TravelGo: A Cloud-Powered Real-Time Travel Booking Platform Using AWS:

Hardware Required:

Processor: Intel i5 or equivalent (minimum). RAM: 4 GB (8 GB recommended for Full Stack MERN). Storage: 128 GB SSD or 128 GB HDD. Internet Connectivity: High-speed internet (minimum 10 Mbps per system). Additional: Audio-visual setup for interactive sessions (microphone, speakers, etc.)

Software Required:

Processor: Intel i5 or equivalent (minimum). RAM: 4 GB (8 GB recommended for Full Stack MERN). Storage: 128 GB SSD or 128 GB HDD. Internet Connectivity: High-speed internet (minimum 10 Mbps per system). Additional: Audio-visual setup for interactive sessions (microphone, speakers, etc.).

System Required:

Projector and Audio System for presentations in all labs/classrooms Classrooms/Labs are equipped with systems or provisions for students to join sessions with their own laptops.

Description:

TravelGo is a full-stack, cloud-based travel booking platform designed to simplify the process of reserving buses, trains, flights, and hotels through a unified interface. Built using Flask as the backend framework, the application is deployed on Amazon EC2 and leverages DynamoDB for efficient storage of user data and bookings. TravelGo allows users to register, log in, search for transportation and accommodation options, and book their travel with ease. Once a booking is confirmed or cancelled, users receive real-time email notifications powered by AWS Simple Notification Service (SNS), keeping them informed throughout their journey.

The platform's user-friendly interface supports dynamic seat selection for buses, hotel filtering based on preferences such as luxury or budget, and provides booking summaries along with centralized cancellation management. By combining cloud scalability, responsive design, and secure session handling, TravelGo delivers a seamless and real-time travel planning experience for users.

Scenarios:

Scenario 1: Hassle-Free Multi-Mode Travel Booking Experience

TravelGo offers users a unified platform to search and book buses, trains, flights, and hotels all in one place. For instance, a user planning a trip from Hyderabad to Bangalore can log in, select their preferred mode of transport, choose from available options, and proceed to booking. Flask manages the backend operations such as retrieving travel listings and processing user input in real-time. Hosted on AWS EC2, the platform remains responsive even during high-traffic hours like weekends or holiday seasons, allowing multiple users to browse and book without delay.

Scenario 2: Real-Time Booking Confirmation with AWS SNS

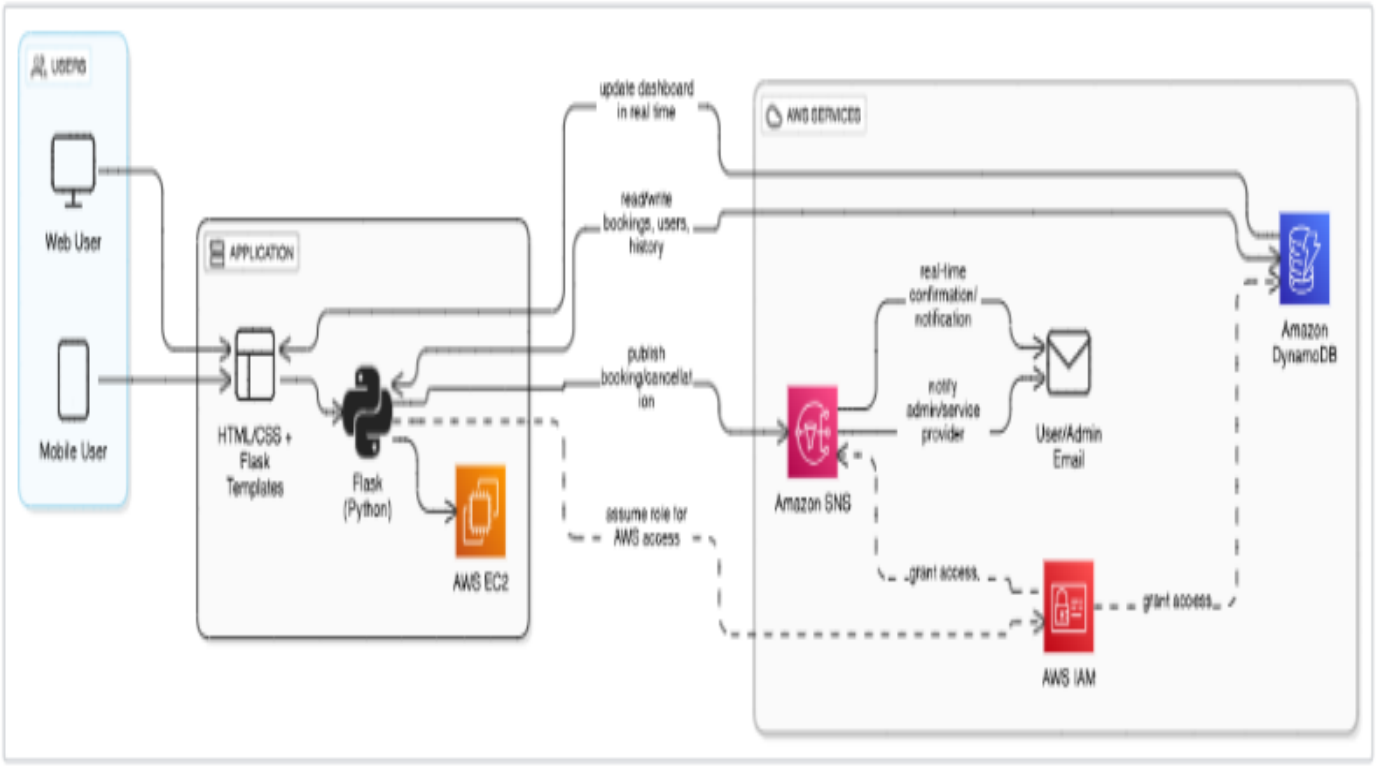
Once a booking is made—whether it's a train ticket or a hotel stay—TravelGo uses AWS SNS to instantly notify the user. For example, after a student books a hotel in Chennai, SNS sends a real-time email notification confirming the booking with all the relevant details. This notification is triggered from the Flask backend after the booking is successfully recorded in DynamoDB. Additionally, SNS can alert admin or service providers, ensuring transparency and real-time updates on every transaction.

Scenario 3: Dynamic Dashboard with Personal Travel History

TravelGo features a dynamic user dashboard that displays all past and upcoming bookings for the logged-in user. For example, a user who has booked a flight and a hotel can view these bookings categorized by type, along with dates, price, and cancellation options. Flask fetches this data from AWS DynamoDB, which persistently stores all user bookings. The dashboard UI, powered by responsive HTML/CSS and Flask templates, ensures users can review or manage bookings anytime, from any device, with real-time updates and quick cancellation workflows supported.

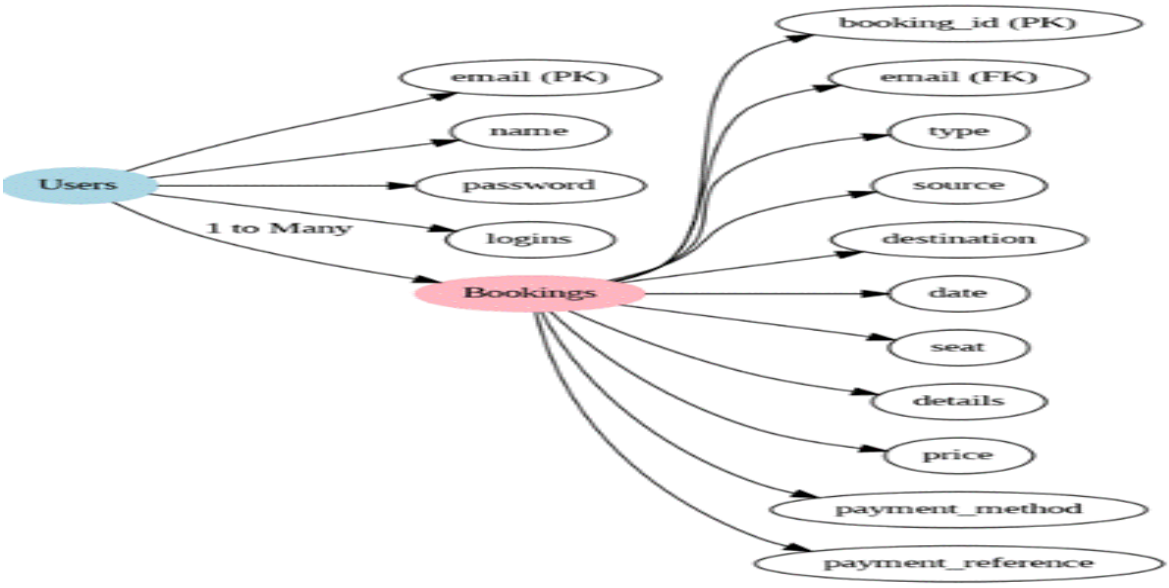
Architecture:

This AWS-based architecture powers a scalable and secure web application using Amazon EC2 for hosting the backend, with a lightweight framework like Flask handling core logic. Application data is stored in Amazon DynamoDB, ensuring fast, reliable access, while user access is managed through AWS IAM for secure authentication and control.



Entity Relationship (ER)Diagram:

An ER (Entity-Relationship) diagram visually represents the logical structure of a database by defining entities, their attributes, and the relationships between them. It helps organize data efficiently by illustrating how different components of the system interact and relate. This structured approach supports effective database normalization, data integrity, and simplified query design.



Pre-requisites:

- AWS Account Setup : https://docs.aws.amazon.com/accounts/latest/reference/getting-started.html
- AWS IAM (Identity and Access Management) : https://docs.aws.amazon.com/IAM/latest/UserGuide/introduction.html
- AWS EC2 (Elastic Compute Cloud) : https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/concepts.html
- AWS DynamoDB : https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Introduction.html
- Amazon SNS : https://docs.aws.amazon.com/sns/latest/dg/welcome.html
- Git Documentation : https://git-scm.com/doc
- VS Code Installation : (download the VS Code using the below link or you can get that in Microsoft store) https://code.visualstudio.com/download

Project WorkFlow:

Milestone 1. Backend Development and Application Setup

- Develop the Backend Using Flask.
- Integrate AWS Services Using boto3.

Milestone 2. AWS Account Setup and Login

- ***Set up an AWS account if not already done.***

- ***Log in to the AWS Management Console***

Milestone 3. DynamoDB Database Creation and Setup

- Create a DynamoDB Table.
- Configure Attributes for User Data and Book Requests.

Milestone 4. SNS Notification Setup

- Create SNS topics for book request notifications.
- Subscribe users and library staff to SNS email notifications.

Milestone 5. IAM Role Setup

- Create IAM Role
- Attach Policies

Milestone 6. EC2 Instance Setup

- Launch an EC2 instance to host the Flask application.
- Configure security groups for HTTP, and SSH access.

Milestone 7. Deployment on EC2

- Upload Flask Files
- Run the Flask App

Milestone 8. Testing and Deployment

- Conduct functional testing to verify user registration, login, book requests, and notifications.

Milestone 1: Web Application Development and Setup

Backend Development and Application Setup focuses on establishing the core structure of the application. This includes configuring the backend framework, setting up routing, and integrating database connectivity. It lays the groundwork for handling user interactions, data management, and secure access.

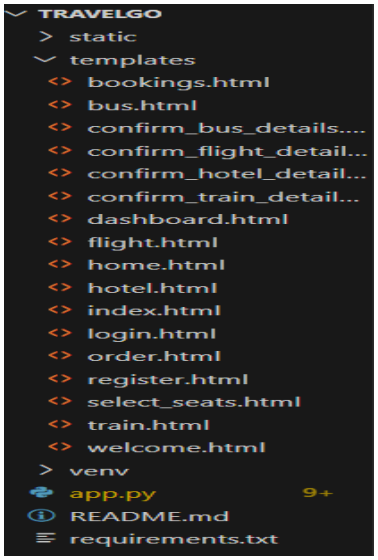
Important Instructions:

- Start by creating the necessary HTML pages and Flask routes (app.py) to build the core functionality of your application.
- During the initial development phase, store and retrieve data using Python dictionaries or lists locally. This will allow you to design, test, and validate your application logic without external database dependencies.
- Ensure your app runs smoothly with local data structures before integrating any cloud services.

Post Troven Access Activation:

- Once Troven Labs access is provided (valid for 3 hours), you must immediately proceedwithMilestone1ofyourGuidedProjectinstructions.
- At this point, modify your app.py and replace local dictionary/list operations with AWS services (such as DynamoDB, RDS, or others as per project requirements).
- Using the temporary credentials provided by Troven Labs, securely connect your application to AWS resources
- Since the AWS configuration is lightweight and already instructed in the milestones, you should be able to complete the cloud integration efficiently within the allotted time.

LOCAL DEPLOYMENT:File explorer:



Description: Organize the project with HTML templates for each feature (e.g., login, wishlist, quiz, checkout) under the templates folder and manage backend logic in application.py.

Activity 3.1:Develop the Backend Using Flask

Import libraries:

```
from flask import Flask, render_template, request, redirect, session, url_for, jsonify,flash
import pymongo
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
import random
import urllib.parse
from bson.objectid import ObjectId
```

Description: Import essential Flask modules for web handling, Boto3 for AWS integration, Werkzeug for password hashing, and date time for timestamp management.

Flask App Initialization:

```
# Initialize the Flask application
application = Flask(__name__)
application.secret_key = 'your_secret_key'
```

Description: Initialize the Flask application and set a secret key to securely manage user sessions and form data.

Database Configuration

```
#Initialize dynamodb resource
dynamodb = boto3.resource('dynamodb', region_name='us-east-1')

#Define dynamodb tables
users_table = dynamodb.Table('travelgo_users')
trains_table = dynamodb.Table('trains')
bookings_table = dynamodb.Table('bookings')
```

Description: Connect to DynamoDB using Boto3 and define references to the User Table,trains and Bookings table

Routes for Core Functionalities:

Home and registration routes:

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

@app.route('/register', methods=['GET', 'POST'])
def register():
    if request.method == 'POST':
        email = request.form['email']
        username = request.form['username']
        password = generate_password_hash(request.form['password'])
        users.insert_one({"email": email, "username": username, "hashed_password": password, "login_count": 0})
        return redirect(url_for('login'))
    return render_template('register.html')
```

Description: Create the home and registration routes, where the registration route securely hashes user passwords and stores user data in DynamoDB upon form submission.

Login routes:

```
@app.route('/login', methods=['GET', 'POST'])
def login():
    if request.method == 'POST':
        email = request.form['email']
        password = request.form['password']
        user = users.find_one({'email': email})
        if user and check_password_hash(user['hashed_password'], password):
            session['email'] = email
            session['username'] = user['username']
            users.update_one({'email': email}, {'$inc': {'login_count': 1}})
            return redirect(url_for('welcome'))
        else:
            return render_template('login.html', error="Invalid credentials.")
    return render_template('login.html')
```

Description: Implement the user login route to validate credentials using DynamoDB and securely manage session data while updating the user's login count.

Welcome:

```
@app.route('/welcome')
def welcome():
    if 'email' not in session:
        return redirect(url_for('login'))
    return render_template('welcome.html', username=session.get('username'))
```

Description: Implement the welcome route to greet users upon login, fetch personalized user details from DynamoDB, and provide access to key TravelGo features through a user-friendly interface.

Dashboard:

```
@app.route('/dashboard')
def dashboard():
    if 'email' not in session:
        return redirect(url_for('login'))
    username = session.get('username', 'User')
    user_email = session['email']
    bookings = []
    for b in bookings_collection.find({'user_email': user_email}):
        booking_type = b.get('booking_type', 'Unknown').capitalize()
        if b.get('booking_type') in ['bus', 'train', 'flight']:
            details = f"{b.get('source')} → {b.get('destination')} @ {b.get('travel_date')}"
        elif b.get('booking_type') == 'hotel':
            details = f"{b.get('hotel_name')} from {b.get('checkin_date')}"
        else:
            details = "Details not available"
        bookings.append({
            'type': booking_type,
            'details': details,
            'date': b.get('booking_date', 'N/A')[:16].replace('T', ' '),
            'booking_id': str(b.get('_id'))
        })
    return render_template('dashboard.html', username=username, bookings=bookings)
```

Description: Secure the user dashboard and implement route to store them in DynamoDB with item details and a timestamp.

Bus,Train,Flight,Hotel routes:

```
@app.route('/bus')
def bus_booking_page():
    return render_template('bus.html') if 'email' in session else redirect(url_for('login'))

@app.route('/train')
def train_booking_page():
    return render_template('train.html') if 'email' in session else redirect(url_for('login'))


@app.route('/flight')
def flight_booking_page():
    return render_template('flight.html') if 'email' in session else redirect(url_for('login'))


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

Description: Implement add routes to allow admins to insert new bus, train, flight, and hotel details into DynamoDB, ensuring proper data validation and seamless integration with the booking system.

Bookings route:

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

    email = session['email']
    bookings = list(bookings_collection.find({'user_email': email}))
    return render_template('bookings.html', bookings=bookings)
```

Description: Implement booking routes to capture user booking details for buses, trains, flights, and hotels, store them in DynamoDB, and trigger AWS SNS notifications upon successful booking.

Select Seats Routes:

```python
@app.route('/select_seats', methods=['GET', 'POST'])
def select_seats():
    if 'pending_booking' not in session:
        return "No pending booking found.", 400

    booking = session['pending_booking']

    if request.method == 'POST':
        selected = request.form.get('selected_seats', '')
        if selected:
            booking['selected_seats'] = selected
            booking['booking_date'] = datetime.now().isoformat()
            db.bookings.insert_one(booking)
            session.pop('pending_booking', None)
            return redirect(url_for('bookings'))  # or dashboard

