Travel Go: A Cloud-Based Instant Travel Reservation Platform

Project Overview:

Travel Go aims to streamline the travel planning process by offering a centralized platform to book buses, trains, flights, and hotels. Catering to the growing need for realtime and convenient travel solutions, it utilizes cloud infrastructure. The application employs Flask for backend services, AWS EC2 for scalable hosting, DynamoDB for fast data handling, and AWS SNS for immediate notifications. Users can sign up, log in, search, and book transport and accommodations. Additionally, they can review booking history and pick seats interactively, ensuring a user-friendly and responsive system.

Use Case Scenarios:

Scenario 1: Instant Travel Booking Experience

After logging into Travel Go, the user picks a travel type (bus/train/flight/hotel) and fills in preferences. Flask manages backend operations by retrieving matching options from DynamoDB. Once a booking is confirmed, AWS EC2 ensures swift performance even under high usage. This real-time setup allows users to secure bookings efficiently, even during rush periods.

Scenario 2: Instant Email Alerts

Upon booking confirmation, Travel Go uses AWS SNS to instantly send emails with booking info. For example, when a flight is booked, Flask handles the transaction and SNS notifies the user via email. The booking details are saved securely in DynamoDB. This smooth integration boosts reliability and enhances user confidence.

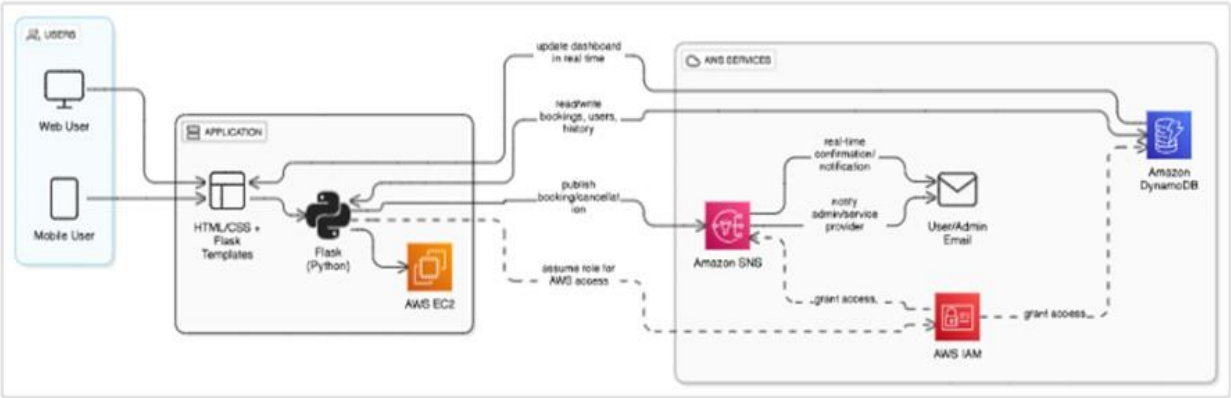
Scenario 3: Easy Booking Access and Control

Users can return anytime to manage previous bookings. For instance, a user checks hotel reservations made last week. Flask quickly fetches this from DynamoDB. Thanks to its cloud-first approach, Travel Go provides uninterrupted access, letting users easily cancel or make new plans. EC2 hosting guarantees consistent performance across multiple users.

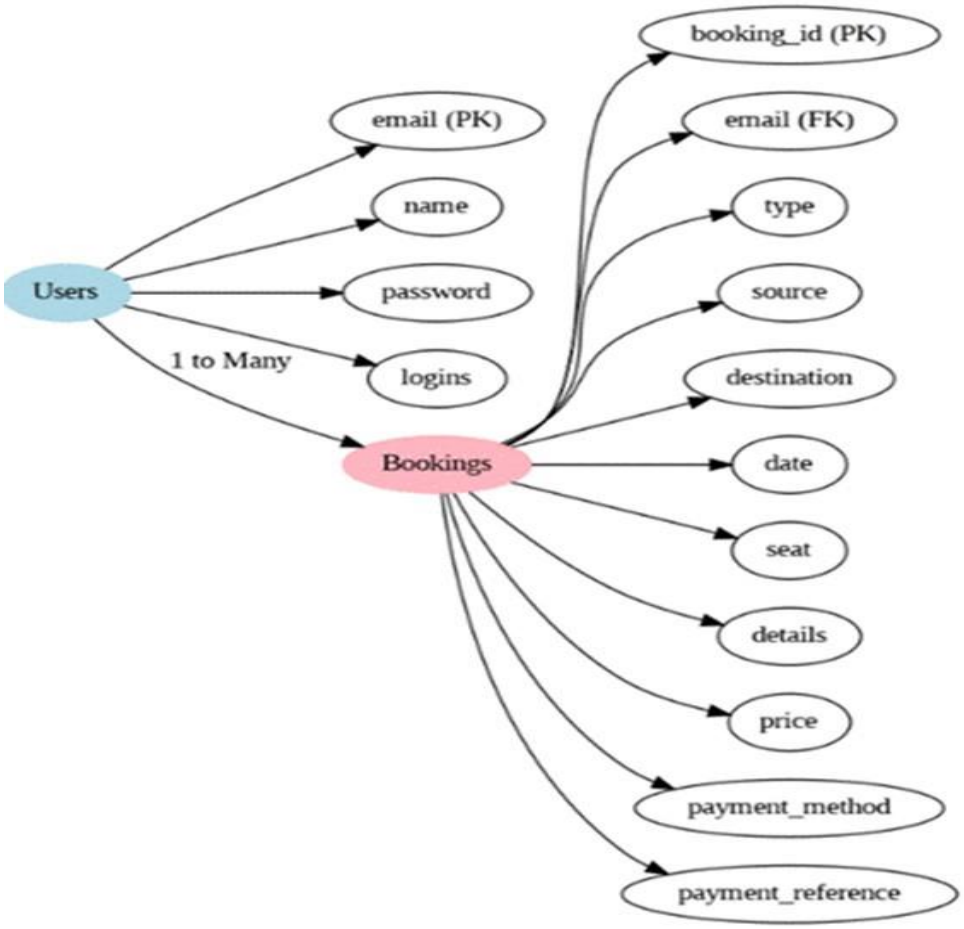
AWS Architecture Diagram

To build a scalable and responsive travel booking platform, a well-structured cloud architecture is essential. Travel Go leverages core AWS services such as EC2, DynamoDB, IAM, and SNS to ensure high availability and fault tolerance. The architecture is designed to support real-time booking, seamless data flow, and secure operations. Below is the visual representation of the AWS setup used in the project.

AWS Architecture



Entity Relationship Diagram



Requirements:

1. AWS Account Configuration
2. IAM Basics Overview
3. EC2 Introduction
4. DynamoDB Fundamentals
5. SNS Concepts
6. Git Version Control

Development Steps:

1. AWS Account Setup and Login

- Activity 1.1: Register for an AWS account if you haven't already.
- Activity 1.2: Access the AWS Management Console using your credentials.

2. DynamoDB Database Initialization

- Activity 2.1: Create a DynamoDB table.
- Activity 2.2: Define attributes for users and travel bookings.

3. SNS Notification Configuration

- Activity 3.1: Create SNS topics to notify users upon booking.
- Activity 3.2: Add user and provider emails as subscribers for updates.

4. Backend Development using Flask

- Activity 4.1: Build backend logic with Flask.
- Activity 4.2: Connect AWS services using the boto3 SDK.

5. IAM Role Configuration

- Activity 5.1: Set up IAM roles with specific permissions.
- Activity 5.2: Assign relevant policies to each role.

6. EC2 Instance Launch

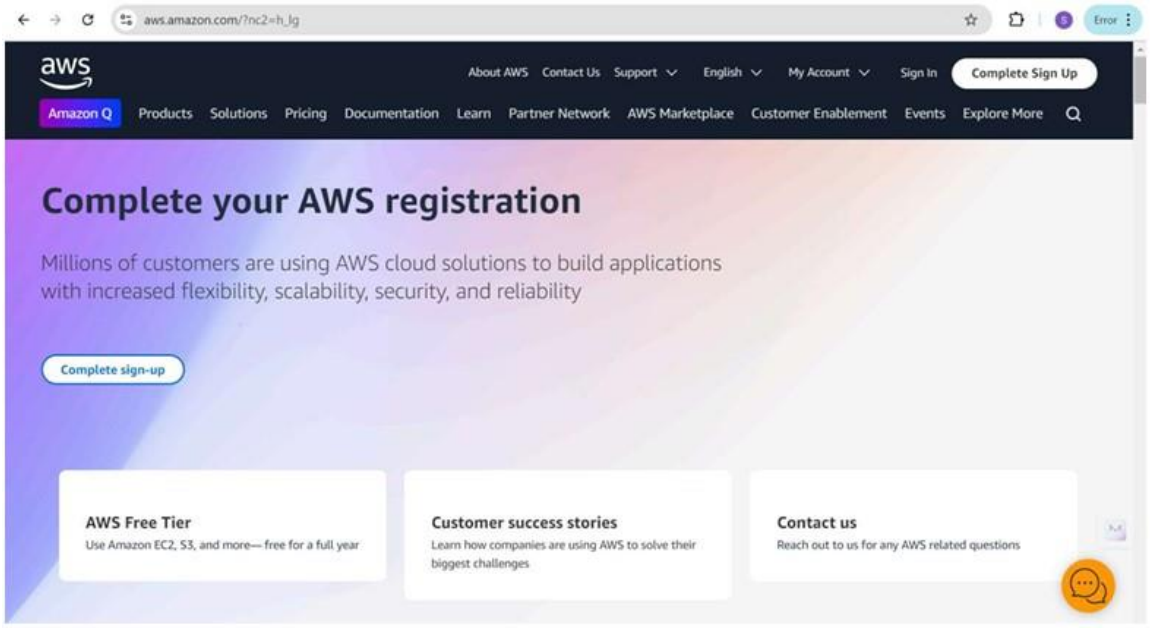
- Activity 6.1: Start an EC2 instance to host the backend.
- Activity 6.2: Set up security rules to allow HTTP and SSH traffic.

7. Flask Application Deployment

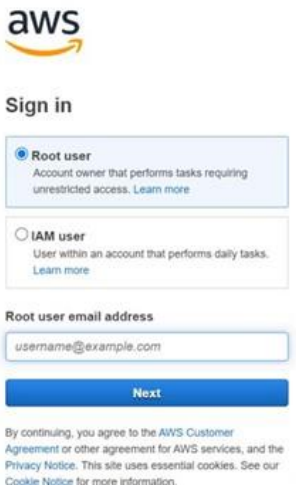
- Activity 7.1: Upload your Flask files to the EC2 instance.
- Activity 7.2: Launch your Flask server from the instance.

8. Testing and Launch

1. AWS Account Setup and Login



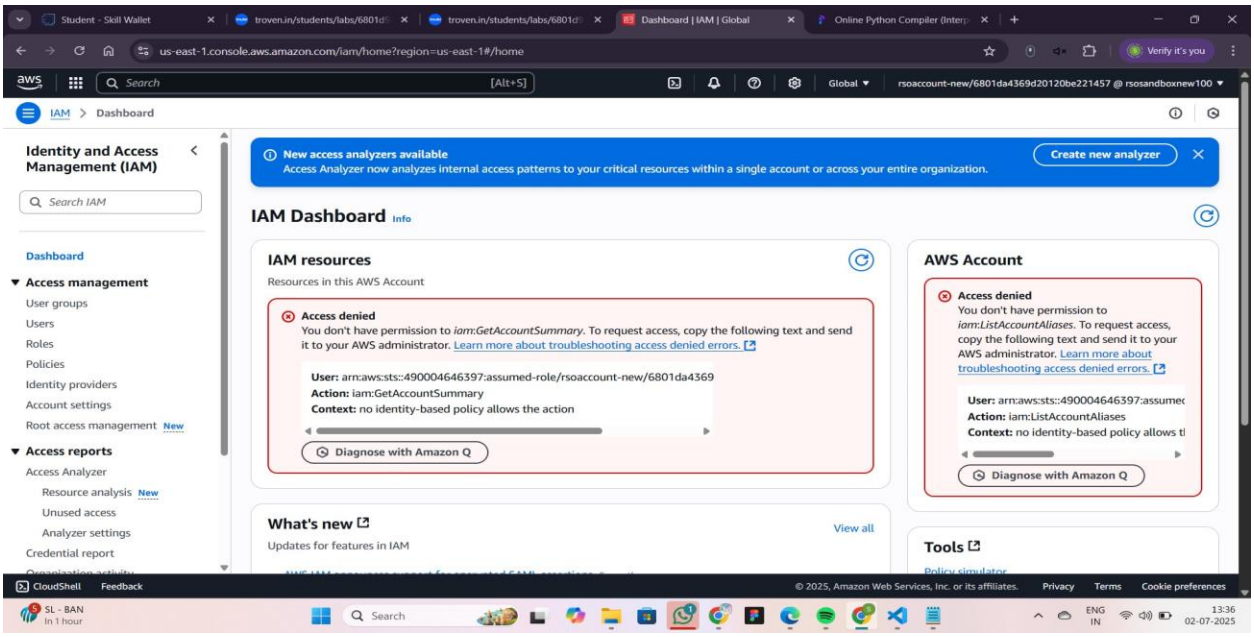
2. Log in to the AWS Management Console

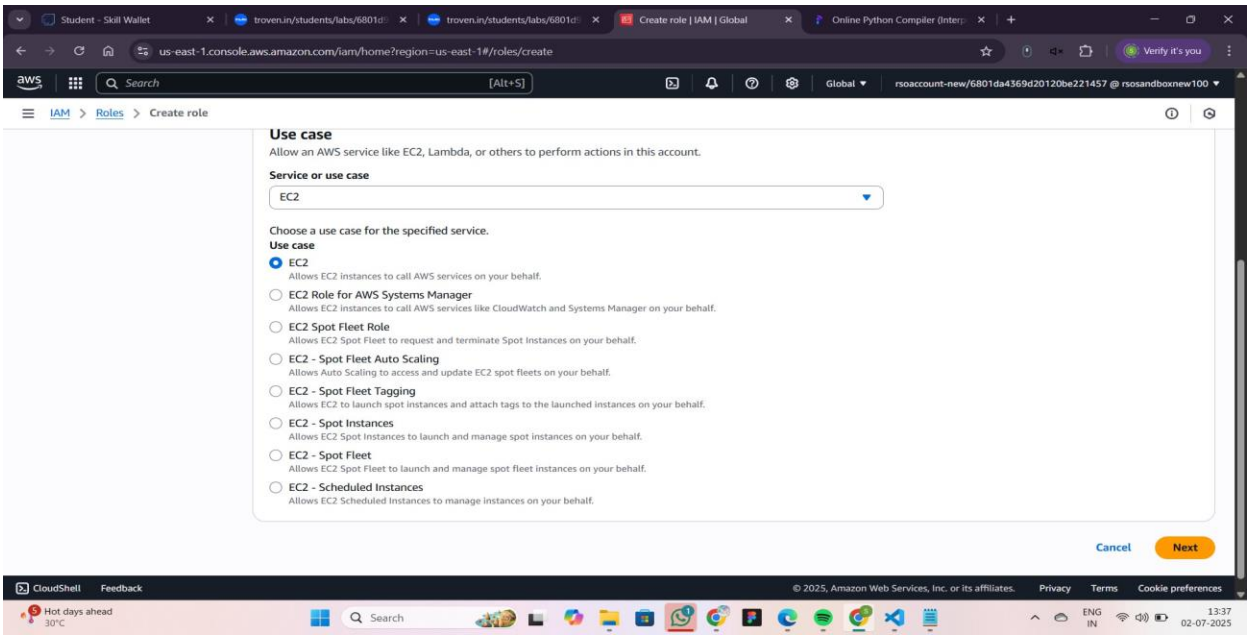


After logging into the AWS Console:

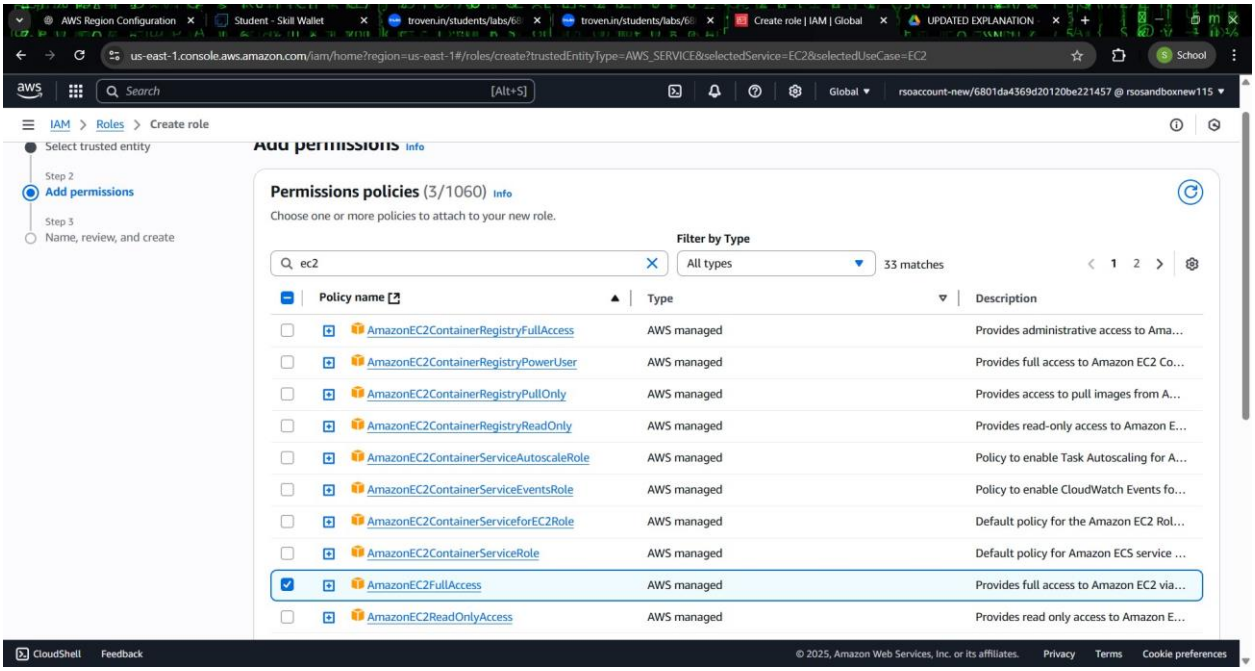
Once you're logged in, you can access a wide range of AWS services from the dashboard. Use the search bar at the top to quickly navigate to services like EC2, DynamoDB, SNS, or IAM as needed for your project setup.

