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.

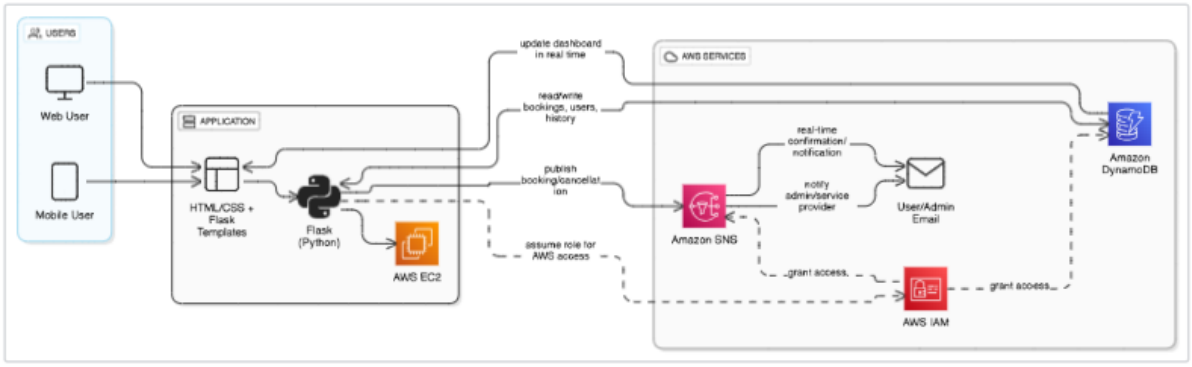
Scenario 3: Dynamic Dashboard with Personal Travel History

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.

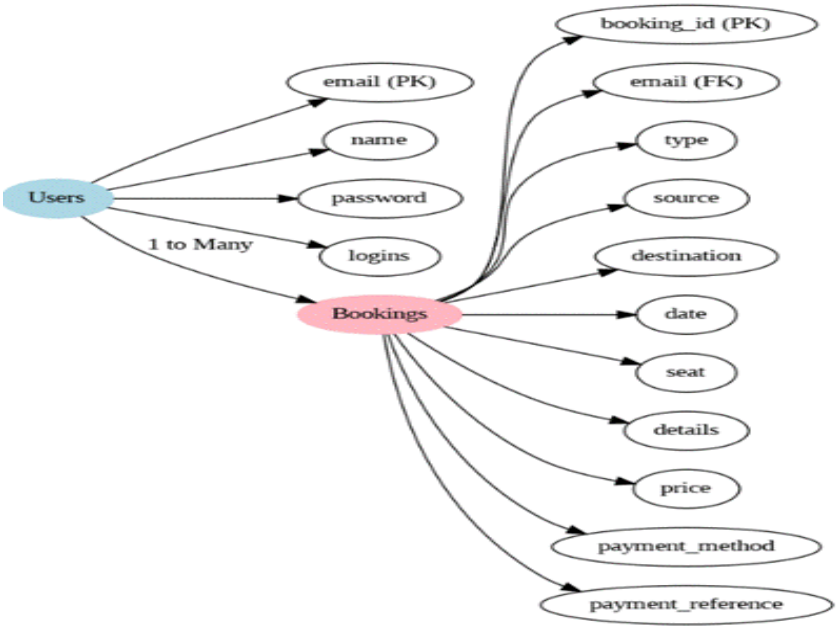
ARCHITECTURE

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. Real-time alerts and system notifications are enabled via Amazon SNS, enhancing communication and user engagement.



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 (Simple Notification Service)**
  https://docs.aws.amazon.com/sns/latest/dg/welcome.html
- **Git Documentation**
  https://git-scm.com/doc
- **Visual Studio Code Download**
  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

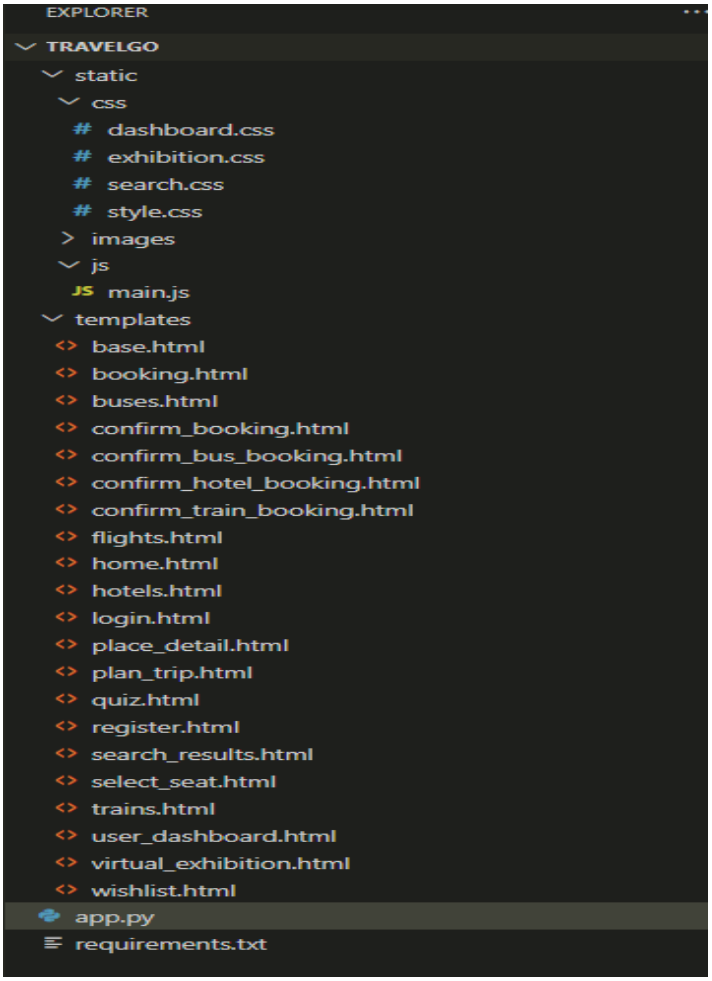
Backend Development and Application Setup focuses on establishing the core structure of the application. This includes configuring the backend framework, setting up routing, and integrating database connectivity. It lays the groundwork for handling user interactions, data management, and secure access.

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.

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:

```
app.py > ...
1  from flask import Flask, render_template, request, redirect, session, url_for, flash, jsonify, abort
2  from pymongo import MongoClient
3  from werkzeug.security import generate_password_hash, check_password_hash
4  from datetime import datetime
5  from functools import wraps
6  from flask_caching import Cache
7  from bson.objectid import ObjectId  # ✅ Needed for _id lookups
8  import os
9
```

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

Flask App Initialization:

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

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

Database Configuration:

```
# Initialize DynamoDB resource
dynamodb = boto3.resource('dynamodb', region_name='ap-south-1')


# DynamoDB Tables
users_table = dynamodb.Table('Users')  # Ensure the 'Users' table
requests_table = dynamodb.Table('Requests')  # Ensure the 'Reques
```

Description: Connect to DynamoDB using Boto3 and define references to the UserTable and WishlistTable for user and wishlist data operations.

Home and registration routes:

```
# Home route redirects to Registration page
@app.route('/')
def home():
    return redirect(url_for('register'))
```

Description: define the home route / to automatically redirect users to the register page when they access the base URL.

```python
# 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 = request.form['password']

        if users_col.find_one({"email": email}):
            return render_template('register.html', error="Email already registered.")

        hashed_password = generate_password_hash(password)
        users_col.insert_one({
            'email': email,
            'username': username,
            'hashed_password': hashed_password,
            'login_count': 0
        })
        return redirect(url_for('login'))

    return render_template('register.html')

# --- Login ---
@app.route('/login', methods=['GET', 'POST'])
def login():
    if request.method == 'POST':
        email = request.form['email']
        password = request.form['password']
        user = users_col.find_one({"email": email})

        if user and check_password_hash(user['hashed_password'], password):
            session['user_id'] = str(user['_id'])
            session['email'] = user['email']
            session['username'] = user['username']
            users_col.update_one({"email": email}, {"$inc": {"login_count": 1}})
            flash('Login successful!', 'success')
            return redirect(url_for('user_dashboard'))

        return render_template('login.html', error="Invalid credentials")

    return render_template('login.html')
```

Description: define /register route to validate registration form fields, hash the user password using Bcrypt, store the new user in DynamoDB with a login count, and send an SNS notification on successful registration
define /login route to validate user credentials against DynamoDB, check the password using Bcrypt, update the login count on successful authentication, and redirect users to the home page

Dashboard and other routes:

```python
# --- Dashboard ---
@app.route('/user_dashboard')
@login_required
def user_dashboard():
    email = session['email']
    bookings = list(bookings_col.find({"email": email}).sort("booked_at", -1))
    bus = list(db['bus_bookings'].find({"email": email}).sort("booked_at", -1))
    train = list(train_bookings_col.find({'email': email}))
    hotels = list(hotel_bookings_col.find({"email": email}).sort("booked_at", -1))
    trips = list(trip_plans_col.find({'email': email}).sort("created_at", -1))

    return render_template('user_dashboard.html', bookings=bookings, bus_bookings=bus,
                           train_bookings=train, hotel_bookings=hotels, trip_plans=trips)


# --- Search Places ---
@app.route('/search_places', methods=['GET'])
def search_places():
    query = request.args.get('location', '').strip()
    category = request.args.get('category', '').strip()

    if not query:
        return render_template("search_results.html", results=[], message="Please enter a location to search.")

    try:
        # Use Atlas Search if available
        pipeline = [{"$search": {"index": "default", "text": {"query": query, "path": ["location", "name"]}}}]
    except:
        # Fallback to basic regex match if $search fails
        pipeline = [{"$match": {
            "$or": [
                {"location": {"$regex": query, "$options": "i"}},
                {"name": {"$regex": query, "$options": "i"}}
            ]
        }}]

    if category:
        pipeline.append({"$match": {"category": {"$regex": category, "$options": "i"}}})

    pipeline.append({"$limit": 20})
    results = list(db.places.aggregate(pipeline))
    for r in results:
        r['_id'] = str(r['_id'])
```

Description: The /dashboard route provides a personalized view where logged-in users can see all their current bookings—flights, buses, trains, and hotels. It fetches data from the database and allows users to manage or cancel their reservations.

The /search (or /booking) route lets users search for travel options by submitting a unified form. Based on the selected tab (flight, bus, train, or hotel), it filters results from available data and shows relevant options for booking.

Request booking Routes:

```python
@app.route('/booking', methods=['GET', 'POST'])
def booking():
    active_tab = 'flights'  # default tab
    flights = buses = trains = hotels = []

    if request.method == 'POST':
        session['check_in'] = request.form.get('check_in')
        session['check_out'] = request.form.get('check_out')
        session['passengers'] = int(request.form.get('passengers', 1))
        active_tab = request.form.get('active_tab', 'flights')
        form_type = request.form.get('form_type')

        if form_type == 'flight':
            origin = request.form.get('origin')
            destination = request.form.get('destination')
            date = request.form.get('date').strip()
            passengers = int(request.form.get('passengers', 1))
            flights = [f for f in flights_data if f['origin'] == origin and f['destination'] == destination and f['departure_time'].startswith(date)]
            return render_template('booking.html', flights=flights, active_tab=active_tab, passengers=passengers)

        elif form_type == 'bus':
            origin = request.form.get('origin')
            destination = request.form.get('destination')
            date = request.form.get('date')
            buses = [b for b in buses_data if b['origin'] == origin and b['destination'] == destination and b['departure'] == date]
            return render_template('booking.html', buses=buses, active_tab=active_tab)

        elif form_type == 'train':
            origin = request.form.get('origin')
            destination = request.form.get('destination')
            date = request.form.get('date')
            trains = [t for t in trains_data if t['origin'] == origin and t['destination'] == destination and t['date'] == date]
            return render_template('booking.html', trains=trains, active_tab=active_tab)

        elif form_type == 'hotel':
            location = request.form.get('location')
            check_in = request.form.get('check_in')
            check_out = request.form.get('check_out')
            hotels = [h for h in hotels_data if h['location'].lower() == location.lower()]
            return render_template('booking.html', hotels=hotels, active_tab=active_tab)

    return render_template('booking.html',flights=flights,  buses=buses, trains=trains, hotels=hotels, active_tab=active_tab)

@app.route('/confirm_booking/<int:flight_id>', methods=['GET', 'POST'])
```

Description: The /booking route accepts both GET and POST requests. On GET, it shows the search interface with tabs for flights, buses, trains, and hotels. On POST, it processes the form

submission, filters available options based on origin, destination, date, and passengers, and renders the results accordingly.