    return render_template('select_seats.html', booking=booking)
```

Description: Create seat selection routes that display available seats dynamically, allow users to choose their preferred seats, and update the selected status in DynamoDB upon confirmation.

Initiating Flask app:

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

Description: Start the Flask application on host 0.0.0.0 and port 5000 in debug mode to enable live reloads and detailed error logs during development.

Milestone 2 : AWS Account Setup

Important Notice: Use Troven Labs for AWS Access

Students are strictly advised not to create their own AWS accounts, as doing so may incur charges. Instead, we have set up a dedicated section called “Labs” on the Troven platform, which provides temporary and cost-free access to AWS services.

Once your website is locally deployed and fully functional, you must proceed with integrating AWS services only through the Troven Labs environment. This ensures secure, controlled access to AWS resources without any risk of personal billing.

All steps involving AWS (such as deploying to EC2, connecting to DynamoDB, or using SNS) must be carried out within the Troven Labs platform, as we've configured temporary credentials for each student.

Reminder: You must complete the Web Development task before gaining access to Troven. Once accessed, the AWS Console via Troven is available for only 3 hours—please plan your work accordingly.

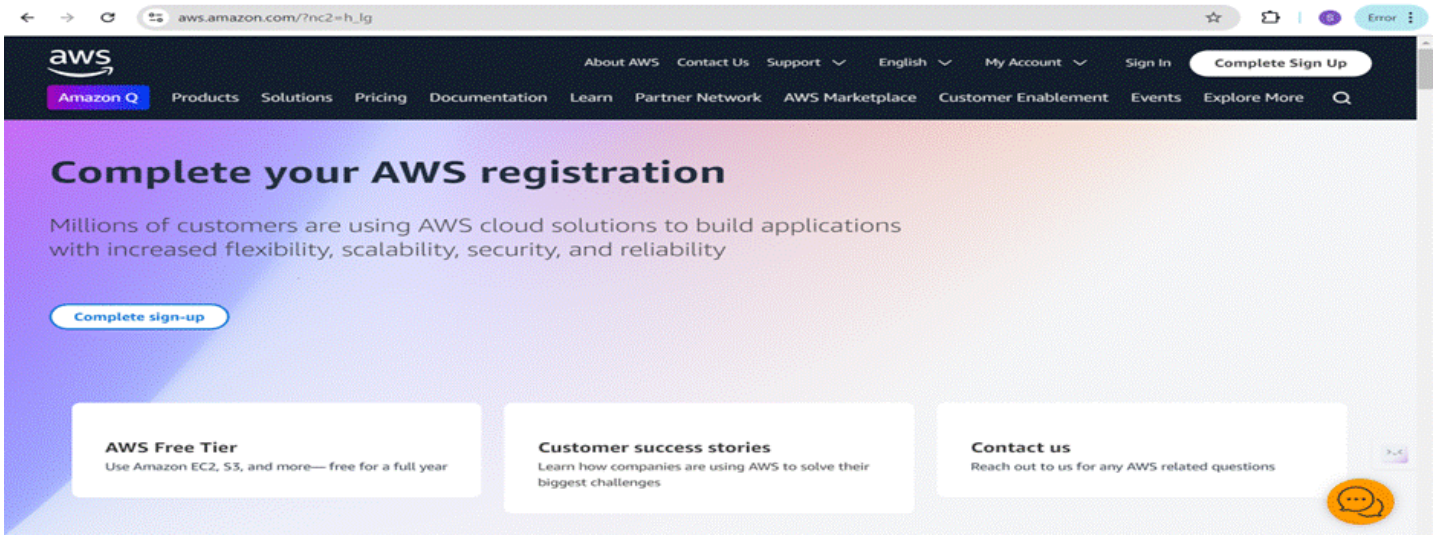
Please follow the provided guidelines and access AWS exclusively through Troven to avoid unnecessary issues.

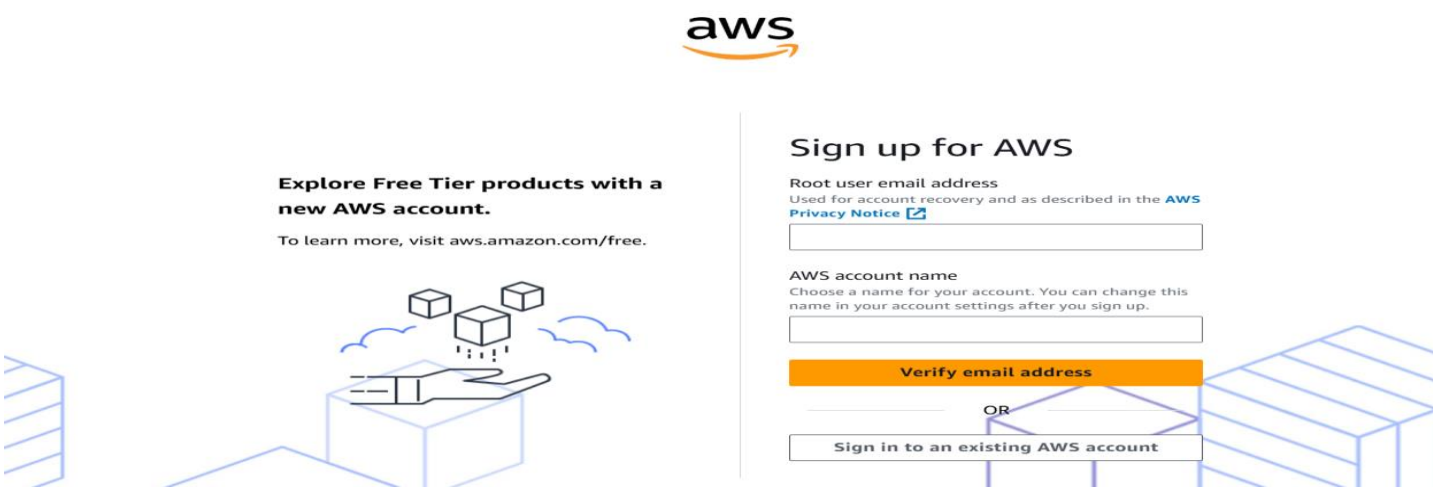
Please refer the below link - https://drive.google.com/file/d/1HzWc7AMJ2BrxhV-uaw5s0vWtcd-28qgI/view?usp=sharing

AWS Account Setup and Login:

This is for your understanding only, please refrain from creating an AWS account. A temporary account will be provided via Troven.

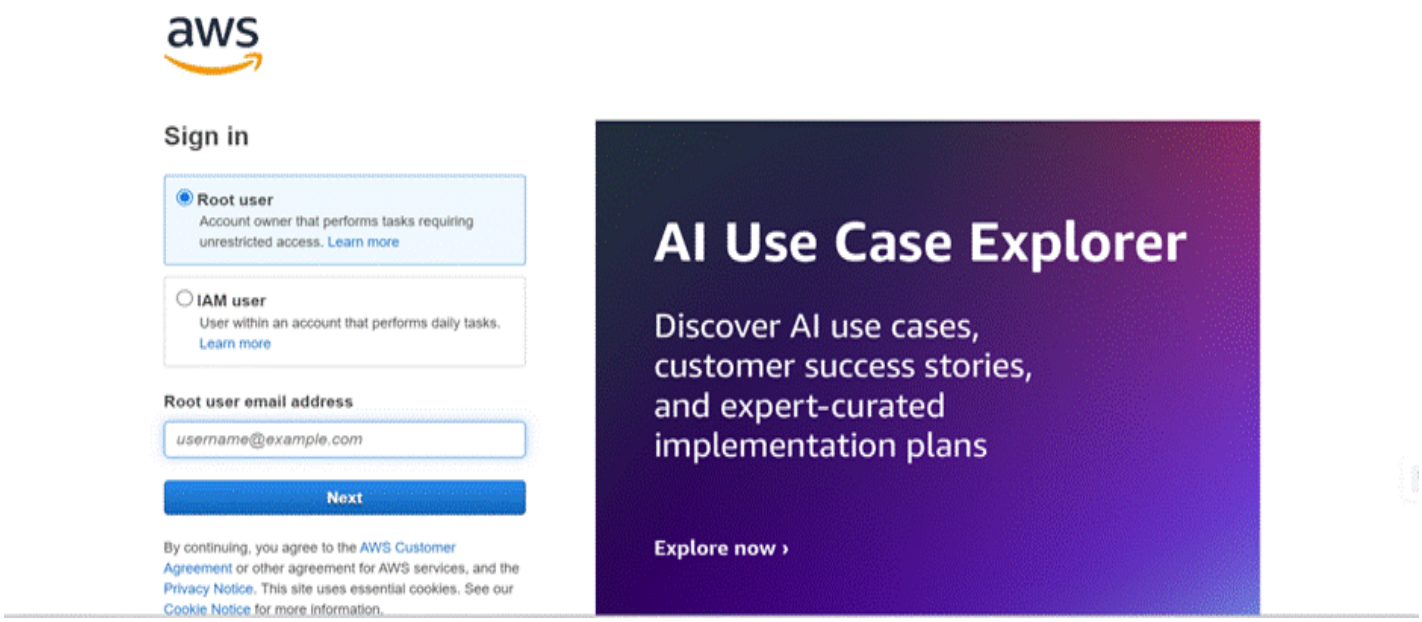
- Go to the AWS website (https://aws.amazon.com/).
- Click on the "Create an AWS Account" button.
- Follow the prompts to enter your email address and choose a password.
- Provide the required account information, including your name, address, and phone number.
- Enter your payment information. (Note: While AWS offers a free tier, a credit card or debit card is required for verification.)
- Complete the identity verification process.
- Choose a support plan (the basic plan is free and sufficient for starting).
- Once verified, you can sign in to your new AWS accounts.





Log in to the AWS Management Console

- After setting up your account, log in to the AWS Management Console.



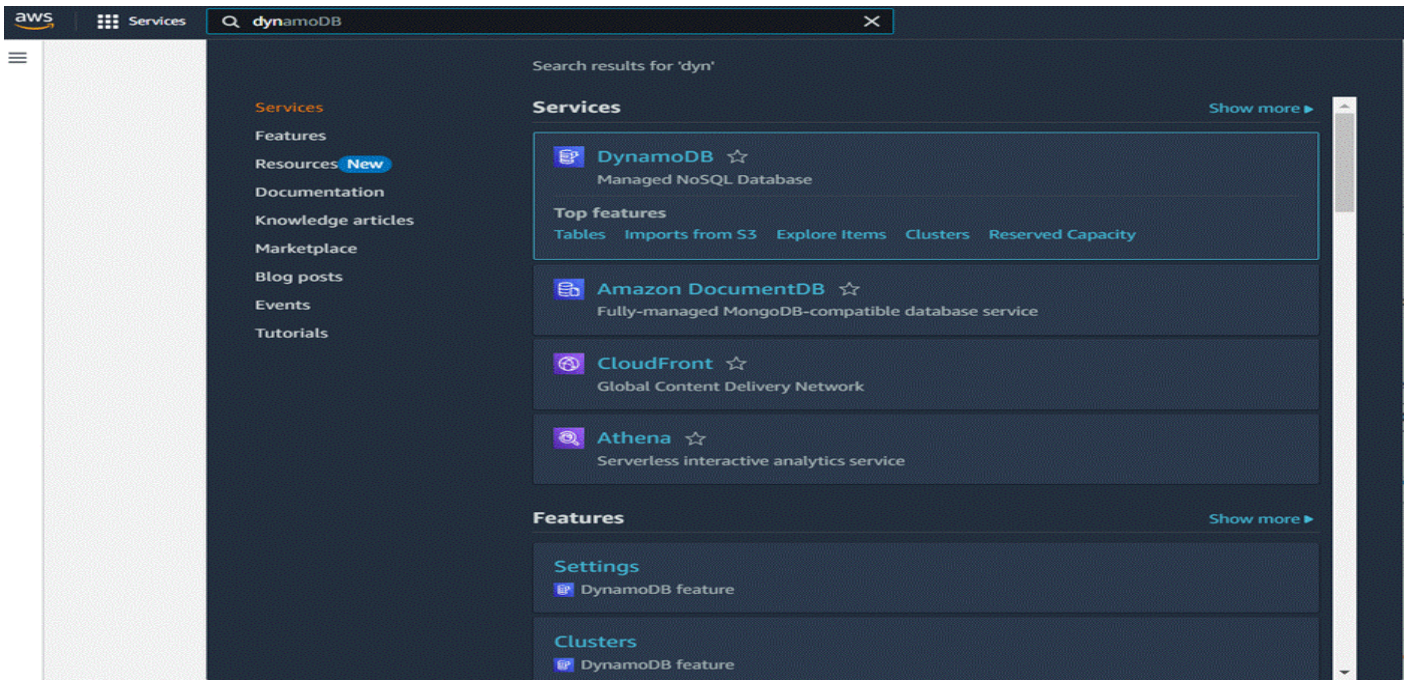
Milestone 3 : DynamoDB Database Creation and Setup

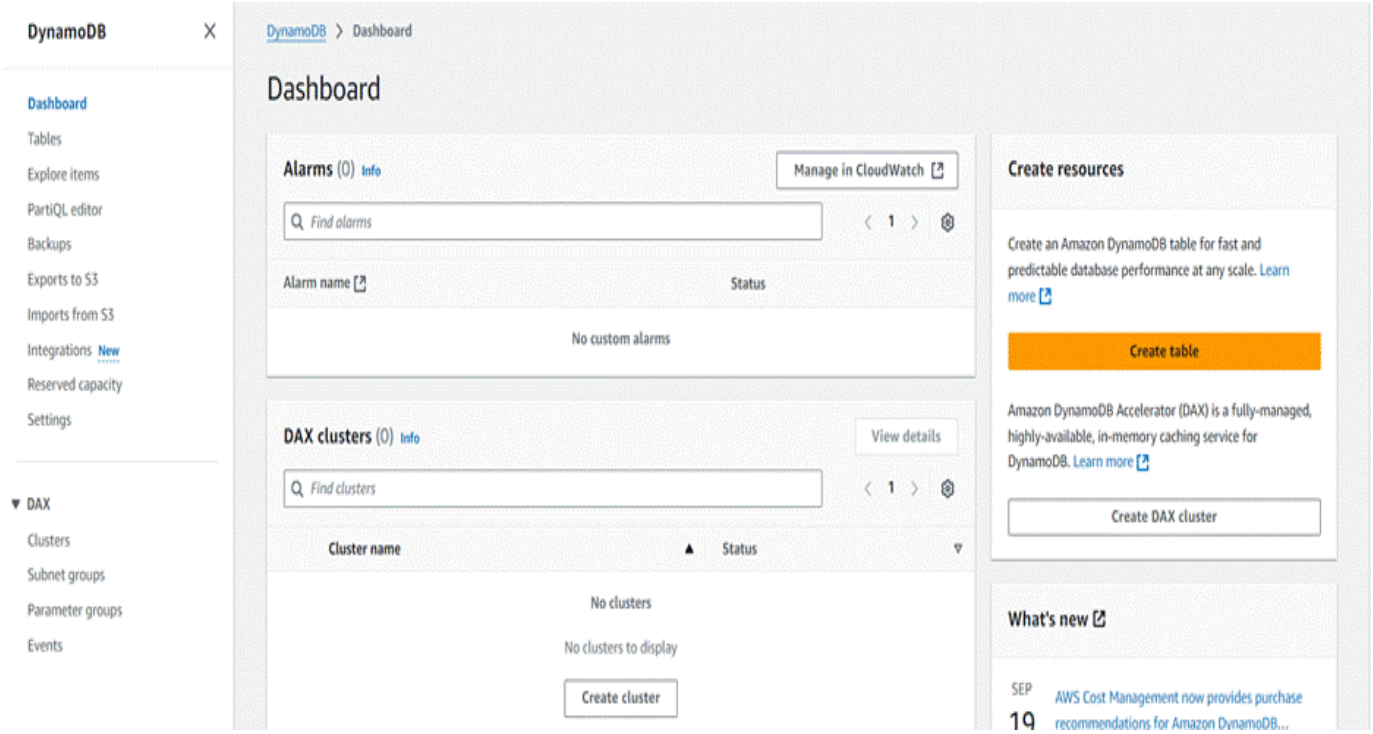
Database Creation and Setup involves initializing a cloud-based NoSQL database to store and manage application data efficiently. This step includes defining tables, setting primary keys, and configuring read/write capacities. It ensures scalable, high-performance data storage for seamless backend operations.

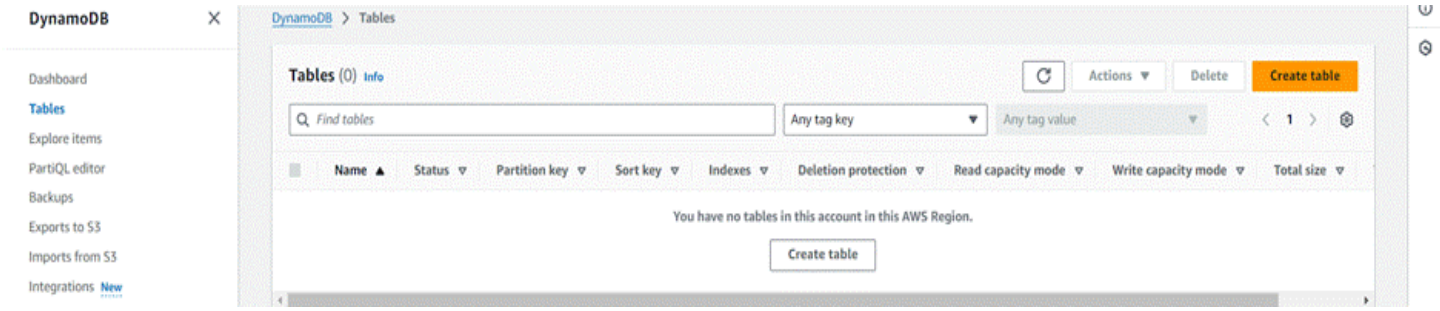
Navigate to the DynamoDB:

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

SMARTBRIDGE

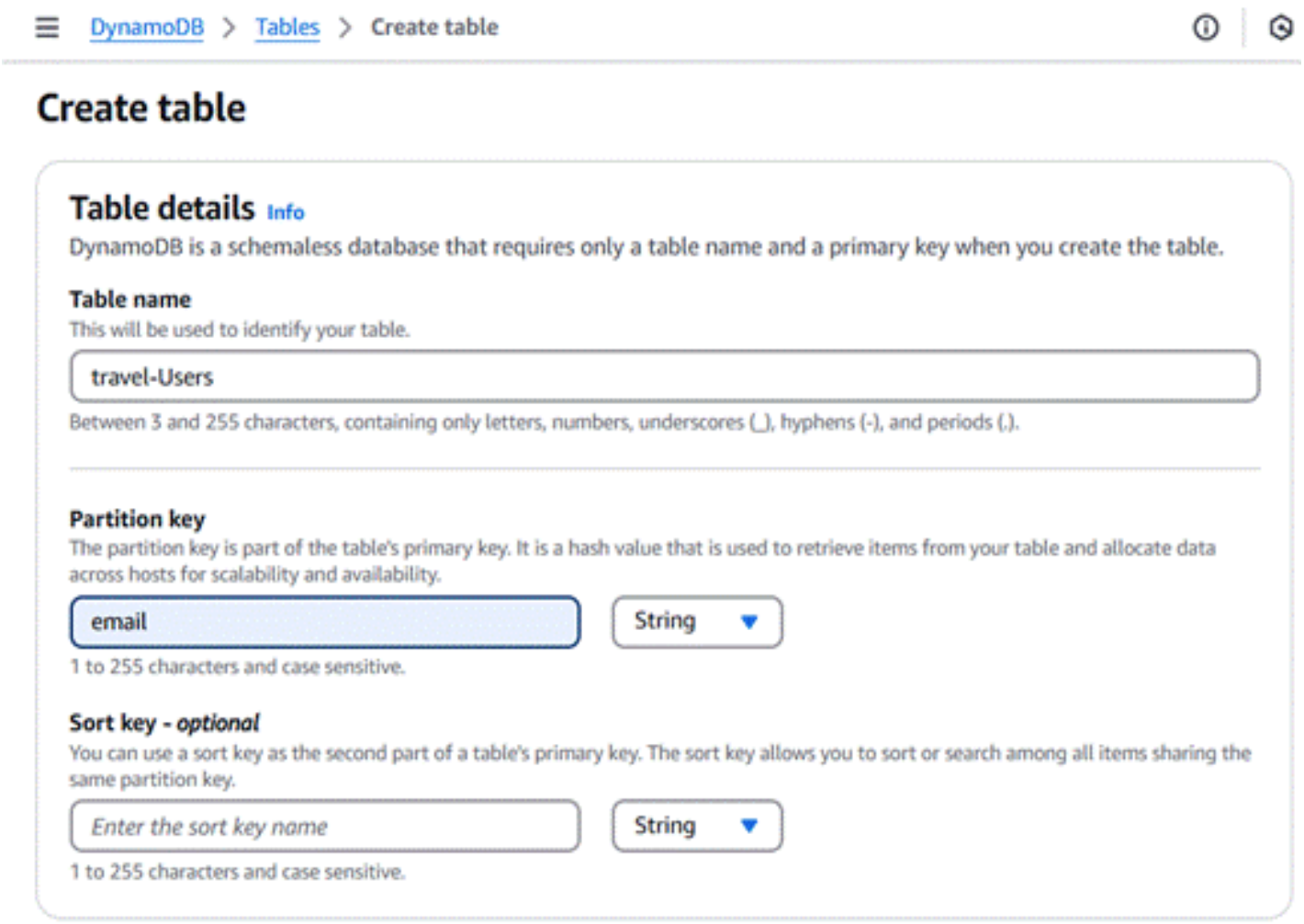


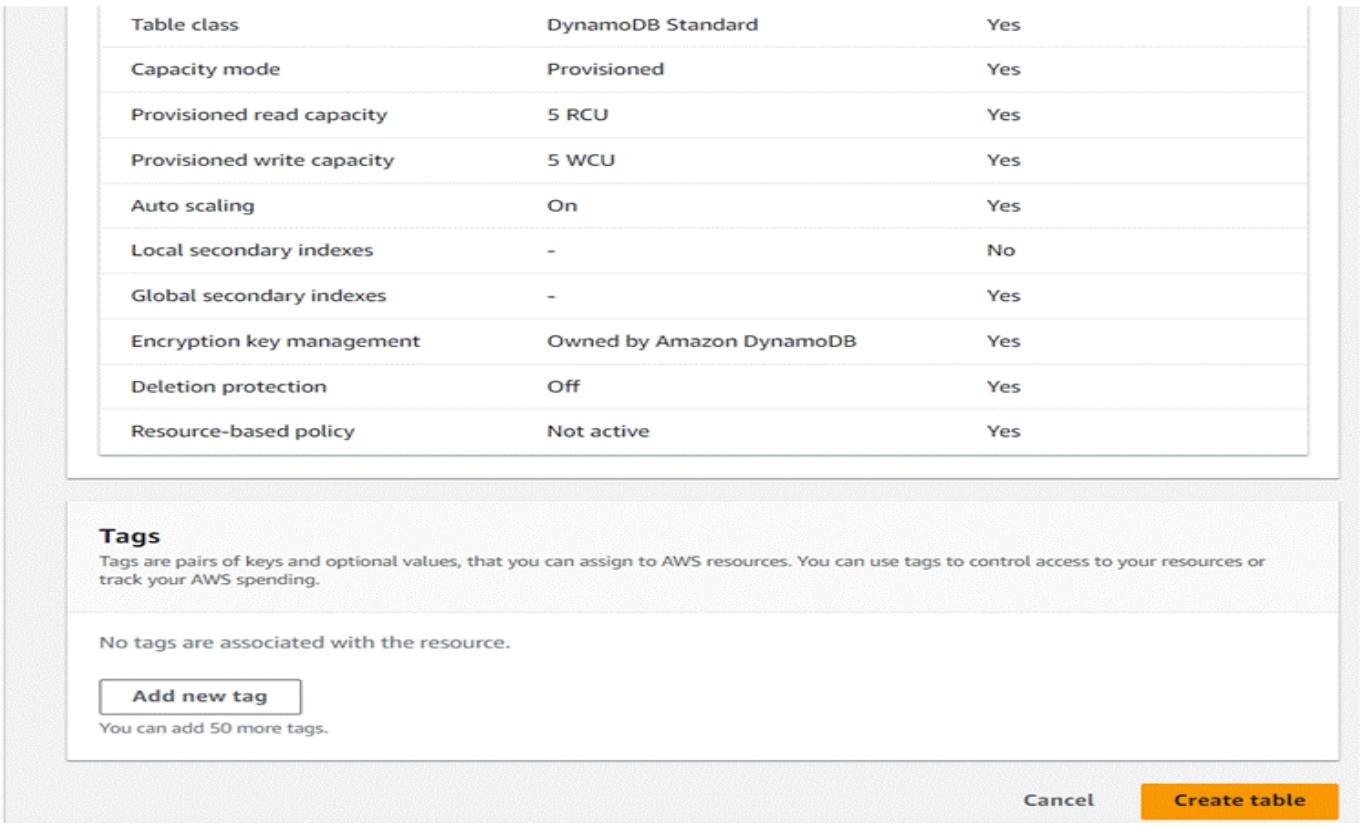




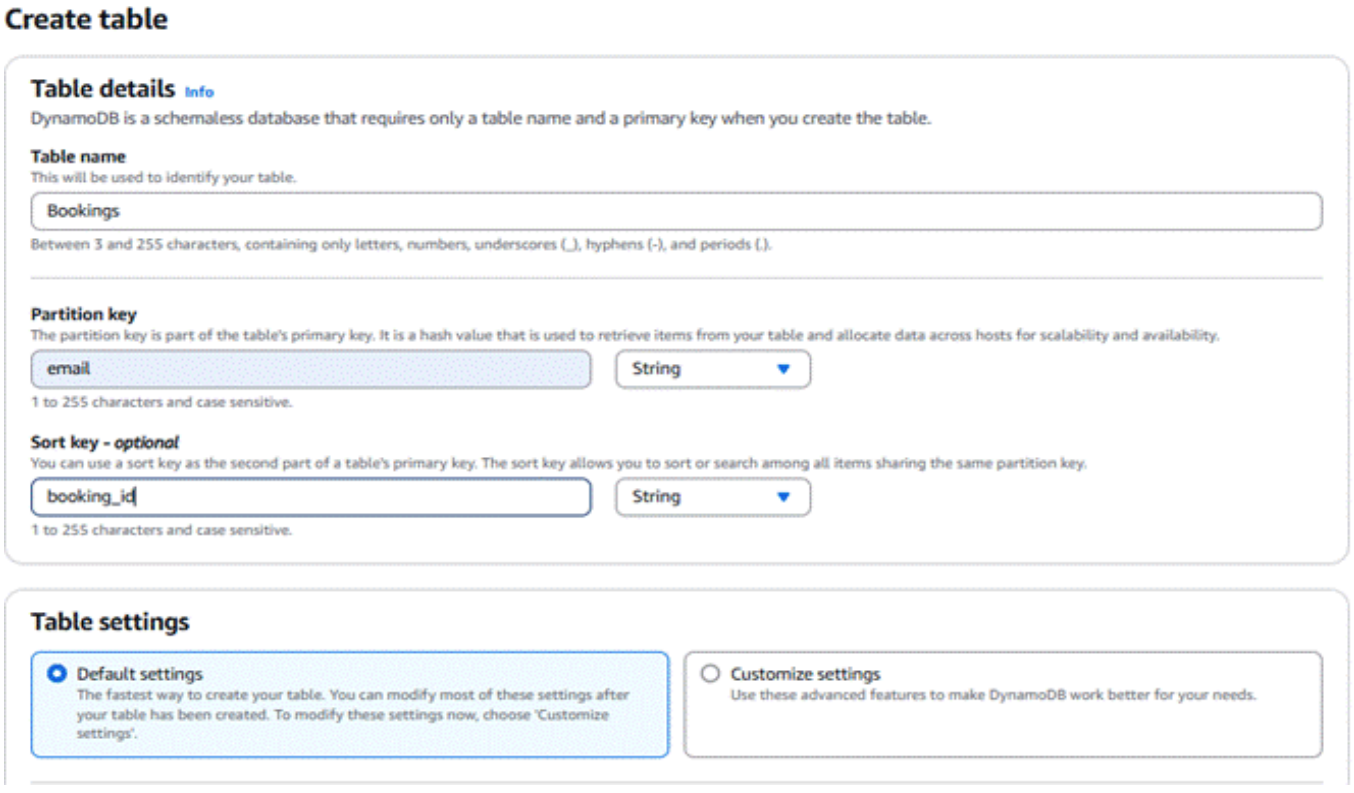
Create an DynamoDB table for storing data

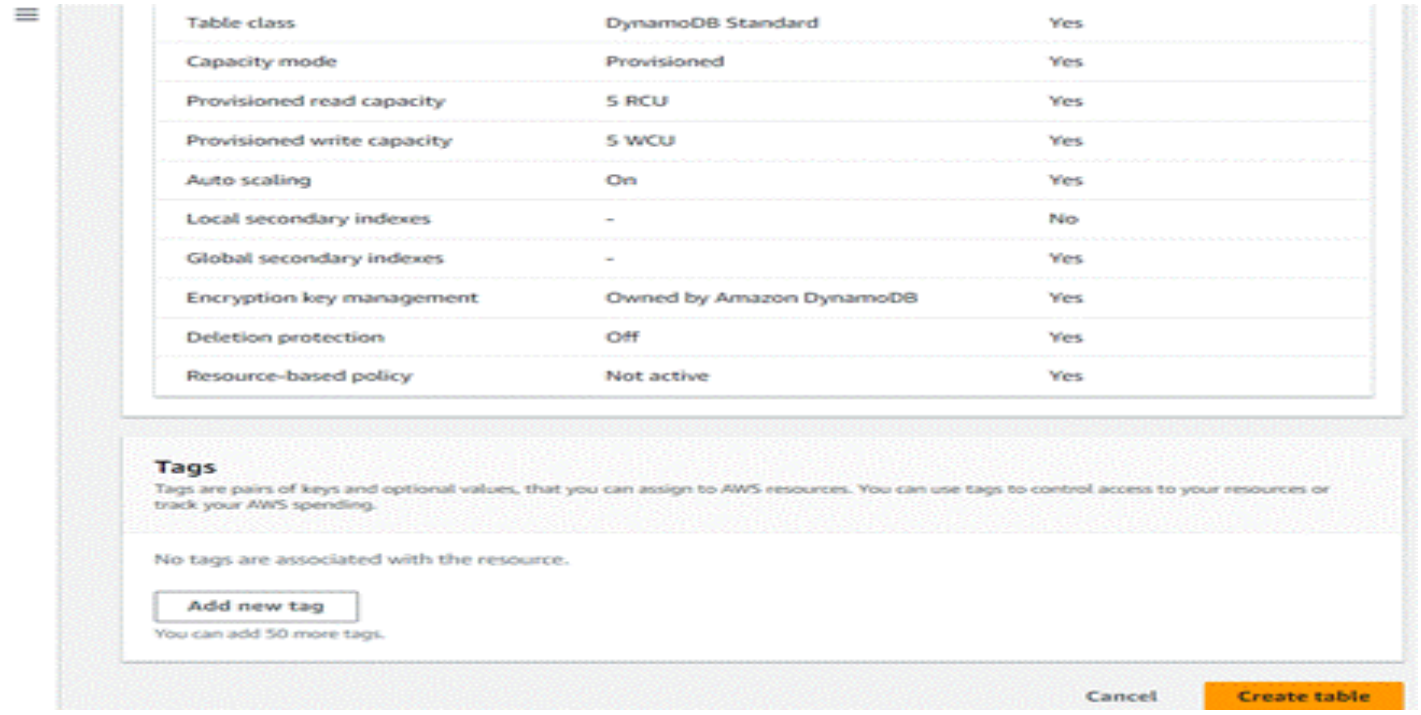
Create a travel-Users table for storing registered user details with partition key “Email” with type String and click on create tables.