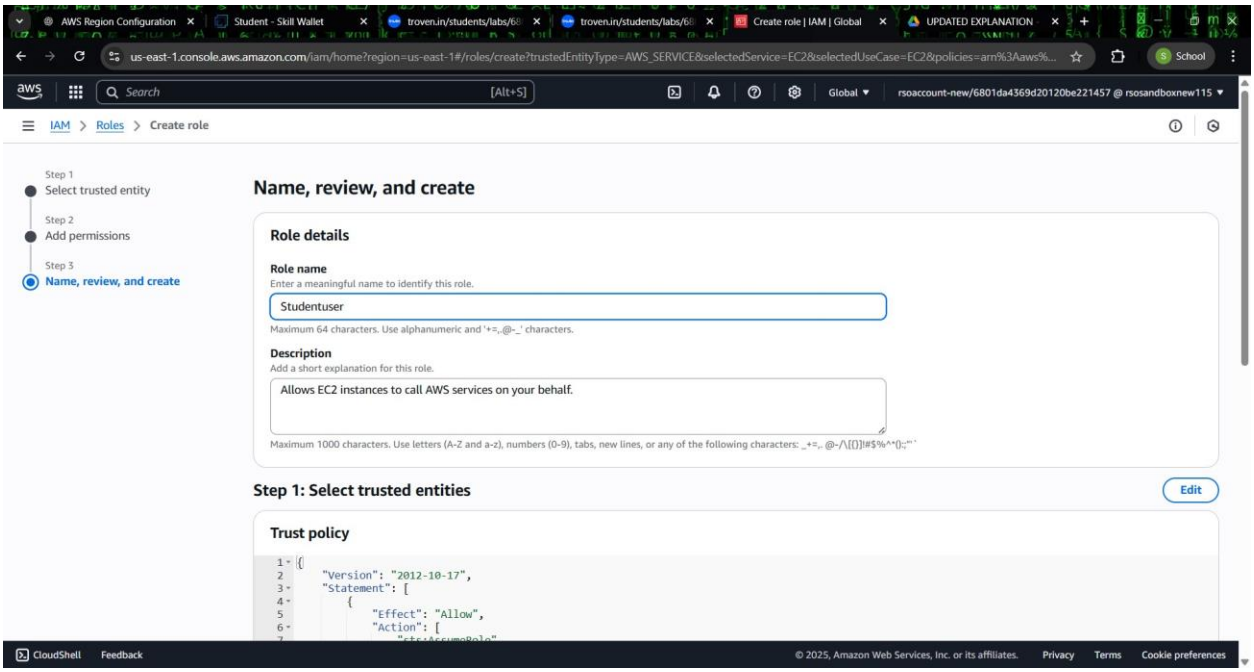
3. IAM Roles Creation





Before assigning permissions, ensure that the IAM role is created and ready to be attached with appropriate AWS-managed policies for each required service.



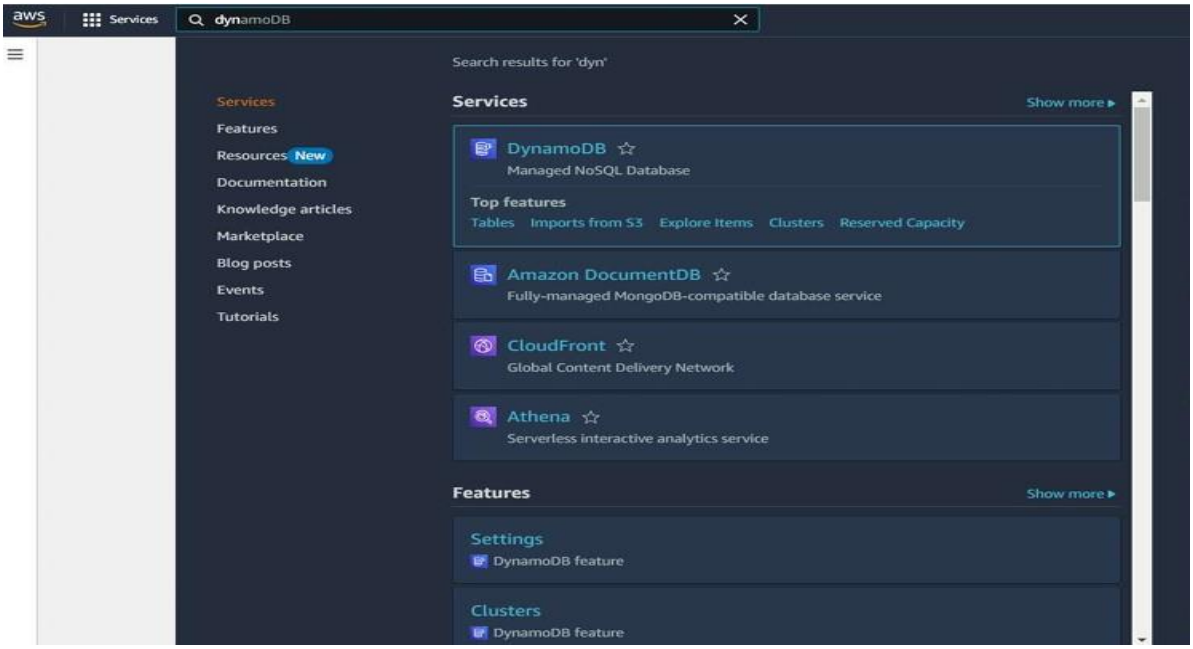


4. DynamoDB Database Creation and Setup

Familiarize yourself with the DynamoDB service interface. DynamoDB offers a fast and flexible NoSQL database ideal for serverless applications like Travel Go. You will now proceed to create tables to store user and booking data.In this setup, you'll define partition keys to uniquely identify records—such as using “Email” for users or “Train Number” for trains. Properly configuring your attributes is crucial to ensure efficient querying and data retrieval. DynamoDB’s schema-less

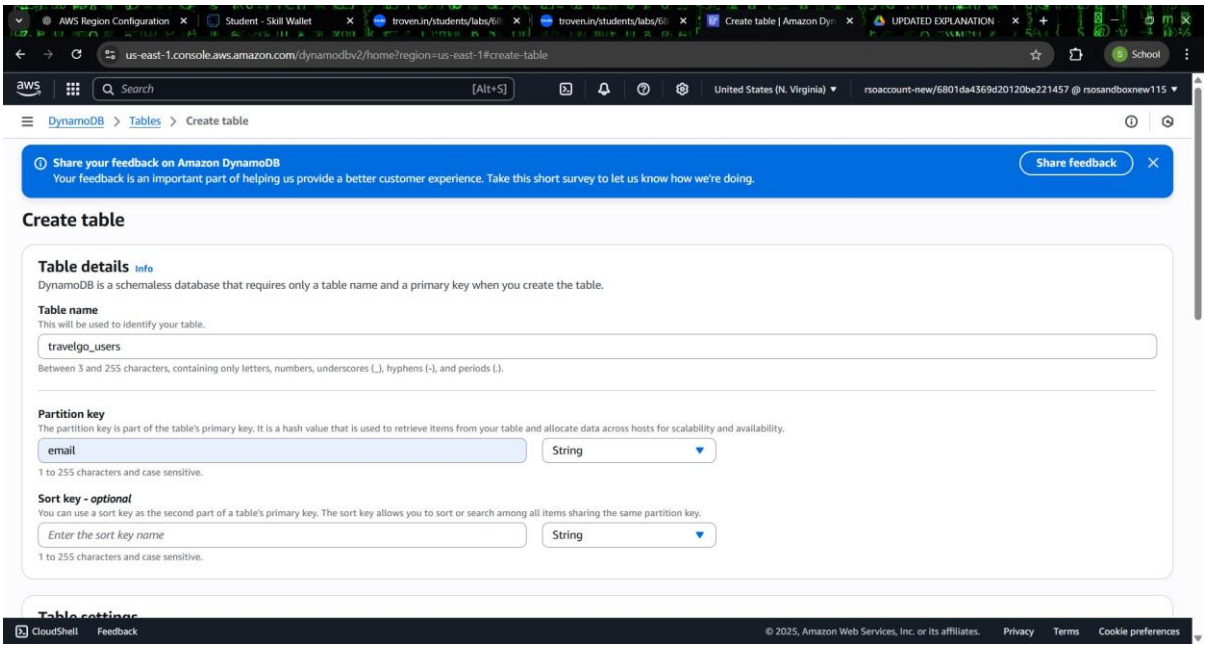
design allows flexibility while maintaining performance, making it well-suited for handling diverse booking data types in Travel Go.

• In the AWS Console, navigate to DynamoDB and click on create tables



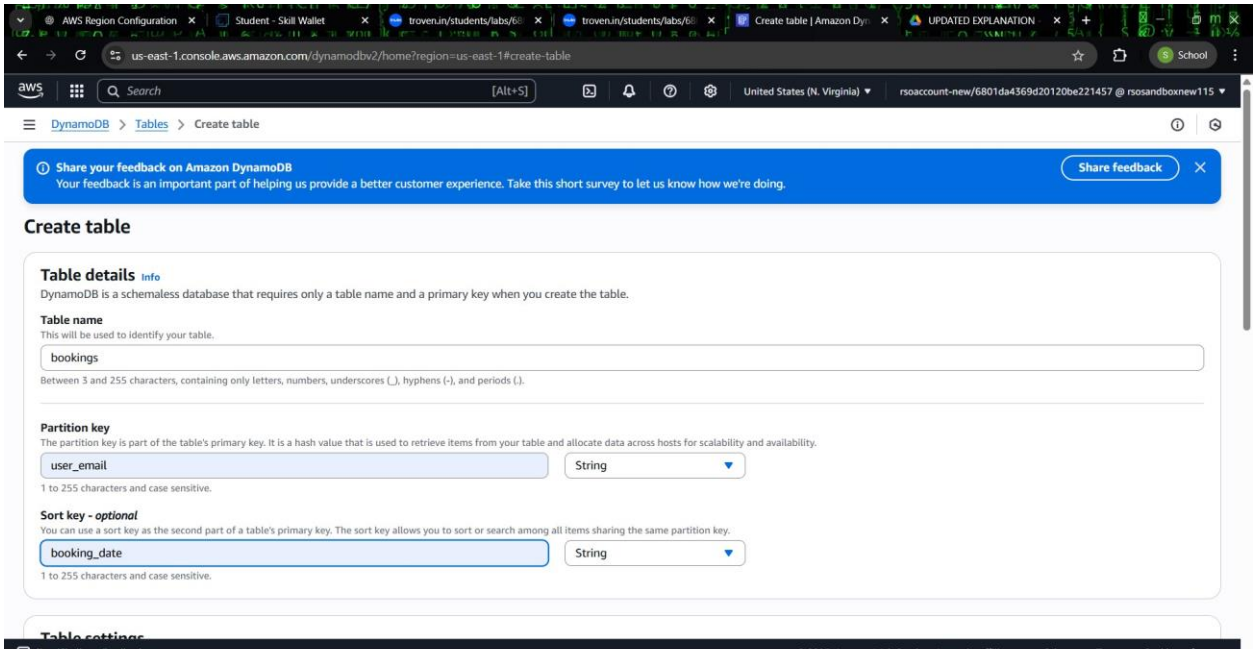
Create a DynamoDB table for storing registration details and book Requests

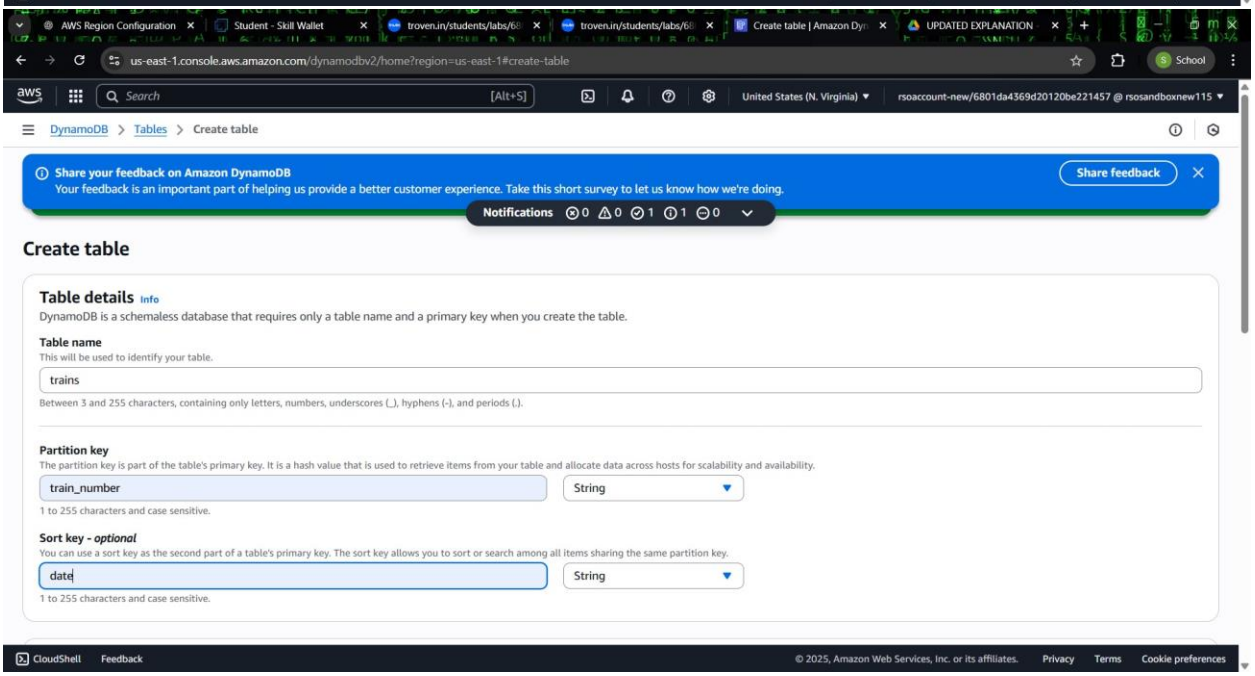
1. Create Users table with partition key “Email” with type String and click on create tables.