Train booking route:

```python
@app.route('/train/confirm/<int:train_id>', methods=['GET', 'POST'])
def confirm_train_booking(train_id):
    train = next((t for t in trains_data if t['id'] == train_id), None)
    if not train:
        return "Train not found", 404

    try:
        passenger_count = int(request.args.get('passengers') or session.get('passengers', 1))
    except ValueError:
        passenger_count = 1

    if request.method == 'POST':
        action = request.form.get('action')
        selected_seats = request.form.getlist('seats[]')

        if action == 'cancel':
            flash("Train booking cancelled.", "info")
            return redirect(url_for('booking'))

        if not selected_seats:
            flash("Please select at least one seat before confirming.", "warning")
            return redirect(request.url)

        if len(selected_seats) > passenger_count:
            flash(f"You selected {len(selected_seats)} seats but only {passenger_count} passenger(s).", "danger")
            return redirect(request.url)

        for seat in selected_seats:
            if train['seats'].get(seat) == 'booked':
                flash(f"Seat {seat} is already booked. Please choose another.", "danger")
                return redirect(request.url)

        passenger_names = request.form.getlist('passenger_names[]')
        if len(passenger_names) != len(selected_seats):
            flash("Number of passenger names does not match selected seats.", "danger")
            return redirect(request.url)

        for seat, name in zip(selected_seats, passenger_names):
            train['seats'][seat] = 'booked'
            train_bookings_col.insert_one({
        'email': session['email'],
        'train_id': train['id'],
        'train_number': train['train_number'],
        'origin': train['origin'],
        'destination': train['destination'],
```

Description: The /train/confirm/<int:train_id> route handles train booking. It shows train info and seat selection on GET, and processes booking or cancellation on POST. It validates seat availability, matches passenger details, saves the booking, and redirects to the dashboard.

Bus booking route:

```python
@app.route('/bus/confirm/<int:bus_id>', methods=['GET', 'POST'])
def confirm_bus_booking(bus_id):
    bus = next((b for b in buses_data if b['id'] == bus_id), None)
    passenger_names = request.form.getlist('passenger_names[]')
    if not bus:
        return "Bus not found", 404

    if request.method == 'POST':
        action = request.form.get('action')
        selected_seats = request.form.getlist('seats')  # ✅ get list of selected seats

        if action == 'confirm':
            if not selected_seats:
                flash('Please select at least one seat.', 'danger')
                return redirect(request.url)

            for seat in selected_seats:
                if bus['seats'].get(seat) == 'booked':
                    flash(f"Seat {seat} already booked.", "danger")
                    return redirect(request.url)

            # Book seats
            for i,seat in enumerate(selected_seats):
                bus['seats'][seat] = 'booked'
                bus_bookings_col.insert_one({
                    'email': session['email'],
                    'bus_id': bus['id'],
                    'bus_number': bus['bus_number'],
                    'origin': bus['origin'],
                    'destination': bus['destination'],
                    'departure_time': bus['departure_time'],
                    'arrival_time': bus['arrival_time'],
                    'price': bus['price'],
                    'seat': seat,
                    'passenger_name': passenger_names[i] if i < len(passenger_names) else '',
                    'booked_at': datetime.utcnow()
                })

            flash(f'Bus booking confirmed! Seats: {", ".join(selected_seats)}', 'success')
            return redirect(url_for('user_dashboard'))
        else:
            flash('Bus booking cancelled.', 'info')
            return redirect(url_for('booking'))

    return render_template('confirm_bus_booking.html', bus=bus, passengers=session.get('passengers', 1))
```

Description: The /bus/confirm/<int:bus_id> route manages bus bookings. It displays bus details and available seats, handles booking confirmation or cancellation, checks seat availability, and stores confirmed bookings in the database before redirecting to the user dashboard.

Hotel booking routes:

```python
app.py > login
@app.route('/hotel/confirm/<int:hotel_id>', methods=['GET', 'POST'])
def confirm_hotel_booking(hotel_id):
    hotel = next((h for h in hotels_data if h['id'] == hotel_id), None)
    if not hotel:
        return "Hotel not found", 404

    # Get check-in and check-out dates from session
    check_in = session.get('check_in')
    check_out = session.get('check_out')

    # Calculate total nights
    try:
        nights = (datetime.strptime(check_out, '%Y-%m-%d') - datetime.strptime(check_in, '%Y-%m-%d')).days
        if nights <= 0:
            nights = 1
    except (TypeError, ValueError):
        nights = 1  # fallback if dates are missing or malformed

    if request.method == 'POST':
        if request.form['action'] == 'confirm':
            hotel_bookings_col.insert_one({
                'email': session['email'],
                'hotel_id': hotel['id'],
                'hotel_name': hotel['name'],
                'location': hotel['location'],
                'price': hotel['price_per_night'],
                'rating': hotel['rating'],
                'check_in': check_in,
                'check_out': check_out,
                'booked_at': datetime.~~utcnow~~()
            })
            flash('Hotel booking confirmed!', 'success')
            return redirect(url_for('user_dashboard'))
        else:
            flash('Hotel booking cancelled.', 'info')
            return redirect(url_for('booking'))

    return render_template(
        'confirm_hotel_booking.html',
        hotel=hotel,
        check_in=check_in,
        check_out=check_out,
        nights=nights
    )
```

Description: The /hotel/confirm/<int:hotel_id> route handles hotel booking confirmation. It retrieves hotel info, calculates nights based on check-in/check-out dates, and on confirmation, saves the booking in the database, then redirects to the user dashboard.

Exit Route:

```python
# --------------- LOGOUT ---------------
@app.route('/logout')
def logout():
    session.pop('email', None)
    return redirect(url_for('login'))
```

Description: define /exit route to render the exit.html page when the user chooses to leave or close the application.

Deployment Code:

```
850
851   # ---------------- RUN ----------------
852   if __name__ == '__main__':
853       app.run(host='0.0.0.0', port=80, debug=True)
854
```

Description: start the Flask server to listen on all network interfaces (0.0.0.0) at port 80 with debug mode enabled for development and testing.

Milestone 2 : AWS Account Setup

Important Notice: Use Troven Labs for AWS Access

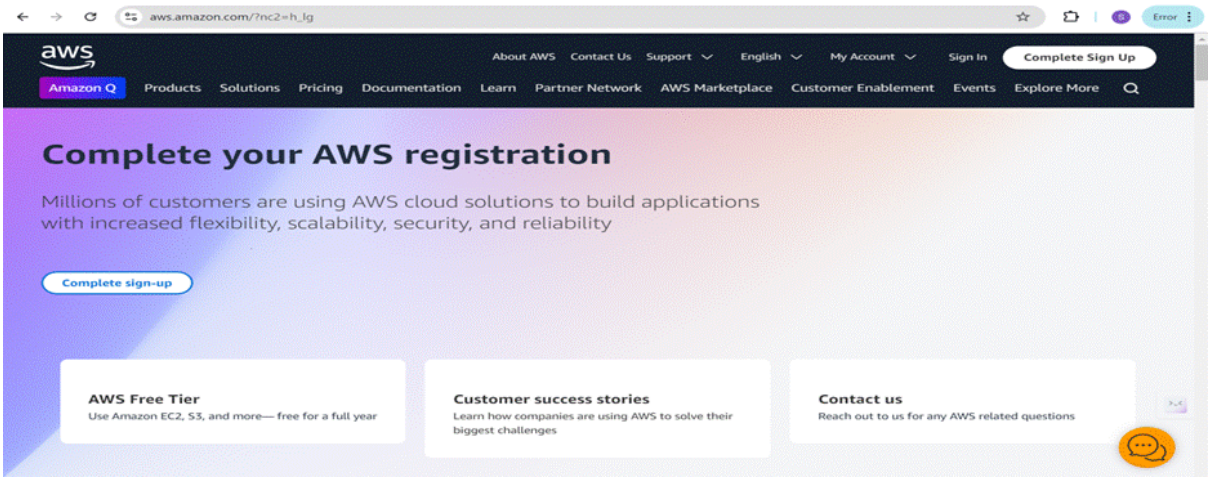
Students are strictly advised not to create their own AWS accounts, as doing so may incur charges. Instead, we have set up a dedicated section called “Labs” on the Troven platform, which provides temporary and cost-free access to AWS services.

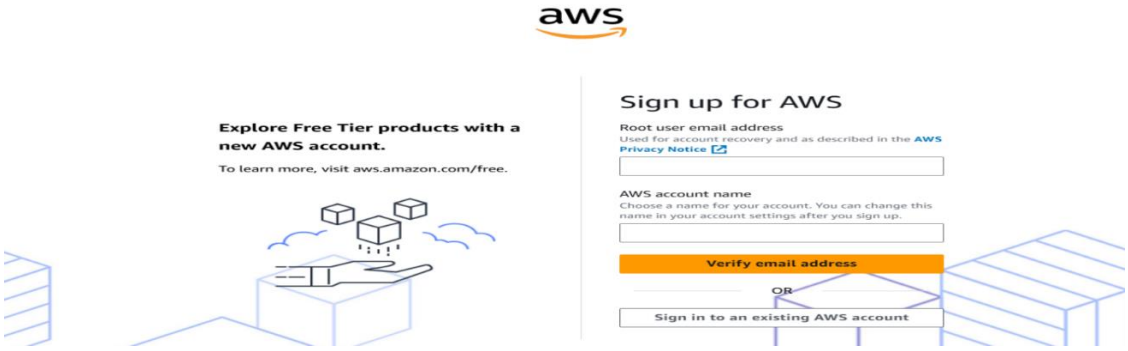
Once your website is locally deployed and fully functional, you must proceed with integrating AWS services only through the Troven Labs environment. This ensures secure, controlled access to AWS resources without any risk of personal billing.

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.