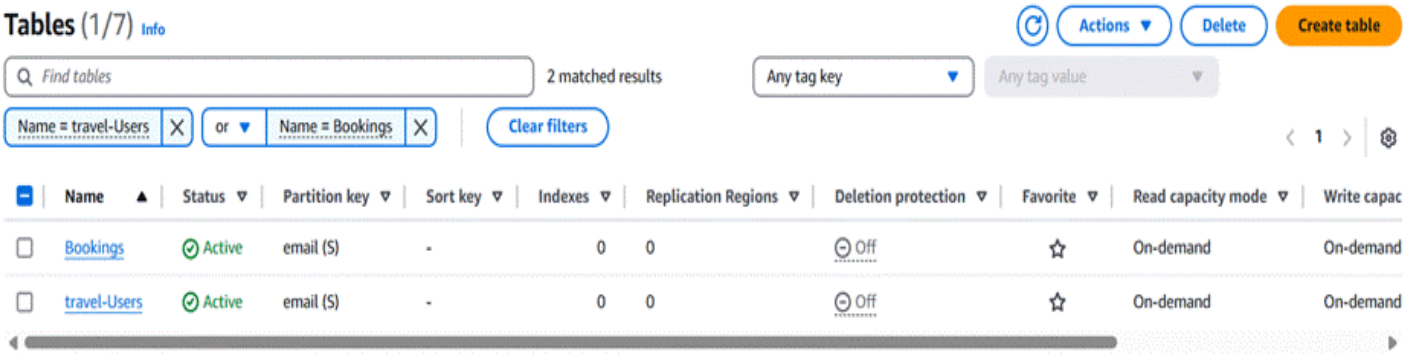




Follow the same steps to create a Bookings for storing booking records with email as the partition key and booking_id as the sort key.





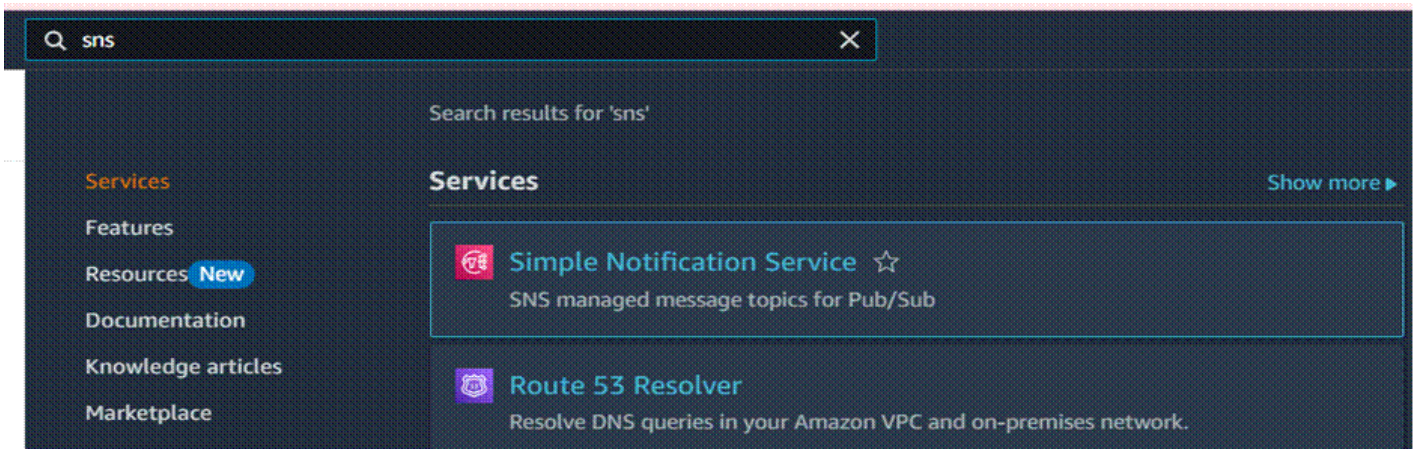


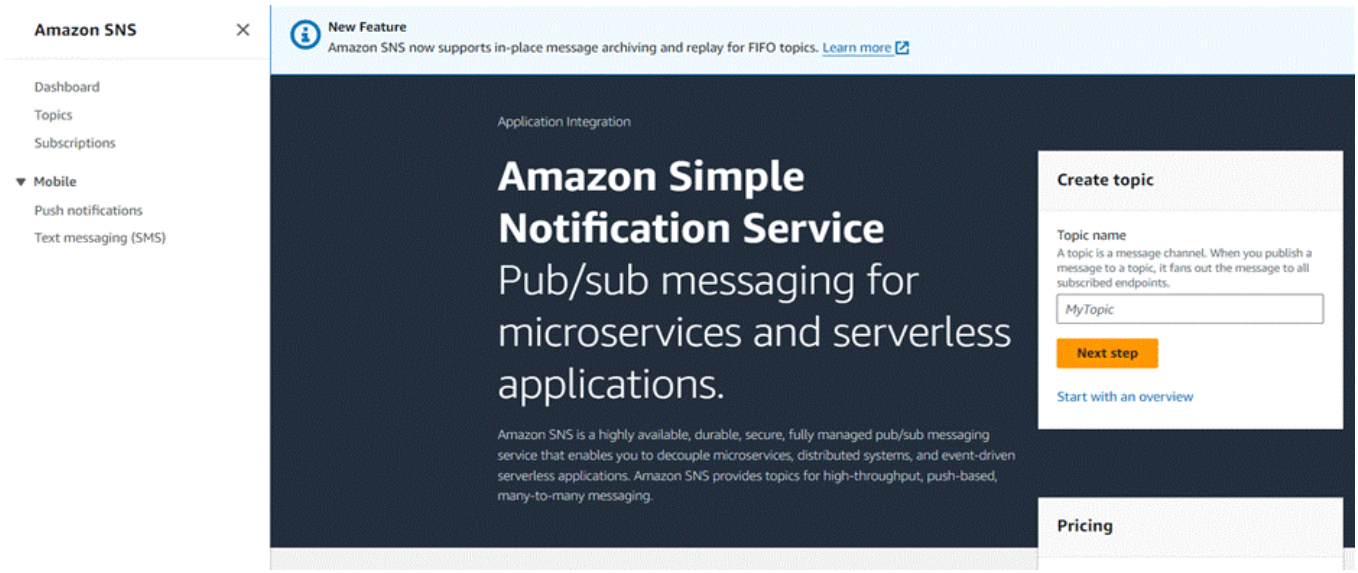
Milestone 4 : SNS Notification Setup:

Amazon SNS is a fully managed messaging service that enables real-time notifications through channels like SMS, email, or app endpoints. You create topics, configure subscriptions, and integrate SNS into your app to send notifications based on specific events.

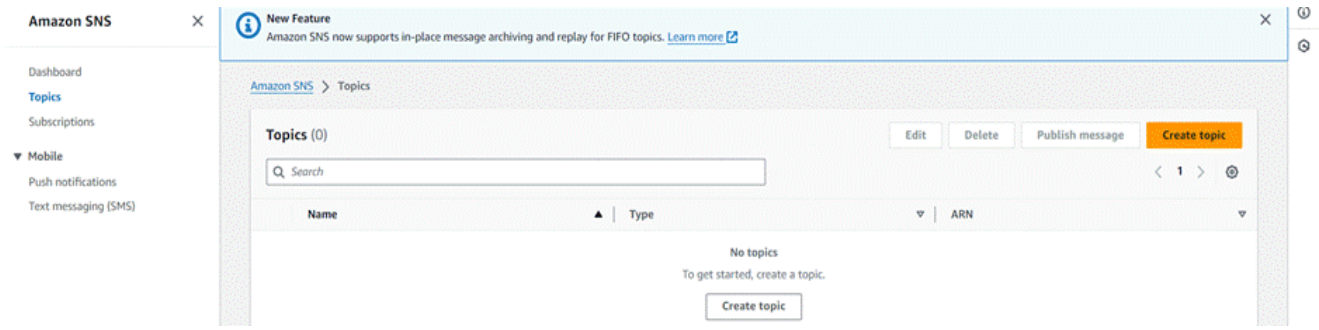
SNS Topics for email notifications

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





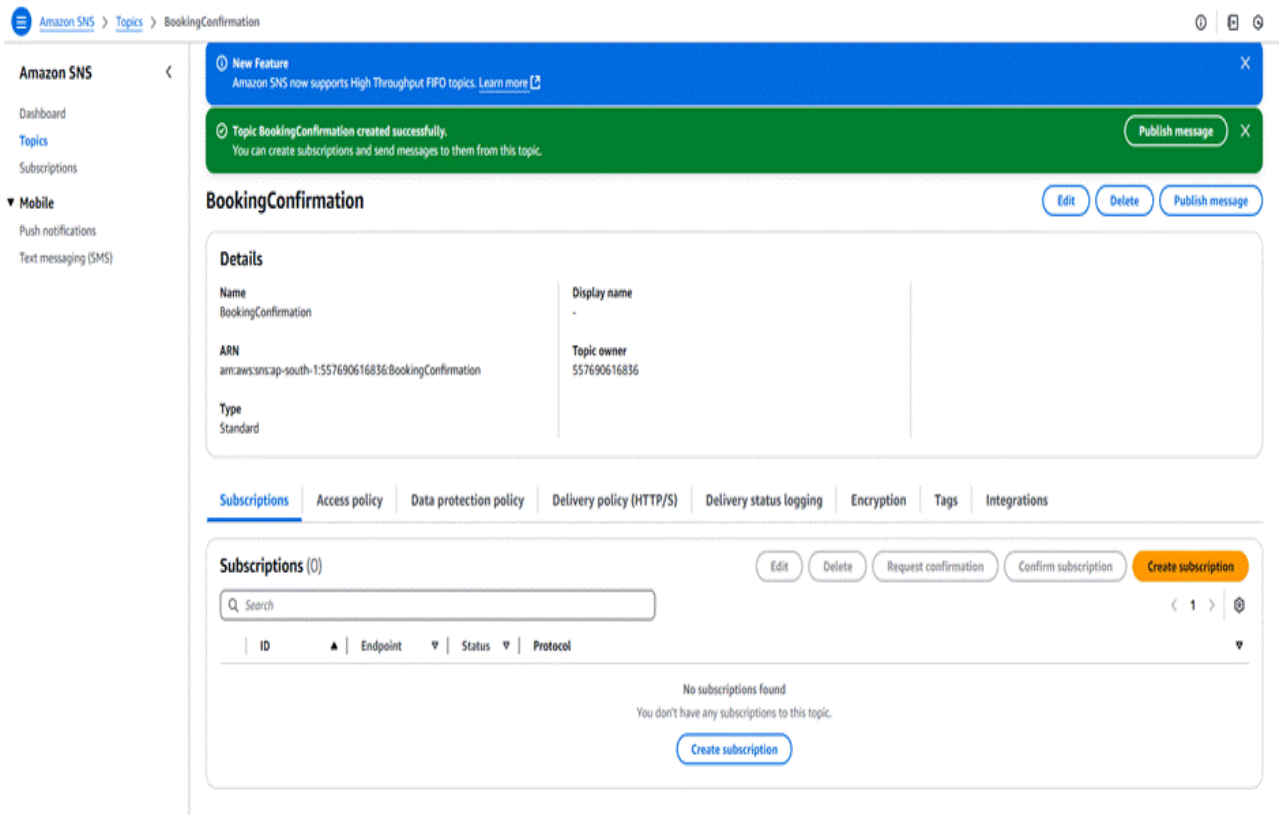
- Click on Create Topic and choose a name for the topic.



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

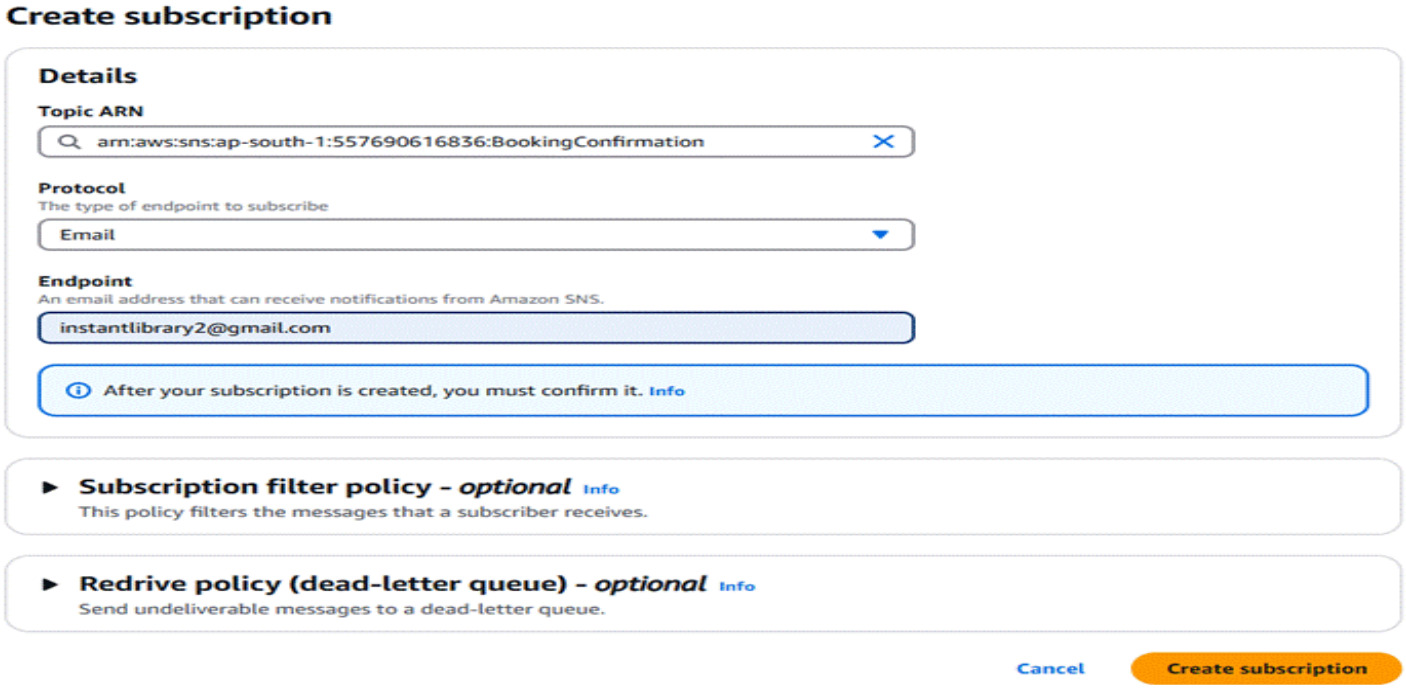
SMARTBRIDGE

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

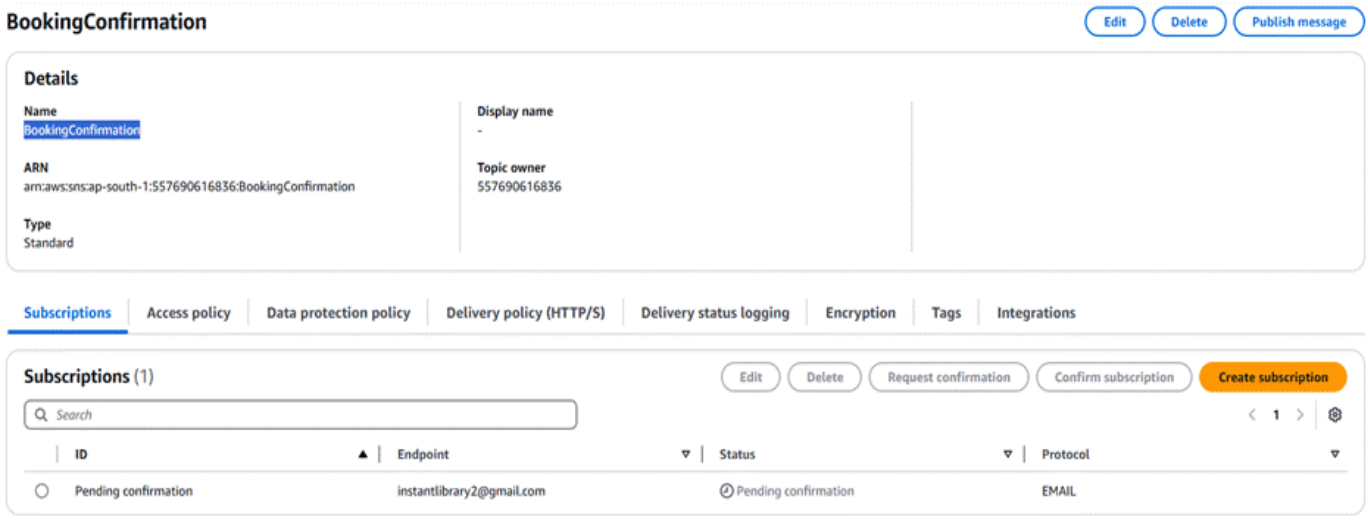


Subscribe users and admin

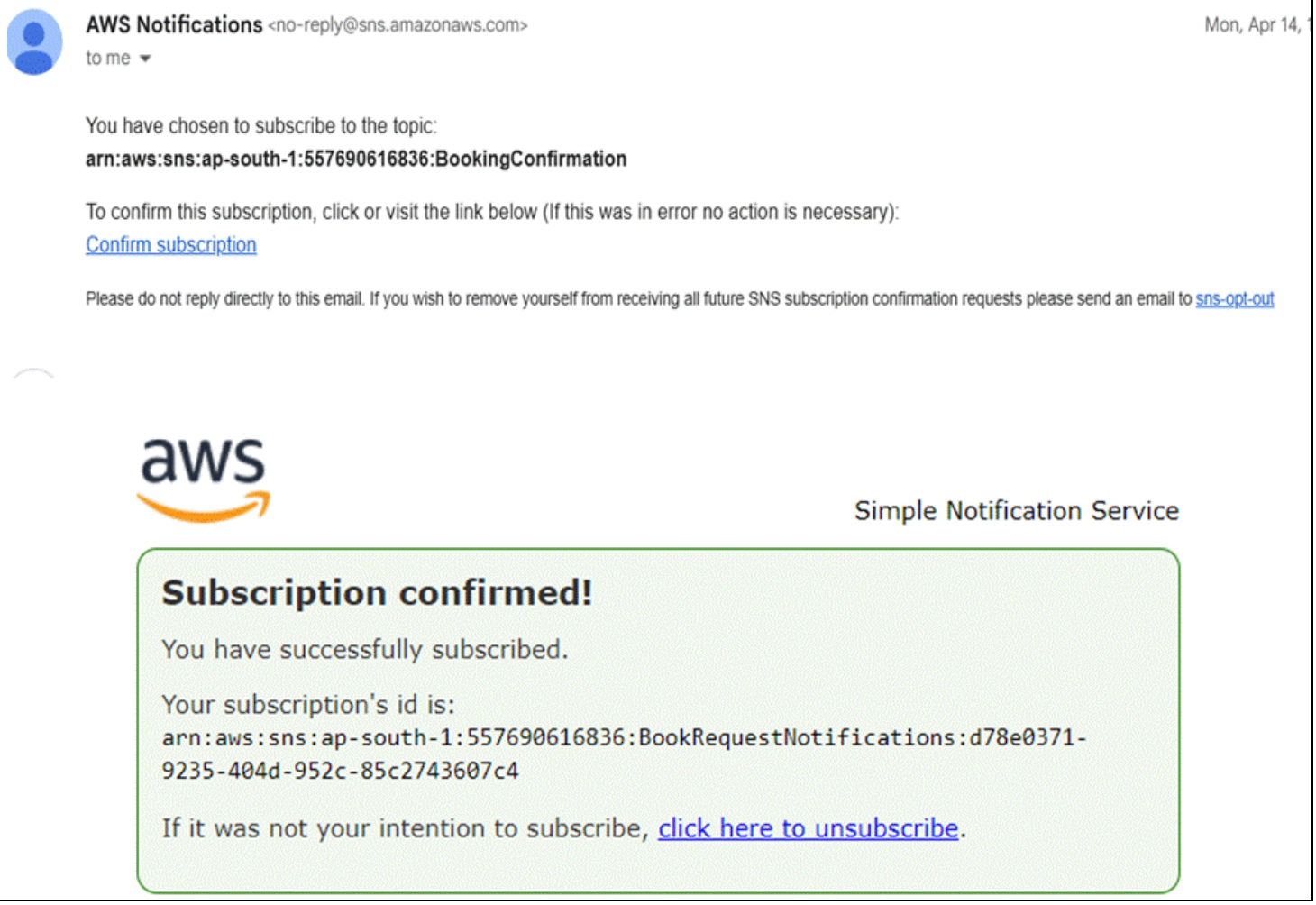
- Subscribe users (or admin staff) to this topic via Email. When a book request is made, notifications will be sent to the subscribed emails.



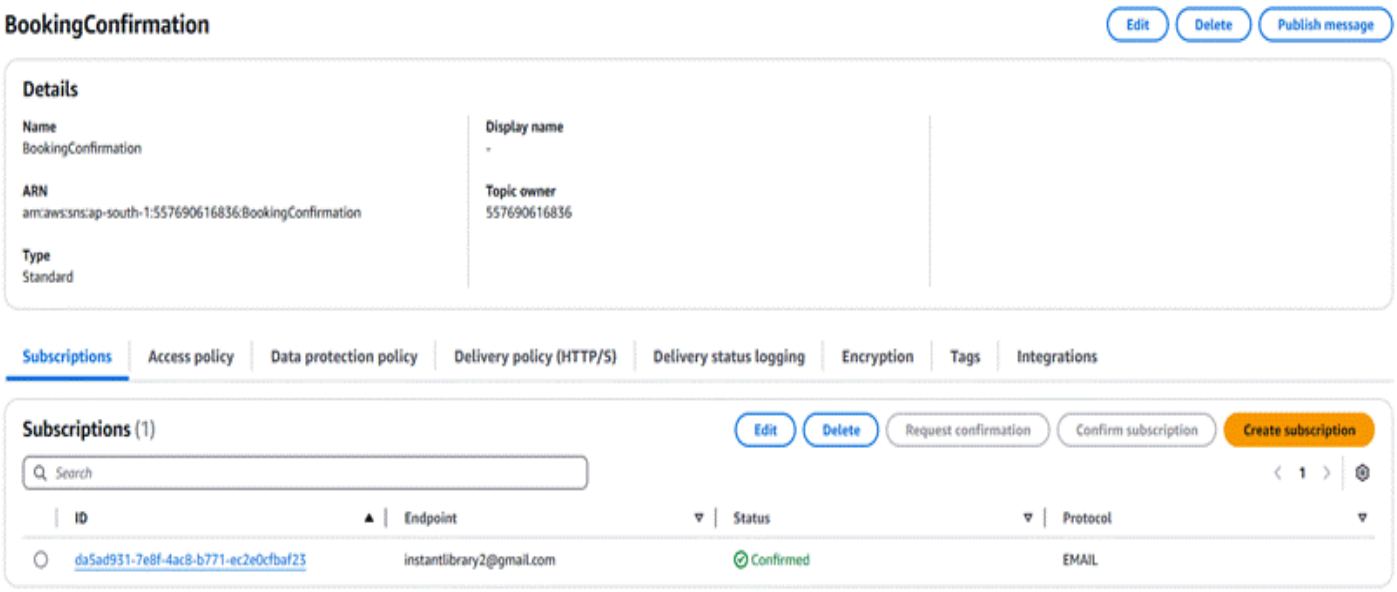
After subscription request for the mail confirmation



- 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.

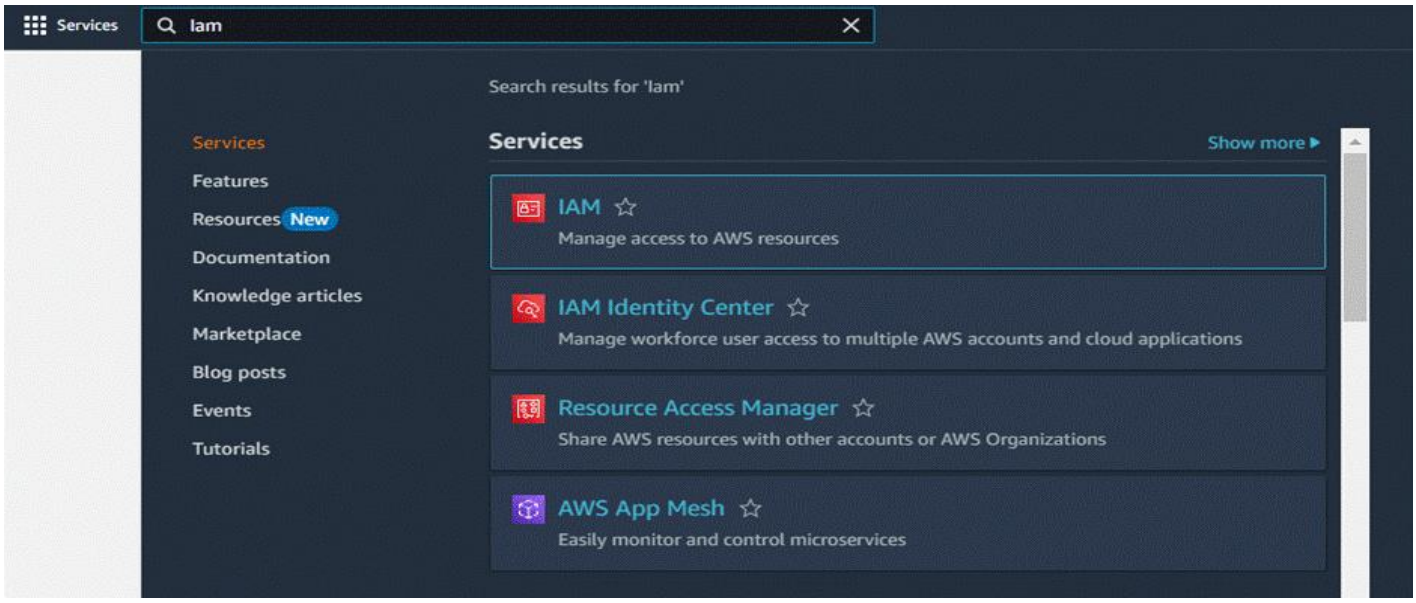


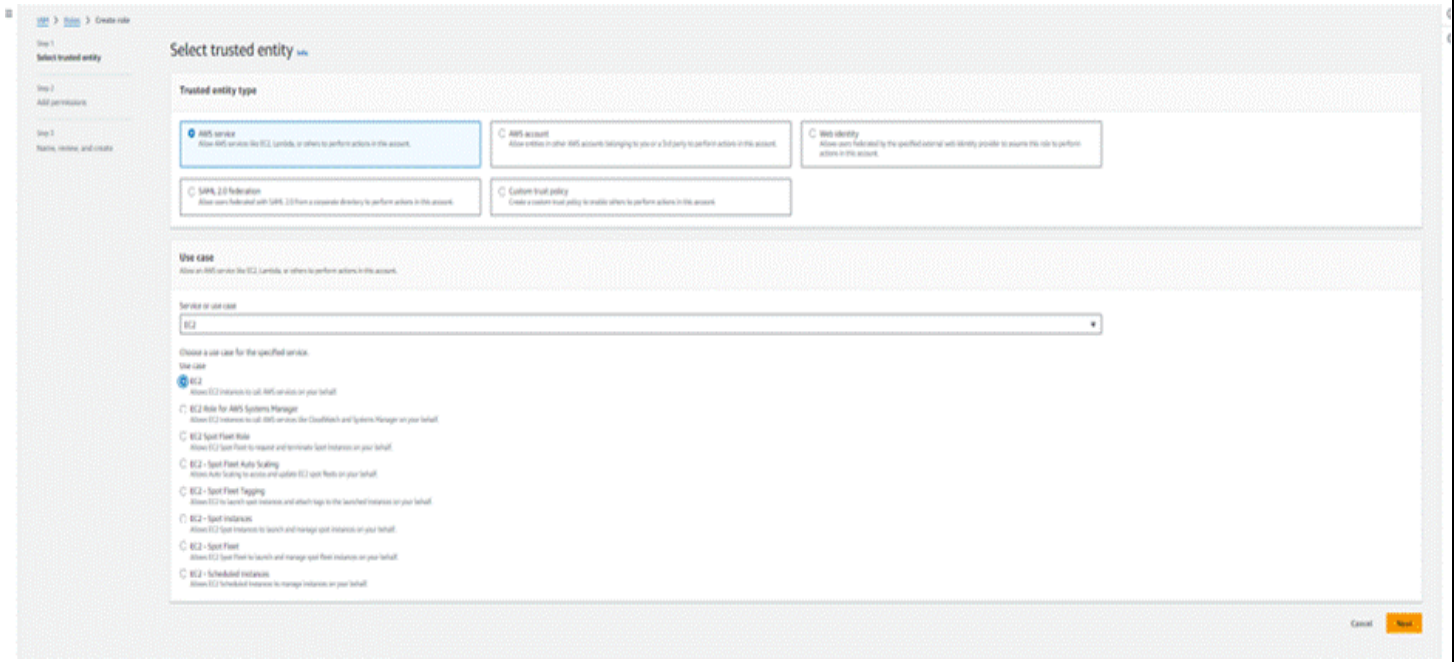
Milestone 5 : IAM Role Setup:

IAM (Identity and Access Management) role setup involves creating roles that define specific permissions for AWS services. To set it up, you create a role with the required policies, assign it to users or services, and ensure the role has appropriate access to resources like EC2, S3, or RDS. This allows controlled access and ensures security best practices in managing AWS resources.