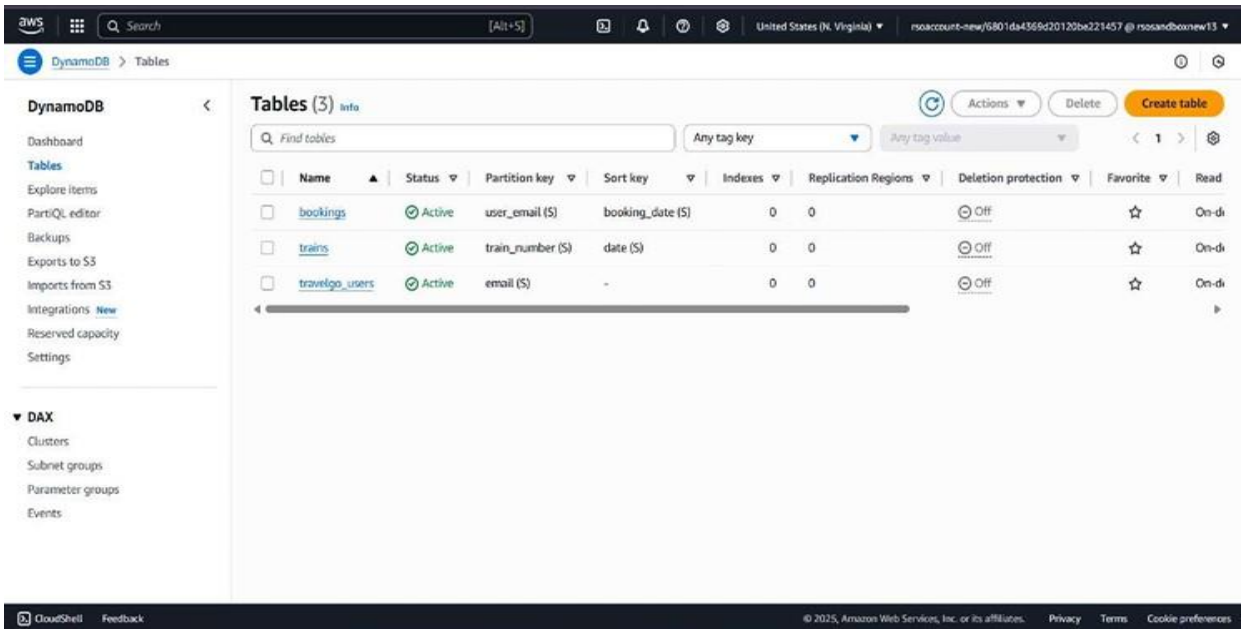


Repeat the same procedure to create additional tables:

• **Train Table:** Use “Train Number” as the partition key to uniquely identify each train. • **Bookings Table:** Set “User Email” as the partition key and “Booking Date” as the sort key to efficiently organize and retrieve individual user booking records based on date.

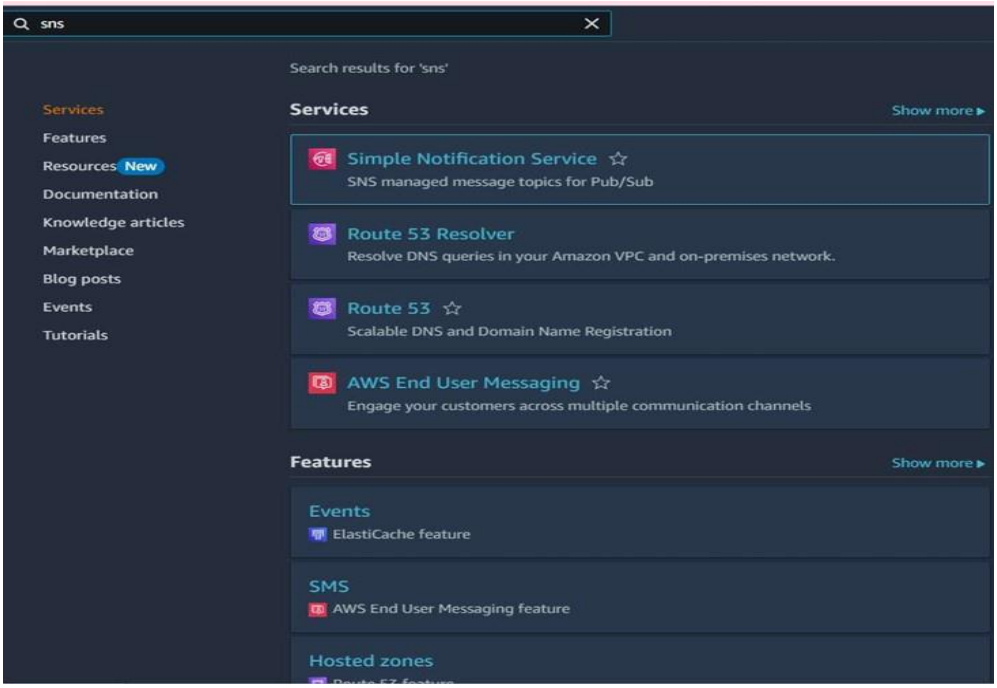


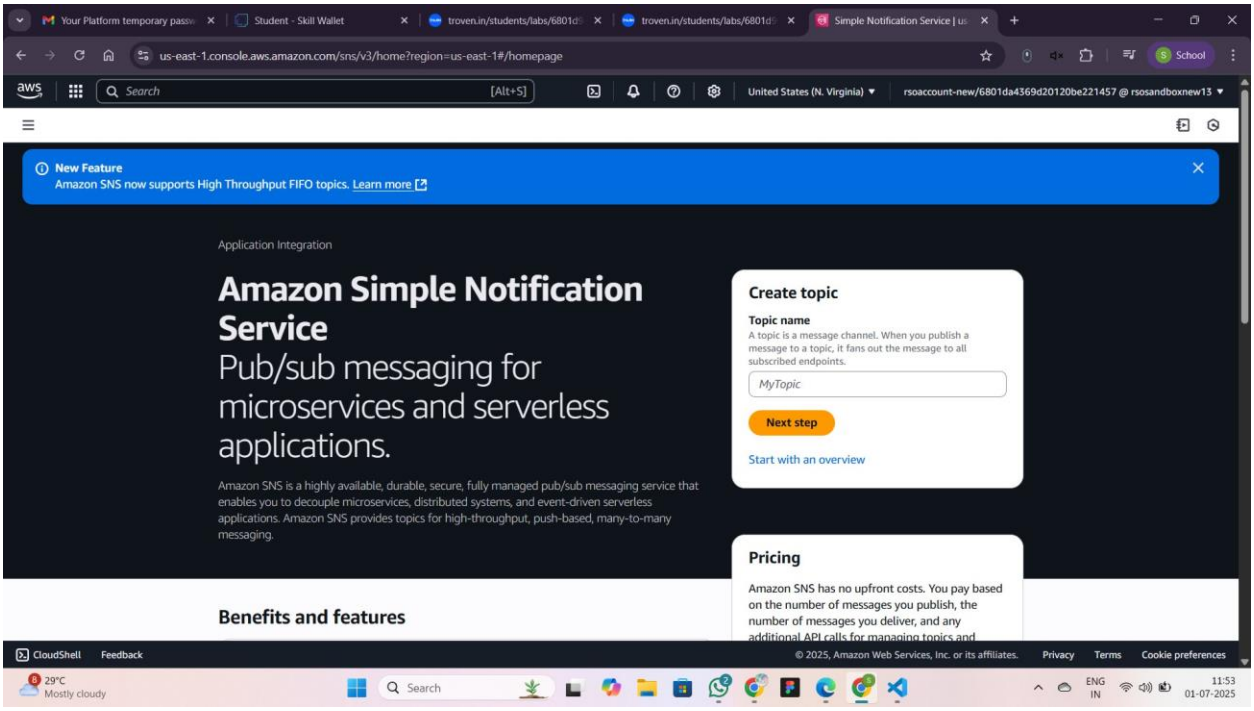




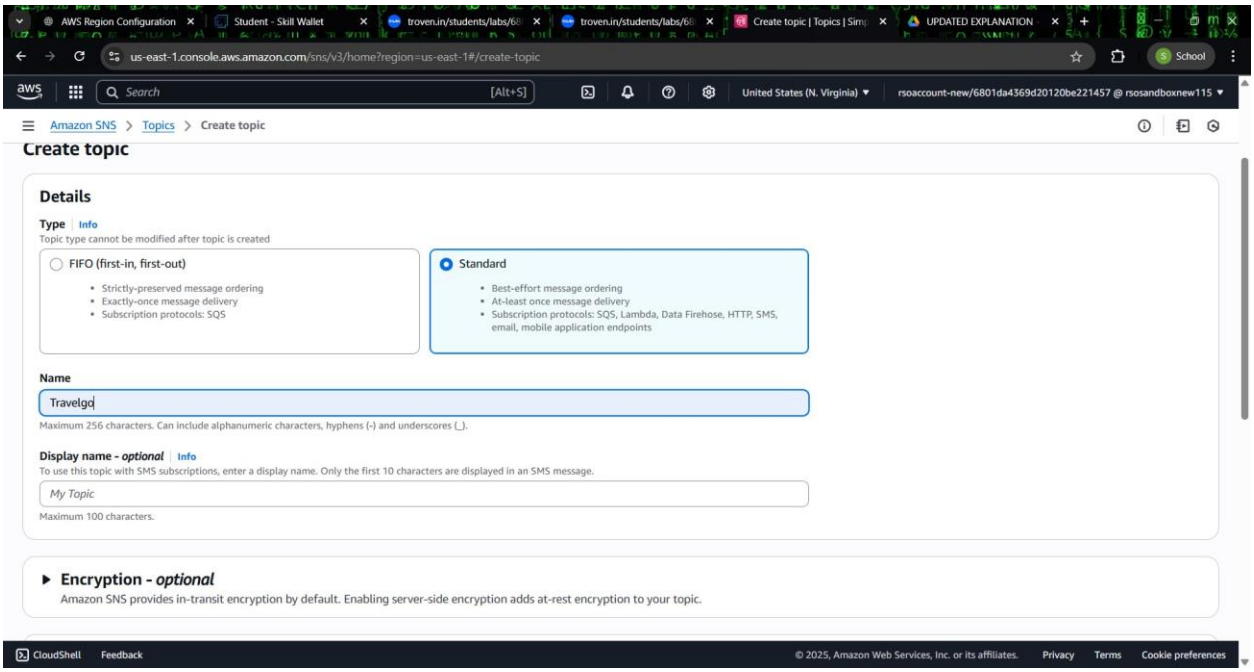
5. SNS Notification Setup

1. Create SNS topics for sending email notifications to users.





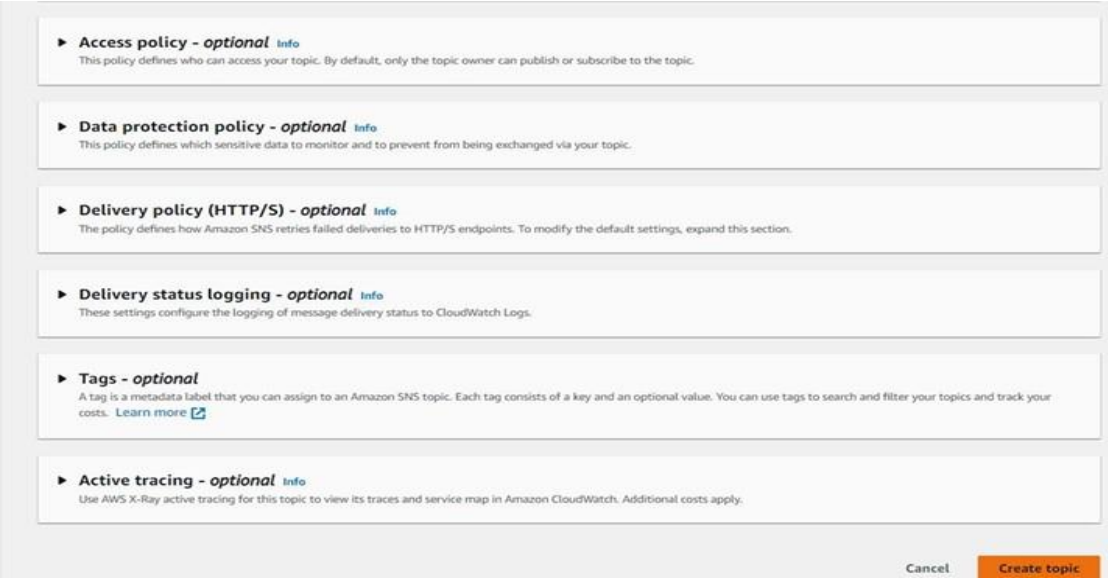
1. Click on Create Topic and choose a name for the topic.



2. Click on create topic

To begin configuring notifications, navigate to the SNS dashboard and click on “Create Topic.” Choose the topic type (Standard or FIFO) based on your requirements. Provide a meaningful name

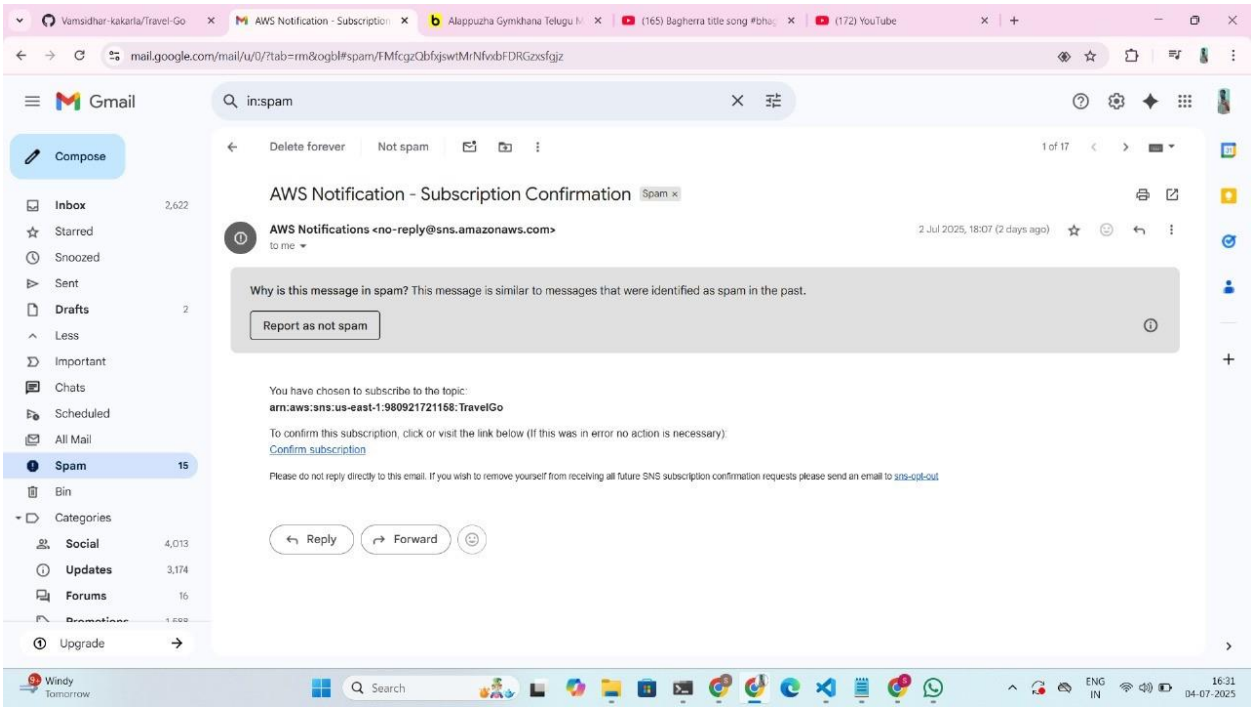
for the topic that reflects its purpose (e.g., booking-alerts). This topic will serve as the communication channel for sending notifications to subscribed users.



3. Configure the SNS topic and note down the **Topic ARN**.
4. Click on create subscription.

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

Once the email subscription is confirmed, the endpoint becomes active for receiving notifications. This ensures that users will instantly get booking confirmations or alerts. You can manage or remove subscriptions anytime via the SNS dashboard. It's recommended to test the setup by publishing a sample message to verify successful delivery.