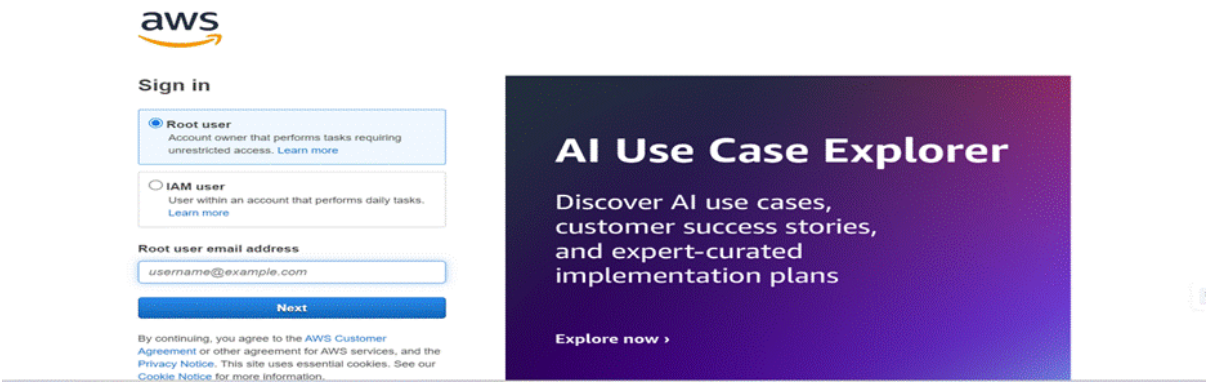
AWS Account Setup and Login

- 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]

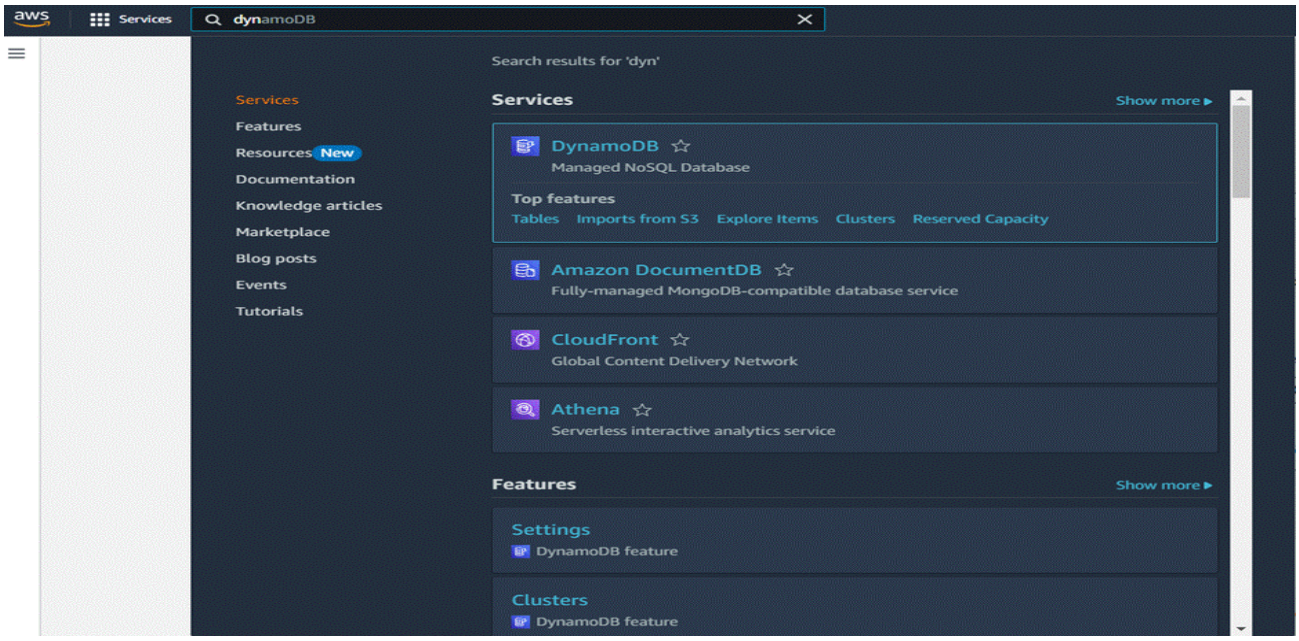


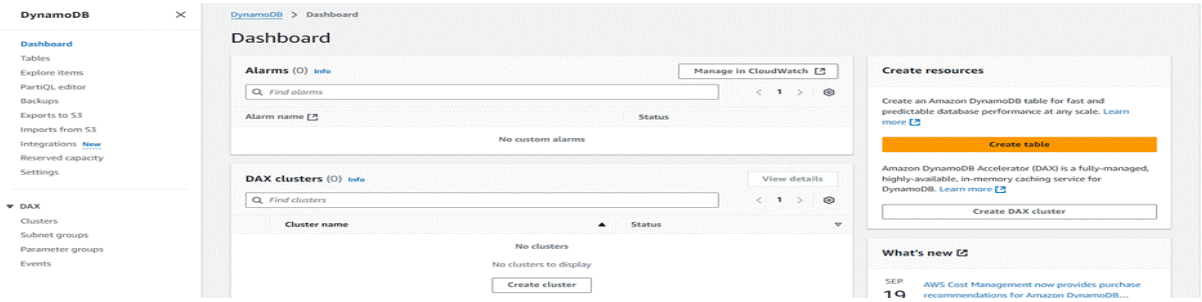
Milestone 3 : DynamoDB Database Creation and Setup

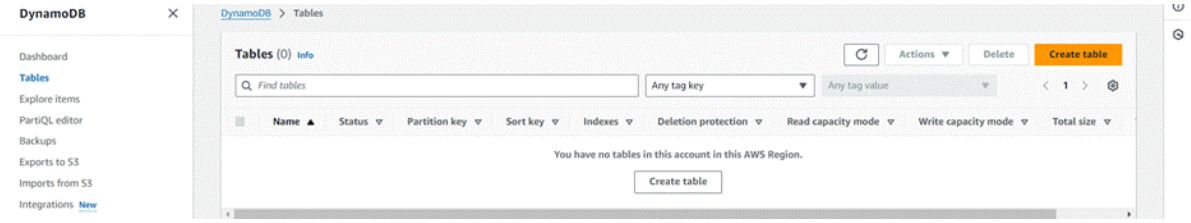
Database Creation and Setup involves initializing a cloud-based NoSQL database to store and manage application data efficiently. This step includes defining tables, setting primary keys, and configuring read/write capacities. It ensures scalable, high-performance data storage for seamless backend operations.