Create IAM Role.

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



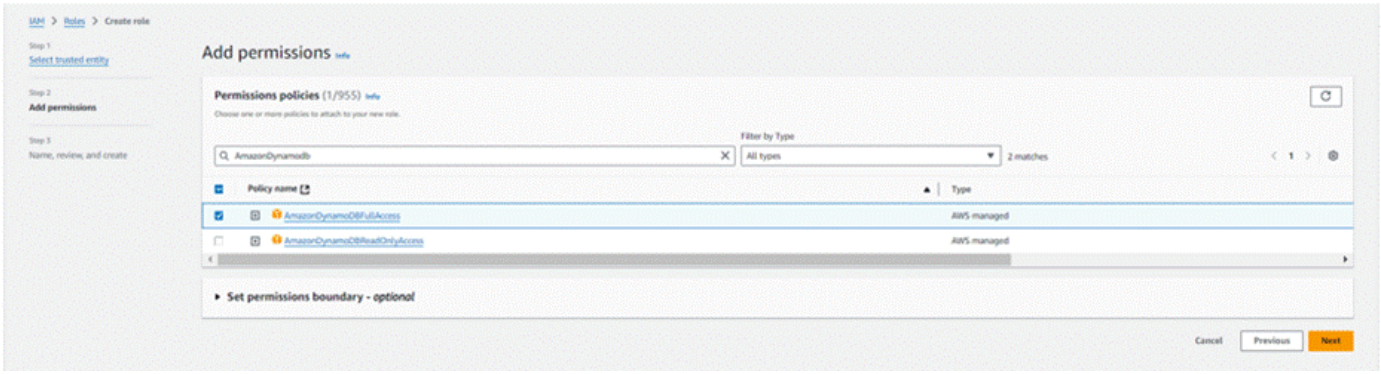


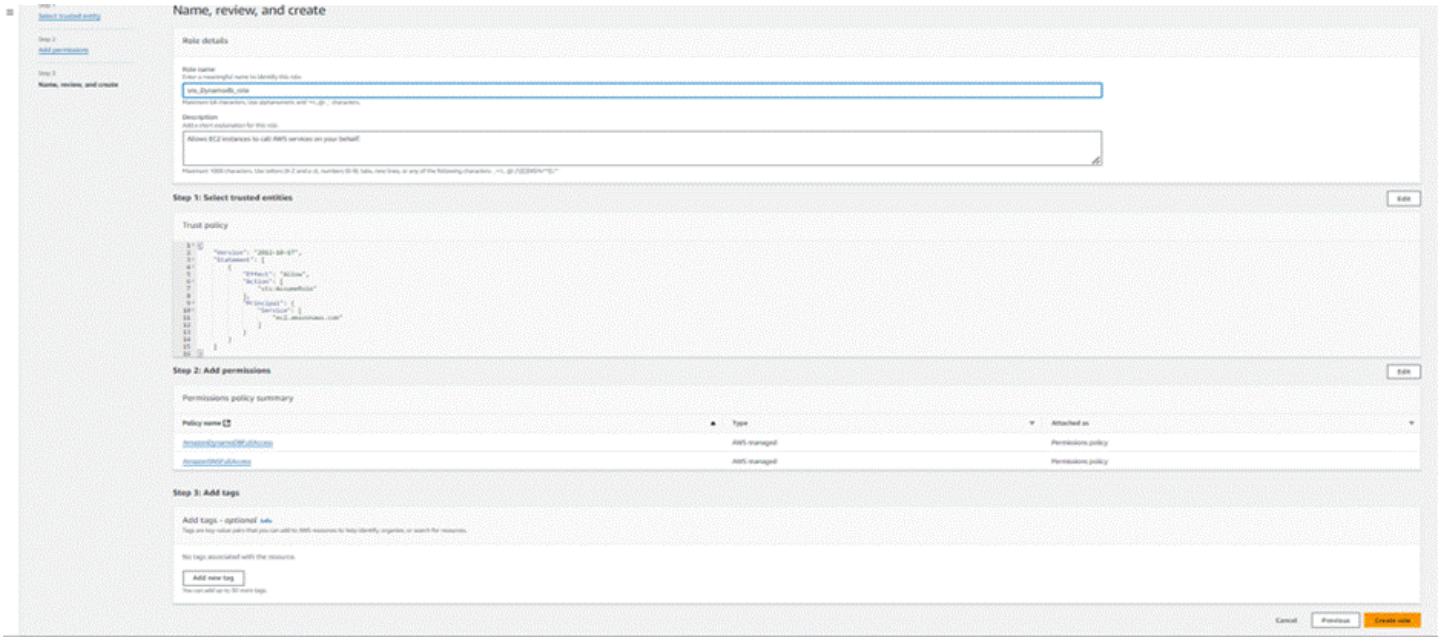
Attach Policies.

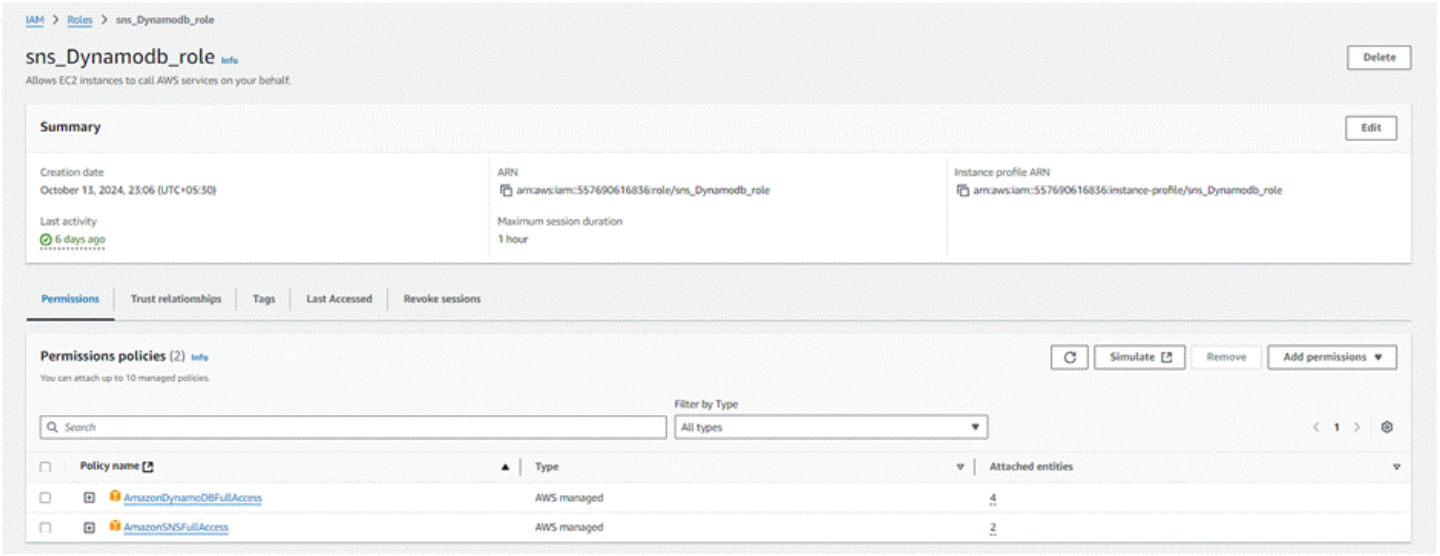
Attach the following policies to the role:

- **AmazonDynamoDBFullAccess**: Allows EC2 to perform read/write operations on DynamoDB.
- **AmazonSNSFullAccess**: Grants EC2 the ability to send notifications via SNS

SMARTBRIDGE



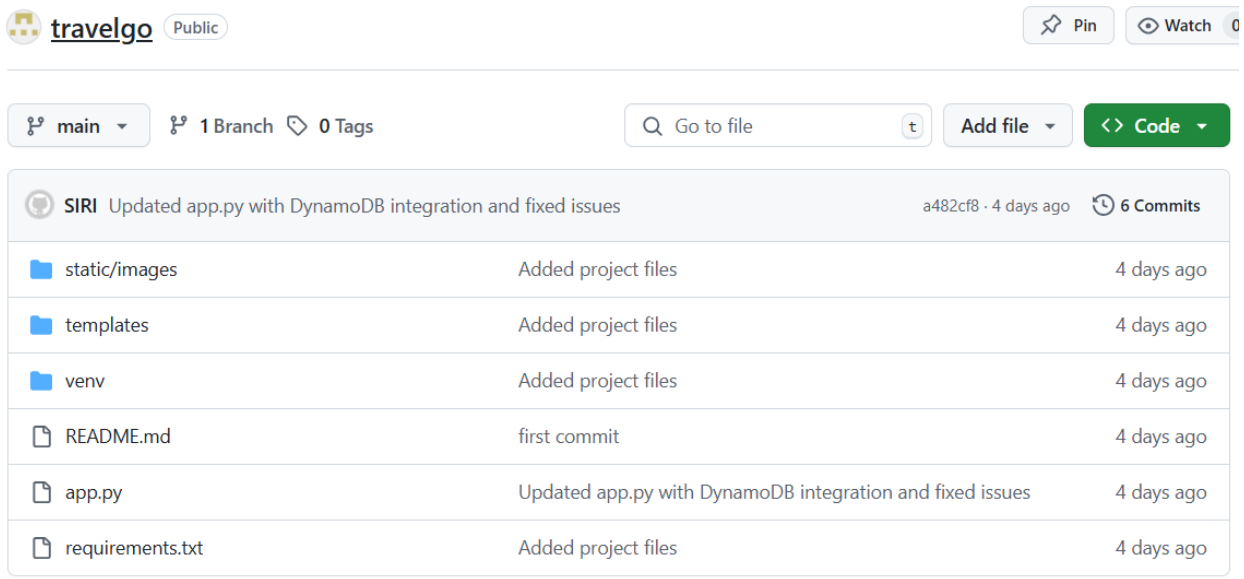


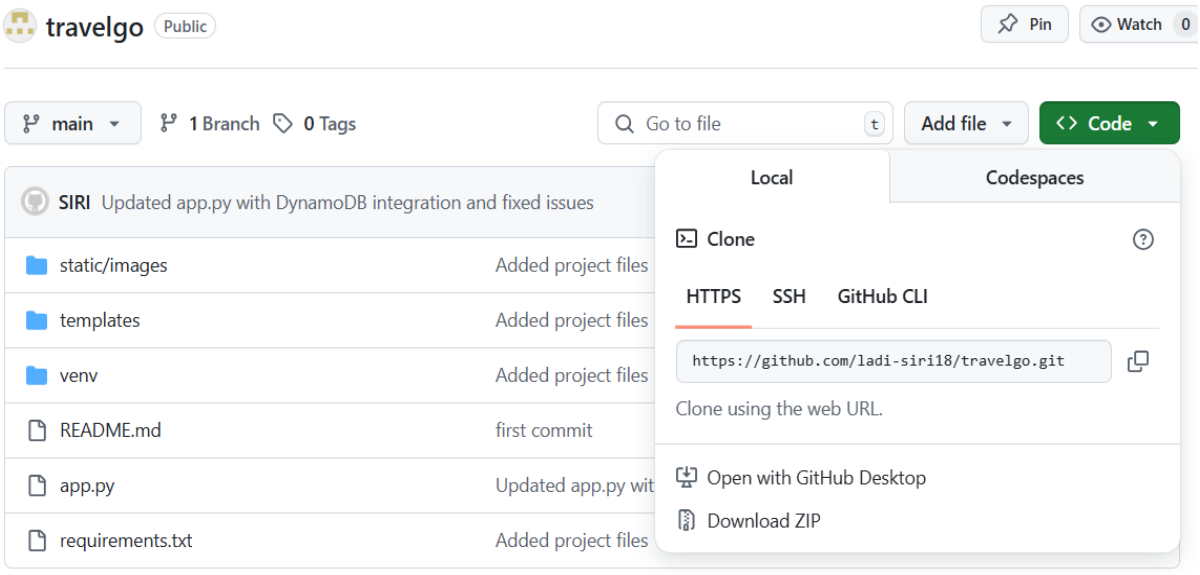


Milestone 6 : EC2 Instance Setup

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

To set up a public EC2 instance, choose an appropriate Amazon Machine Image (AMI) and instance type. Ensure the security group allows inbound traffic on necessary ports (e.g., HTTP/HTTPS for web applications). After launching the instance, associate it with an elastic IP for consistent public access, and configure your application or services to be publicly accessible.

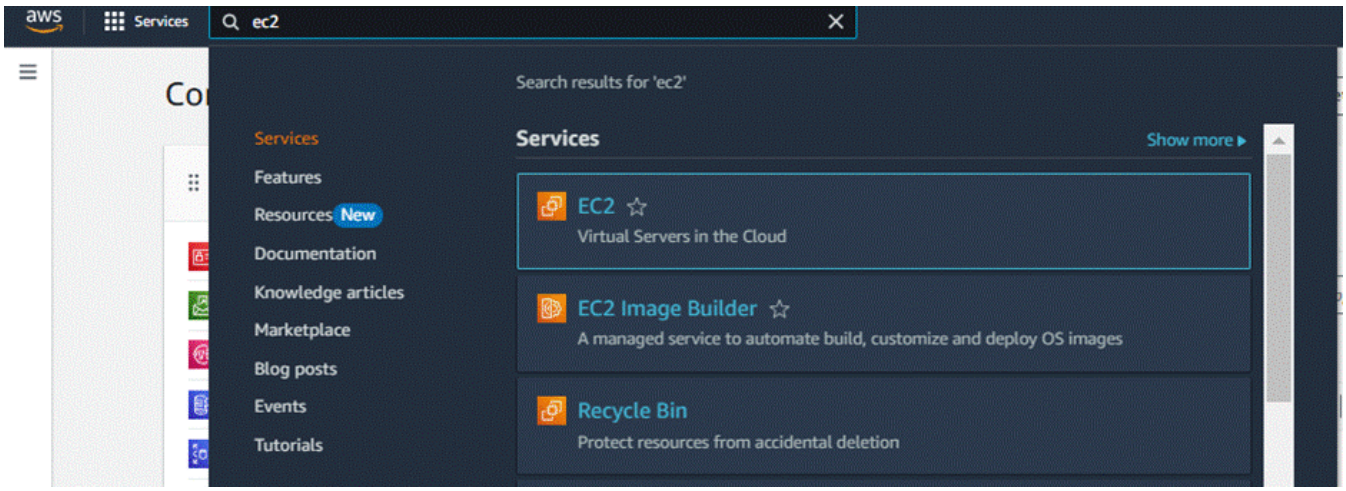


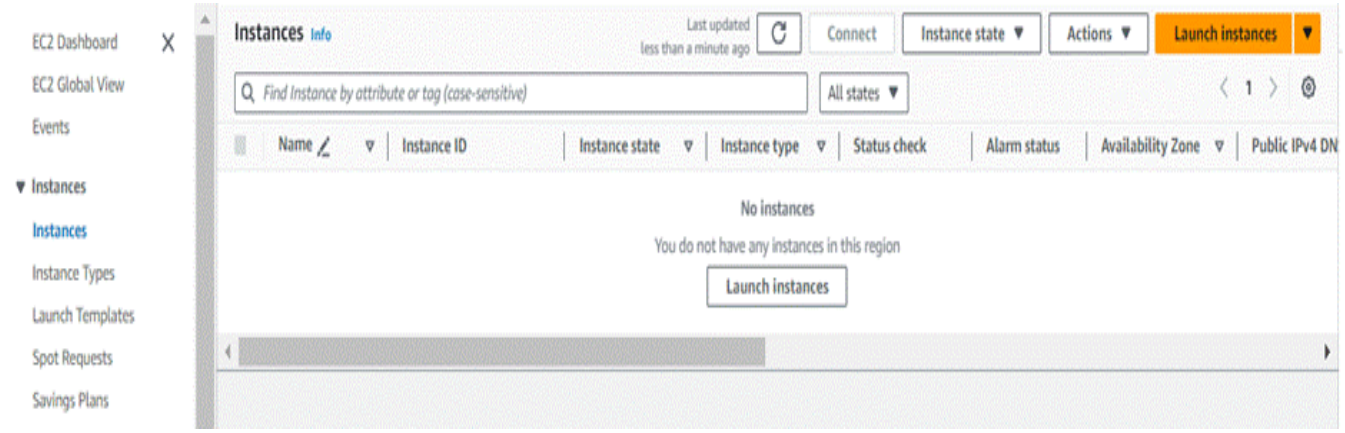


Load your Project Files to Github:

Launch EC2 Instance

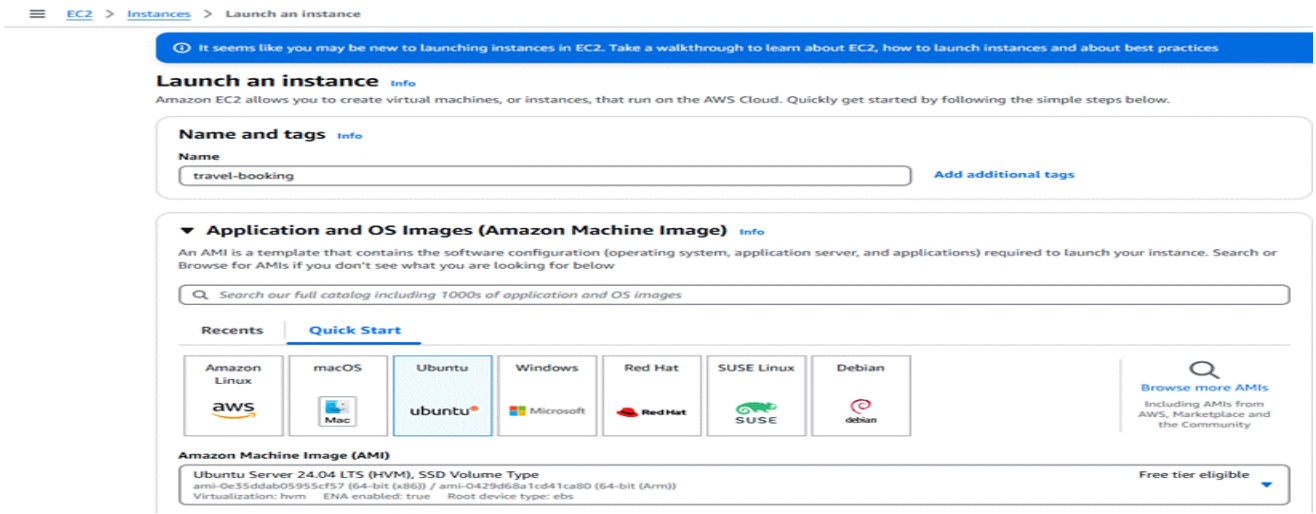
In the AWS Console, navigate to EC2 and launch a new instance.



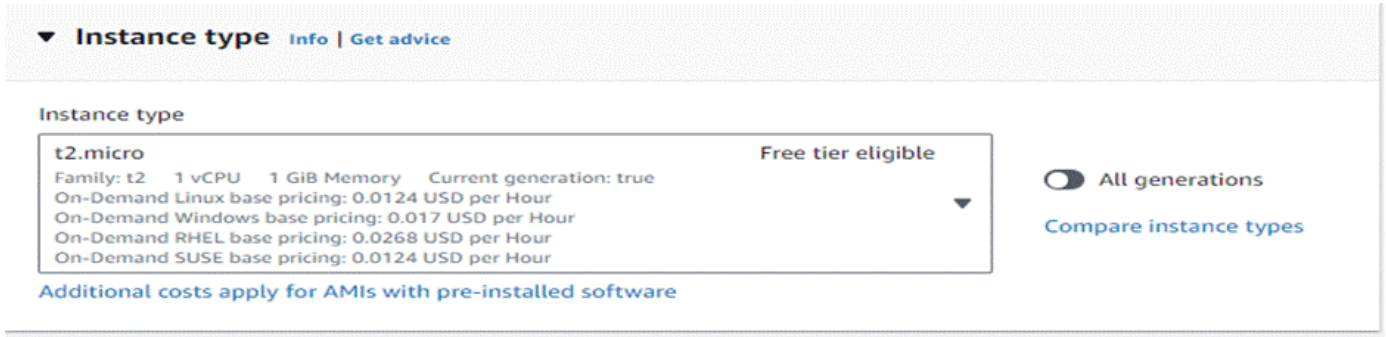


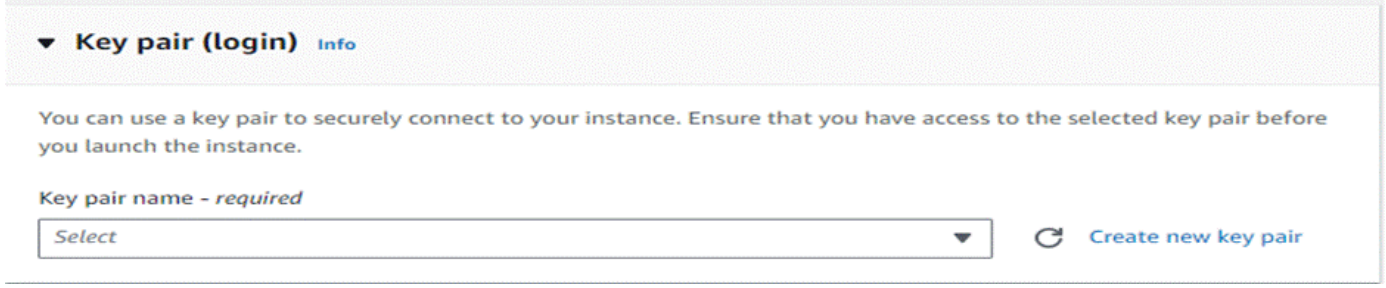
Click on Launch instance to launch EC2 instance

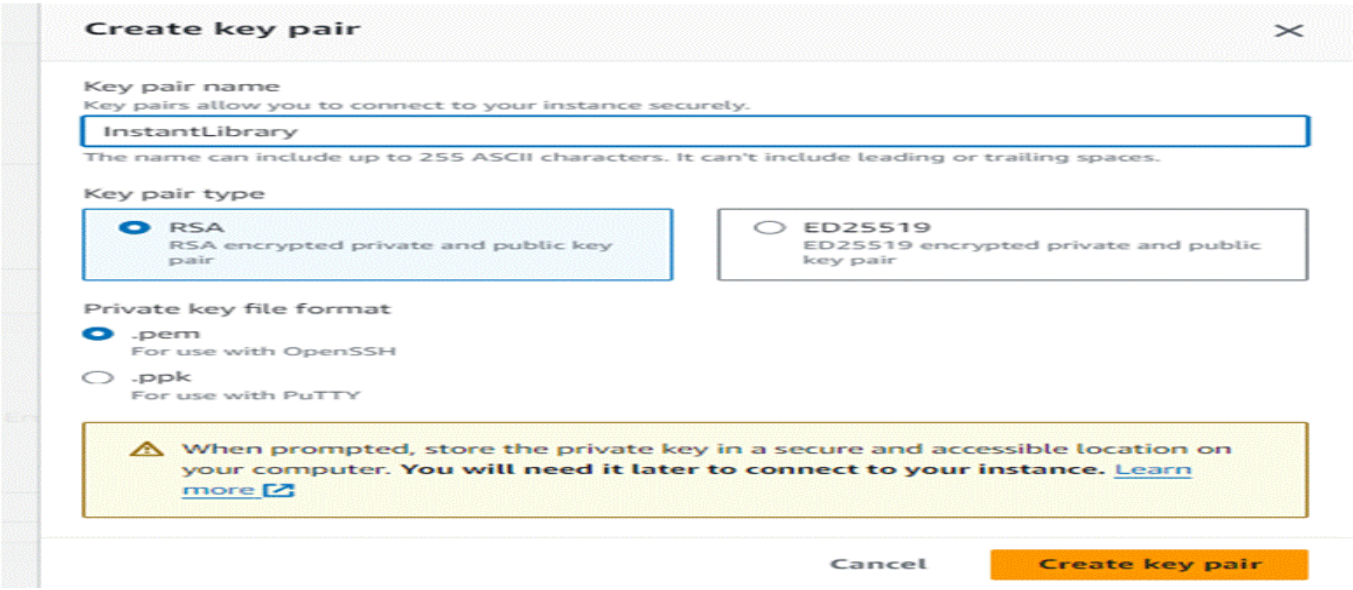
- Choose Amazon Linux 2 or Ubuntu as the AMI and t2.micro as the instance type (free-tier eligible).

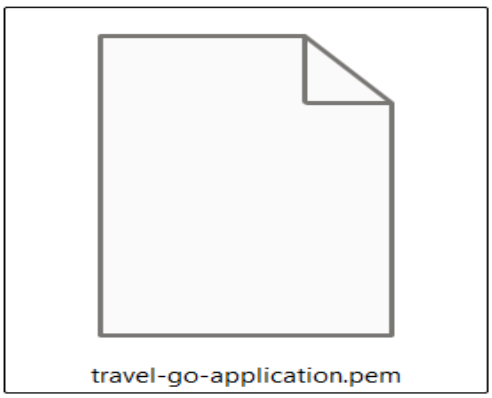


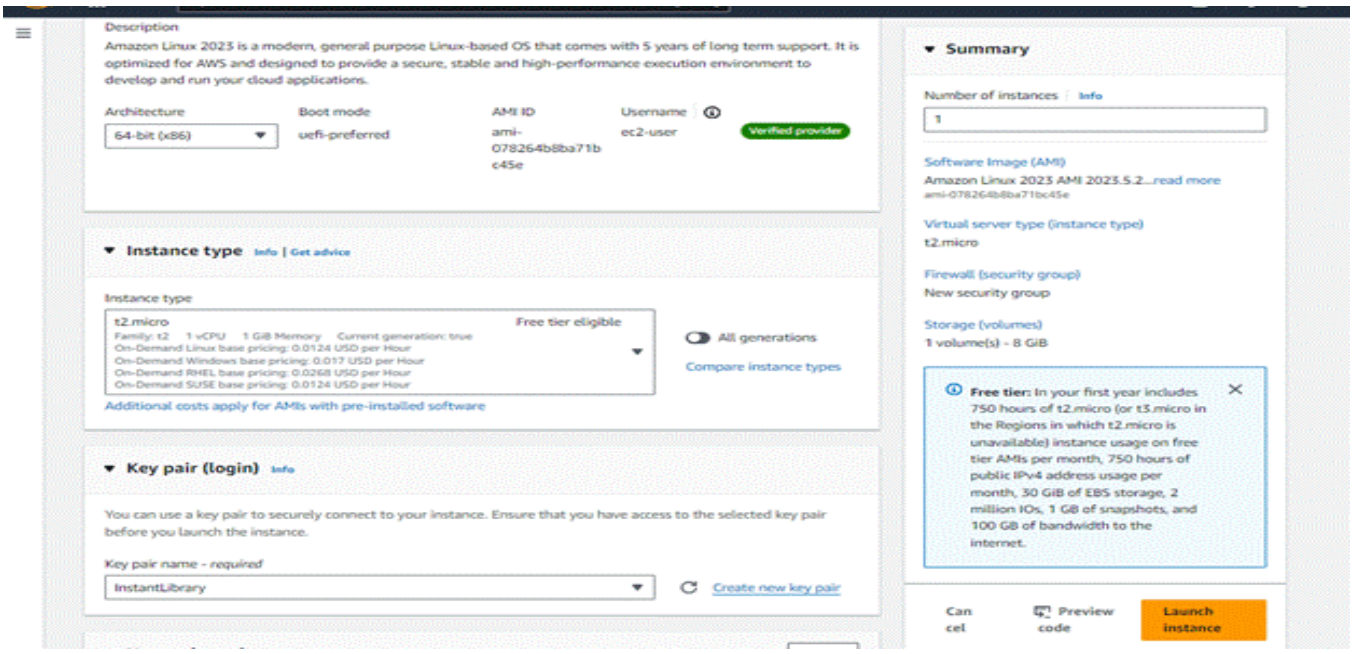
- Create and download the key pair for Server access.