Backend Configuration and Coding

```
1  from flask import Flask, render_template, request, redirect, url_for, session, flash, jsonify
2  from pymongo import MongoClient
3  from werkzeug.security import generate_password_hash, check_password_hash
4  from datetime import datetime
5  from bson.objectid import ObjectId
6  from bson.errors import InvalidId
```

- Flask: Initiates web application
- render_template: Loads Jinja2 HTML templates
- request: Collects input from forms/API
- redirect/url_for: Page redirection logic
- session: Keeps user-specific info
- jsonify: Returns data in JSON structure

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

Begin building the web application by initializing the Flask app instance using Flask(__name__), which sets up the core of your backend framework.

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

SNS connection:

This function sends alerts using AWS SNS by publishing a subject and message to a predefined topic. It uses sns_client.publish() with error handling to catch failures and print debug info. This ensures users receive real-time booking notifications.

```python
SNS_TOPIC_ARN = 'arn:aws:sns:us-east-1:490004646397:Travelgo:372db6a7-7131-4571-8397-28b62c9e08a8'

# 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.
```

Routes:

Route Overview of TravelGo:

The application begins with user onboarding through the Register and Login routes, securing user access. Once authenticated, users are redirected to the Dashboard, which serves as the central hub. From there, they can explore and book through Bus, Train, Flight, and Hotel modules. Each booking passes through a Confirmation step where details are reviewed before final submission. The Final Confirmation route stores the booking and triggers email alerts. Users also have the option to manage or cancel their reservations via the Cancel Booking route, ensuring full control and flexibility.

Let’s take a closer look at the backend code that powers these application routes.

```python
@app.route('/')
def index():
    return render_template('index.html')

@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')
```

```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')
```

```python
@app.route('/logout')
def logout():
    session.pop('email', None)
    flash('You have been logged out.', 'info')
    return redirect(url_for('index'))

@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)
```

```python
@app.route('/train')
def train():
    if 'email' not in session:
        return redirect(url_for('login'))
    return render_template('train.html')
```

```python
@app.route('/confirm_train_details')
def confirm_train_details():
    if 'email' not in session:
        return redirect(url_for('login'))

    booking_details = {
        'name': request.args.get('name'),
        'train_number': request.args.get('trainNumber'),
        'source': request.args.get('source'),
        'destination': request.args.get('destination'),
        'departure_time': request.args.get('departureTime'),
        'arrival_time': request.args.get('arrivalTime'),
        'price_per_person': Decimal(request.args.get('price')),
        'travel_date': request.args.get('date'),
        'num_persons': int(request.args.get('persons')),
        'item_id': request.args.get('trainId'), # This is the train ID
        'booking_type': 'train',
        'user_email': session['email'],
        'total_price': Decimal(request.args.get('price')) * int(request.args.get('persons'))
    }
```

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

    booking_id = request.form.get('booking_id')
    user_email = session['email']
    booking_date = request.form.get('booking_date') # This is crucial as it's the sort key

    if not booking_id or not booking_date:
        flash("Error: Booking ID or Booking Date is missing for cancellation.", 'error')
        return redirect(url_for('dashboard'))

    try:
        # Delete item using the primary key (user_email and booking_date)
        # This does not use GSI, so it remains unchanged.
        bookings_table.delete_item(
            Key={'user_email': user_email, 'booking_date': booking_date}
        )
        flash(f"Booking {booking_id} cancelled successfully!", 'success')
    except Exception as e:
        flash(f"Failed to cancel booking {booking_id}: {str(e)}", 'error')