Navigate to the DynamoDB

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

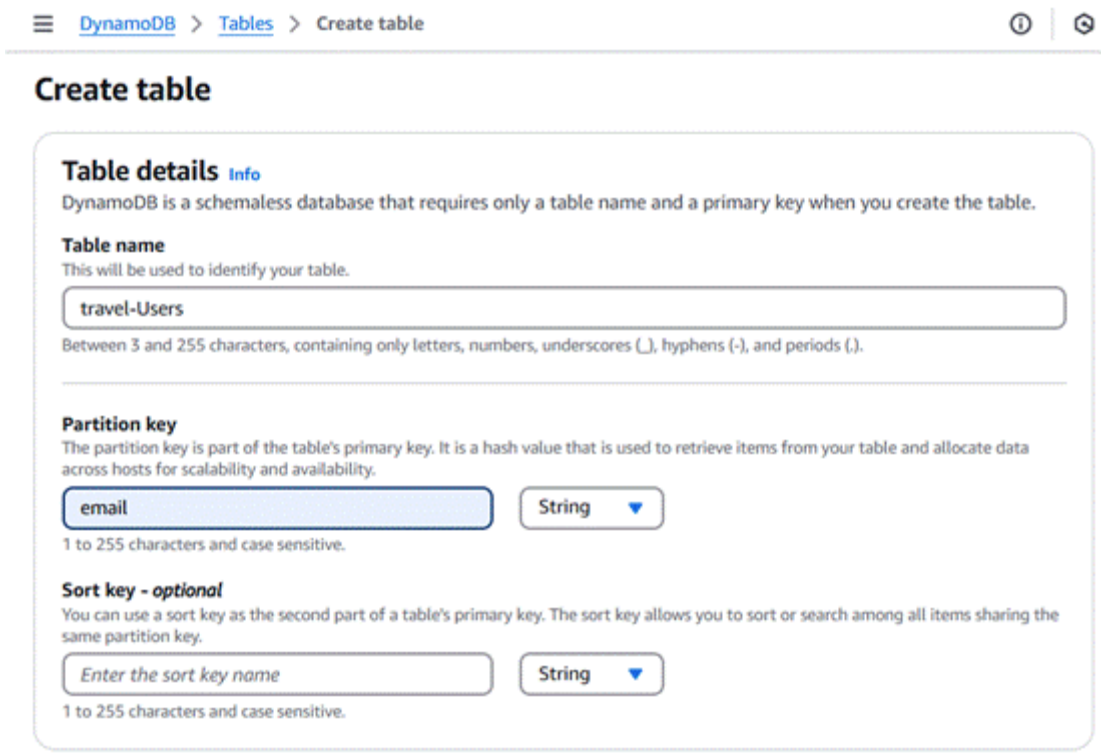




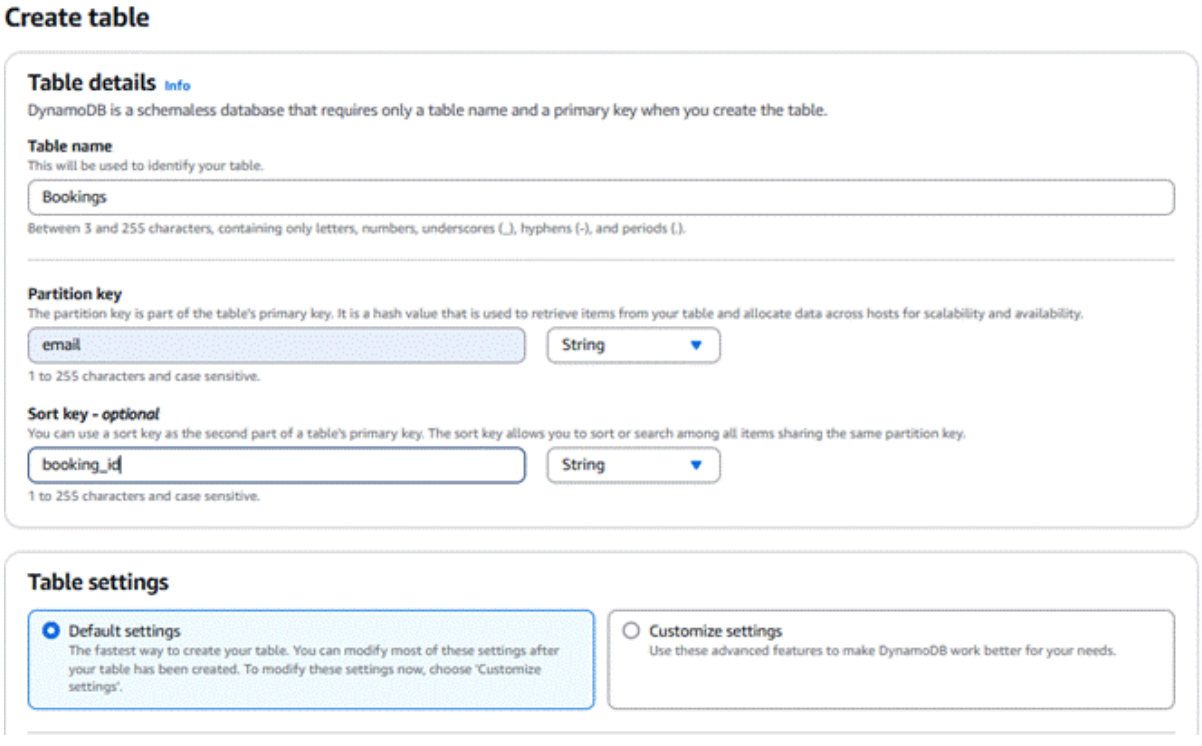


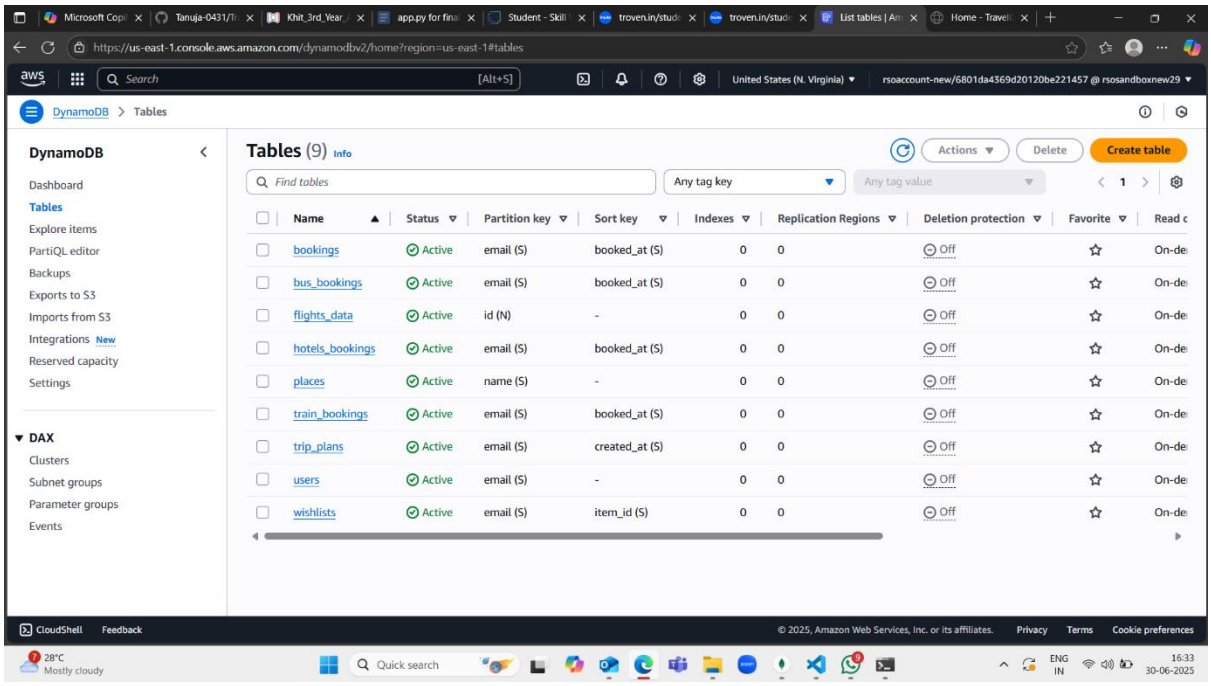
Create an DynamoDB table for storing data

- 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.



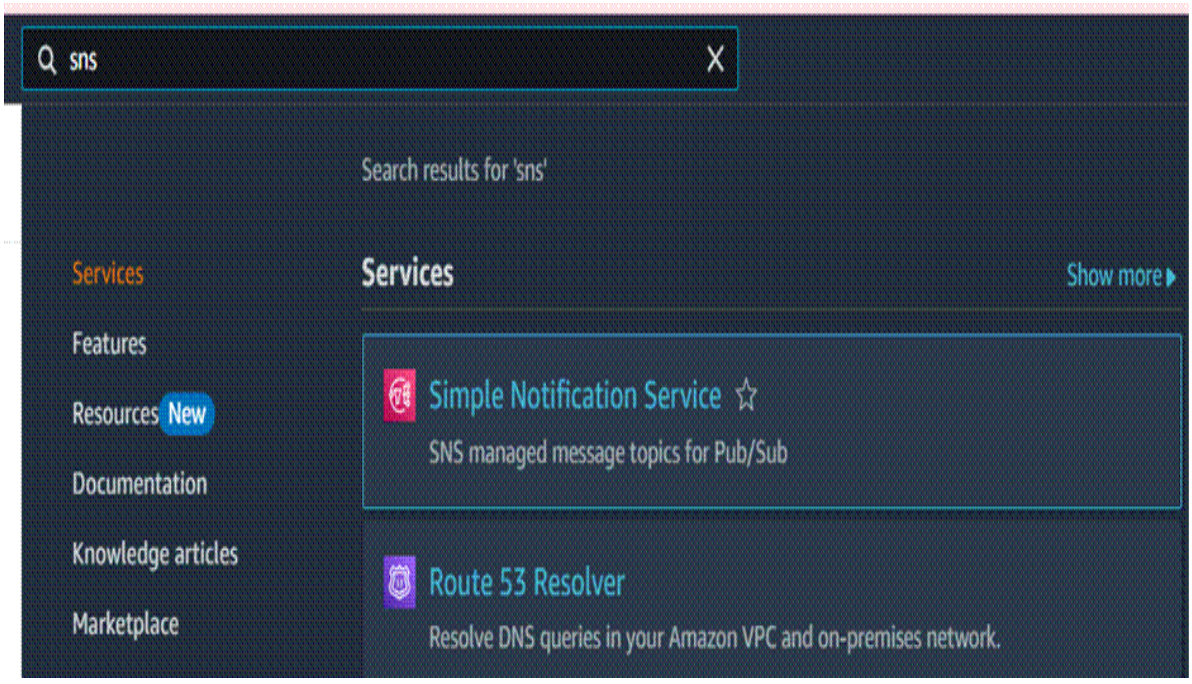


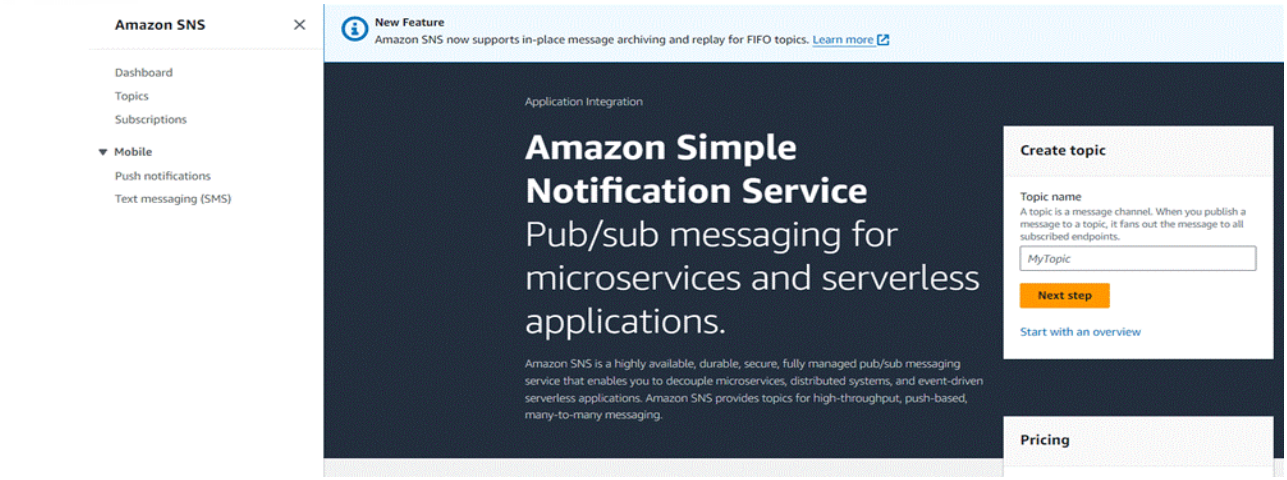
Milestone 4 : SNS Notification Setup

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.

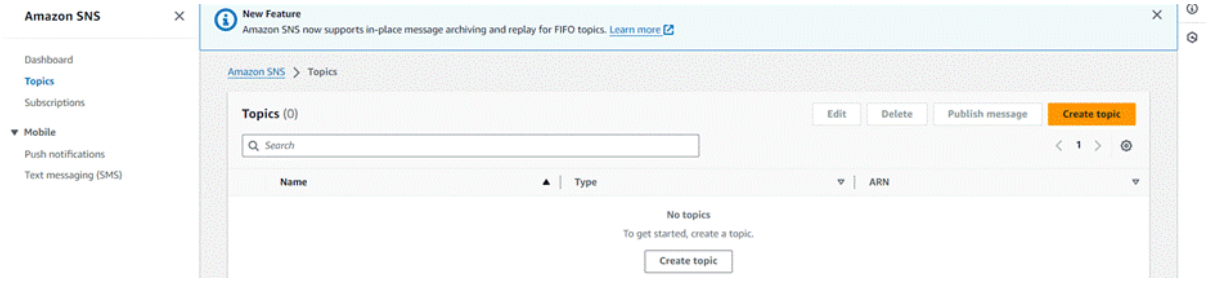
SNS Topics for email notifications

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

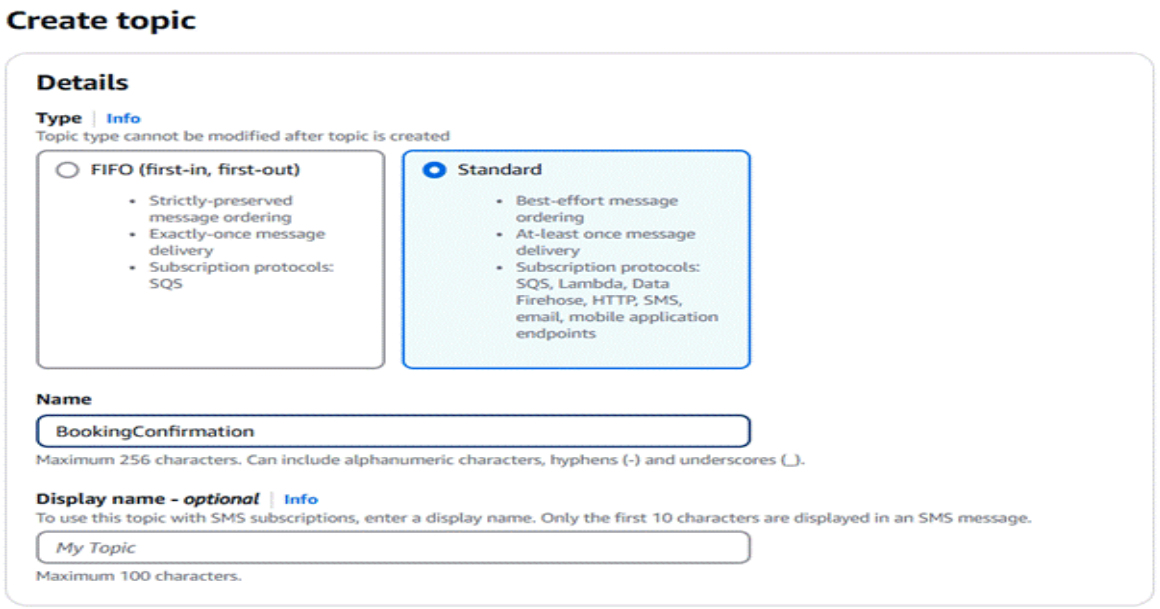


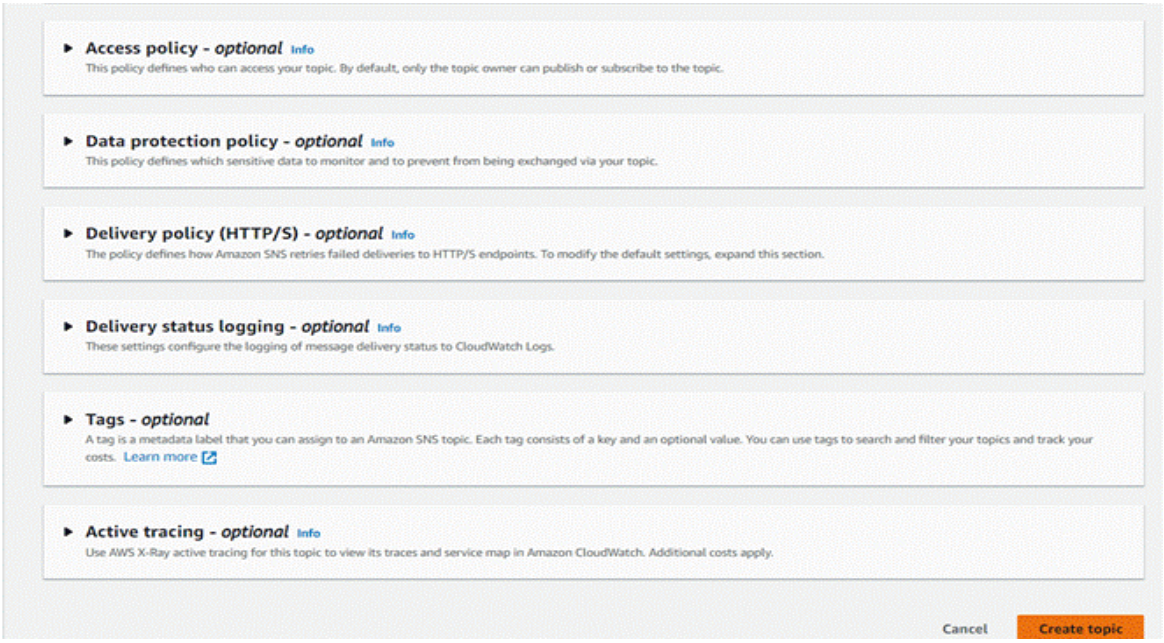


- Click on Create Topic and choose a name for the topic.