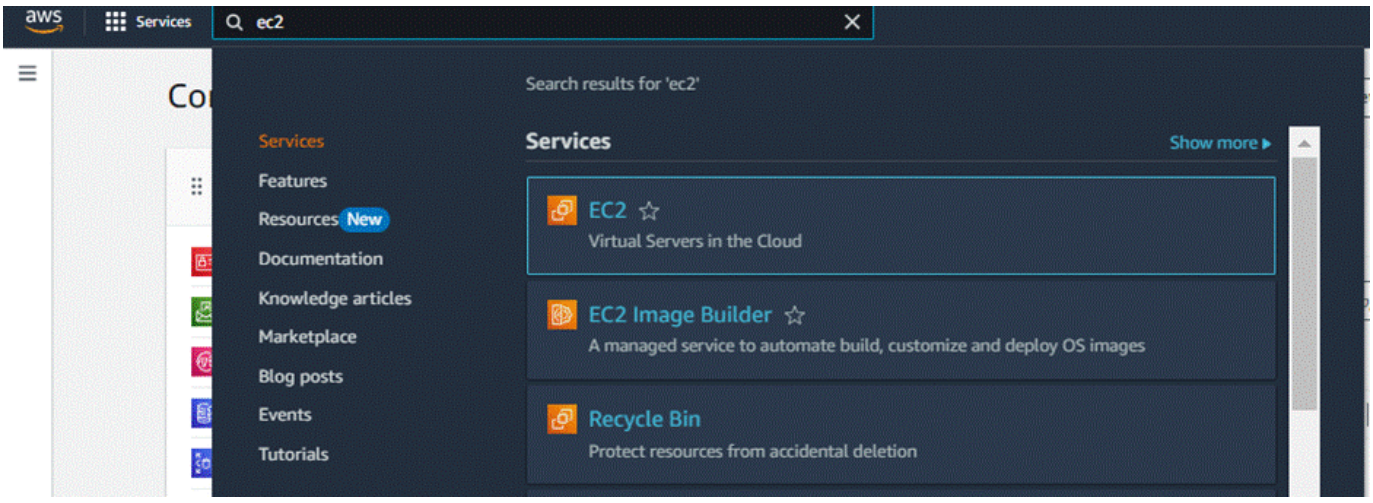




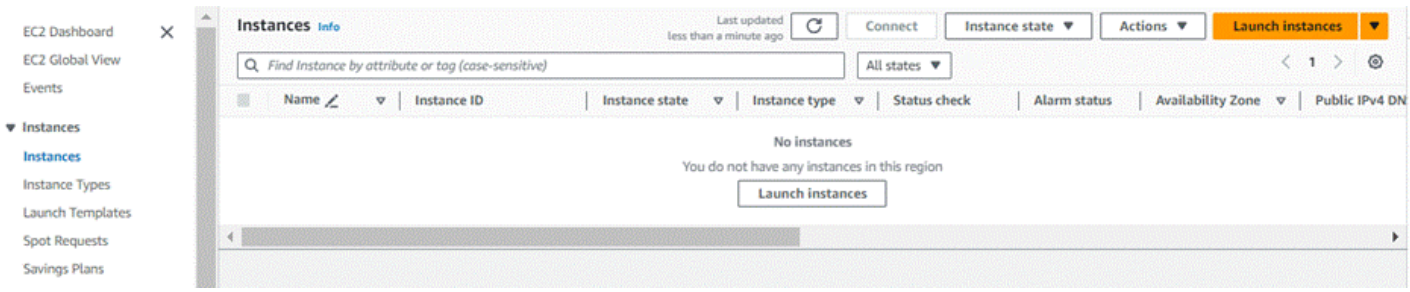
Launch An EC2 instance to host The flask

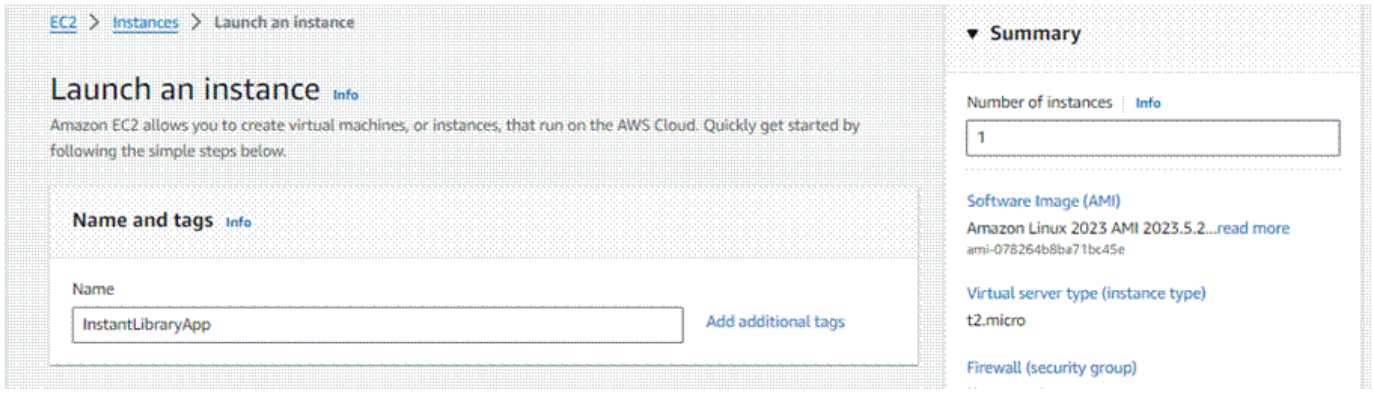
Launch EC2 Instance

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

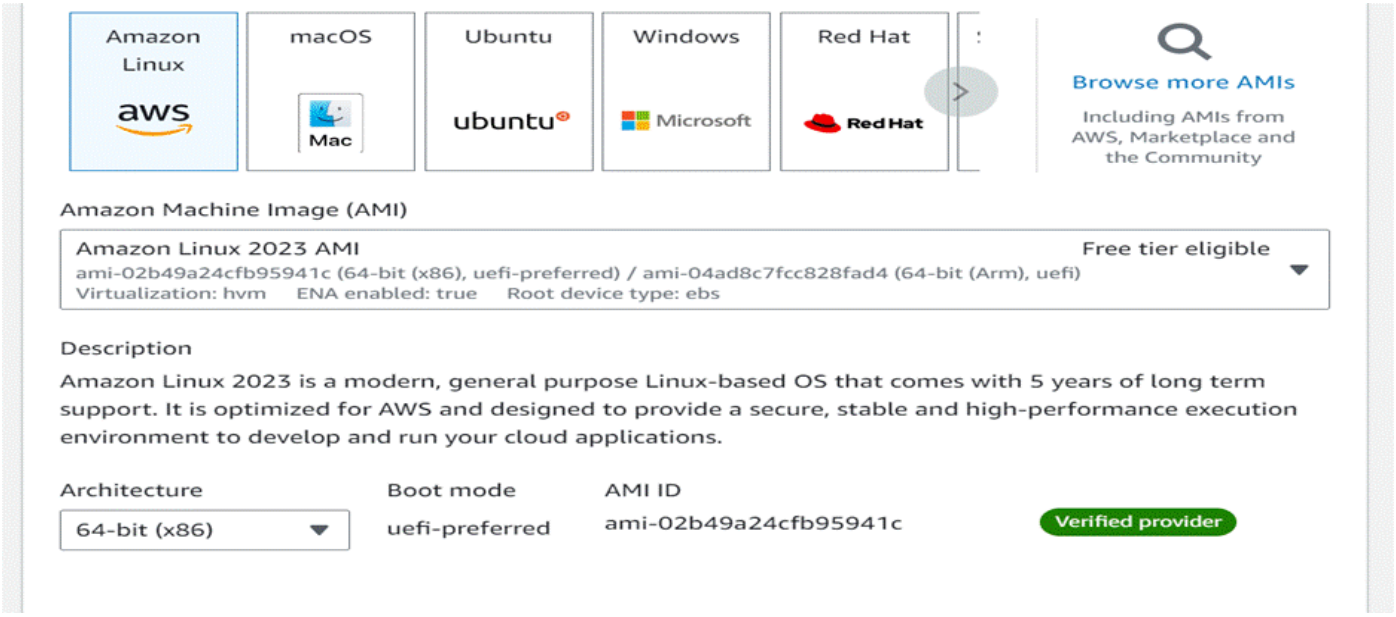


- Click on Launch instance to launch EC2 instance

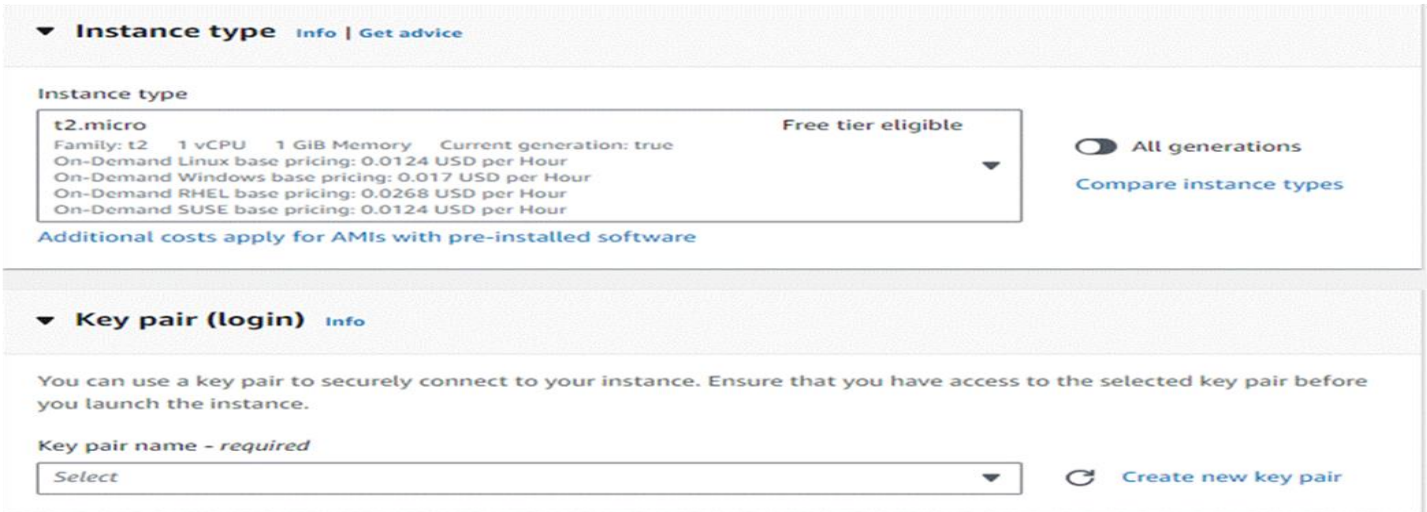




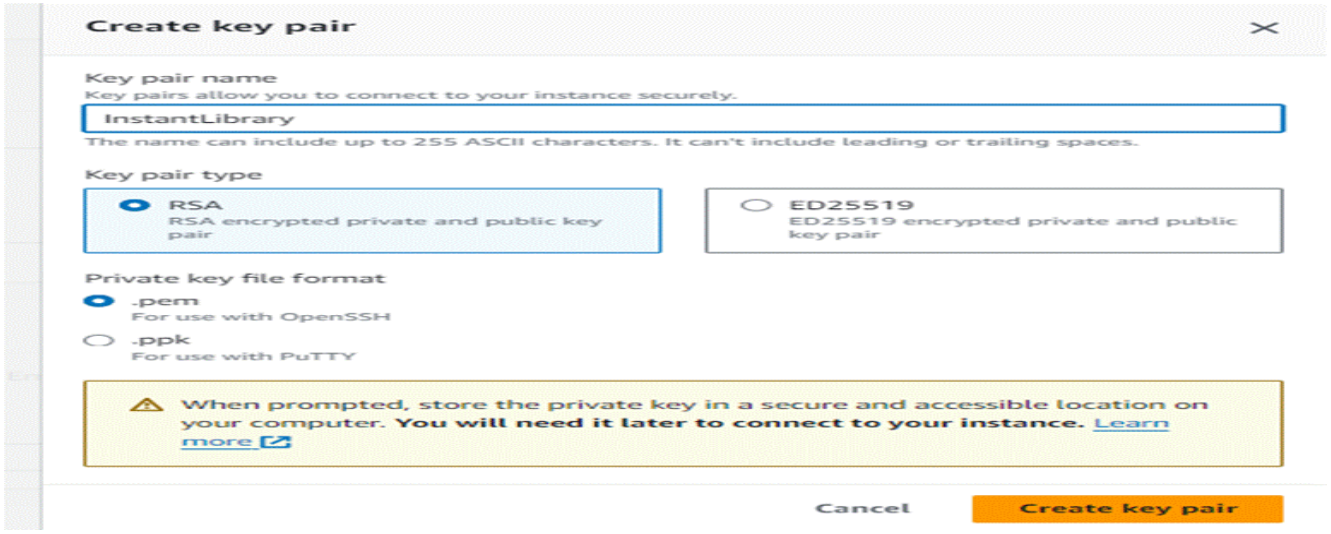
- Choose Amazon Linux 2 or Ubuntu as the AMI and t2.micro as the instance type (free-tier eligible).

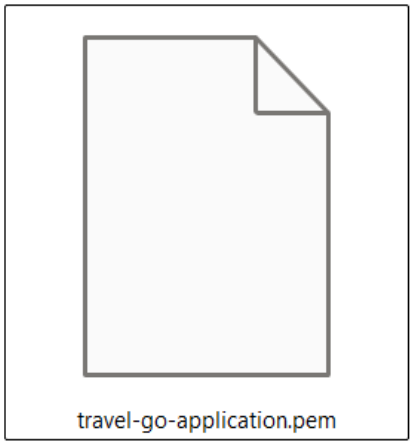


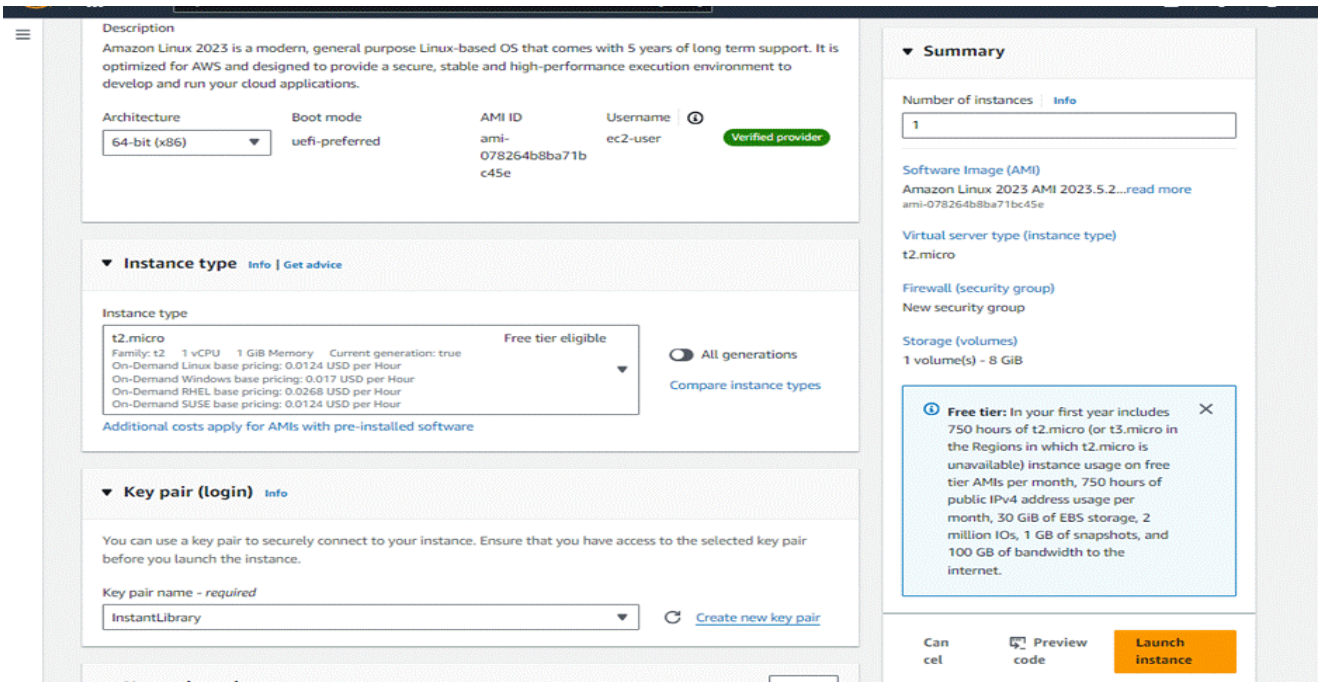
- Create and download the key pair for Server access.



SMARTBRIDGE

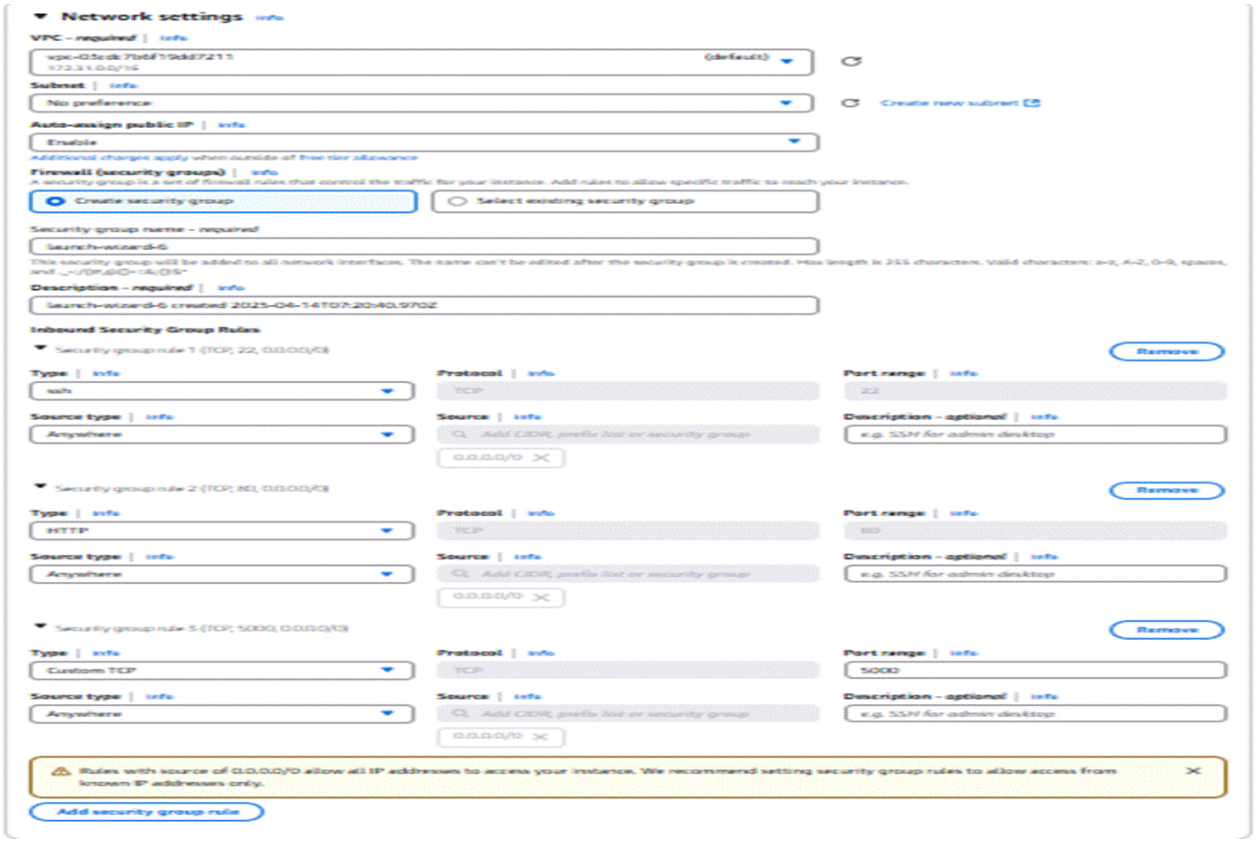


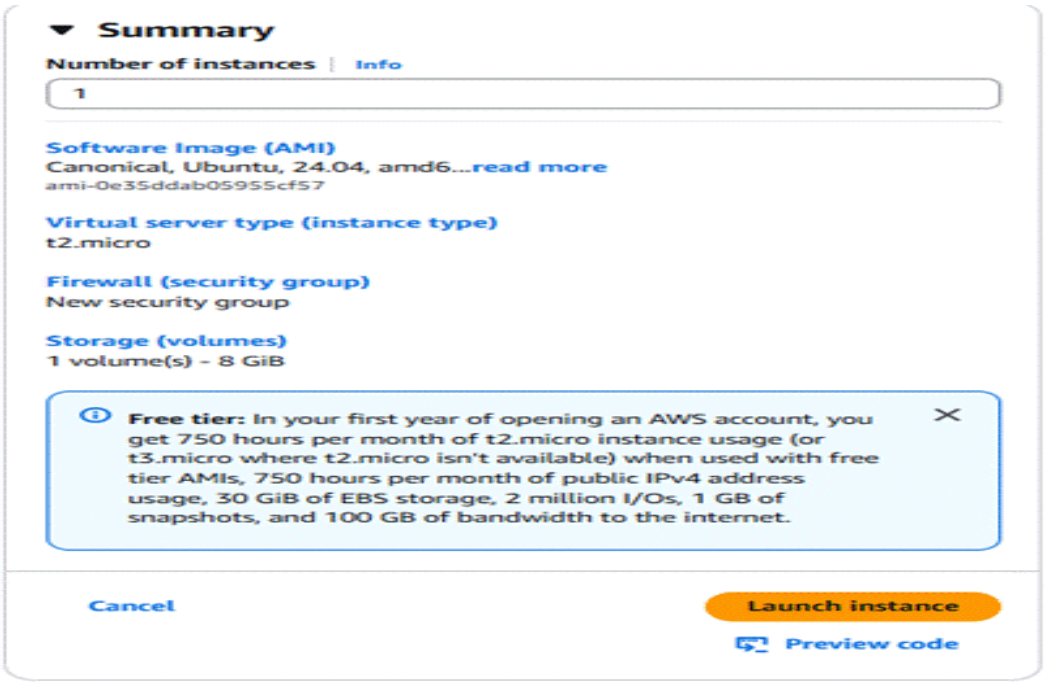


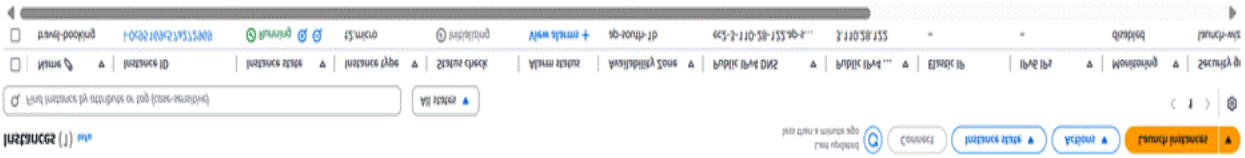


SMARTBRIDGE

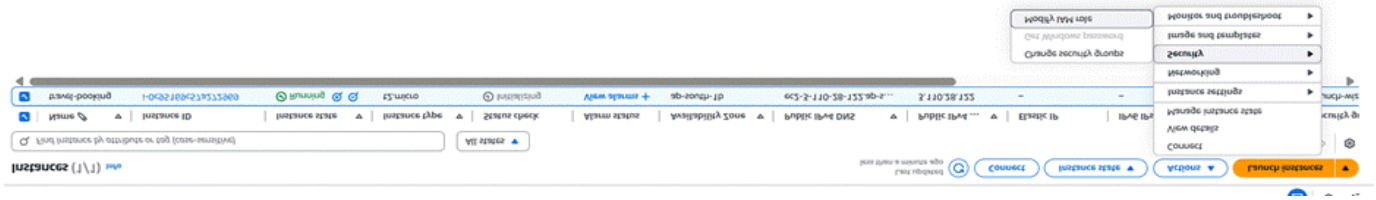
Configure security groups for HTTP, and SSH access:

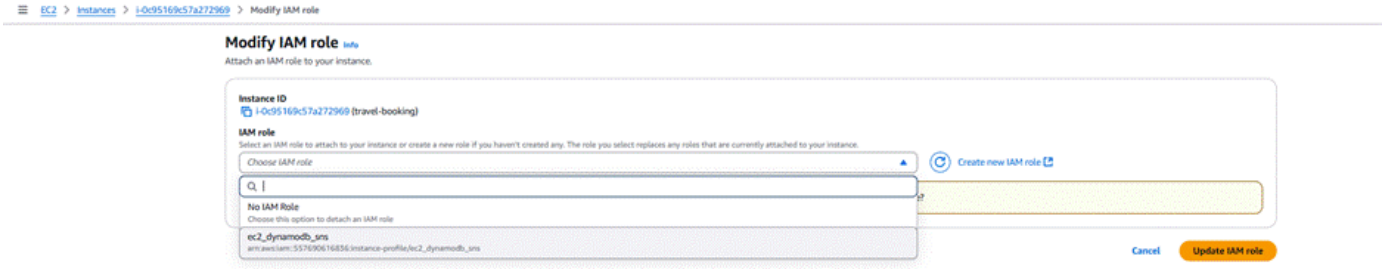




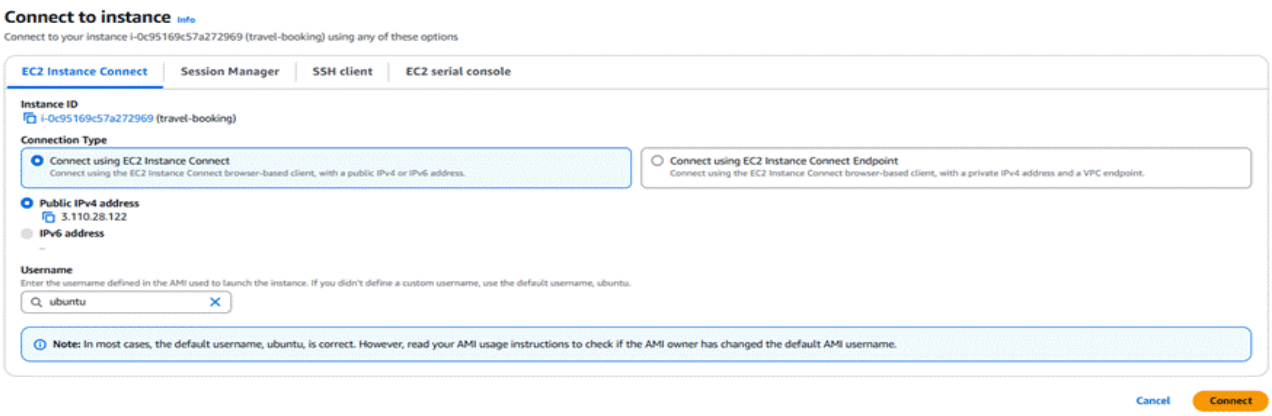


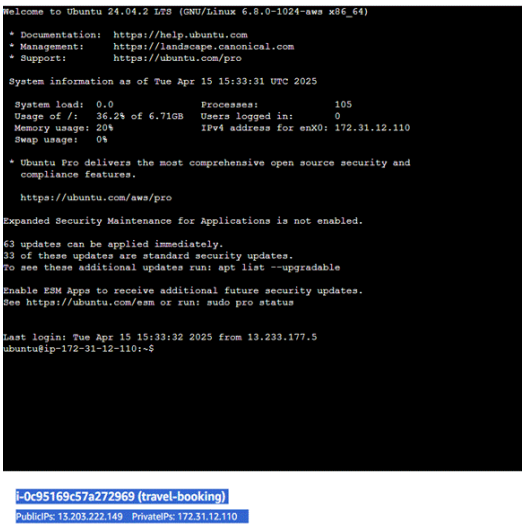
To connect to EC2 using EC2 Instance Connect, start by ensuring that an IAM role is attached to your EC2 instance. You can do this by selecting your instance, clicking on Actions, then navigating to Security and selecting Modify IAM Role to attach the appropriate role. After the IAM role is connected,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

Deployment on an EC2 instance involves launching a server, configuring security groups for public access, and uploading your application files. After setting up necessary dependencies and environment variables, start your application and ensure it's running on the correct port. Finally, bind your domain or use the public IP to make the application accessible online.

Install Software on the EC2 Instance