    return redirect(url_for('dashboard'))
```

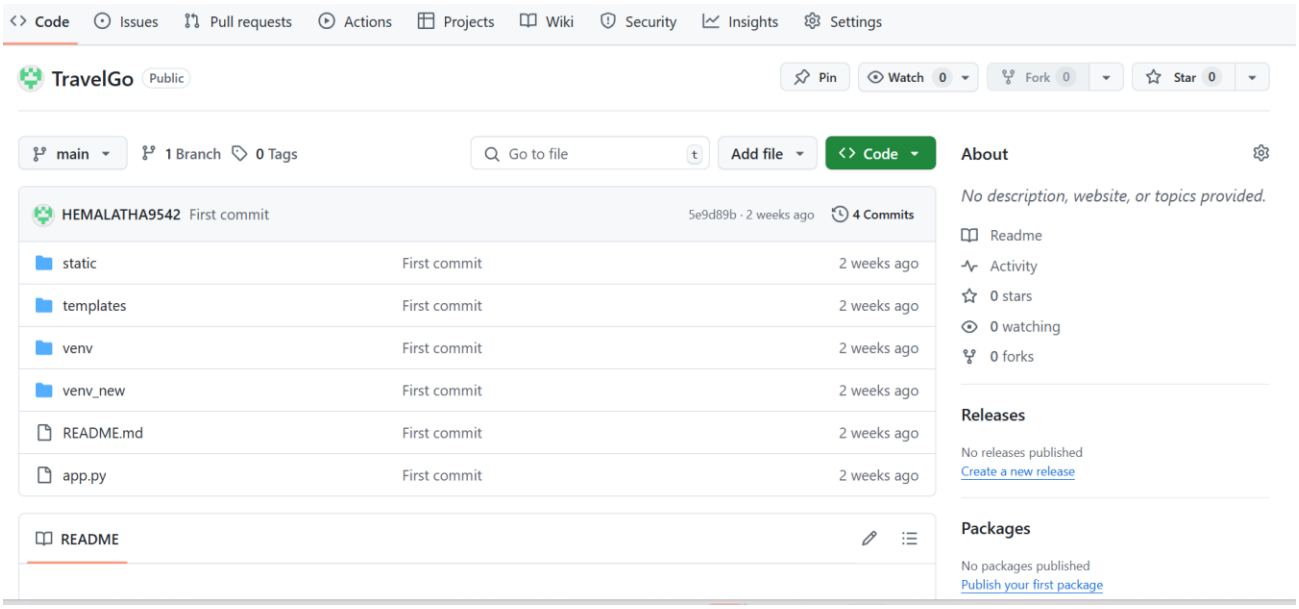
Note: The implementation logic for train booking, train details, and cancellation is consistent across other modules such as bus, flight, and hotel. Each follows a similar structure for handling user input, storing booking data, and managing cancellations using the same backend principles. This modular approach promotes code reusability and simplifies maintenance across the application. Minor adjustments are made in each module to accommodate specific fields like transport type or room preferences. Overall, the core flow—search, select, book, and cancel—remains consistent for all services.

Deployment code:

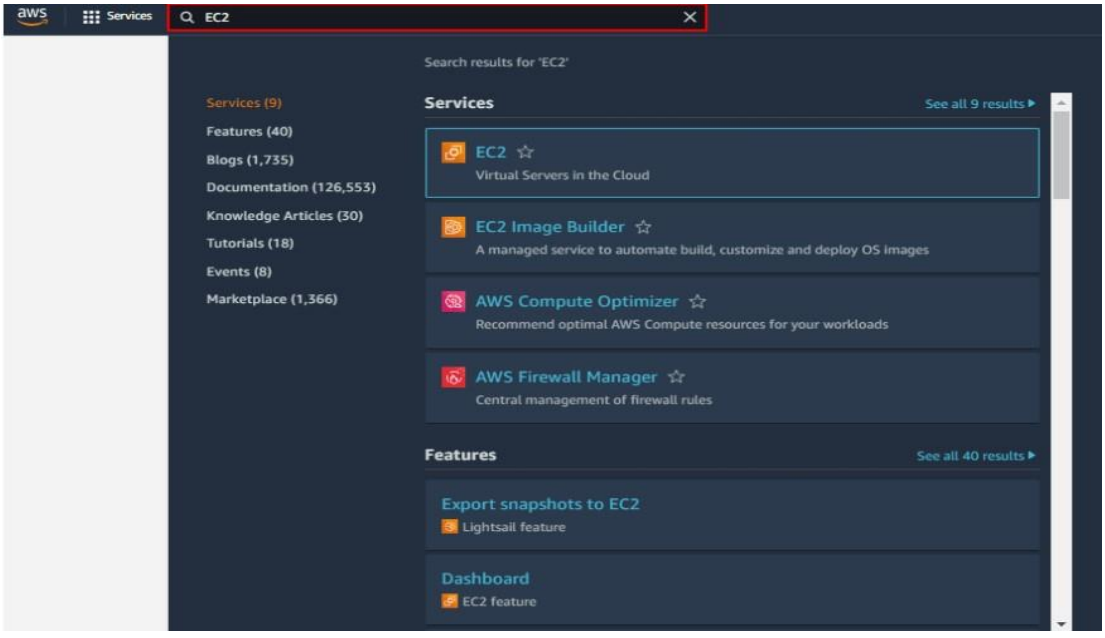
```
518  if __name__ == '__main__':
519      # IMPORTANT: In a production environment, disable debug mode and specify a production-ready host.
520      app.run(debug=True, host='0.0.0.0')
```

Start the Flask server by configuring it to run on all network interfaces (0.0.0.0) at port 5000, with debug mode enabled to support development and live testing.

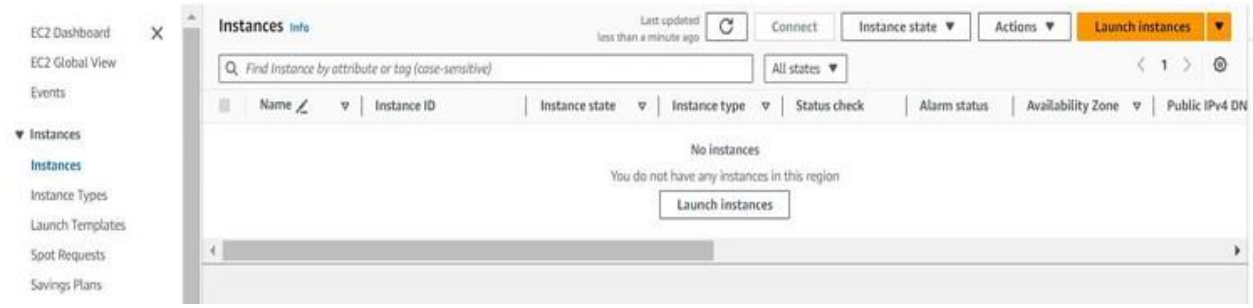
EC2 Instance Setup:



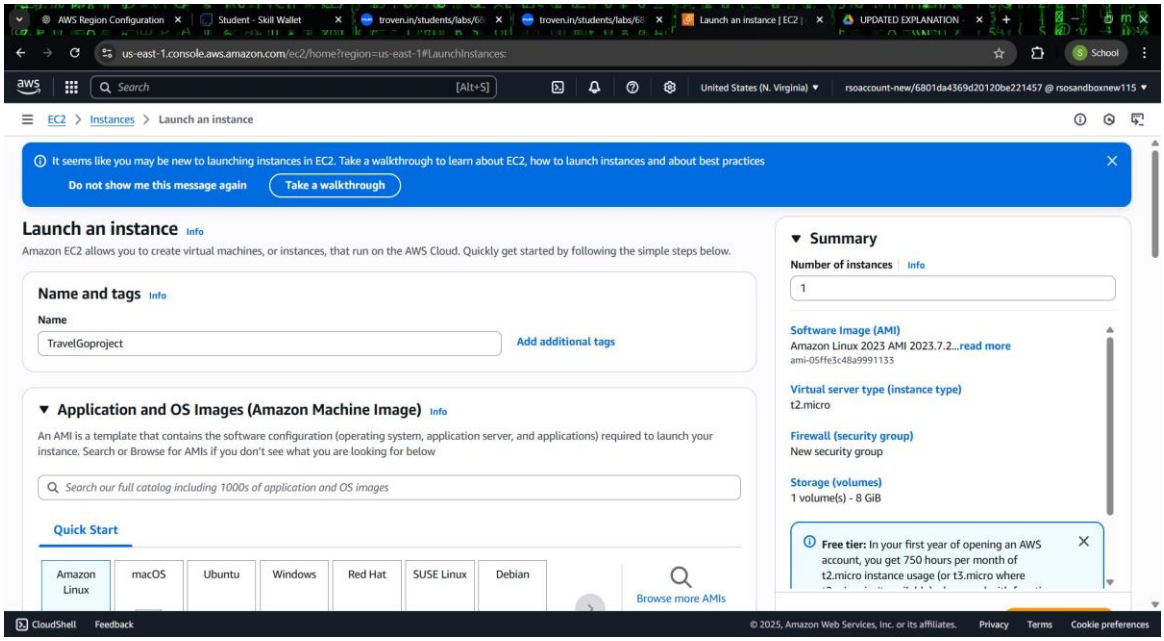
Launch an EC2 instance to host the Flask application.



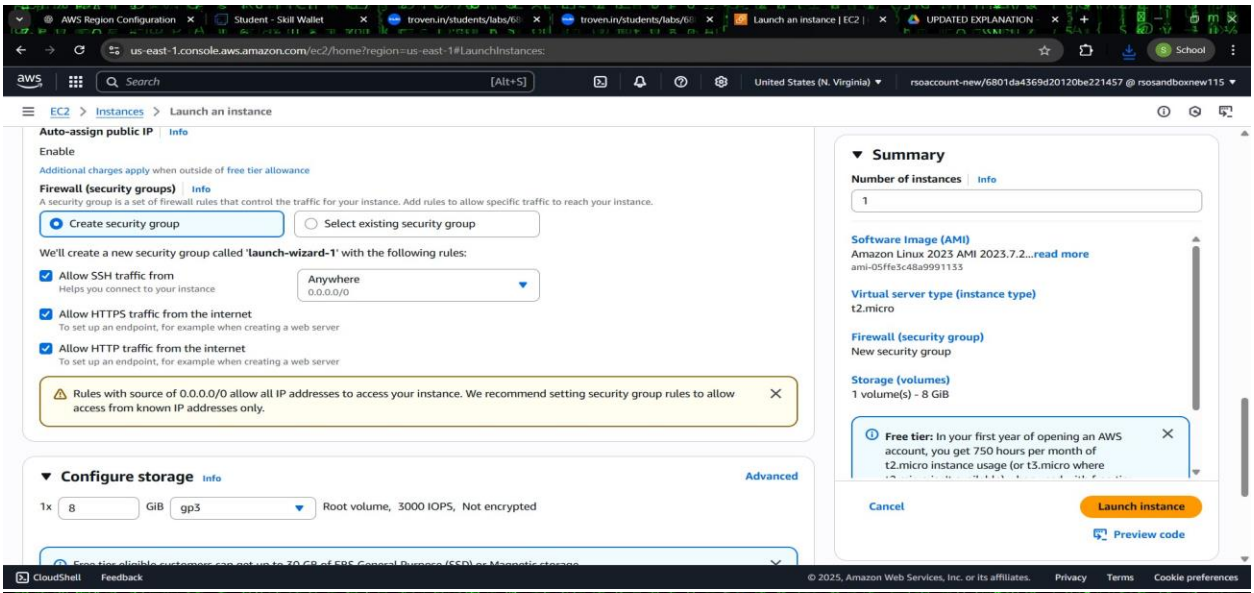
1. Click on launch Instance:

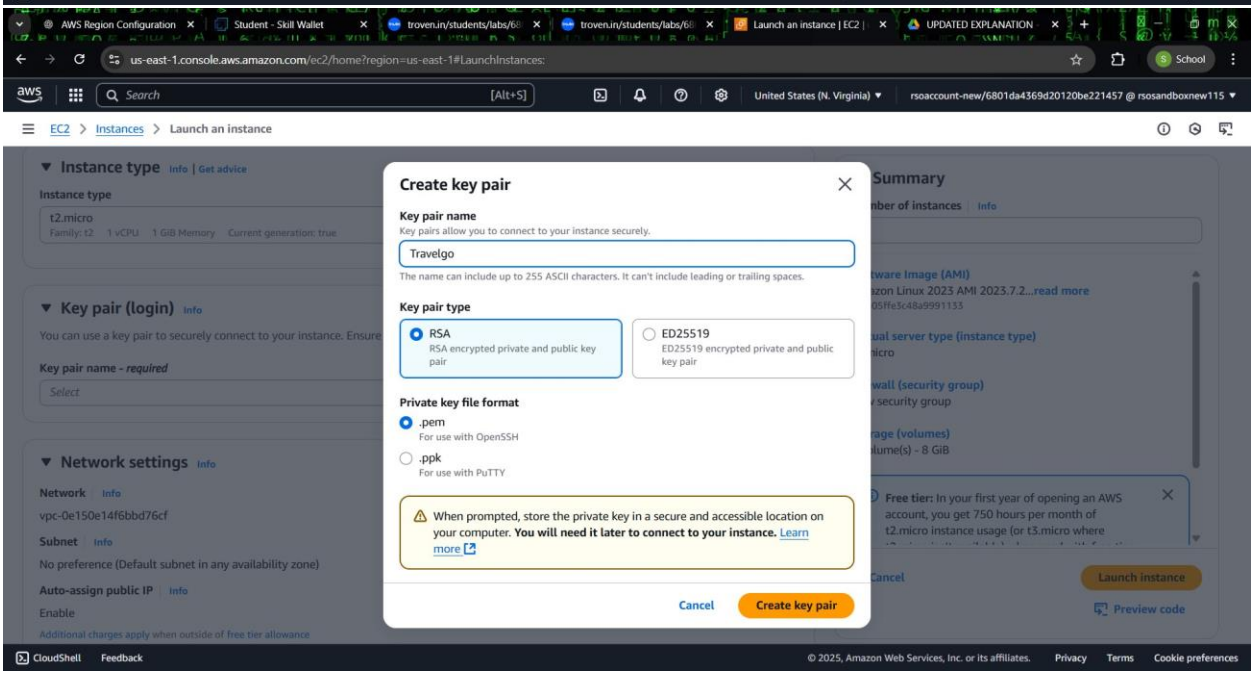


Once the EC2 instance is launched, it acts as the server backbone for your application. Naming the instance as Travelgoproject helps in easy identification during future scaling or monitoring. You can manage performance, logs, and connectivity from the EC2 dashboard. This instance will host and run your Flask backend reliably.

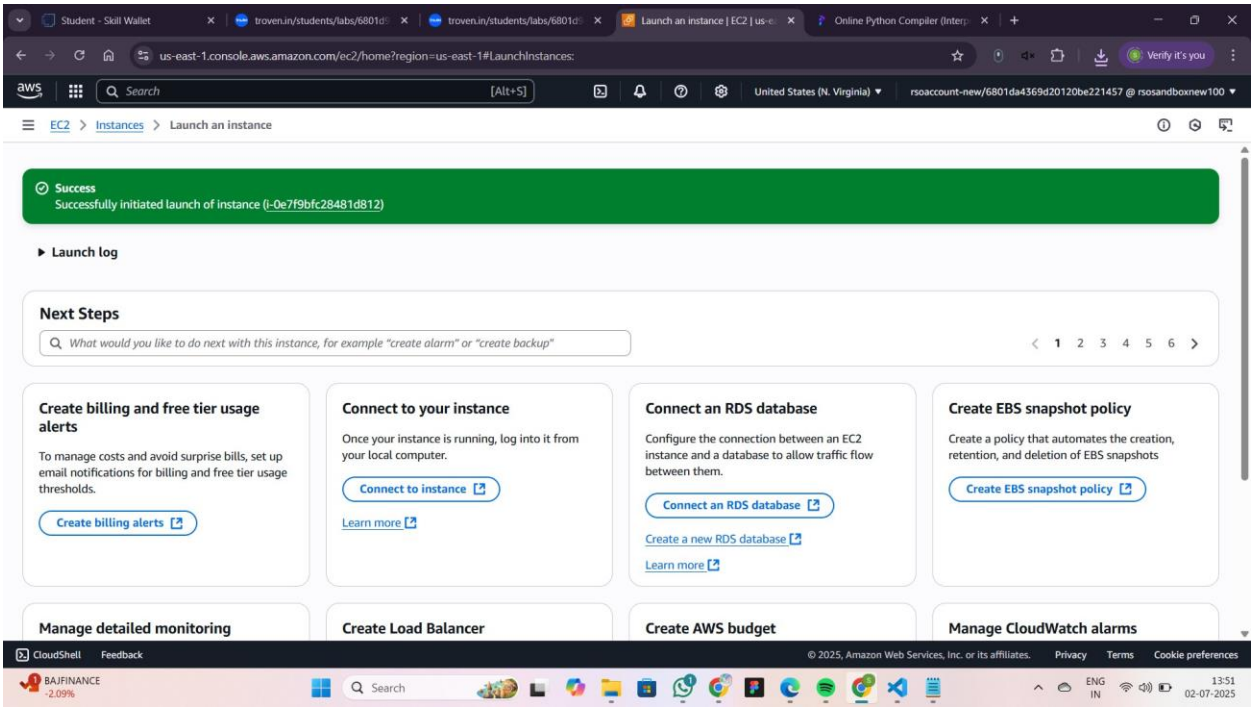


Create a key pair named Travelgo to securely connect to your EC2 instance via SSH. Download and store the .pem file safely, as it will be required for future logins. While setting up the firewall, configure the security group to allow inbound traffic on ports 22 (SSH) and 5000 (Flask). This ensures both secure access and proper functioning of the web application.It's recommended to restrict SSH access to your specific IP range for added security. The Flask port (5000) should be open to all only during development—limit access in production. Properly configured security groups help prevent unauthorized access while keeping your app responsive and accessible.



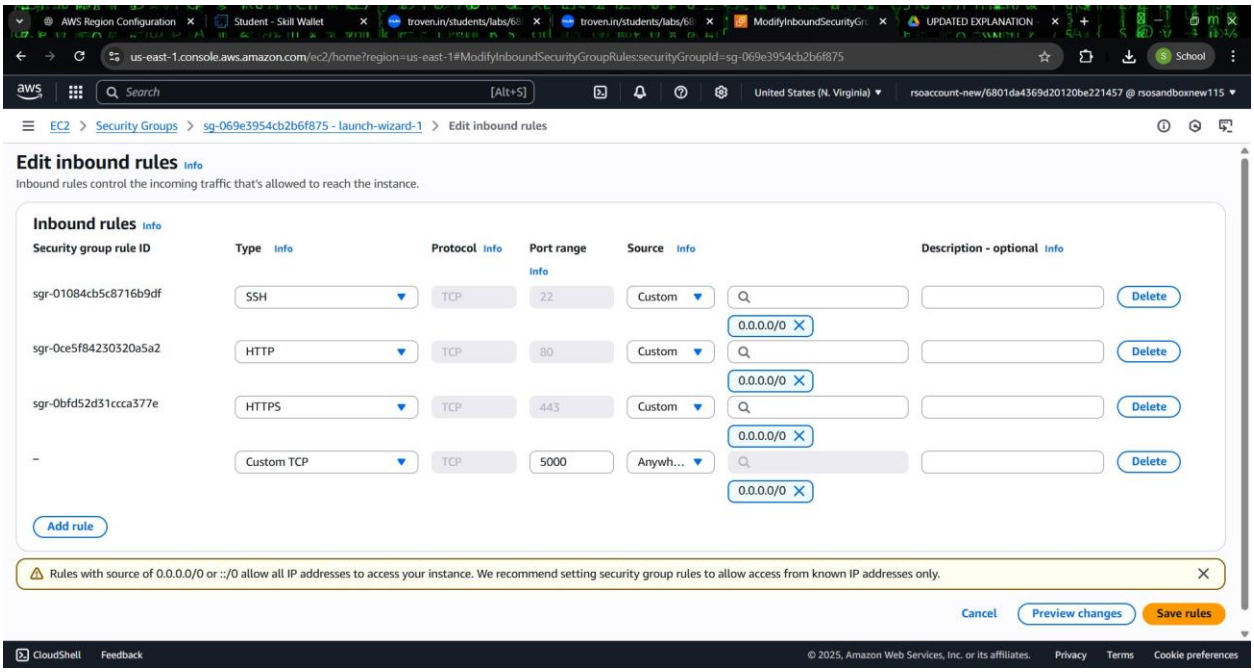