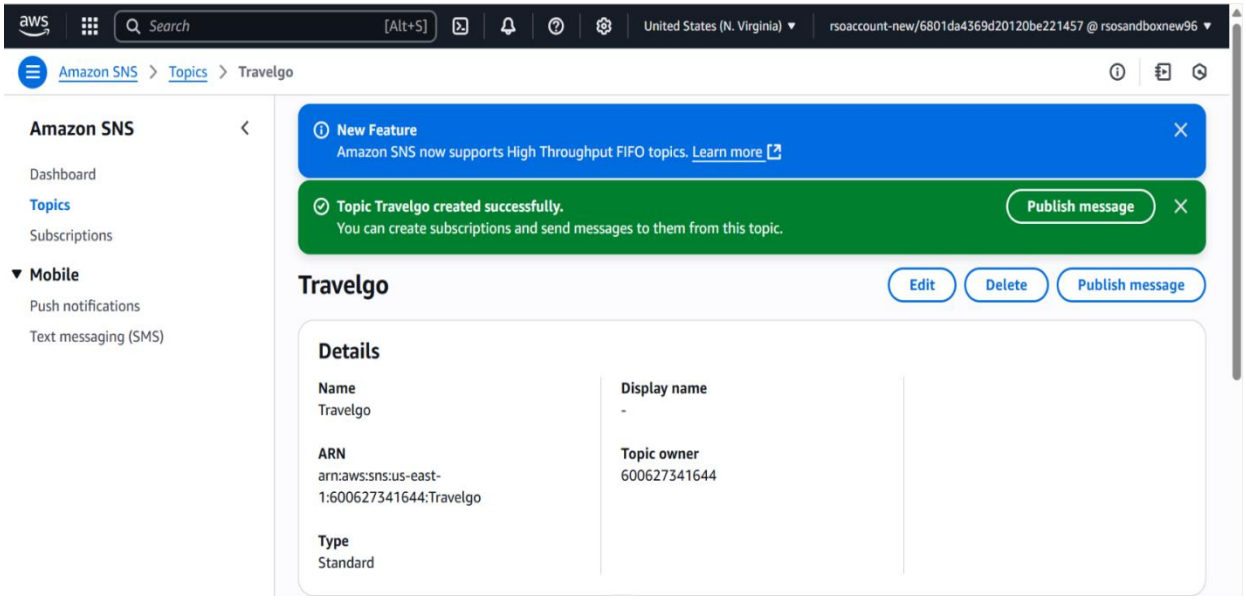


? Choose Standard type for general notification use cases and Click on Create Topic.



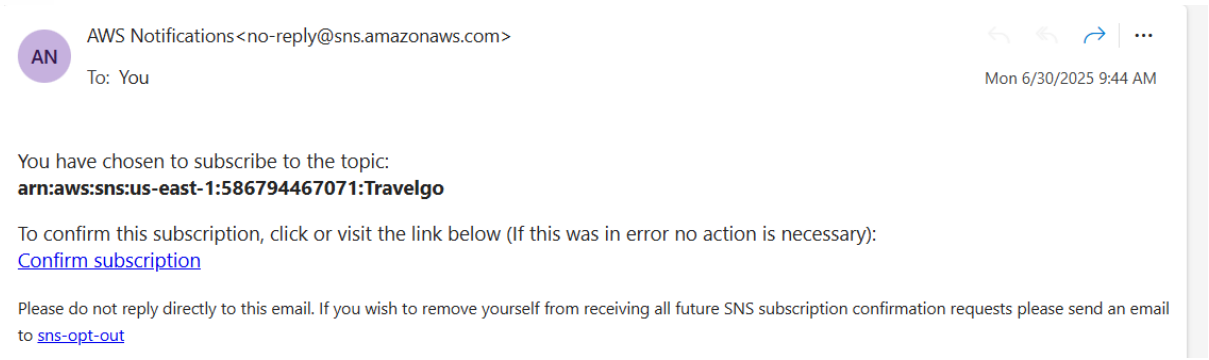


? Configure the SNS topic and note down the Topic ARN.

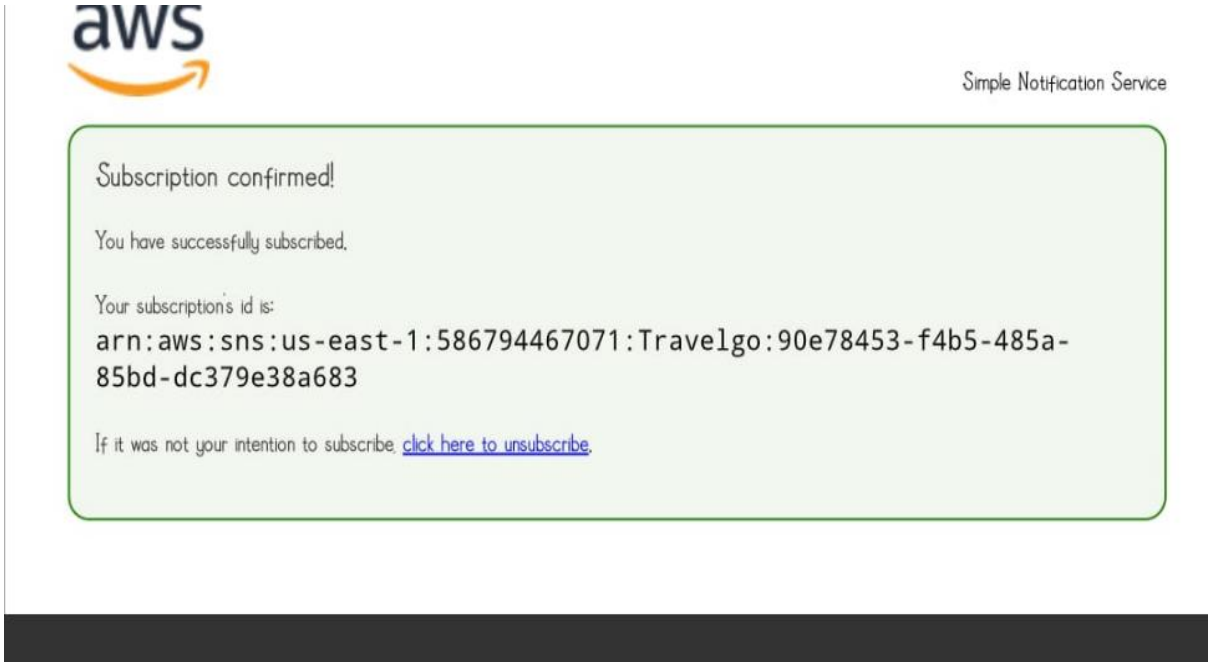


Subscribe users and admin

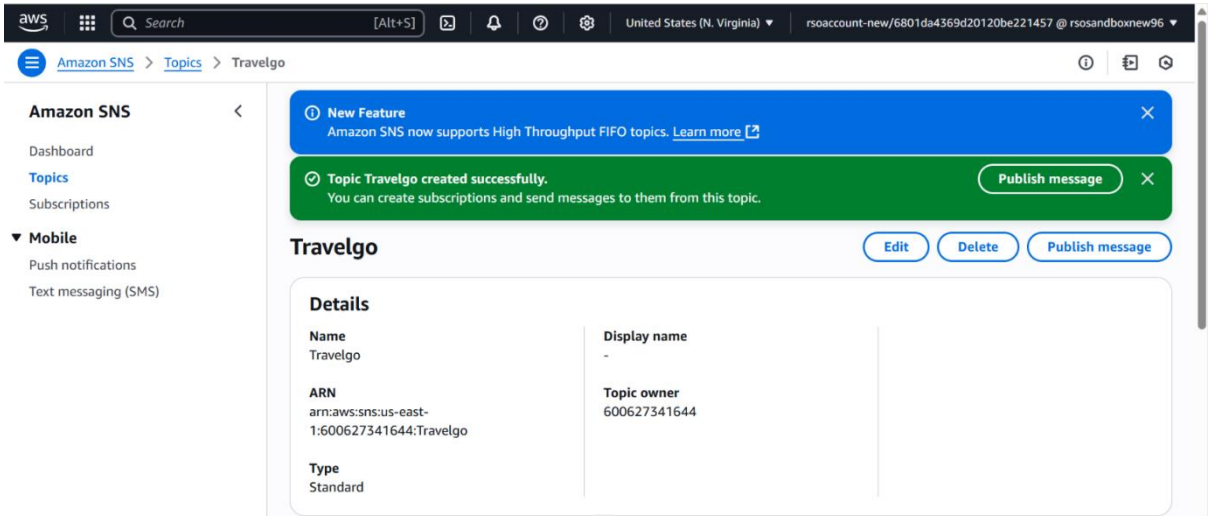
- Subscribe users (or admin staff) to this topic via Email. When a book request is made, notifications will be sent to the subscribed emails.
- After subscription request for the mail confirmation

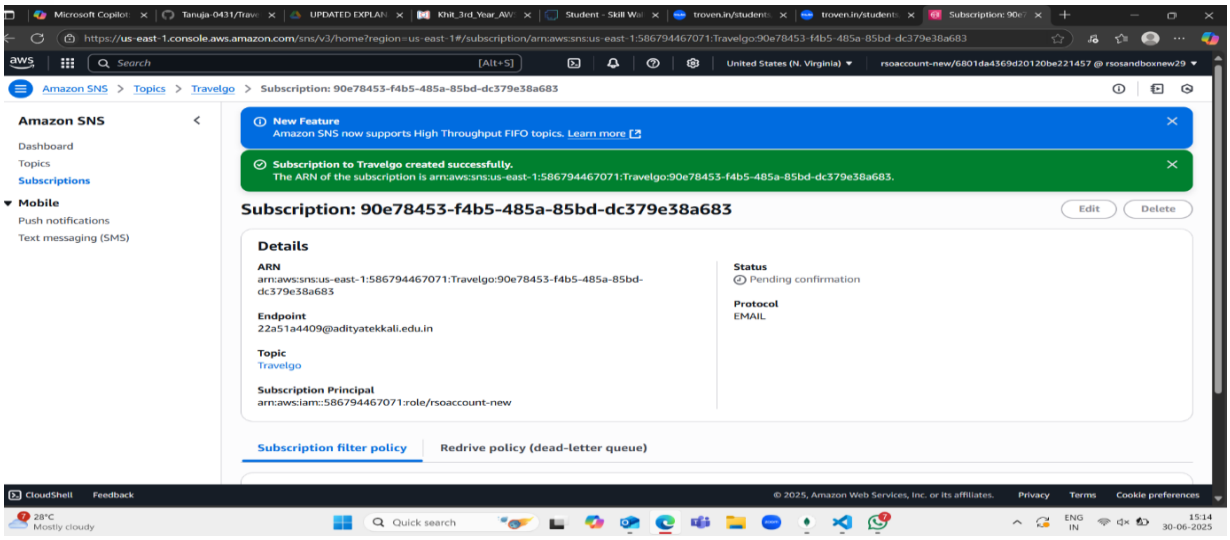


- Navigate to the subscribed Email account and Click on the confirm subscription in the AWS Notification- Subscription Confirmation mail.