Install Python3, Flask, and Git:

- **On Amazon Linux 2:**

  sudo yum update -y

  sudo yum install python3 git

  sudo pip3 install flask boto3

- **Verify Installations:**

  flask --version

  git –version

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: ‘https://github.com/AlekhyaPenubakula/Travel-Booking-website_Ec2_Dynamodb_SNS.git’

This will download your project to the EC2 instance.

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

cd <your-folder>

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

Run the Flask Application

- export SNS_TOPIC_ARN=arn:aws:sns:ap-south-1:557690616836:BookingConfirmation
- git pull origin main (Only if you made changes in git and want to reflect them in EC2 terminal.)

 # Install requirements

pip install flask boto3

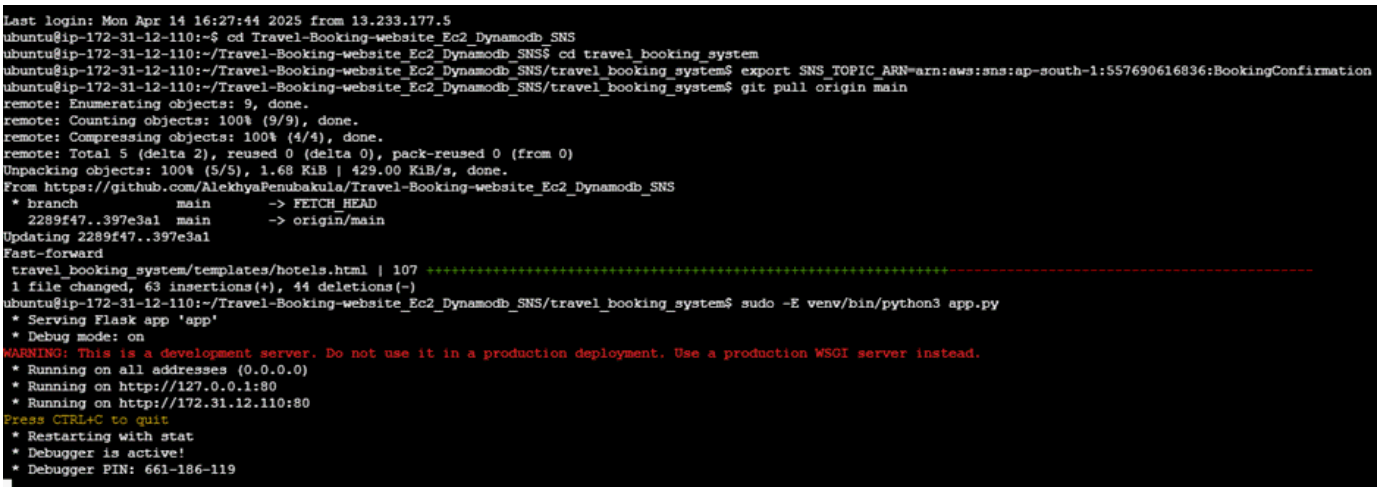
- **Launch the Flask app:**

sudo -E venv/bin/python3 app.py

Verify the Flask app is running:

http://your-ec2-public-ip

- Run the Flask app on the EC2 instance



Access the website through:

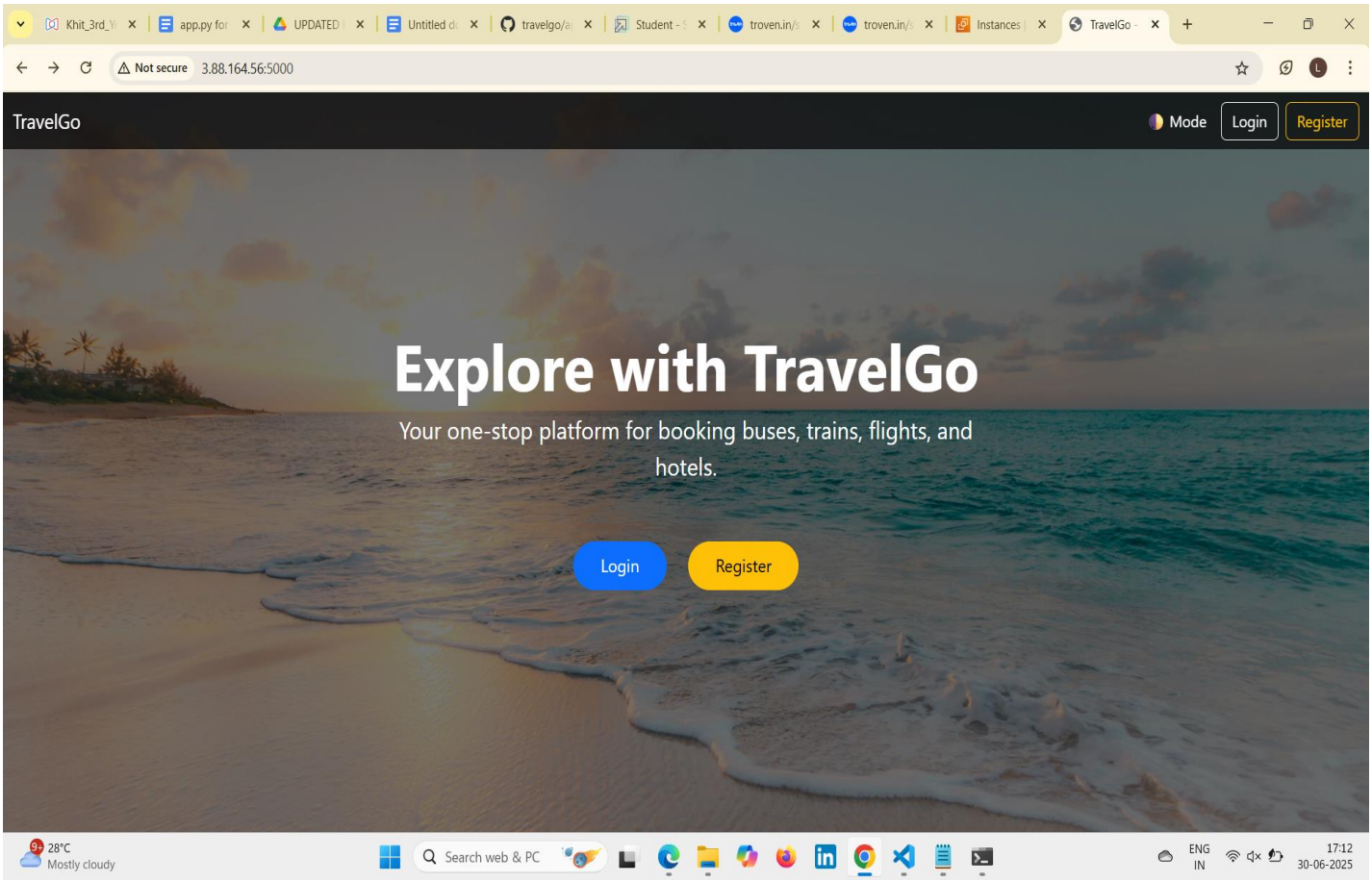
Public IPs: http://3.88.164.56:5000/

Milestone 8 : Testing and Deployment

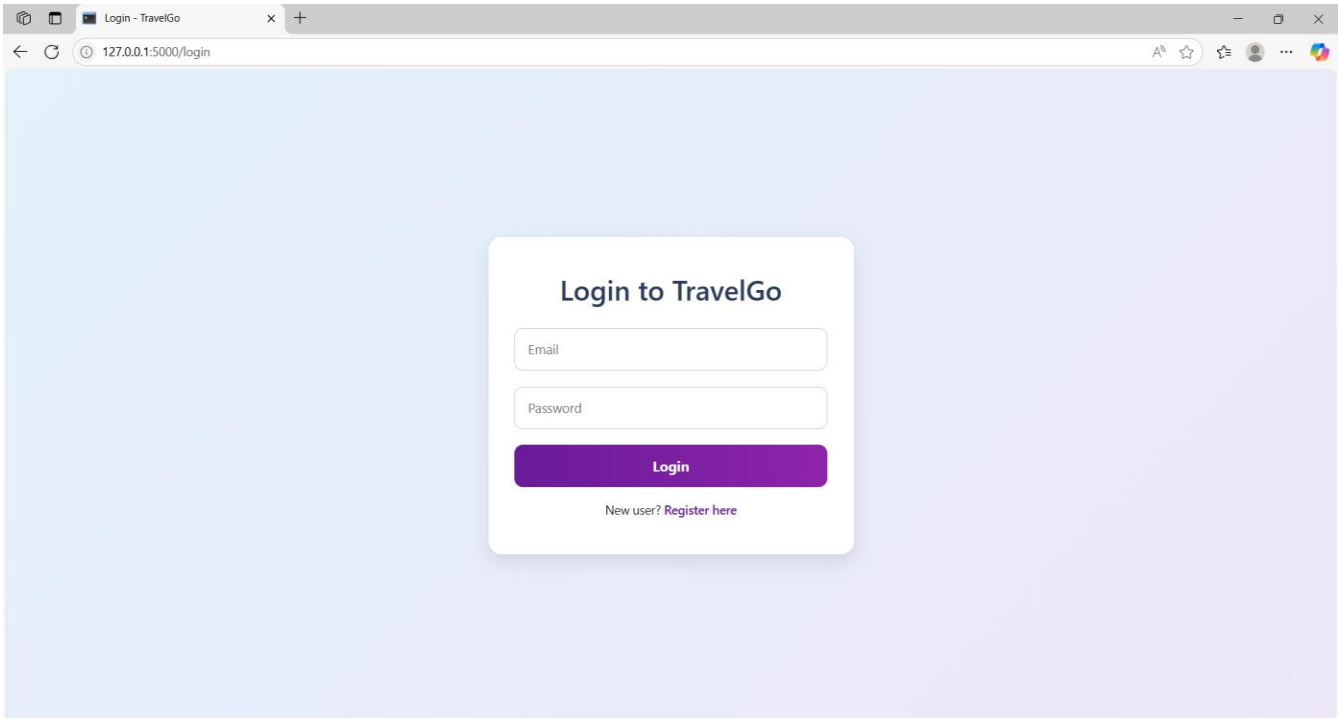
Testing and deployment involve verifying that your application works as expected before making it publicly accessible. Start by testing locally or on a staging environment to catch bugs and ensure functionality. Once tested, deploy the application to an EC2 instance, configure necessary services, and perform a final round of live testing to confirm everything runs smoothly in the production environment.

Functional Testing to verify the Project:

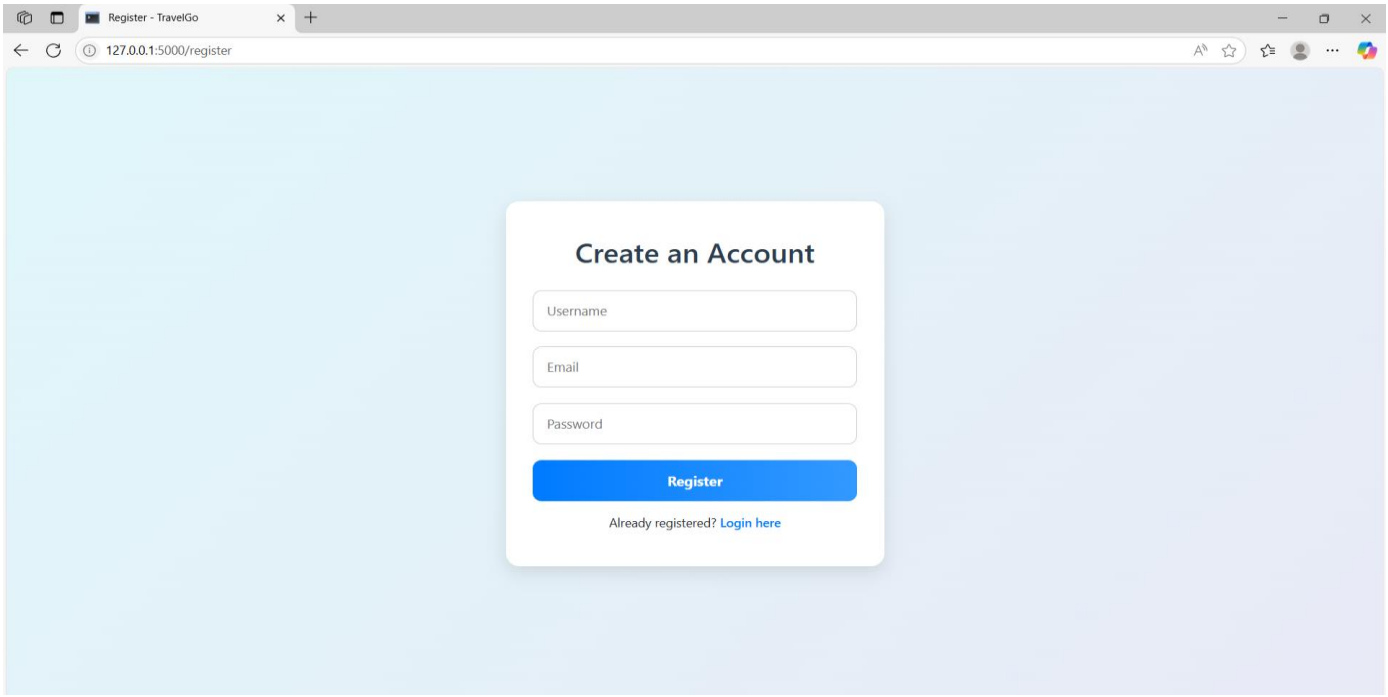
Home Page:



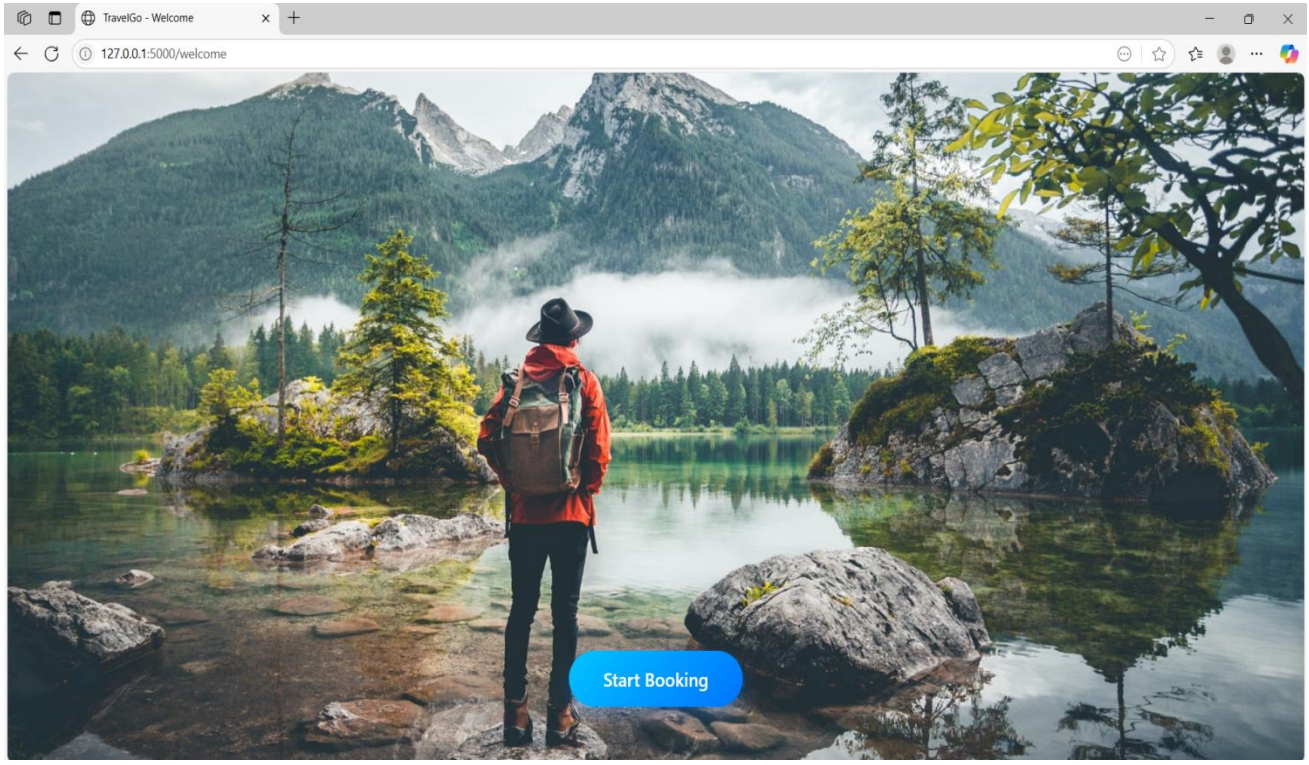
Login Page:



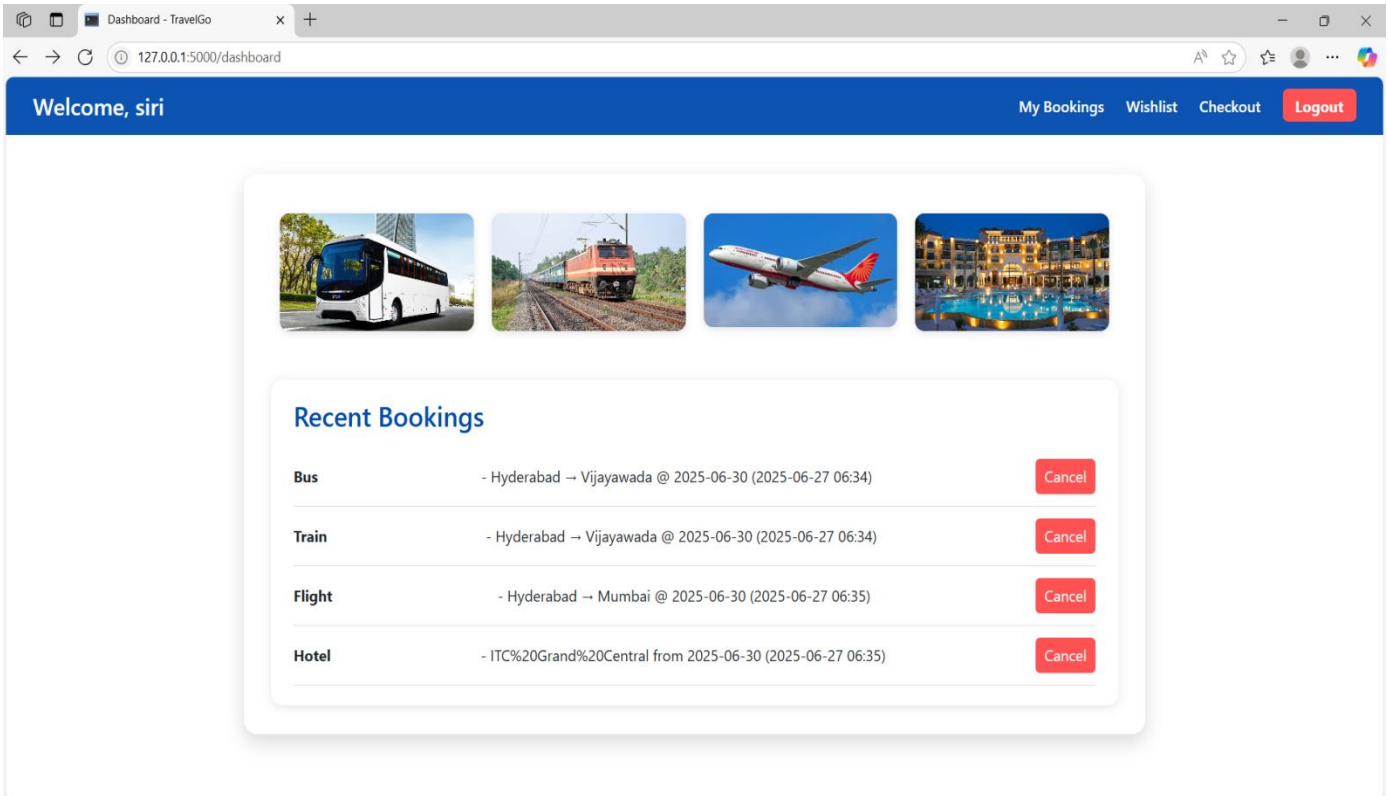