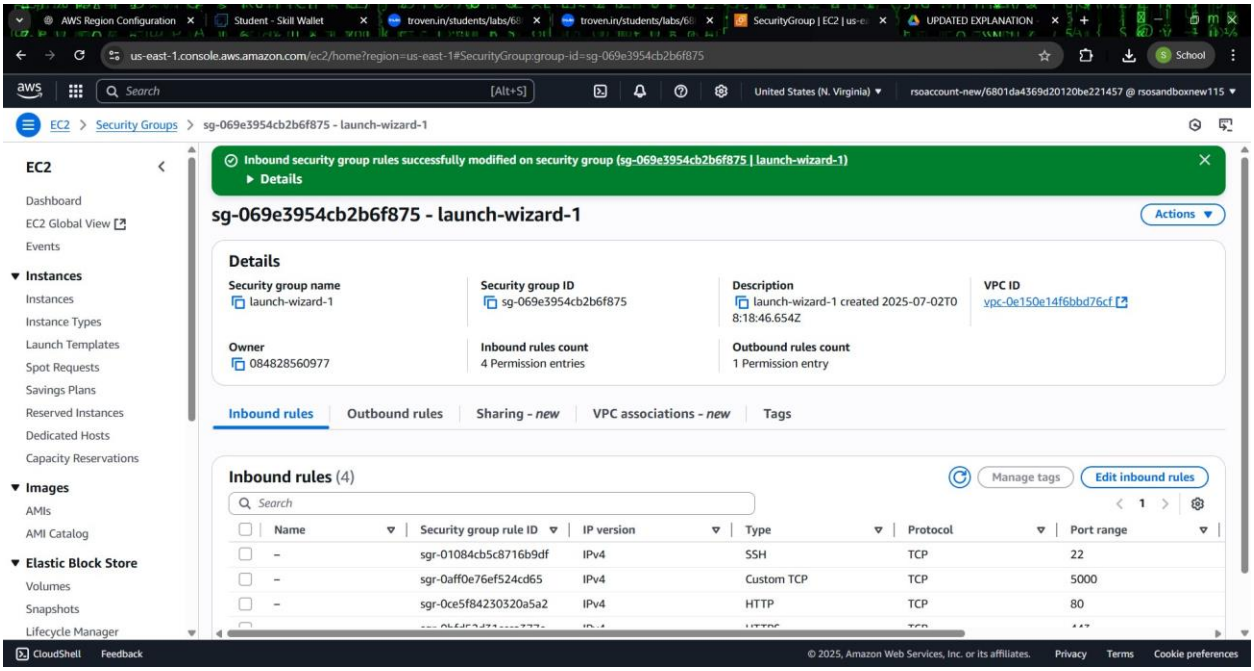
Wait for the confirmation message like “Successfully initiated launch of instance i0e7f9bfc28481d812,” indicating the instance is being provisioned. During this process, create a key pair named Travelgo (RSA type) with .pem file format. Save this private key securely, as it will be needed for future SSH access.



Edit Inbound Rules:

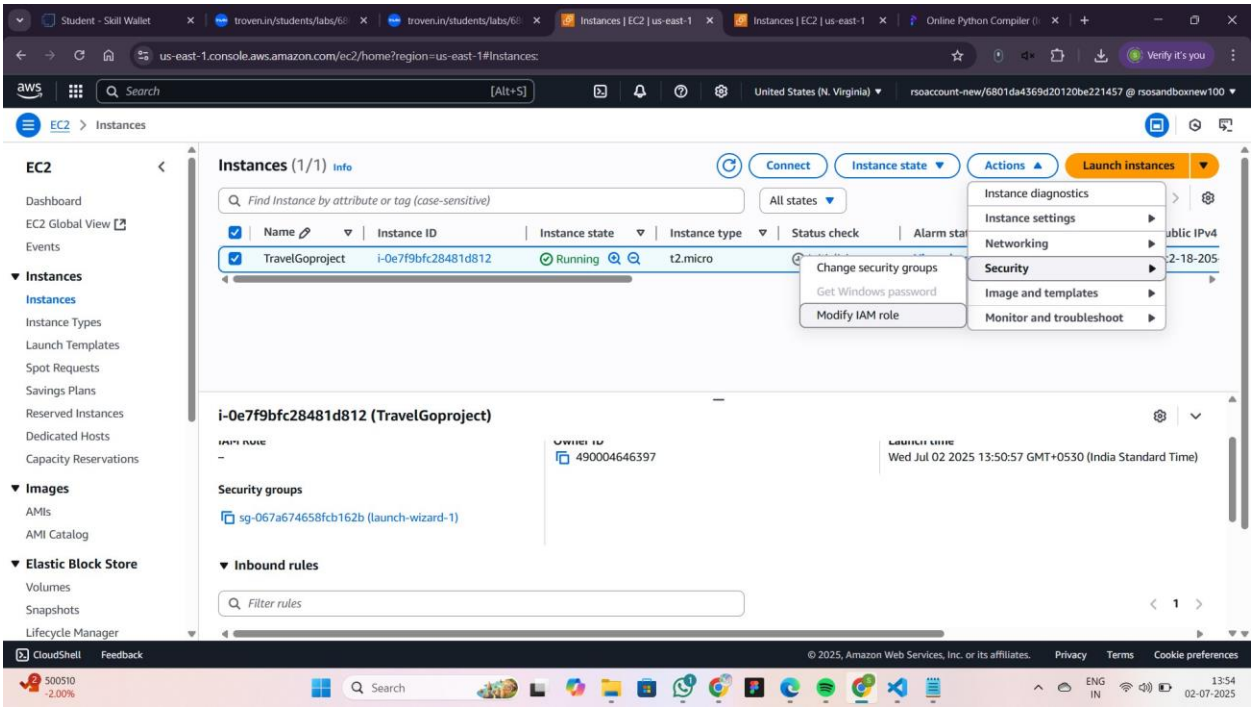
Select the EC2 instance you just launched and ensure it's in the "running" state. Navigate to the "Security" tab, then click "Edit inbound rules." Add a new rule with the following settings: Type – Custom TCP, Protocol – TCP, Port Range – 5000, Source – Anywhere (IPv4) 0.0.0.0/0. This allows external access to your Flask application running on port 5000.

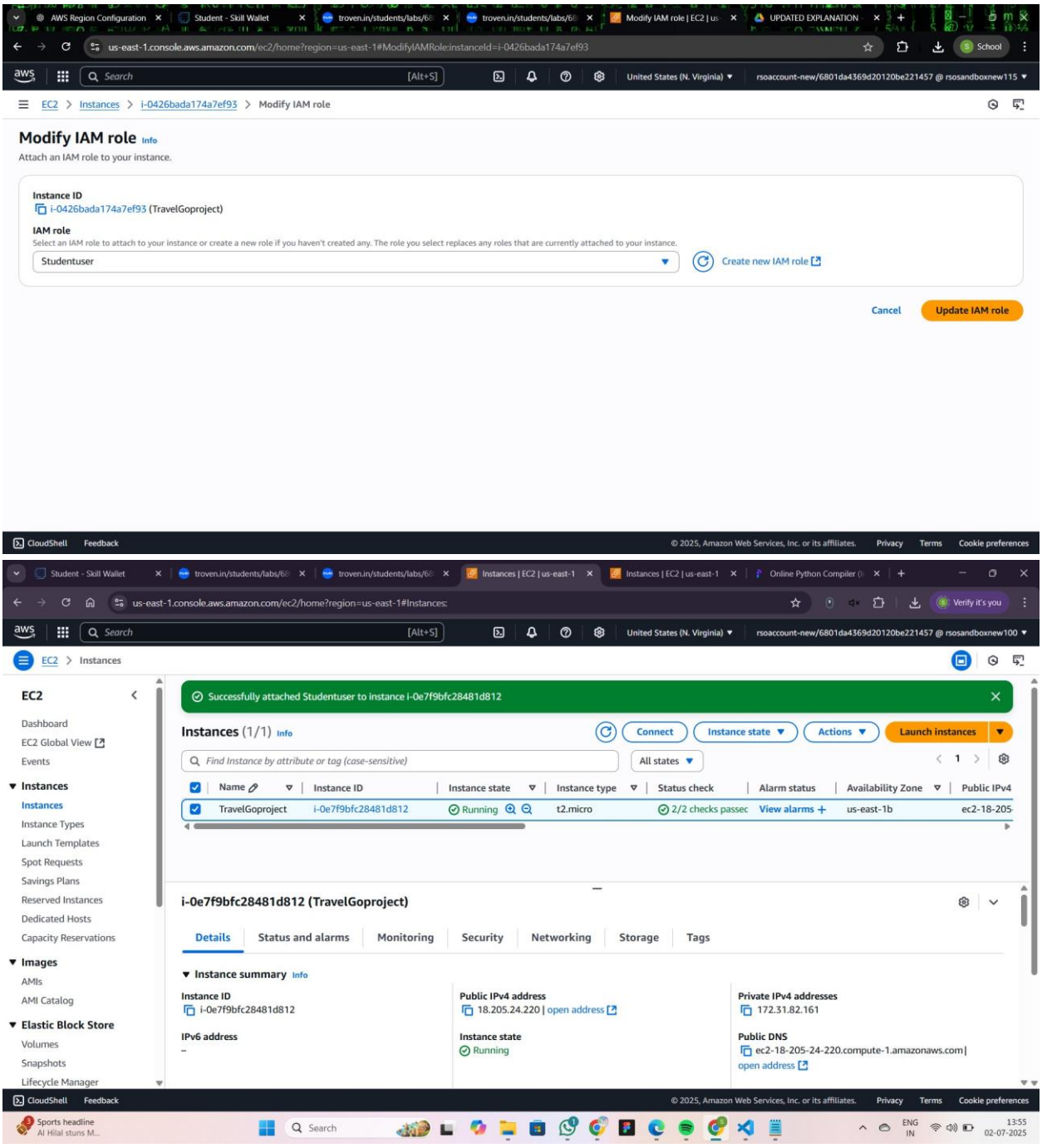




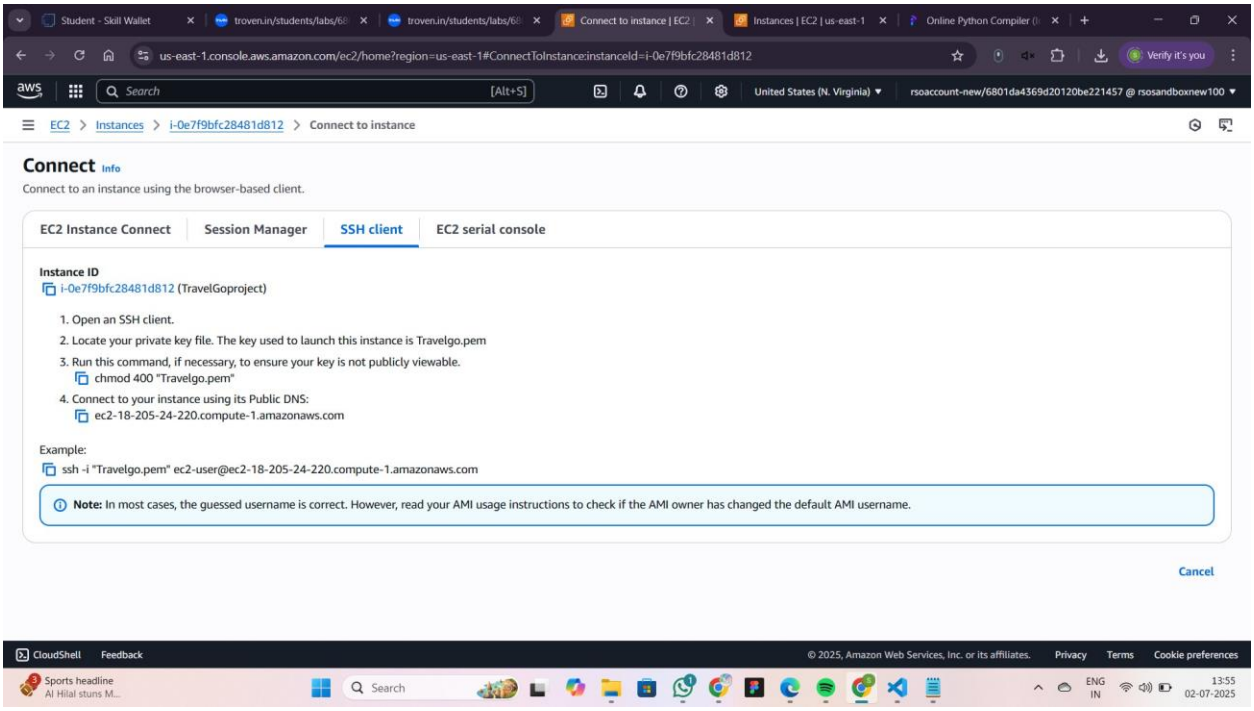
Modify IAM ROLE

Attach the IAM role Studentuser to your EC2 instance to grant secure access to AWS services like DynamoDB and SNS. This avoids hardcoding credentials and ensures your app functions with the required permissions. Make sure the role includes all necessary policies for smooth integration.

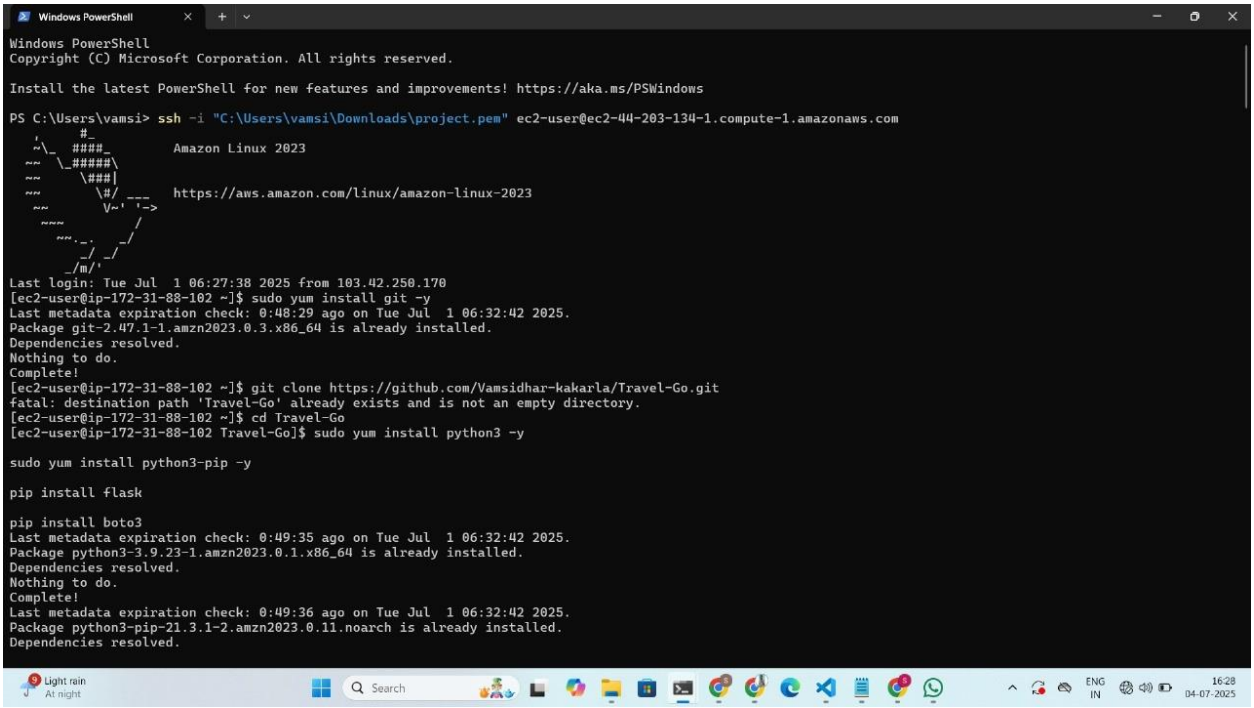


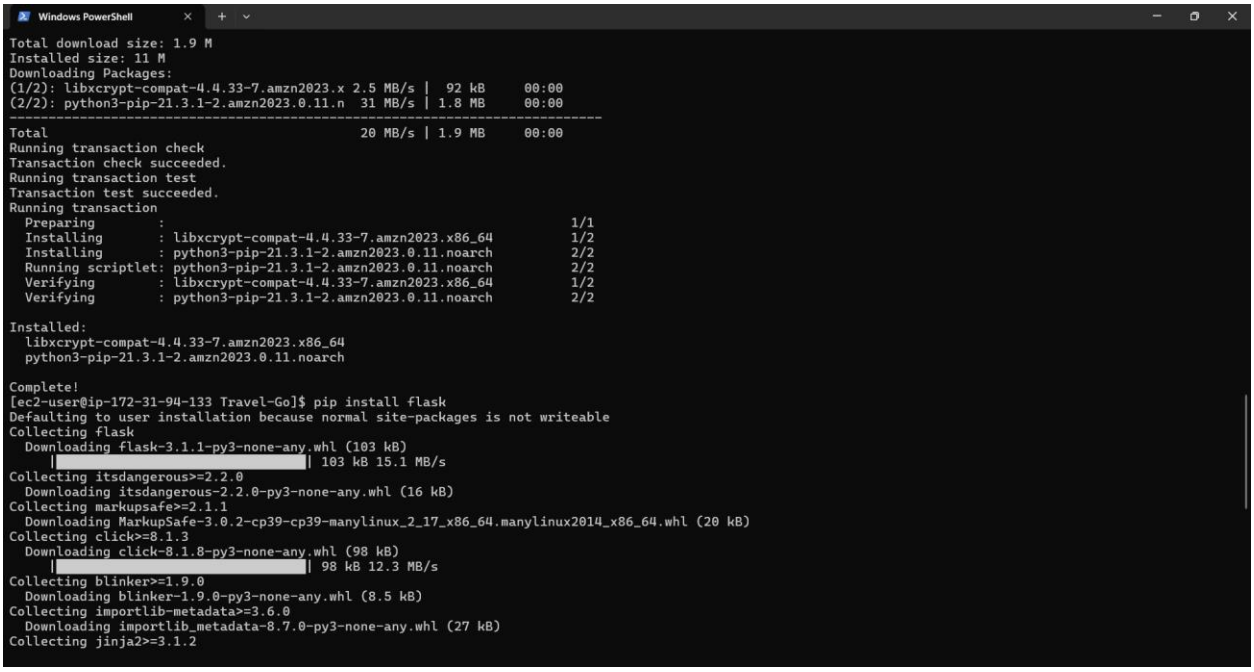


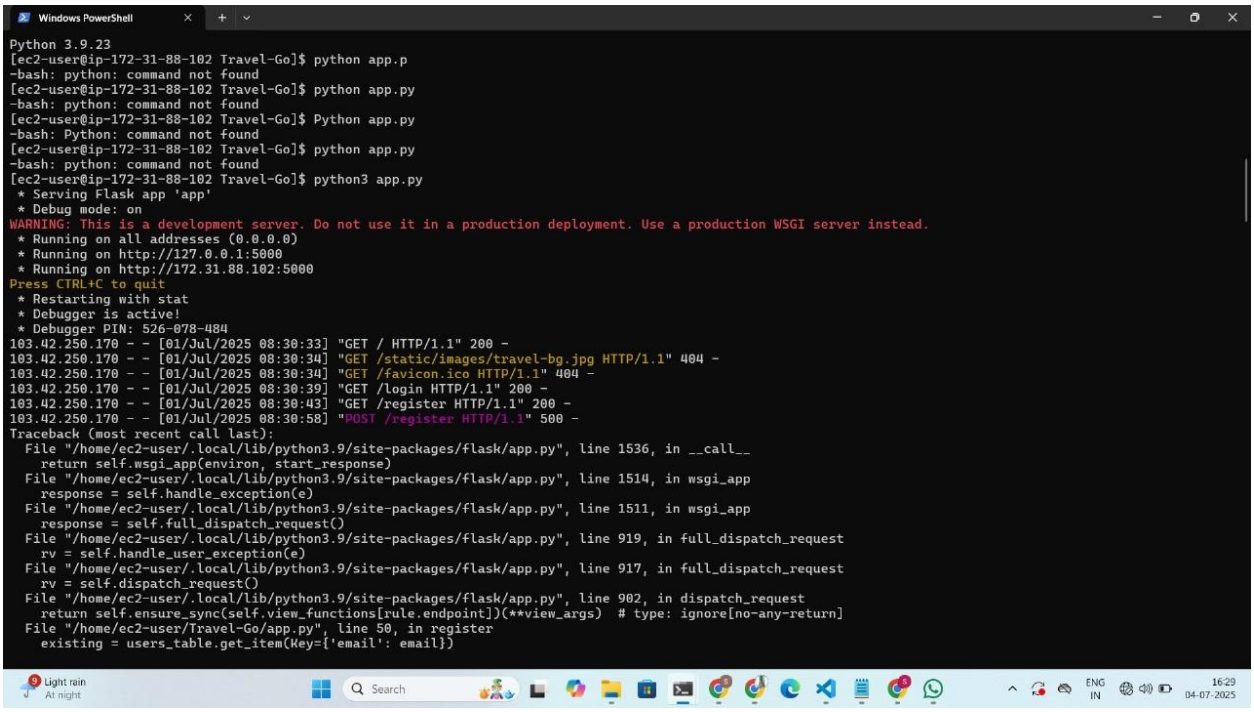
Wait until the confirmation message appears indicating that the Studentuser role has been successfully attached to the instance. This confirms that all required permissions are now active. Once attached, proceed to connect to your EC2 instance and run your GitHub-hosted Flask application code.



The following are the terminal outputs after executing the commands shown below the image. These outputs indicate that the setup steps have been successfully carried out.
The commands summary will be provides at last.







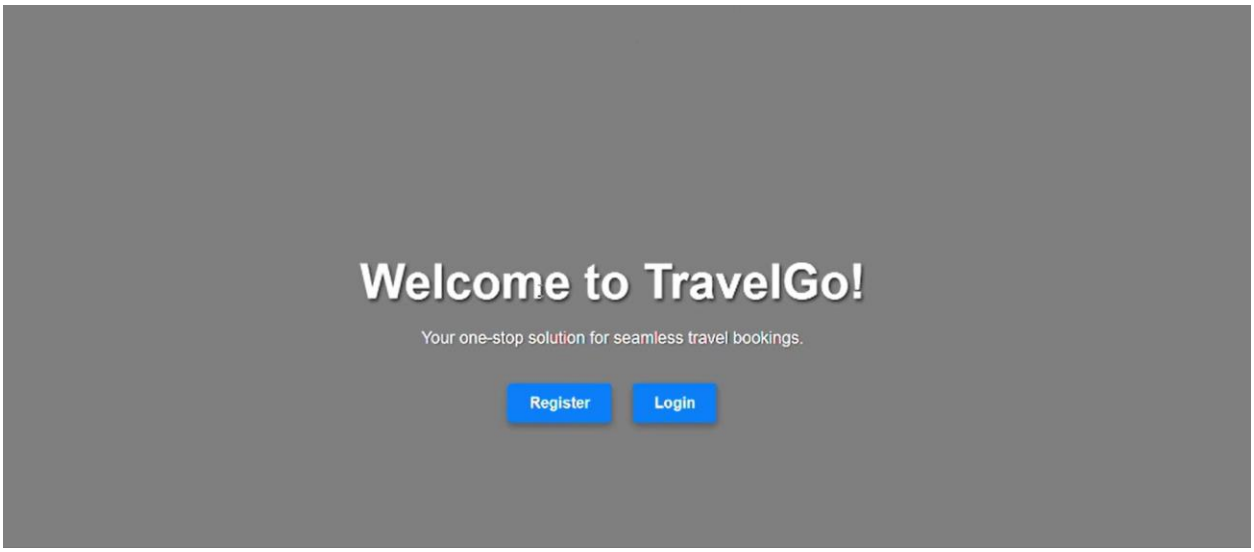
- pip install flask
- pip install boto3
- python3 app.py

Deployment Steps Summary:

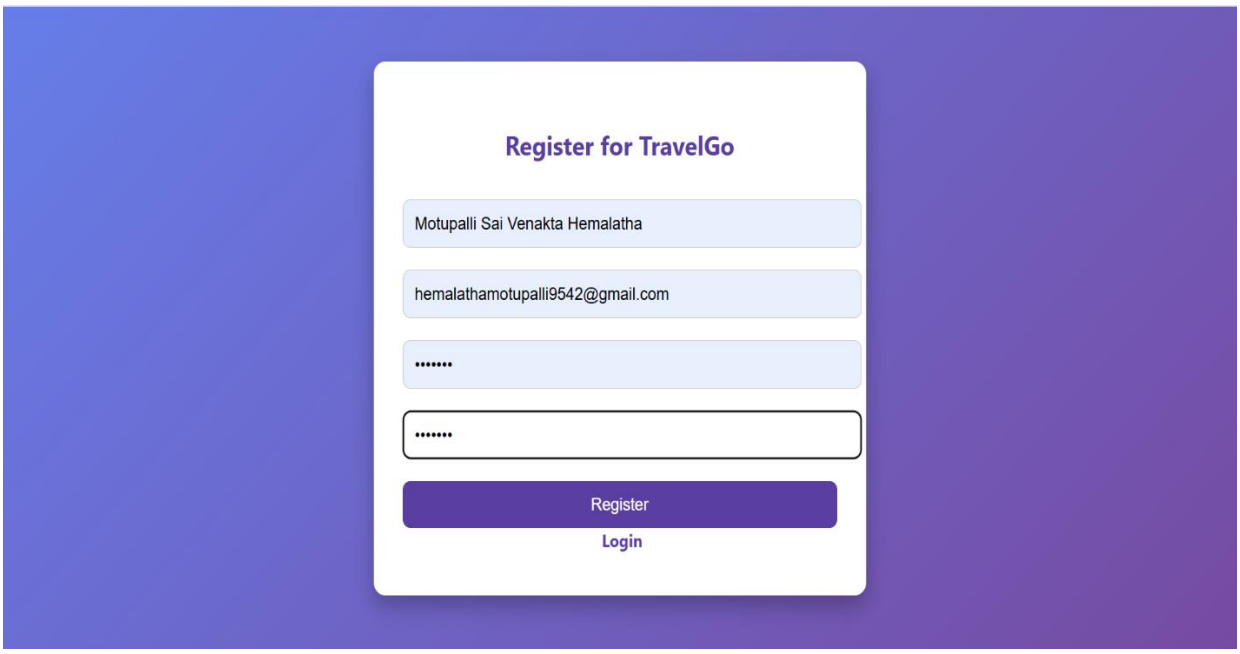
- sudo yum install git -y
- git clone your_repository_url
- cd your_project_directory
- sudo yum install python3 -y
- sudo yum install python3-pip -y
- pip install flask
- pip install boto3 • python3 app.py

User Interface Screens:

- Homepage:

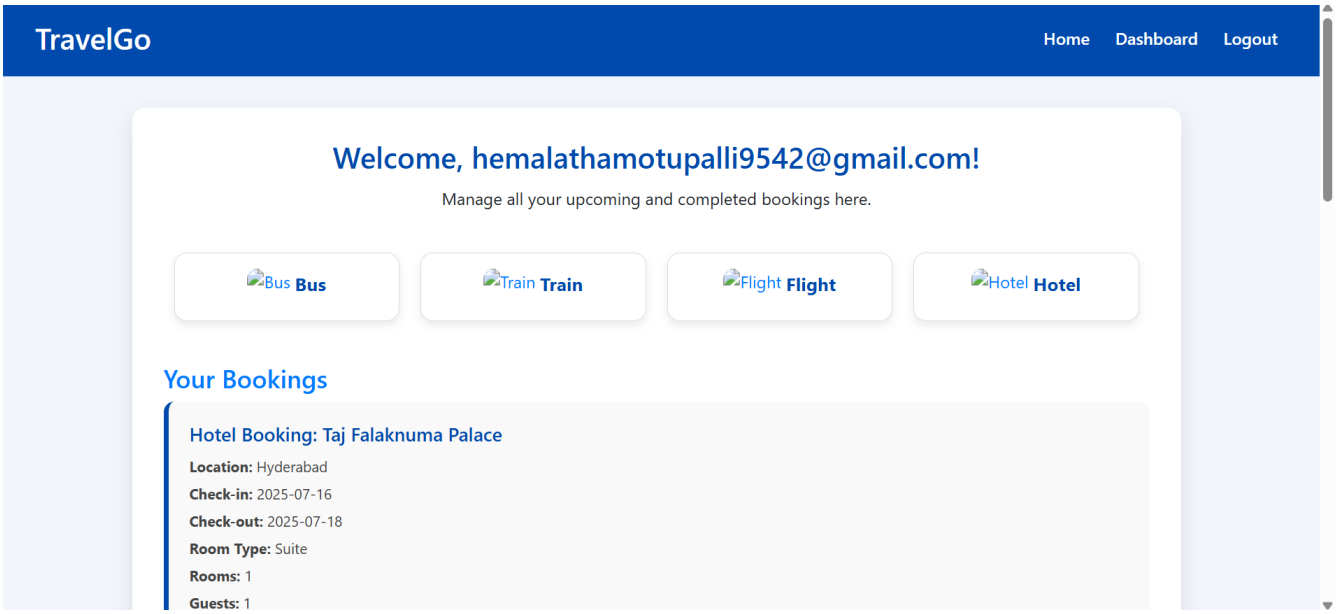


- Register and Login Portals:

- 

• Dashboard View:

The dashboard serves as the central hub for users to access all travel services in one place. It displays a personalized greeting along with quick navigation cards for Bus, Train, Flight, and Hotel bookings. A success alert confirms the user's login, enhancing the user experience. Below, recent and upcoming bookings are listed with full travel details and cancellation options. This intuitive layout ensures users can manage their journeys effortlessly and efficiently.
Lets have a look at my dashboard page!!!

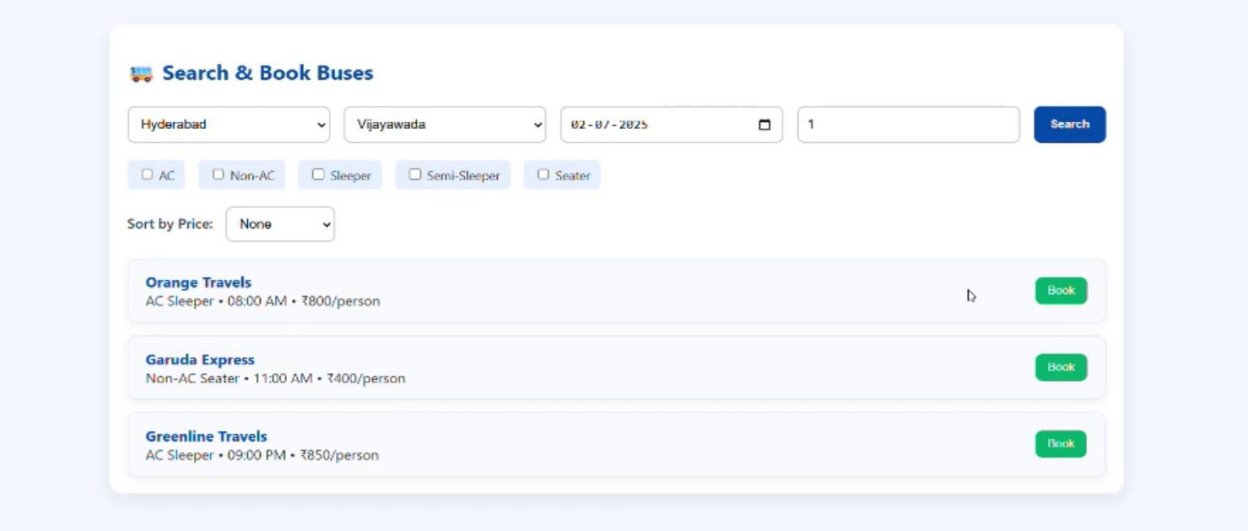


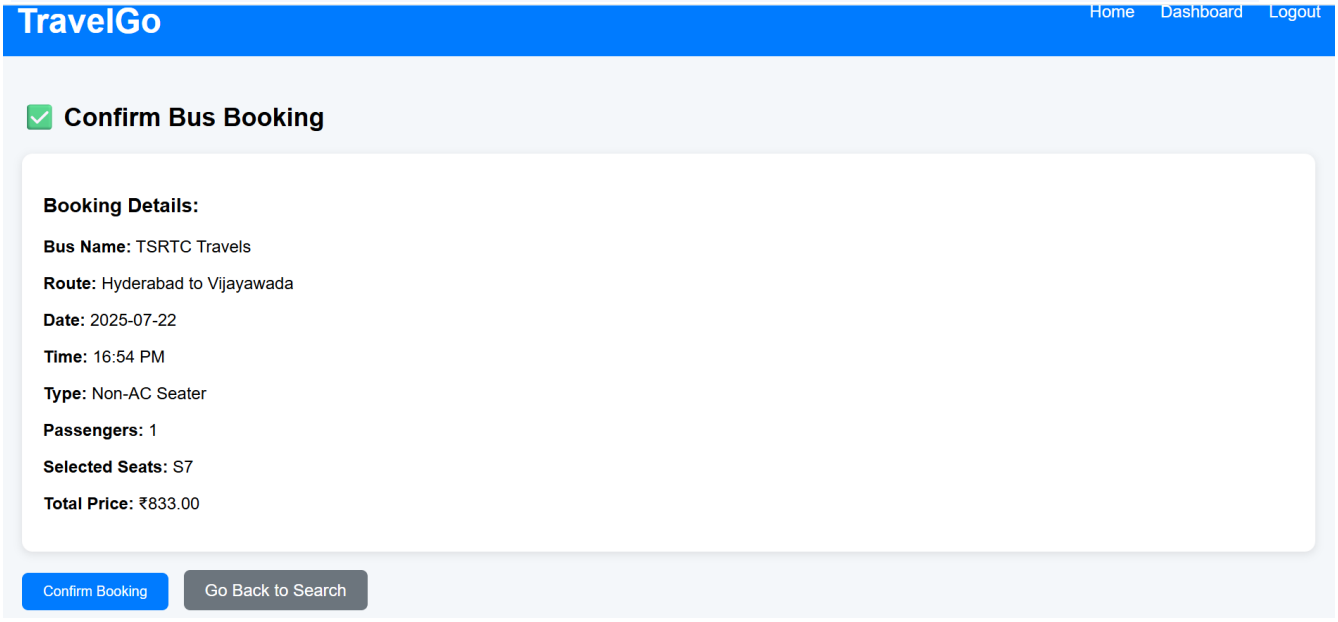
• Booking Tabs: Bus, Train, Flight, Hotel

For demonstration purposes, a seat selection section has also been included in the bus booking module. This allows users to choose their preferred seats during the booking process, enhancing the overall user experience.

This interactive feature simulates real-world booking platforms and showcases the system's dynamic capabilities. It also helps users visualize the layout before confirming their reservation.

• Bus booking:

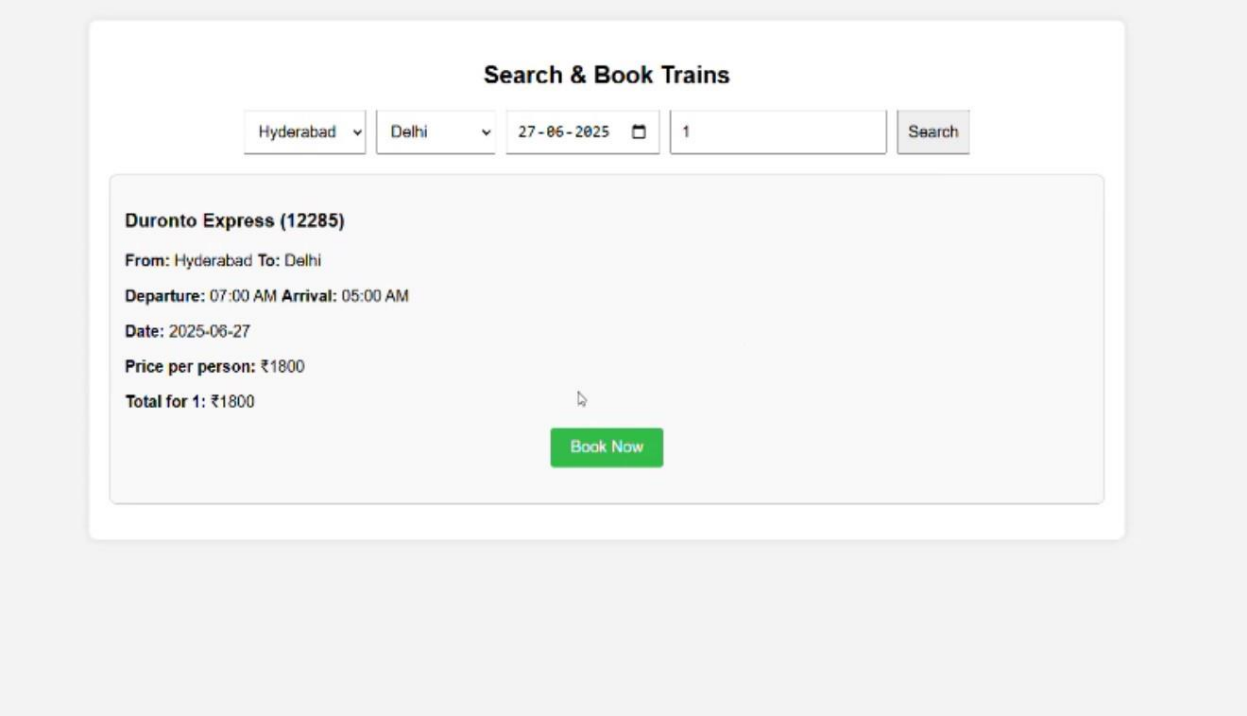


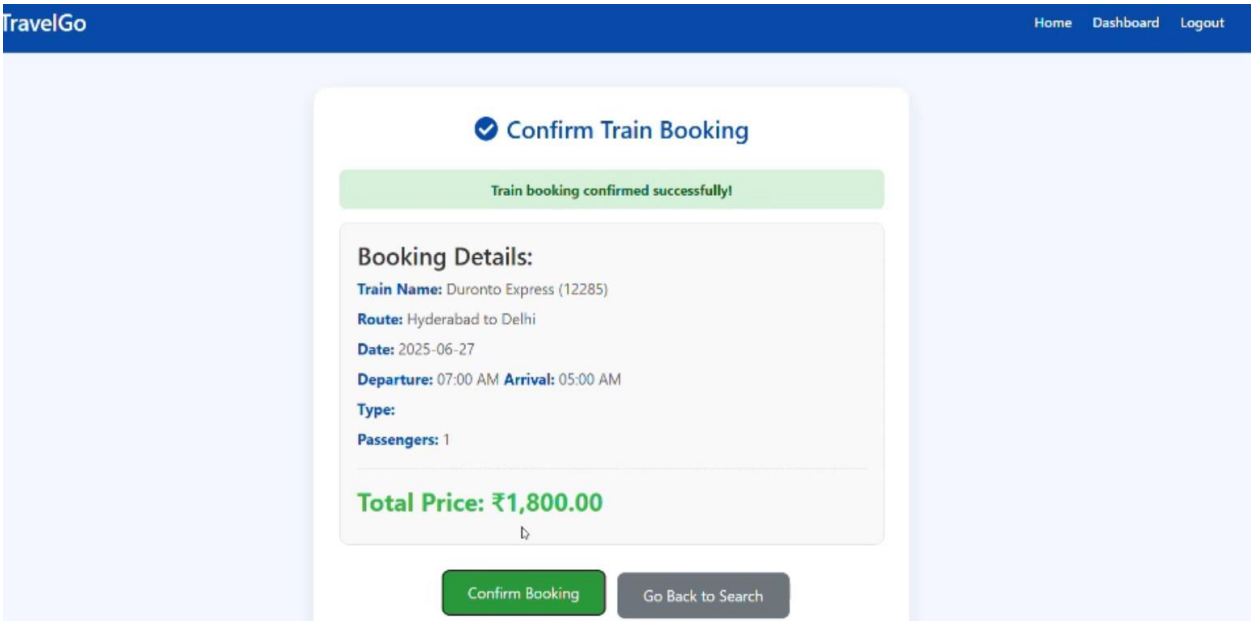


• Train booking:

The train booking module enables users to search and reserve train tickets by selecting routes, dates, and times. It fetches real-time data and displays available options tailored to user input.

Once a booking is made, the system confirms the reservation and stores the details in the database. Bookings are reflected instantly on the user dashboard for easy tracking. This module ensures a seamless and efficient booking experience for train travelers.

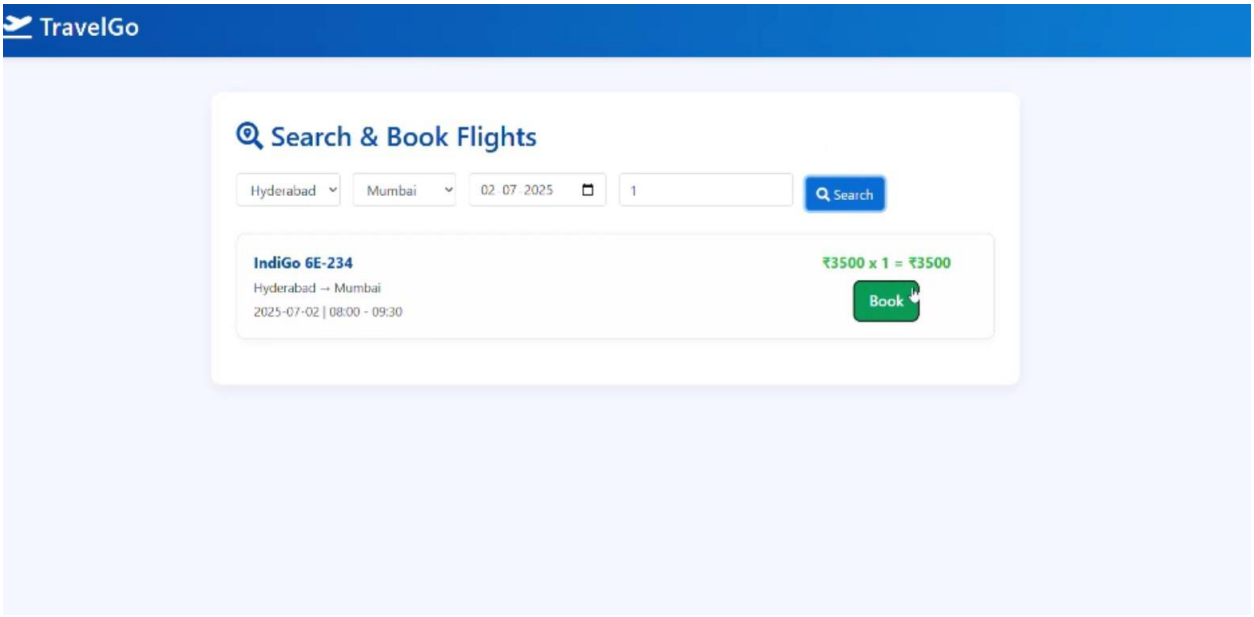


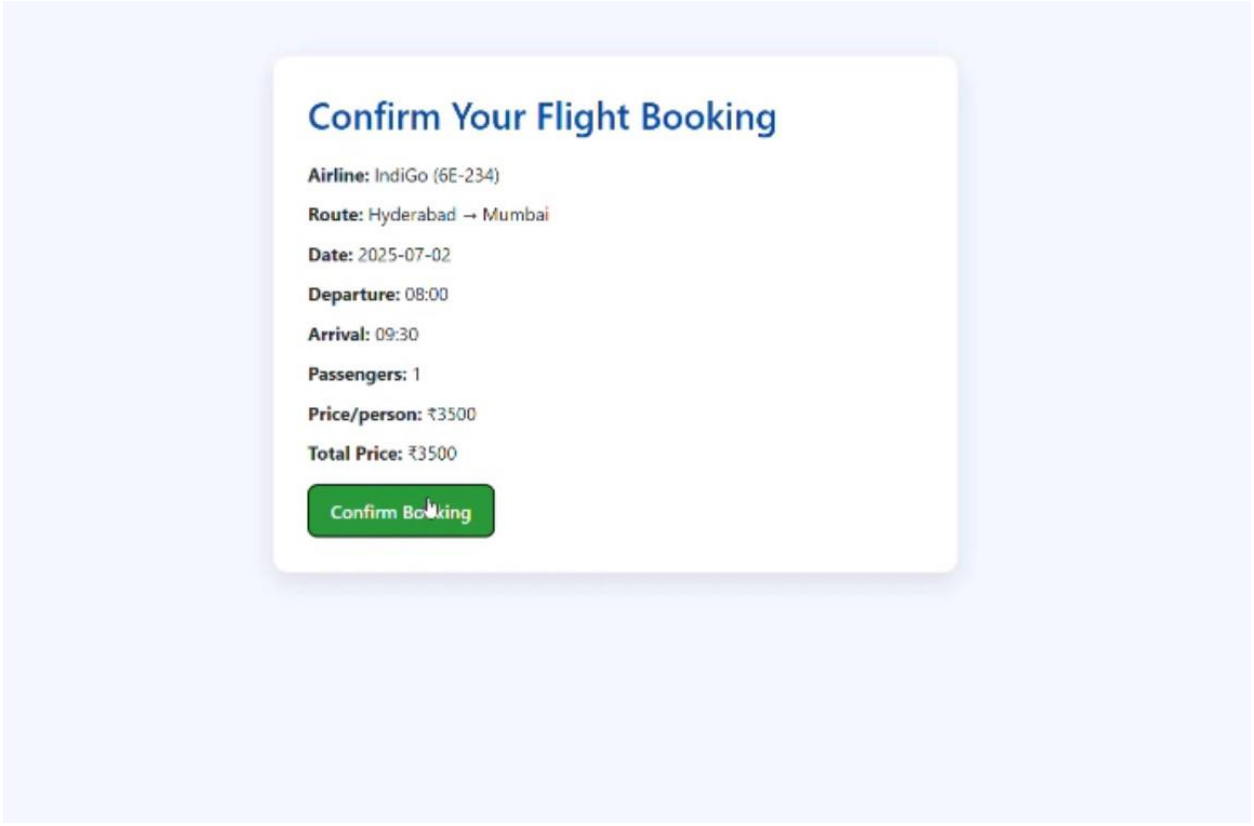


• Flight booking:

✈ The flight booking section enables users to search and reserve flights quickly based on their preferred source, destination, and date. Upon entering the details, the system fetches available flight options and displays them with timing, pricing, and route information. The interface is clean and responsive, allowing users to confirm bookings with just a few clicks. Once booked,

flight details are stored securely and reflected in the dashboard. This module ensures a smooth and efficient booking experience tailored for air travel.

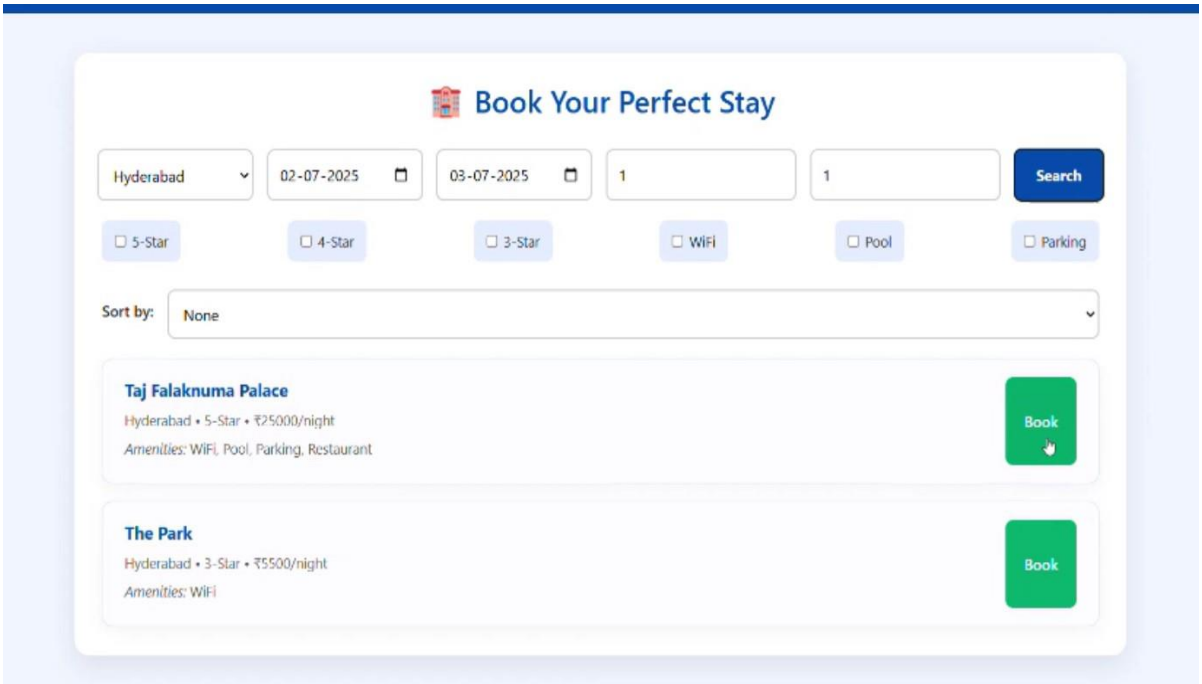


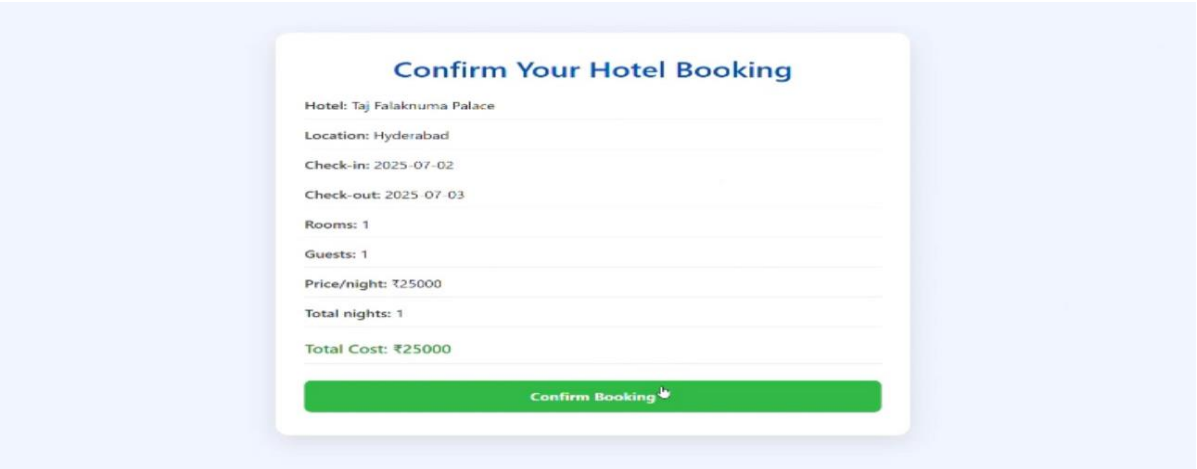


• Hotel booking:

The hotel booking section allows users to search and reserve accommodations based on their destination and travel dates. The interface displays available hotels with details such as location, price, and amenities. Users can easily compare options and proceed with booking in just a few