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



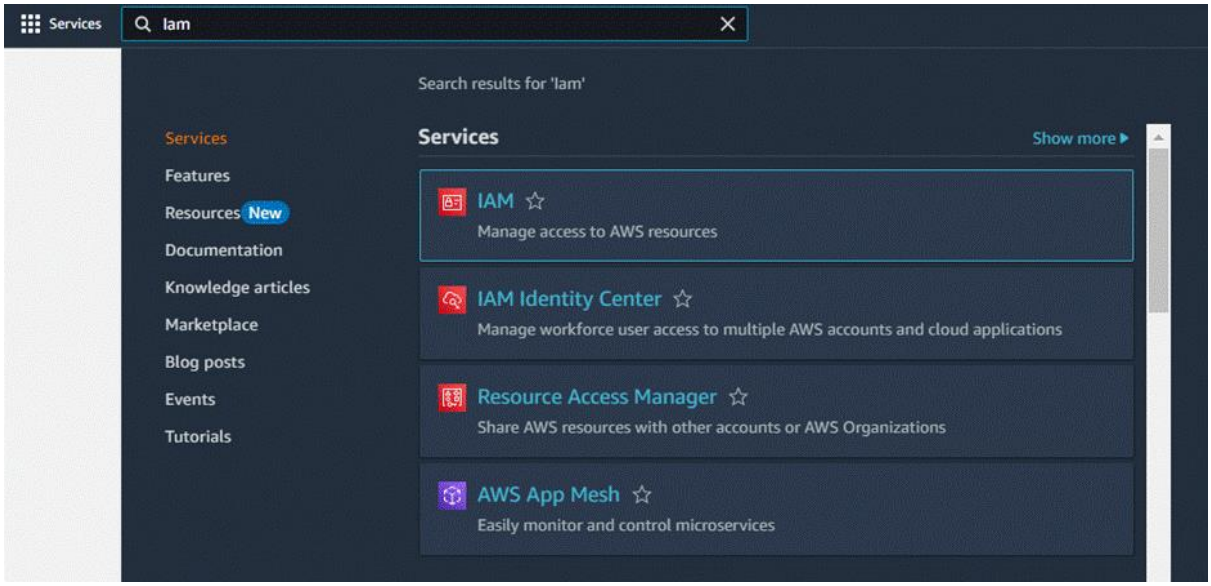


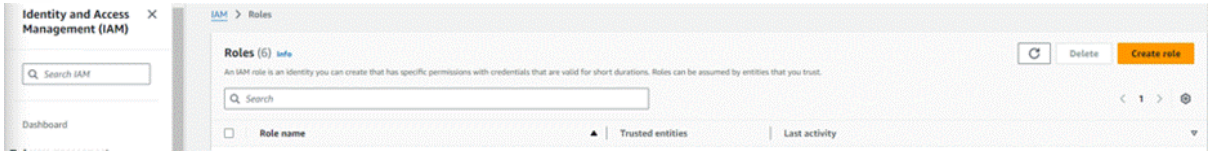
Milestone 5 : IAM Role Setup

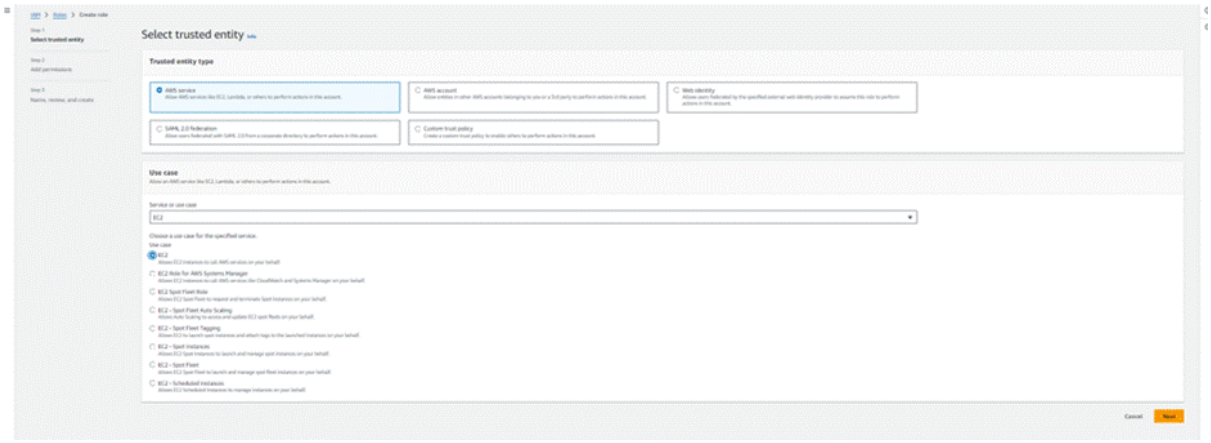
IAM (Identity and Access Management) role setup involves creating roles that define specific permissions for AWS services. To set it up, you create a role with the required policies, assign it to users or services, and ensure the role has appropriate access to resources like EC2, S3, or RDS. This allows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

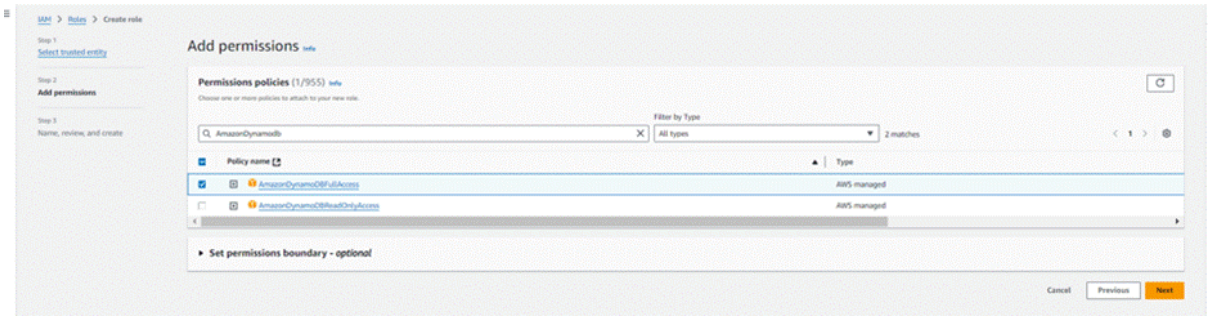
Create IAM Role.

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







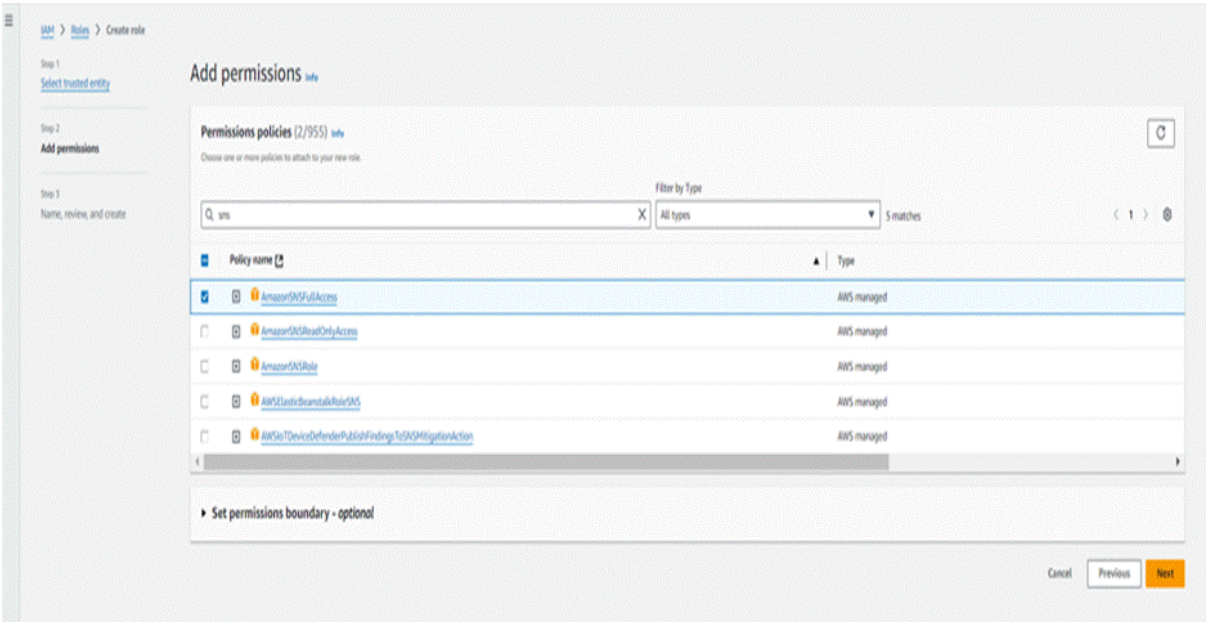


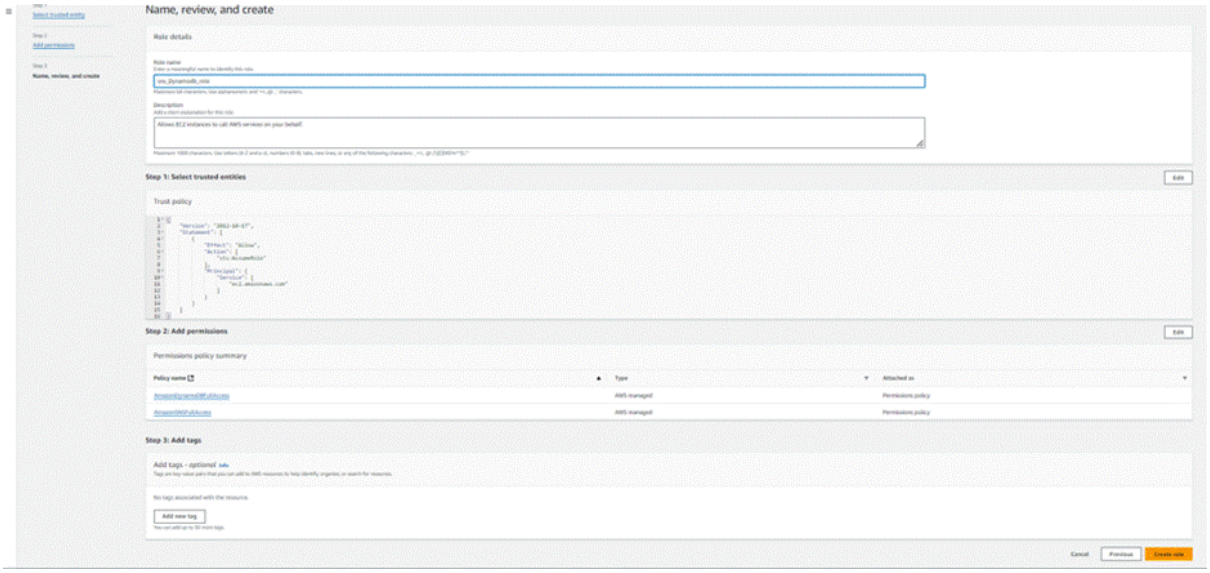
Attach Policies.