Register Page:



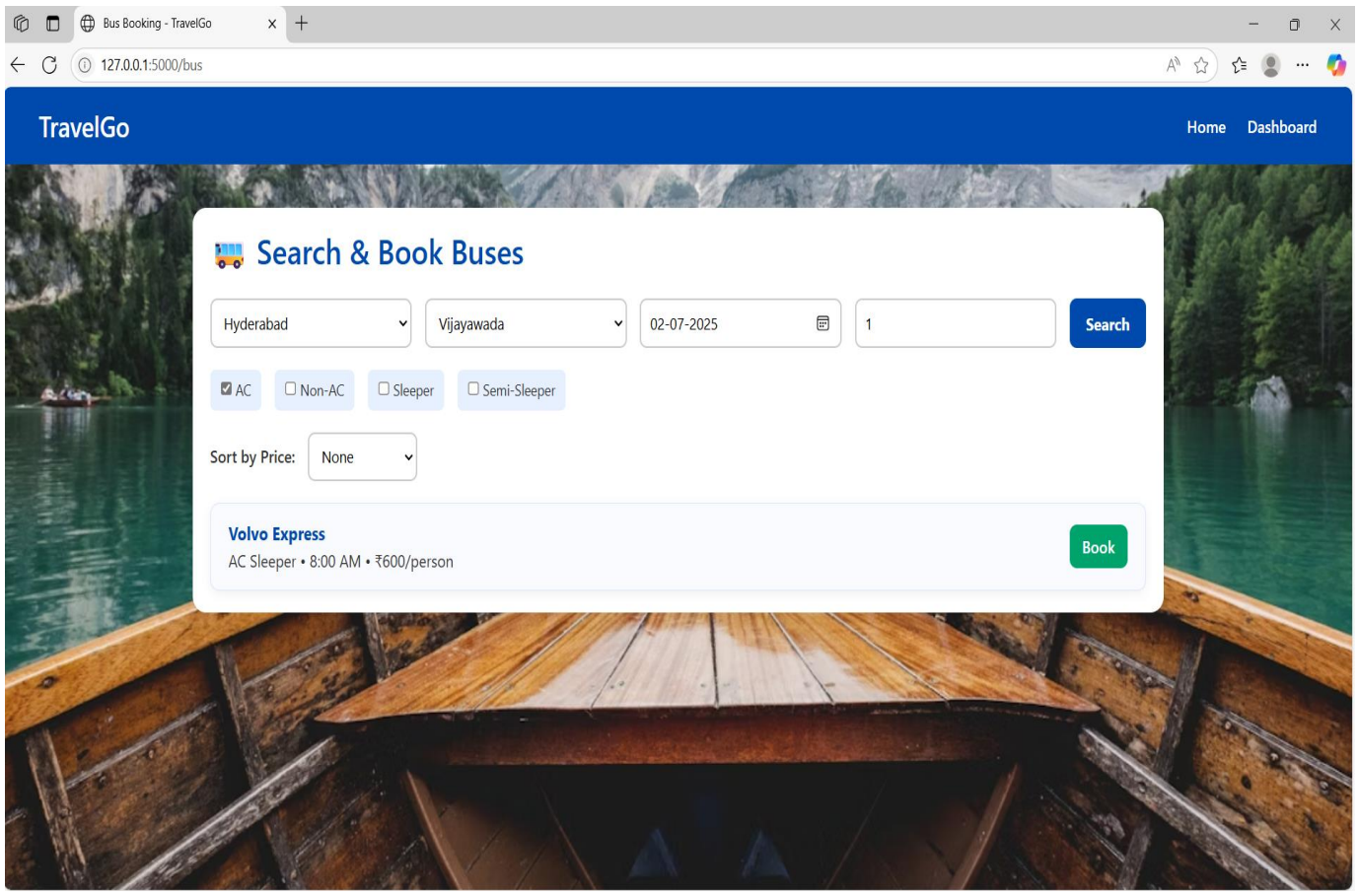
Welcome Page:



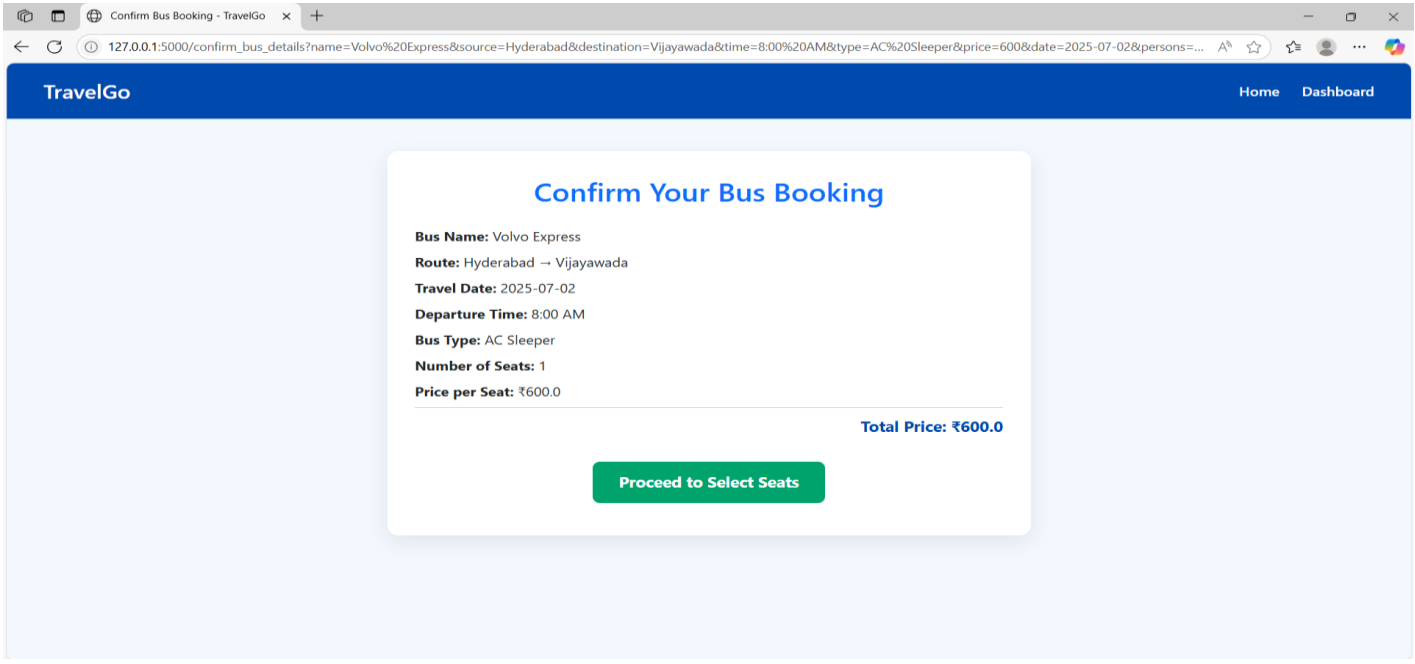
Dashboard Page:



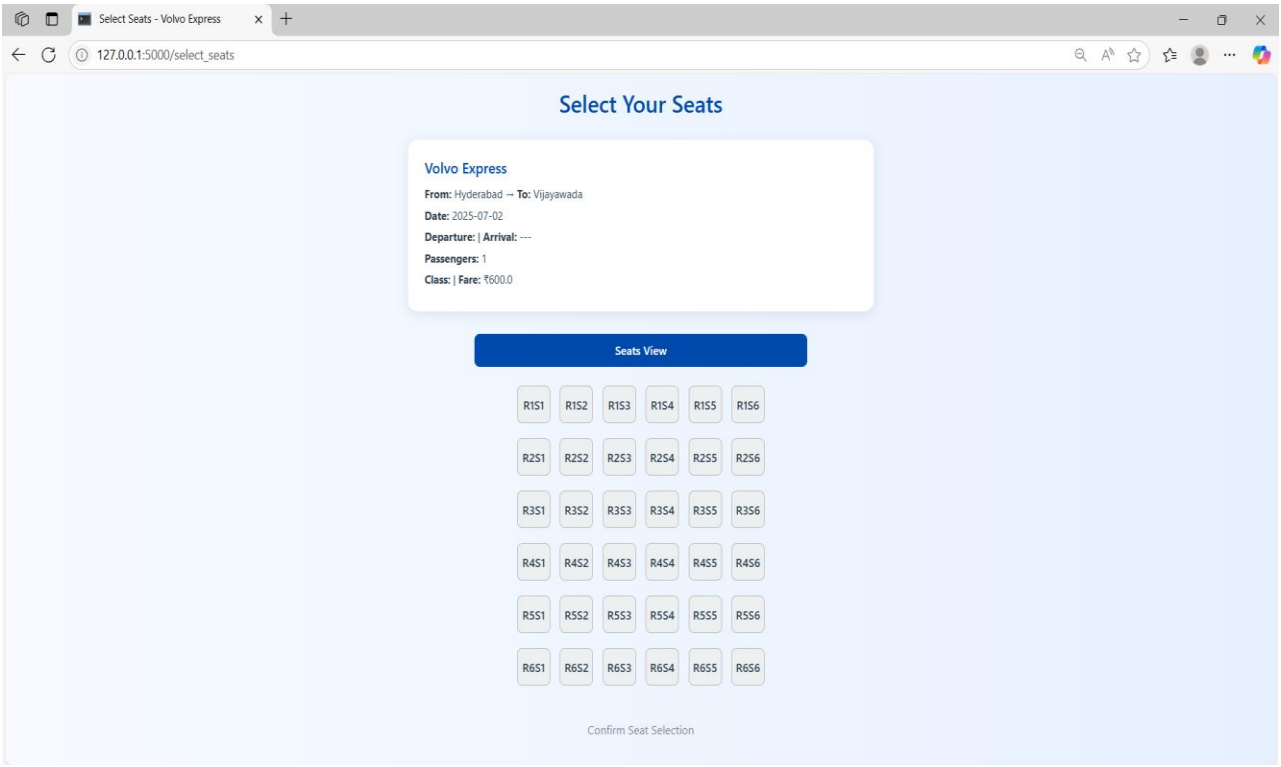
Bus Booking Page:



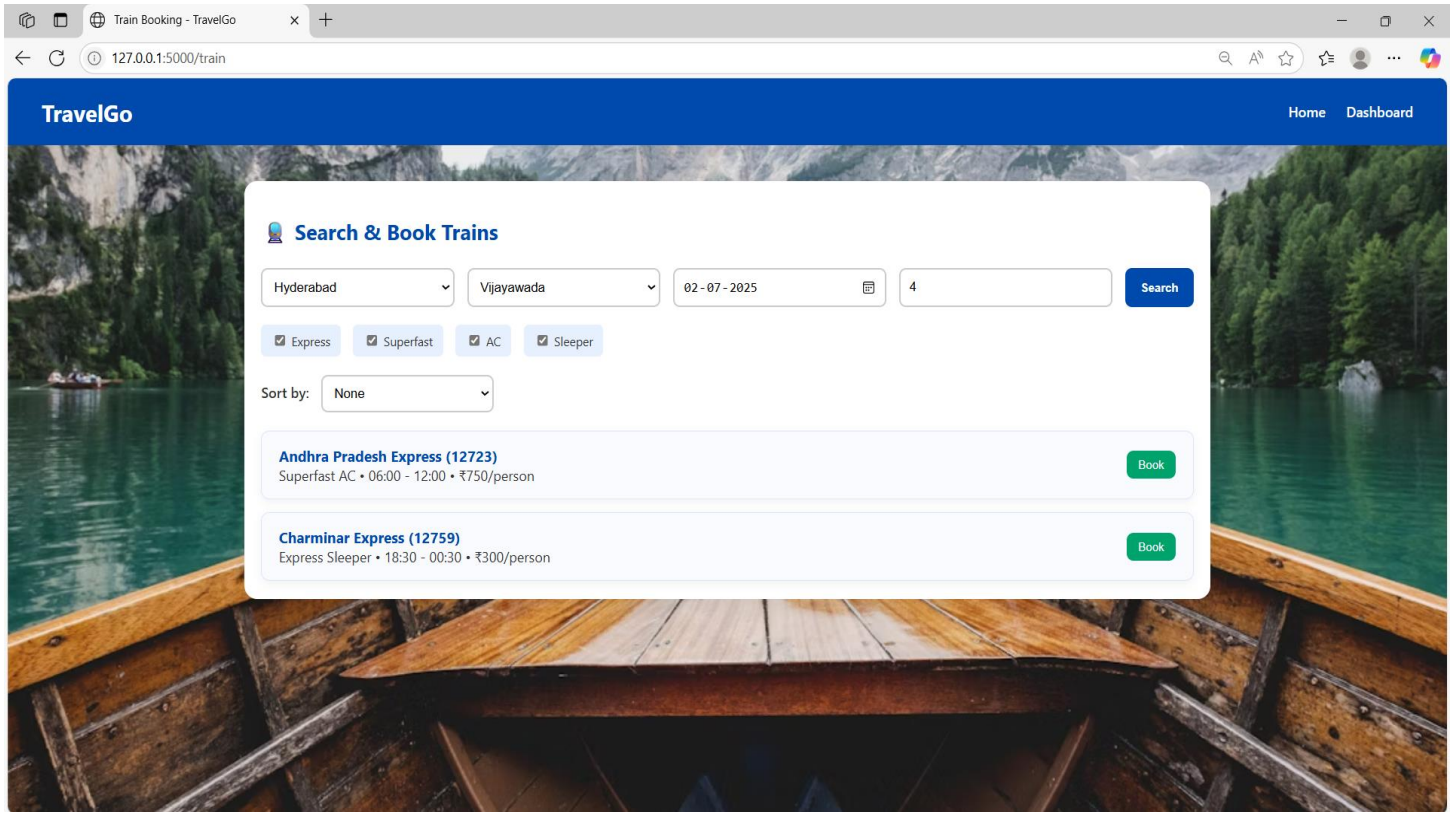
Confirm Bus booking:



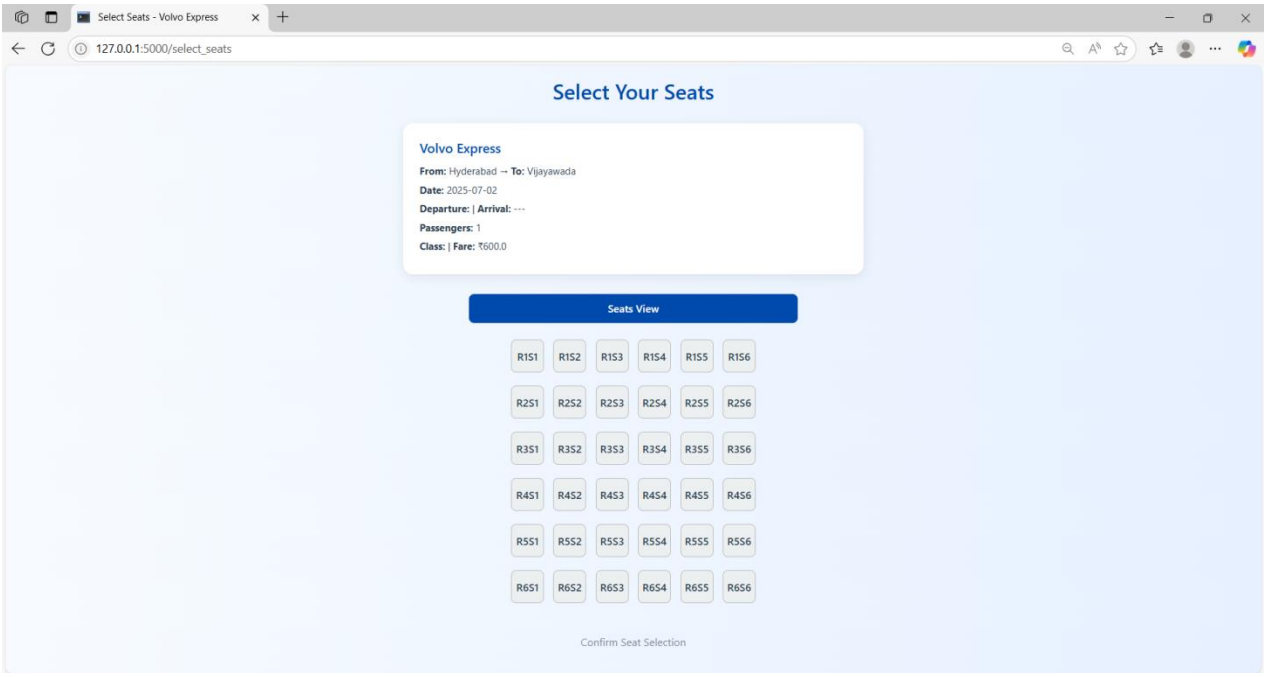
Selecting Seats:



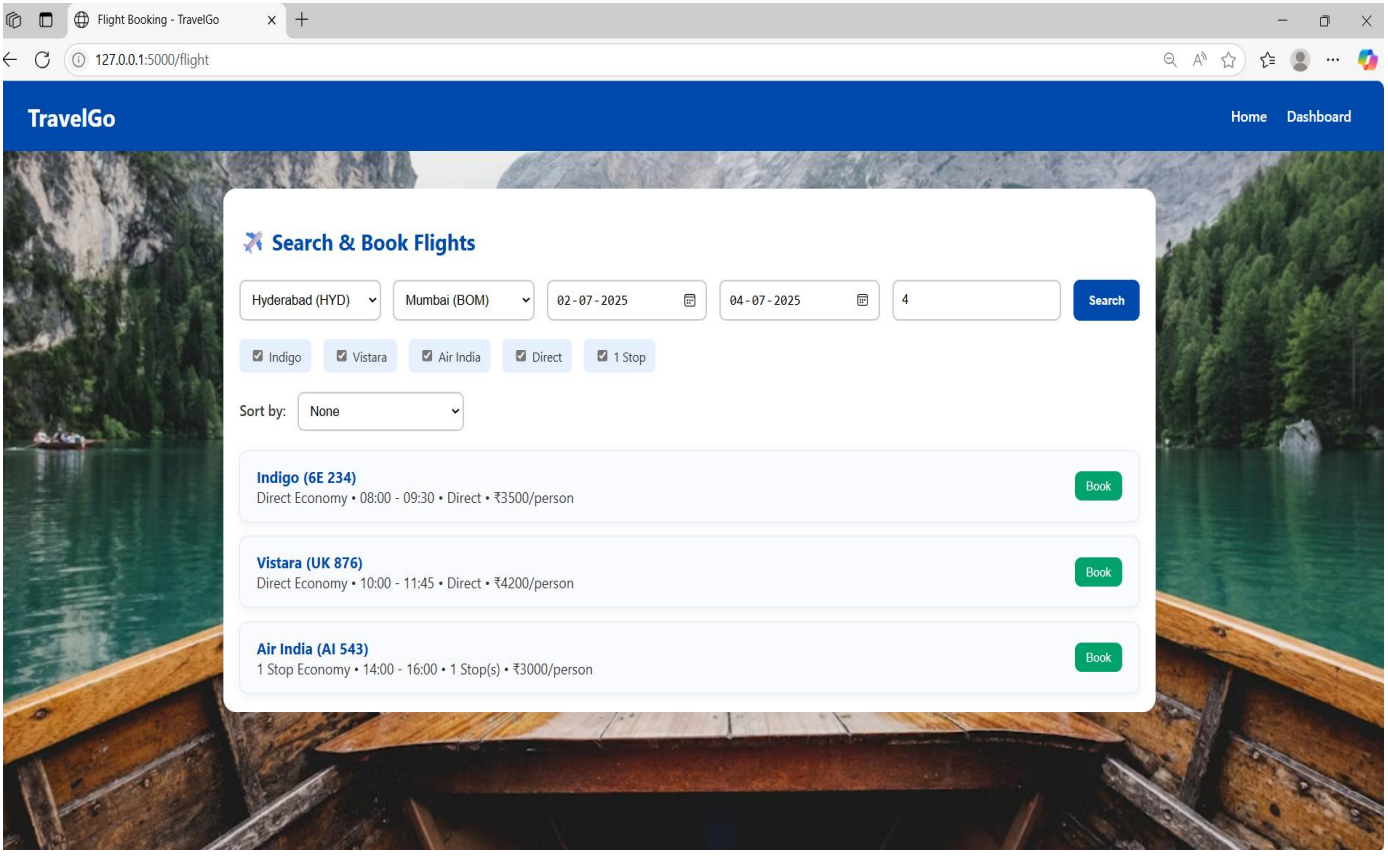
Train Booking Page:



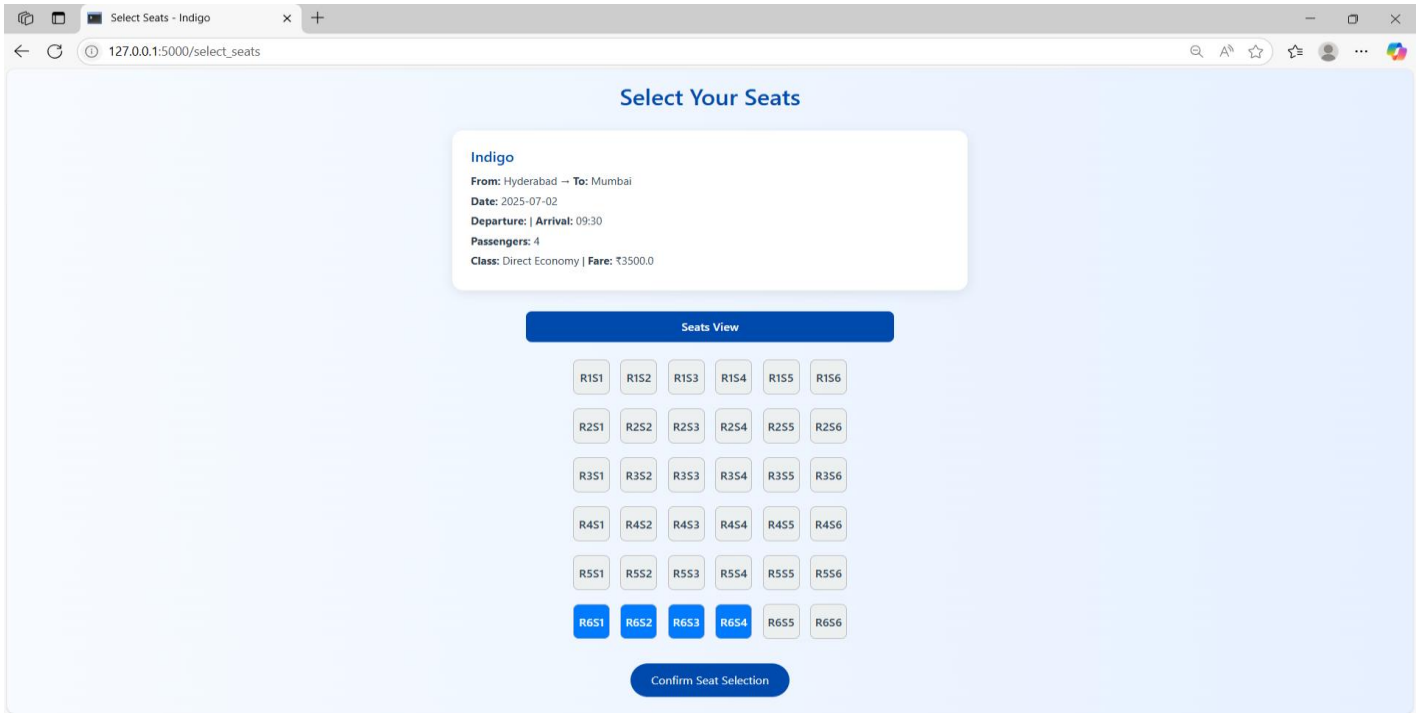
Selecting Seats:



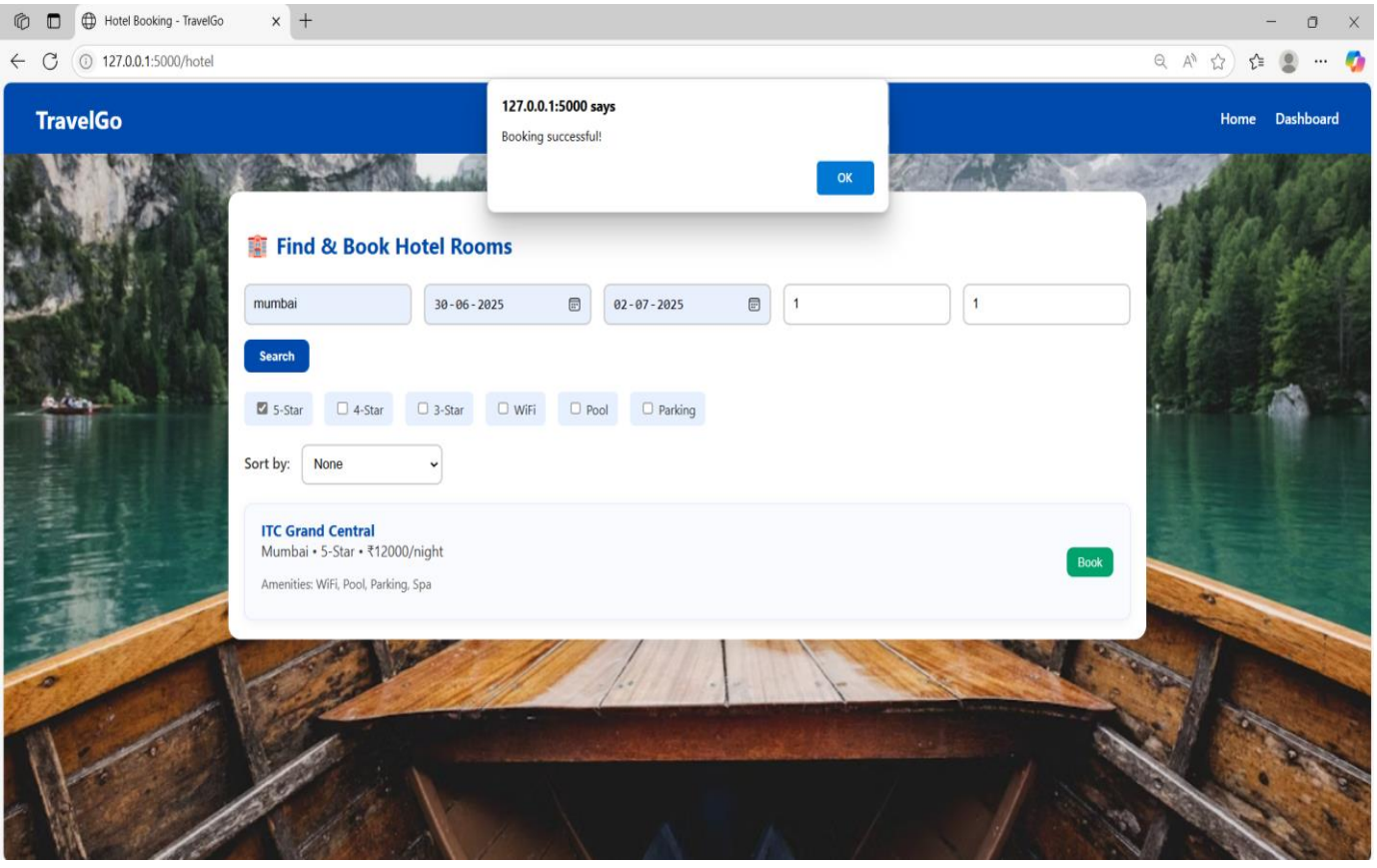
Flight Booking Page



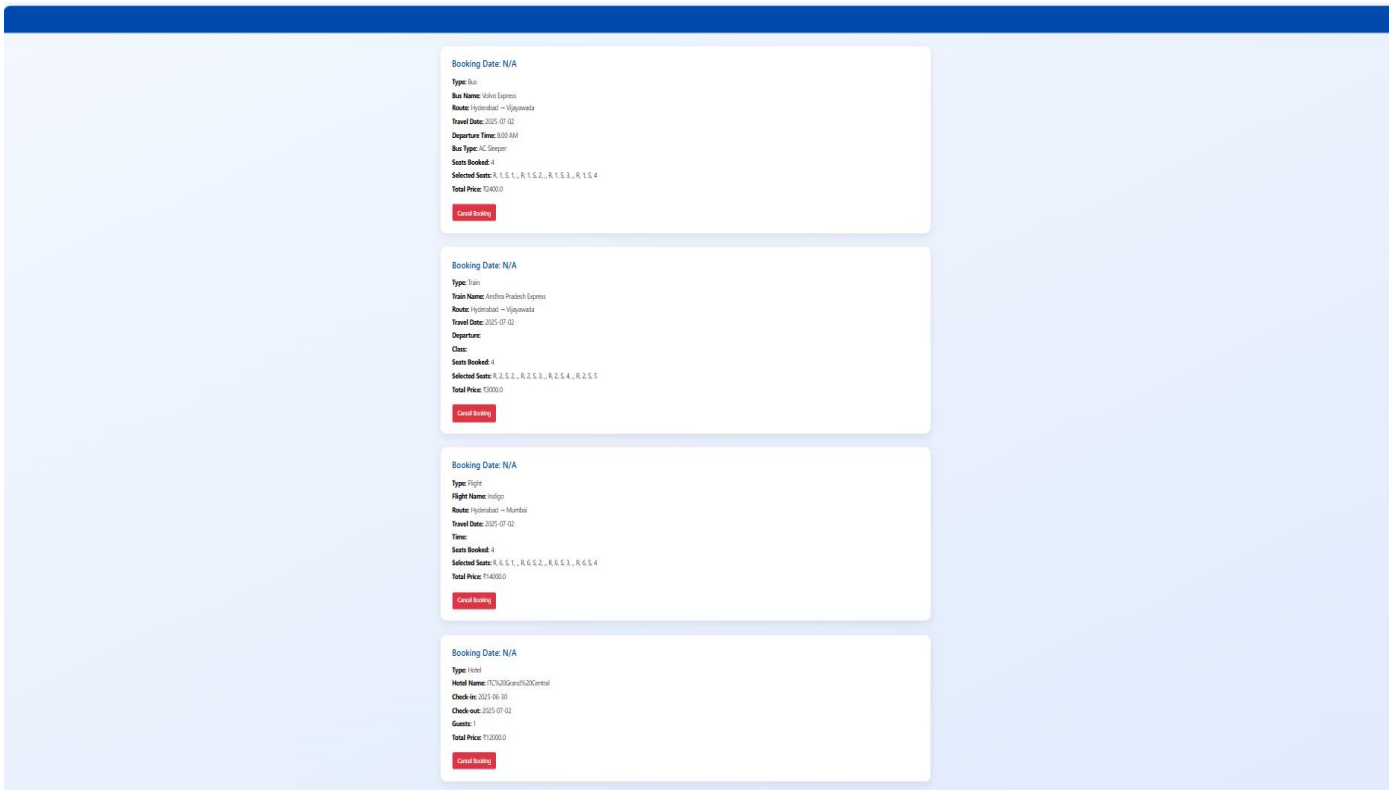
Selecting Seats:



Hotel Booking page:



My Bookings:



Conclusion for TravelGo Cloud-Based Booking Platform Project:

The TravelGo Cloud-Based Booking Platform has been successfully developed and deployed using a scalable architecture built on AWS services. Leveraging EC2 for application hosting, DynamoDB for efficient data management, and SNS for real-time booking notifications, the platform delivers a seamless and reliable experience for users booking bus, train, flight, and hotel services.

This cloud-native solution addresses the common challenges of fragmented travel platforms by offering a unified interface for users to search, book, and manage their travel itineraries. With secure user authentication, dynamic seat selection, and centralized booking history, TravelGo ensures high usability and responsiveness. The system has been thoroughly tested for functionality, ensuring smooth operations from login to booking confirmation.

In conclusion, TravelGo demonstrates the power of cloud-based applications in streamlining multi-modal travel management, enhancing user convenience, and ensuring robust performance under growing demand—making it a scalable solution for the evolving travel industry.