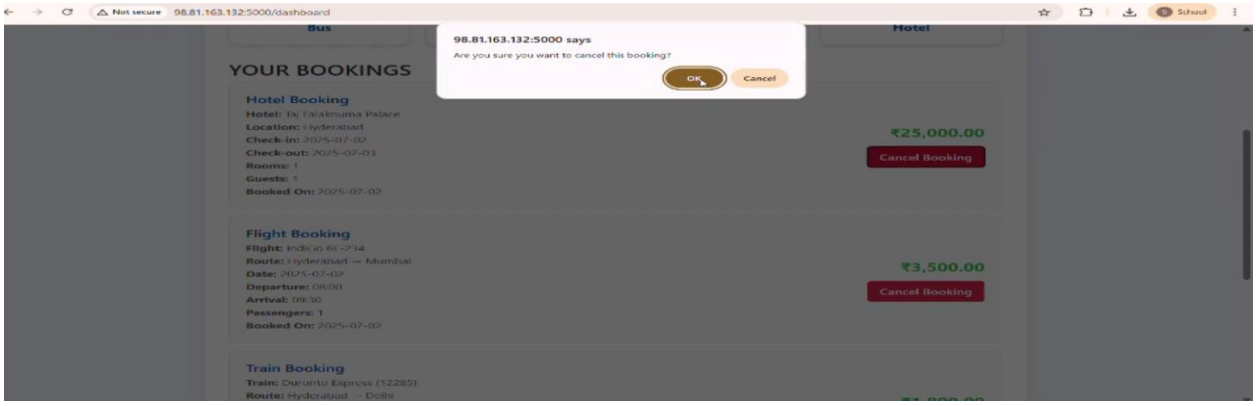
clicks. Once confirmed, the booking details appear on the dashboard for easy tracking. The process is designed to be seamless, intuitive, and efficient for all types of travelers.





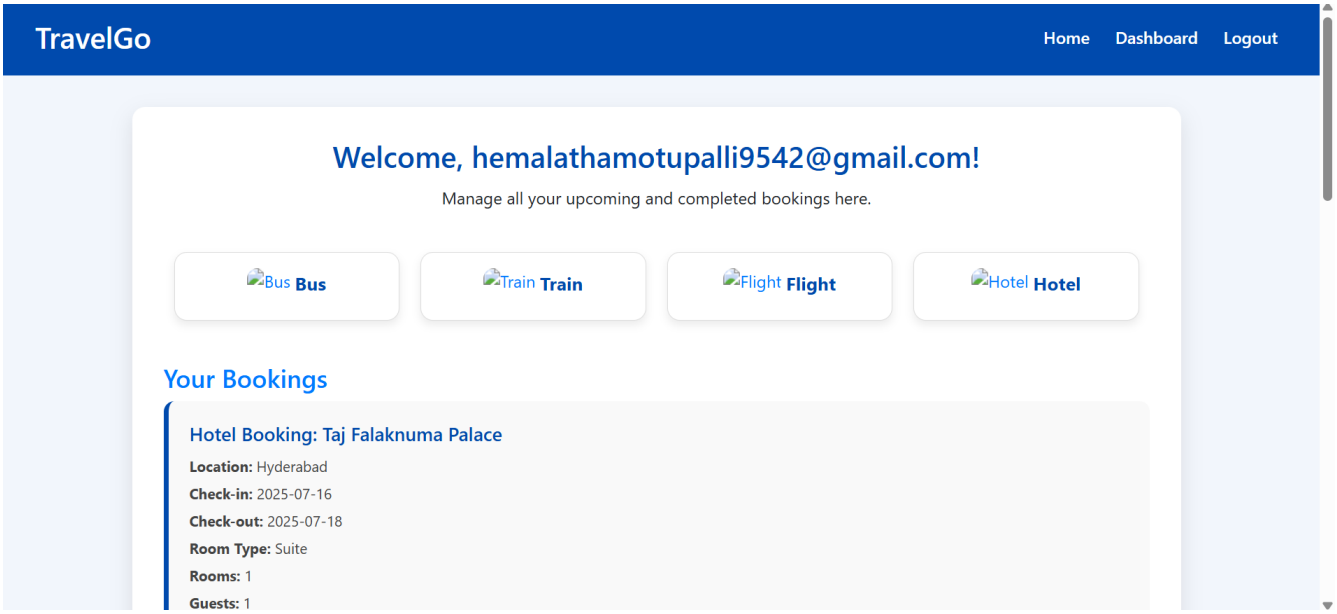
📋 At the bottom of the dashboard, users can view a summary of all their bookings, including travel details, dates, and status. A dedicated “Cancel” button is provided next to each entry, allowing users to easily cancel any upcoming reservations if needed. This section provides a centralized view for managing bookings efficiently. It ensures transparency by displaying every confirmed ticket in an organized format. The cancel feature updates the backend in real time, removing the booking from the list and database. This functionality enhances user control and flexibility while planning or modifying travel. It also improves overall convenience by reducing the need to revisit individual booking sections. All Bookings:

Cancel Bookings:



Once a booking is cancelled, a confirmation message appears indicating the cancellation was successful. This real-time feedback reassures the user that their action has been completed without issues. It also helps avoid confusion or repeated attempts. The booking entry is immediately removed from the dashboard view. This seamless flow enhances the overall usability and responsiveness of the application.

Let's have a look at the page indicating the cancellation was successful.



Final Conclusion – Elevating Travel with TravelGo:

TravelGo stands as a dynamic and cloud-powered solution that redefines how users plan and book their journeys. By seamlessly integrating Flask for backend logic, AWS EC2 for scalable deployment, DynamoDB for reliable data storage, and SNS for real-time notifications, the platform delivers a smooth, responsive, and user-centric experience.

From login to booking, and from instant alerts to effortless cancellations, every feature is built with efficiency and accessibility in mind. Whether it's buses, trains, flights, or hotels—TravelGo offers a unified platform that adapts to diverse travel needs. Its modern architecture ensures high performance, security, and scalability even under peak load conditions. 🌍 **With TravelGo, travel planning is no longer a hassle—it's an experience.**

🚀 Smart. Fast. Reliable. That's the future of travel.