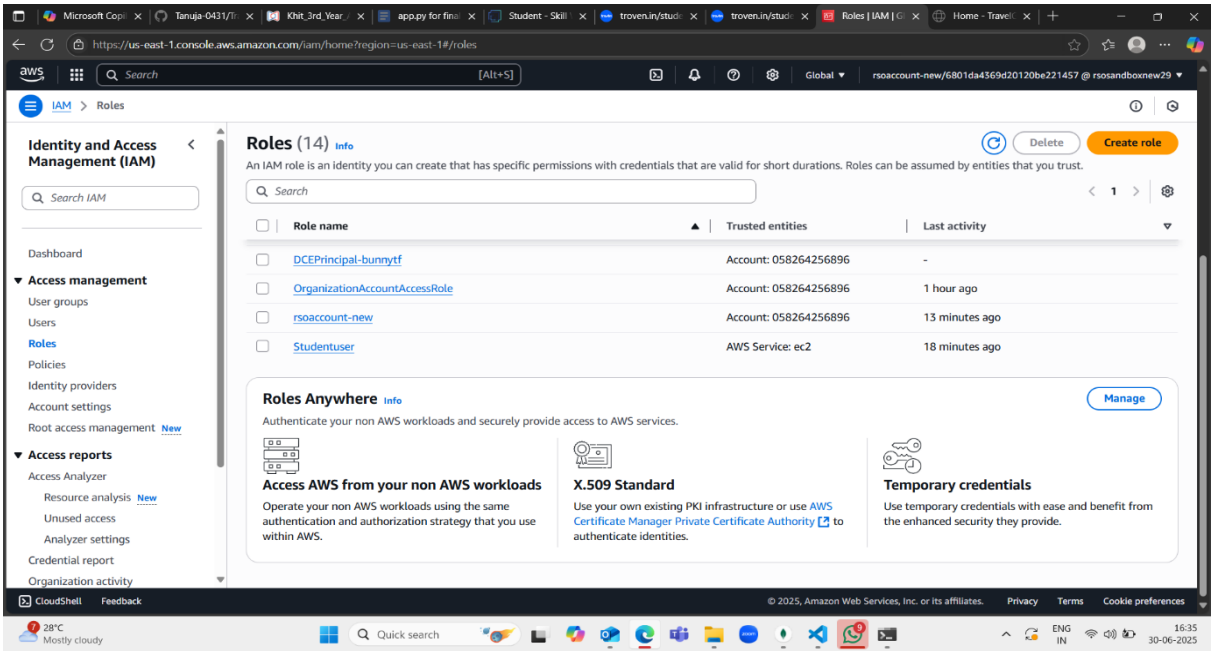
Attach the following policies to the role:

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

SMARTBRIDGE
Let's Bridge the Gap



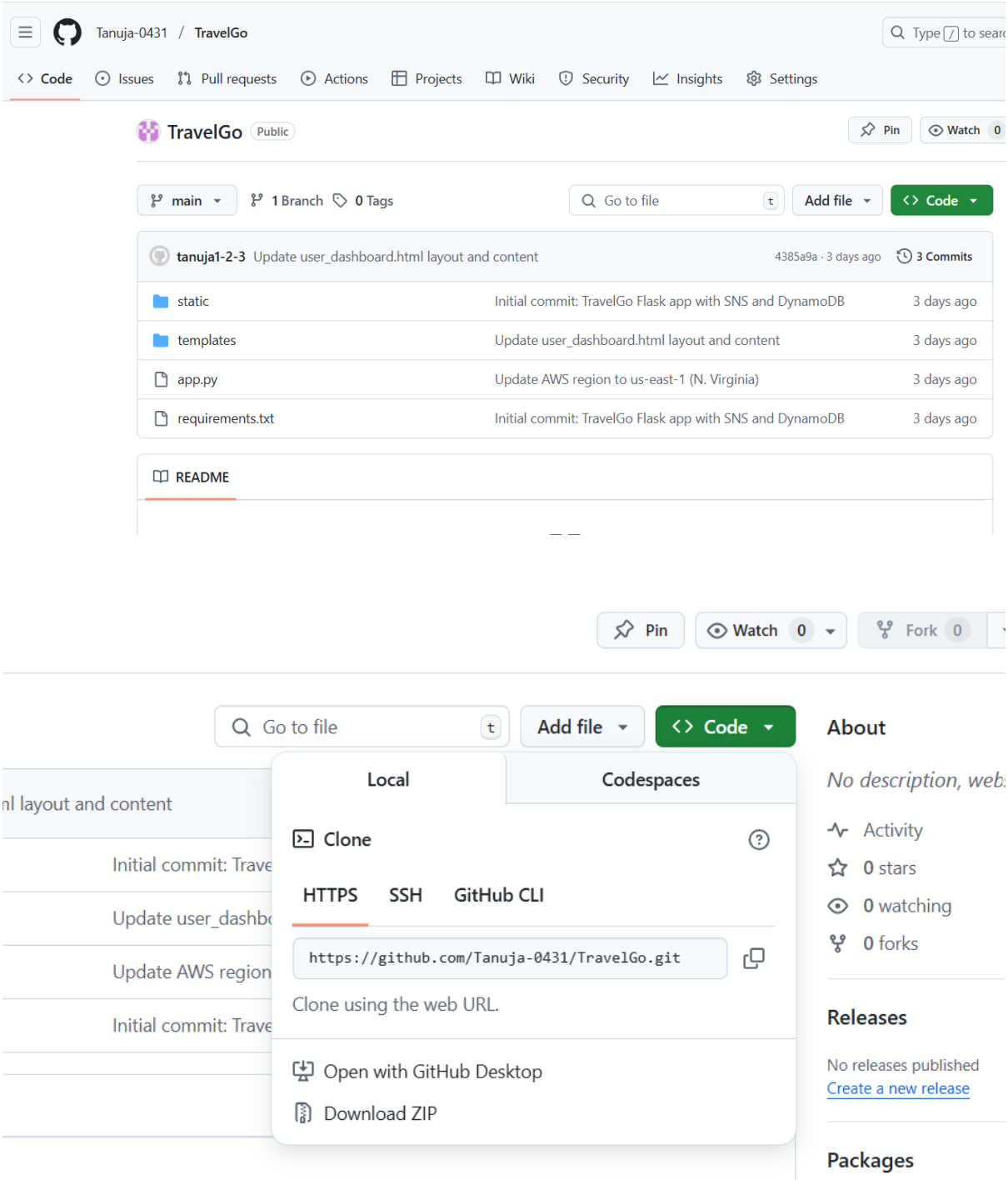




Milestone 6 : EC2 Instance Setup

- 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.

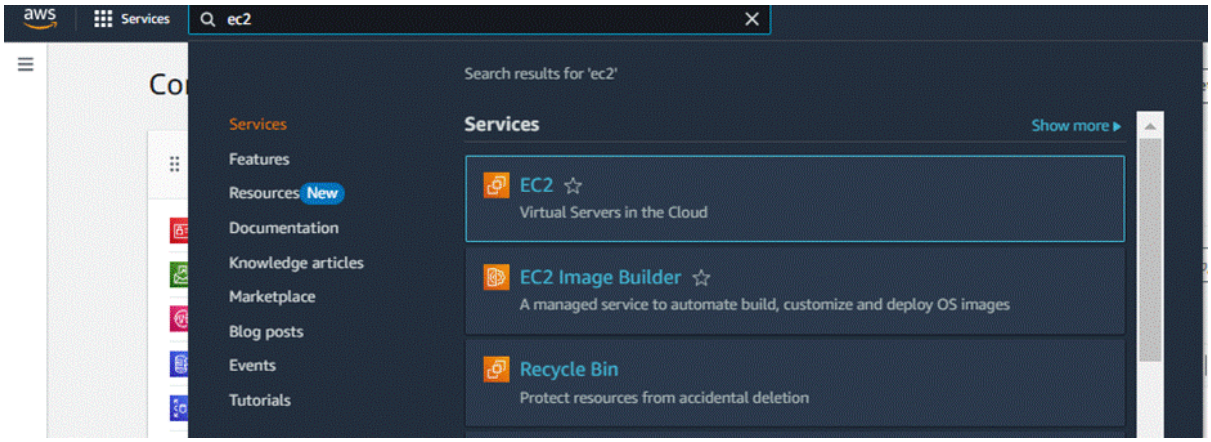


Launch An EC2 instance to host The flask

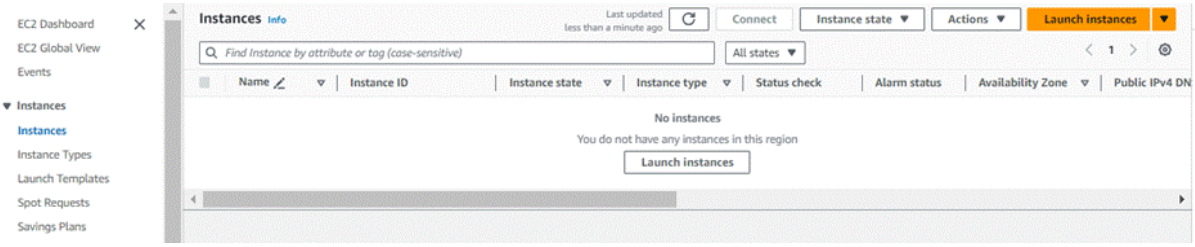
Load your Project Files to Github

- Launch EC2 Instance

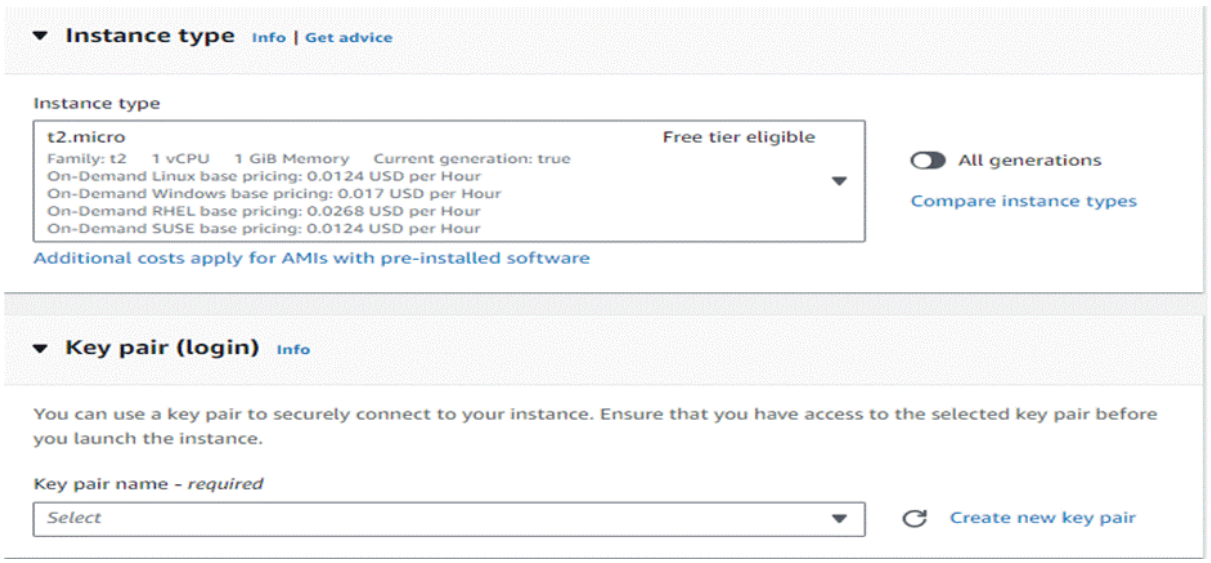
- In the AWS Console, navigate to EC2 and launch a new instance.



