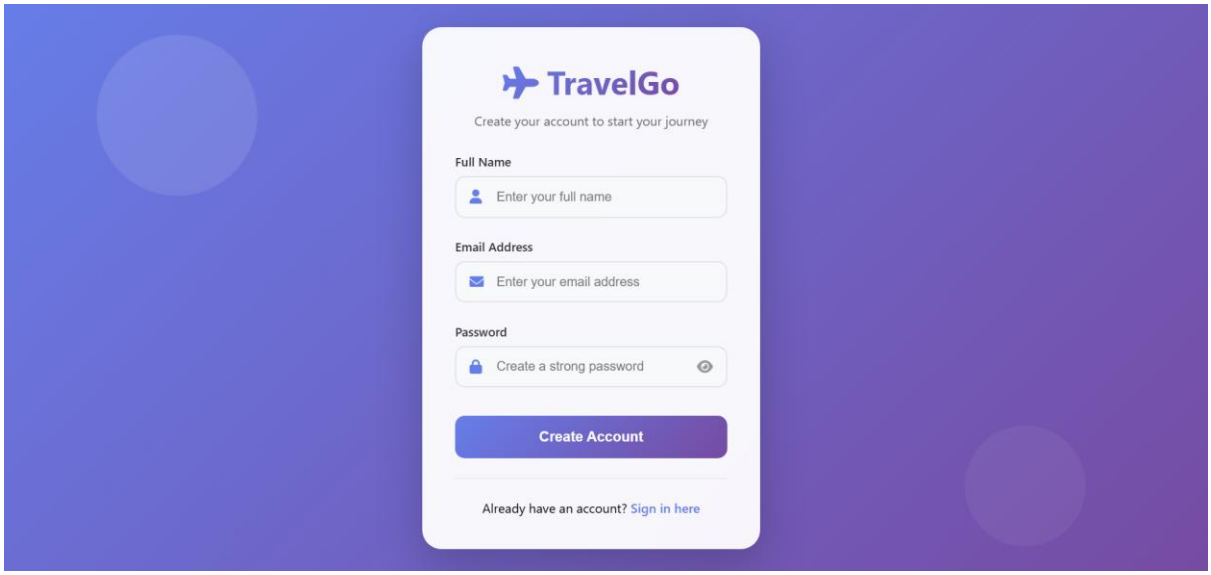
Index page:



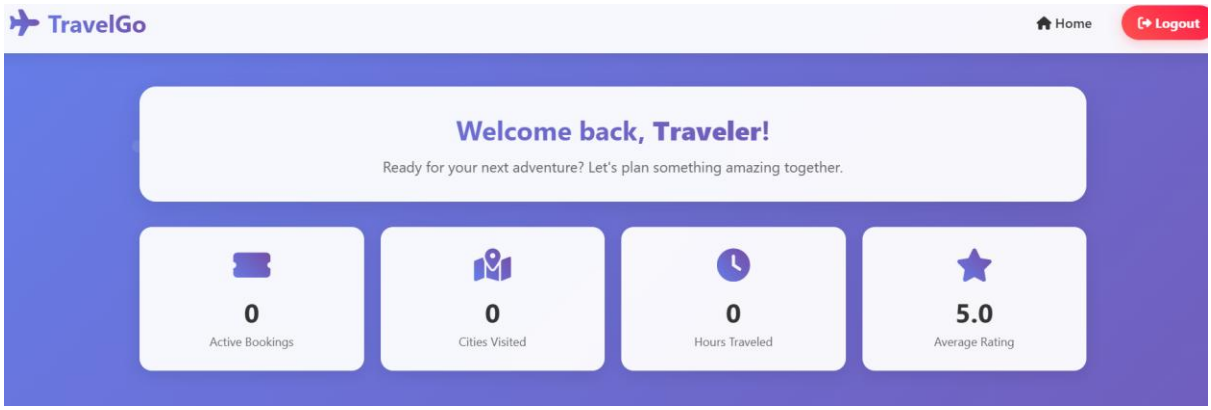
Login page:

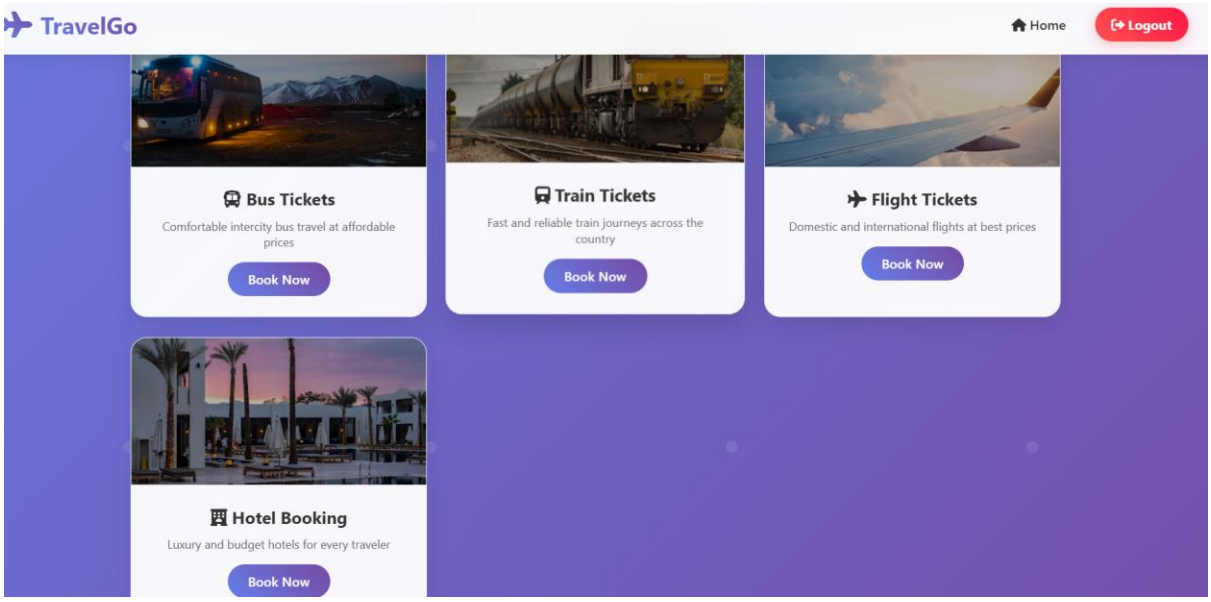


Register page:

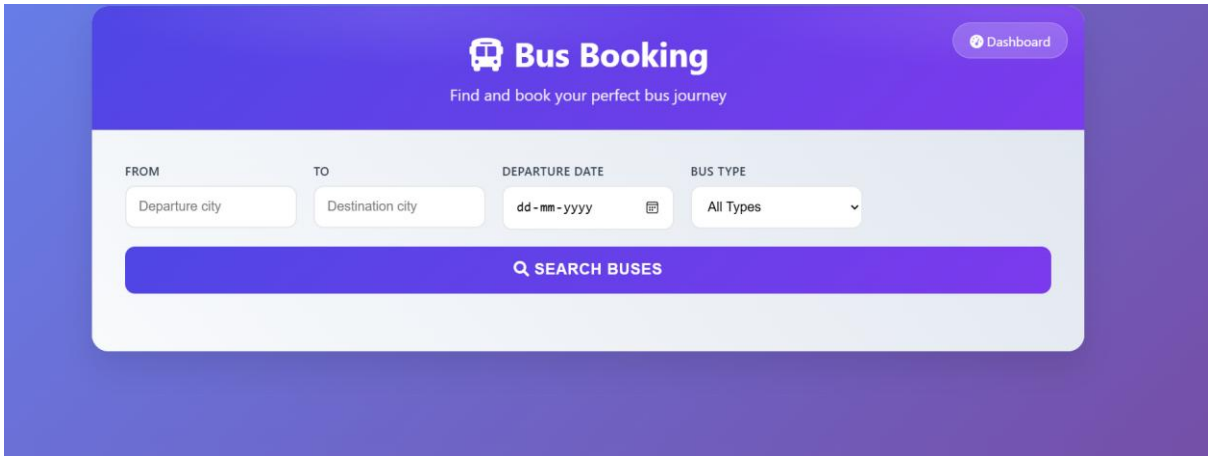


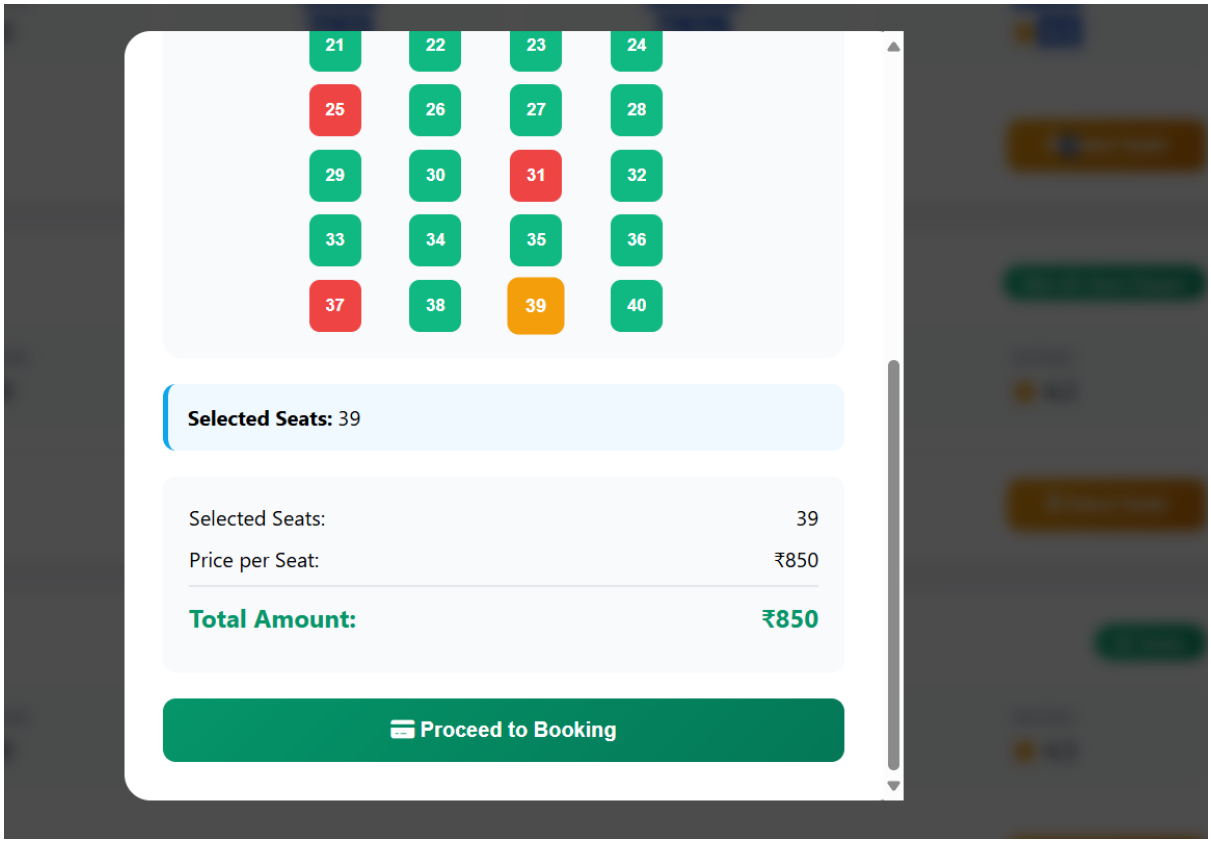
Dashboard page:



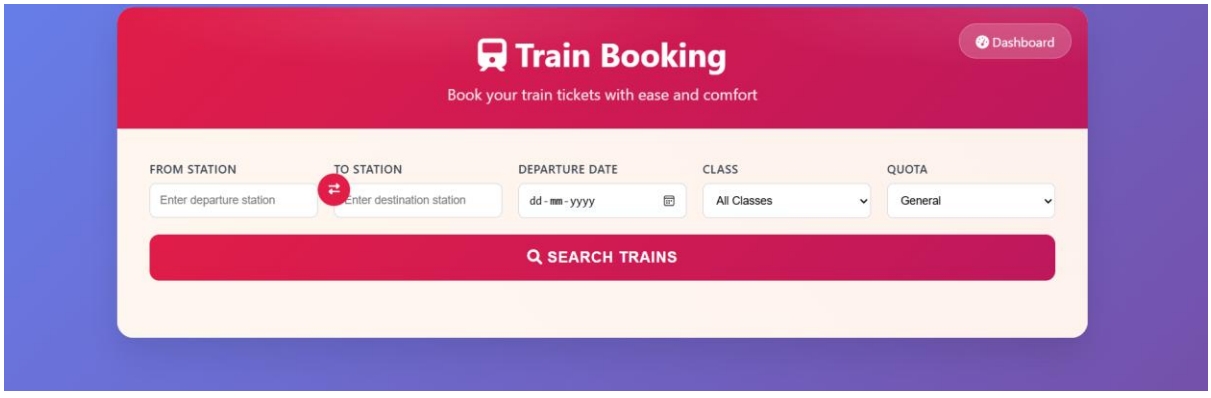


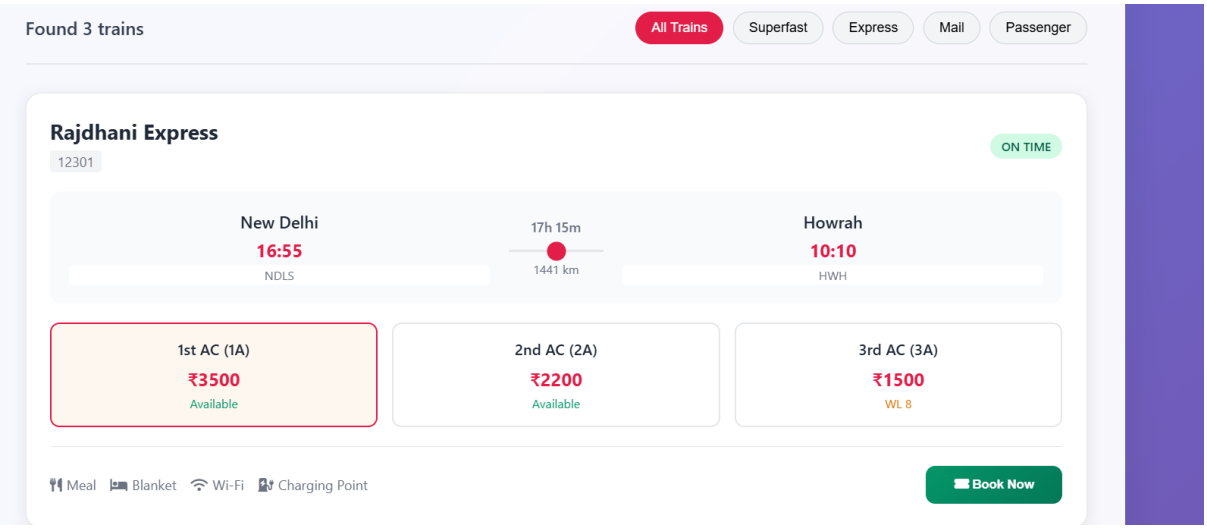
Bus booking page:



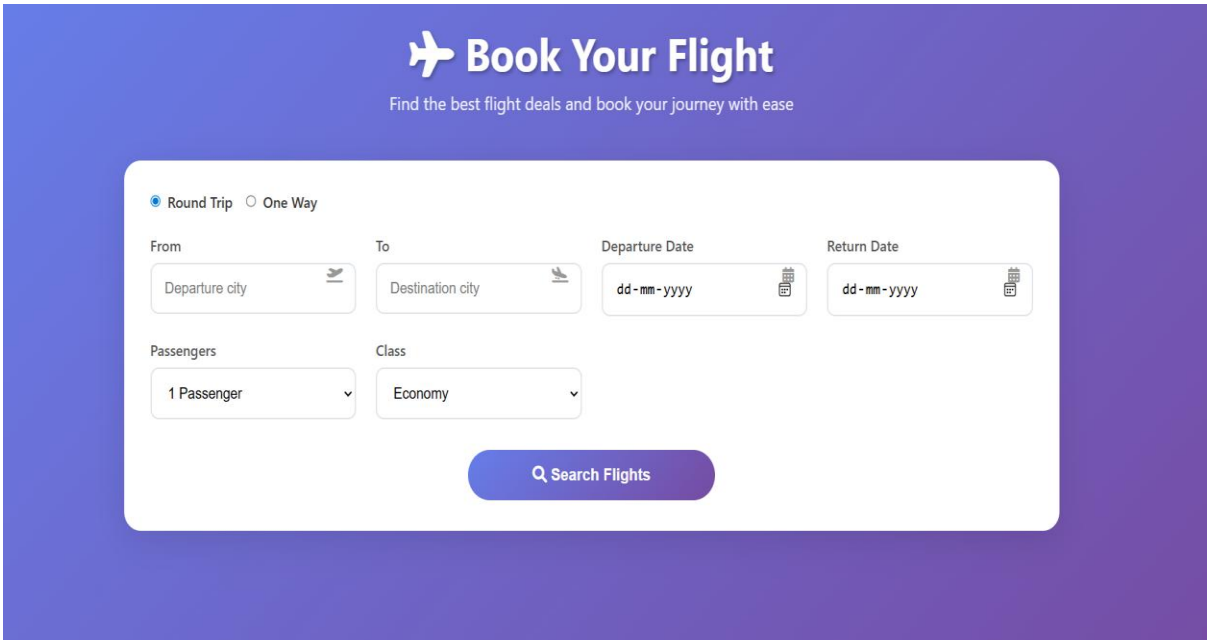


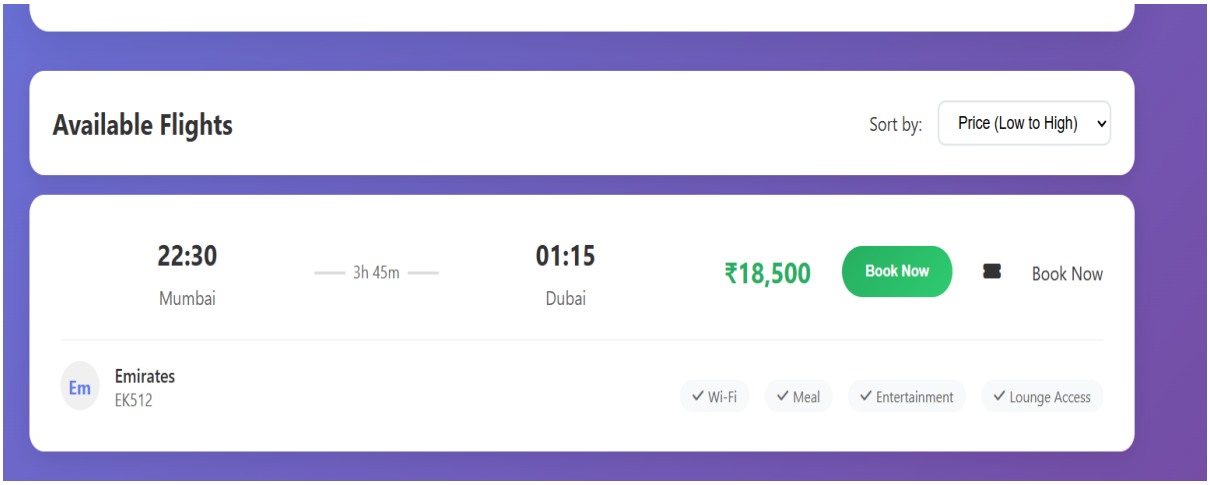
Train booking page:



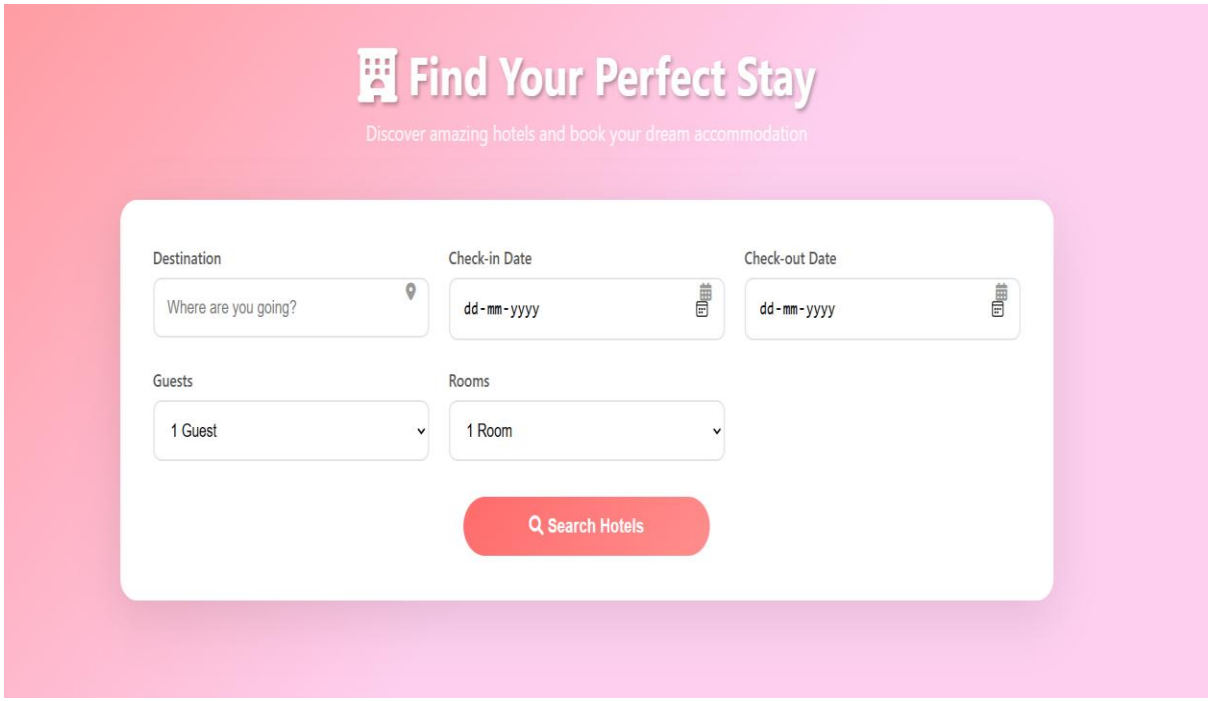


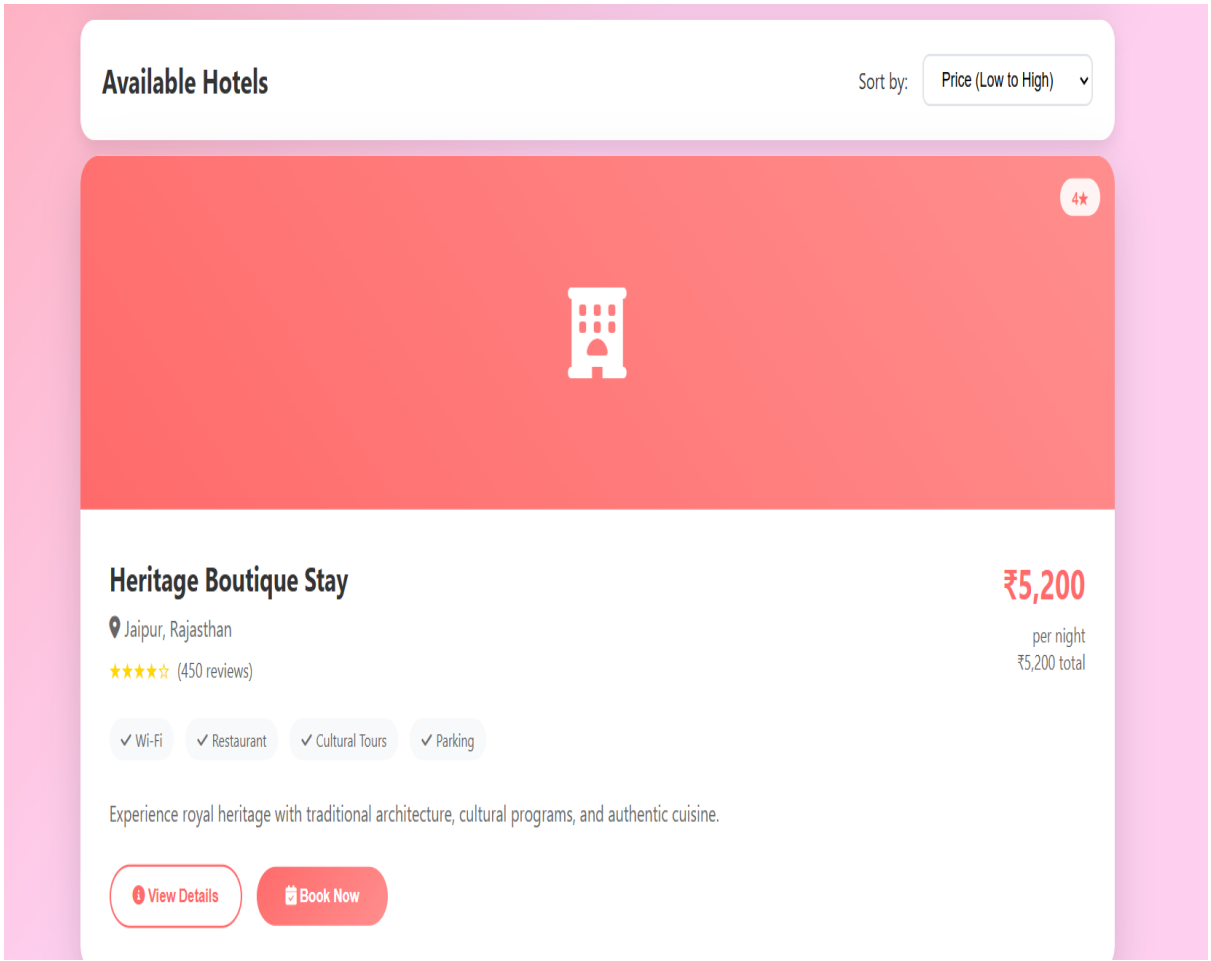
Flight booking page:

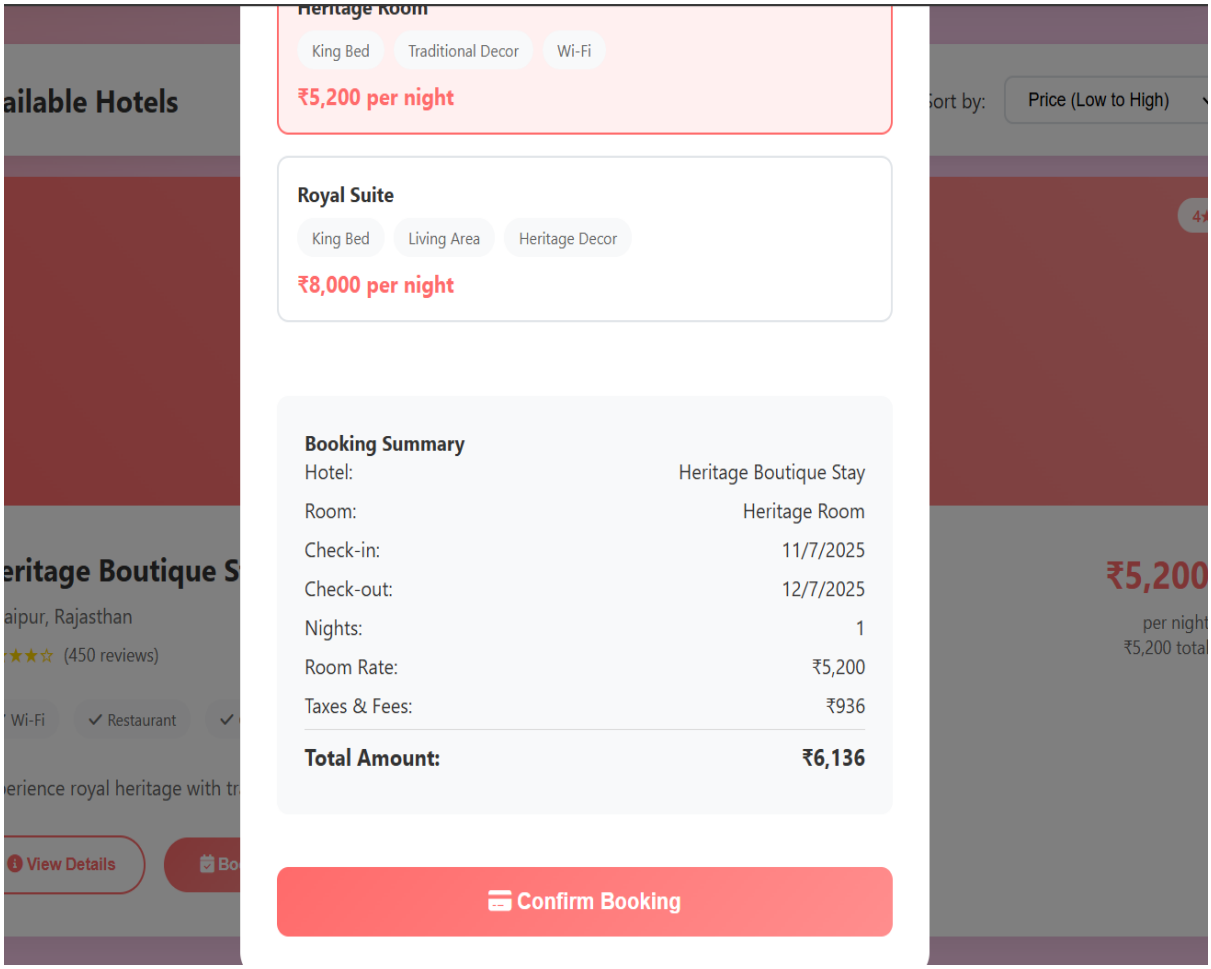




Hotel booking page:







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.