- Click on Launch instance to launch EC2 instance

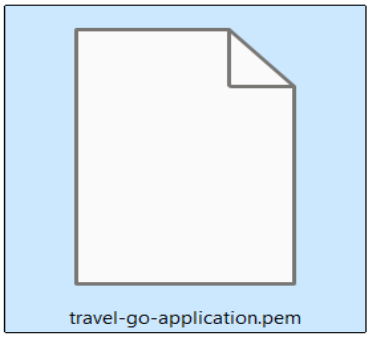


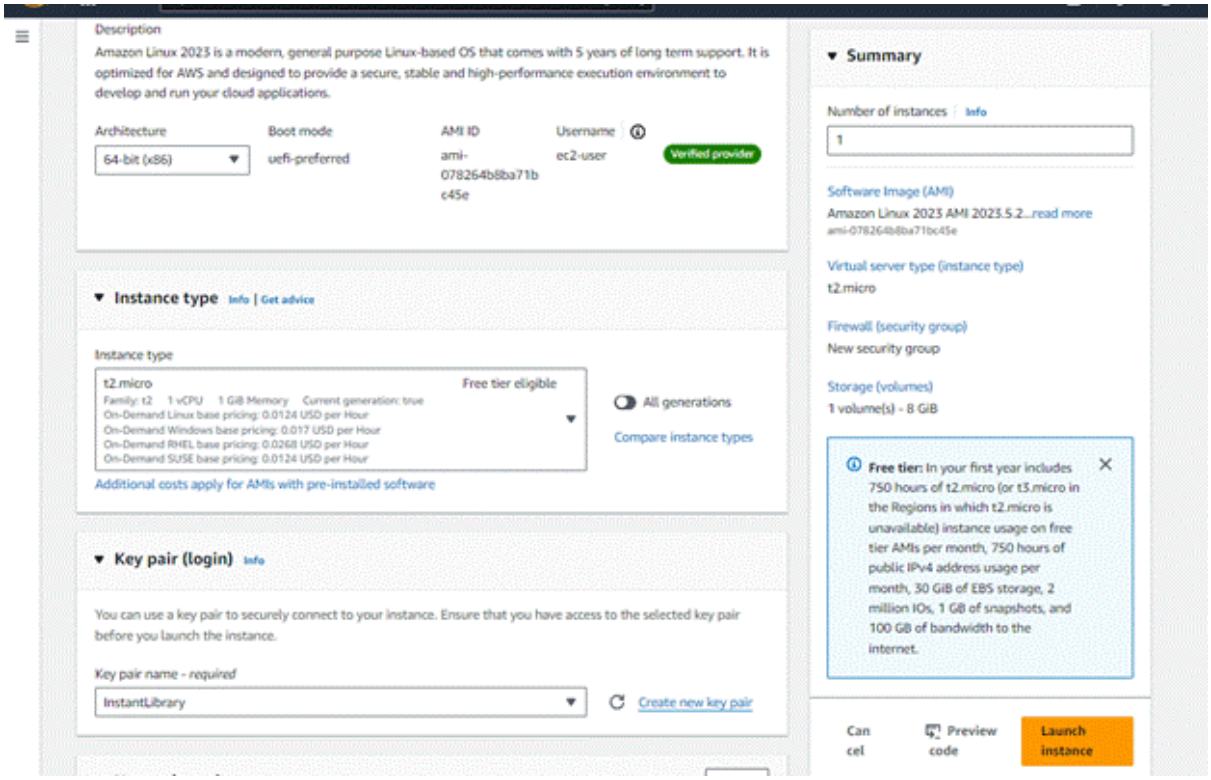
- Choose Amazon Linux 2 or Ubuntu as the AMI and t2.micro as the instance type (free-tier eligible).
- Create and download the key pair for Server access.

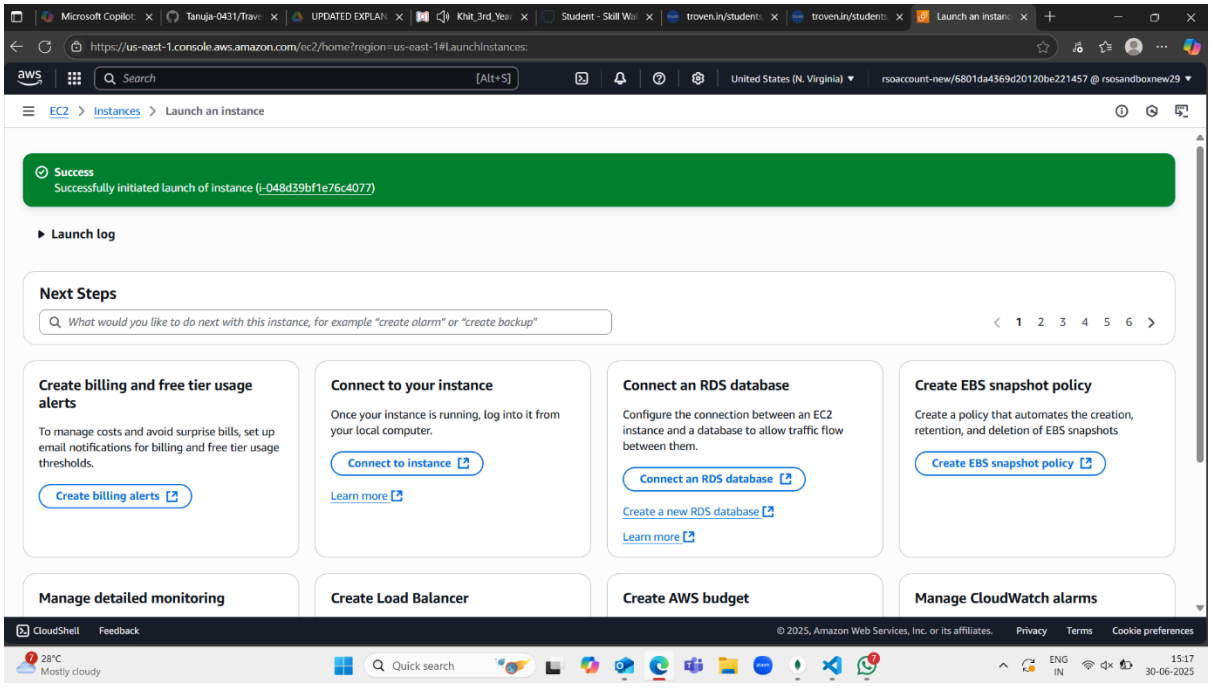


SMARTBRIDGE
Let's Bridge the Gap

Earlier this week



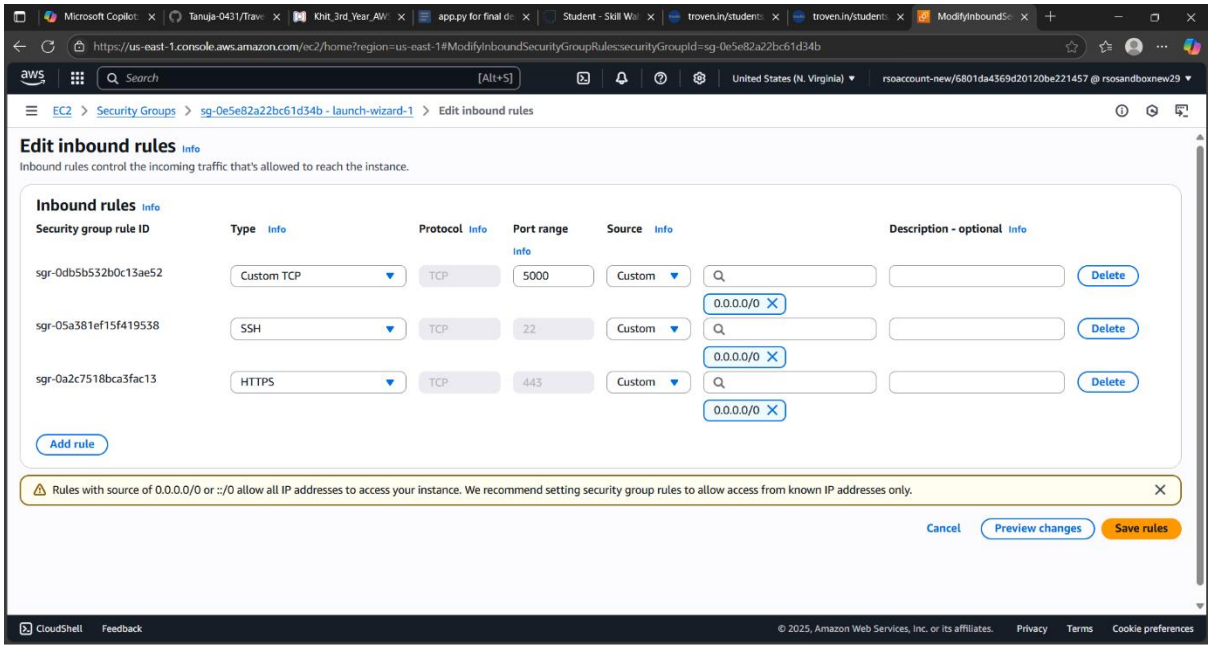


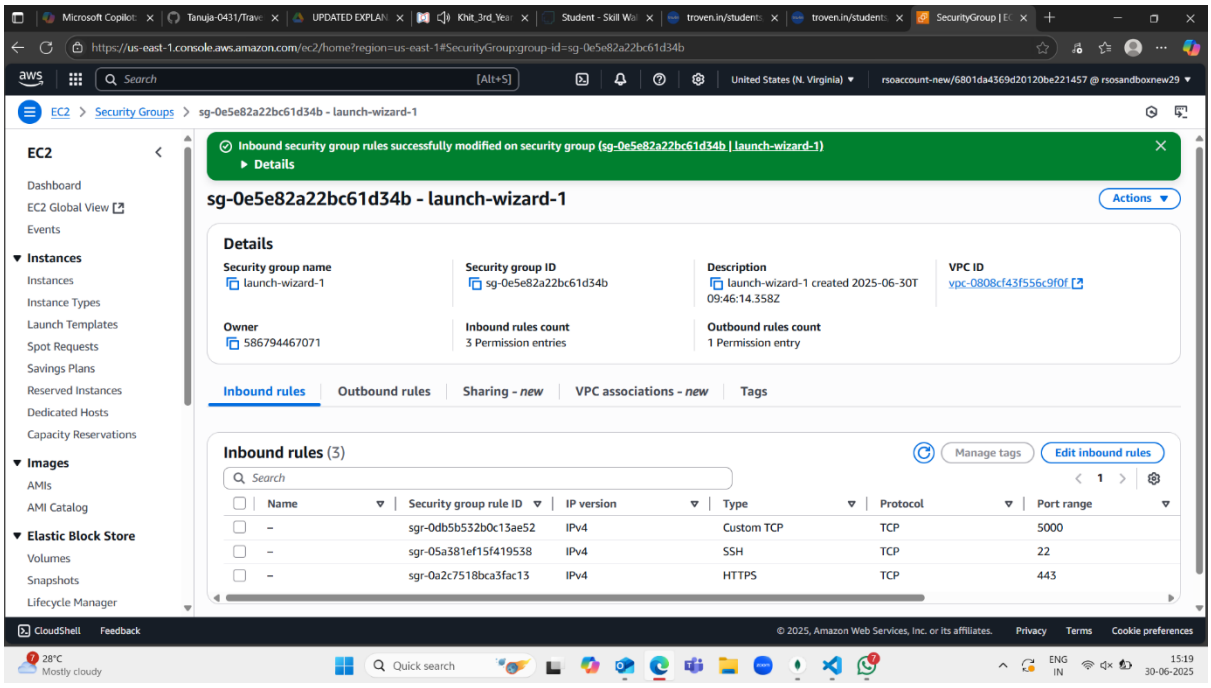


SMARTBRIDGE

Configure security groups for HTTP, and SSH access

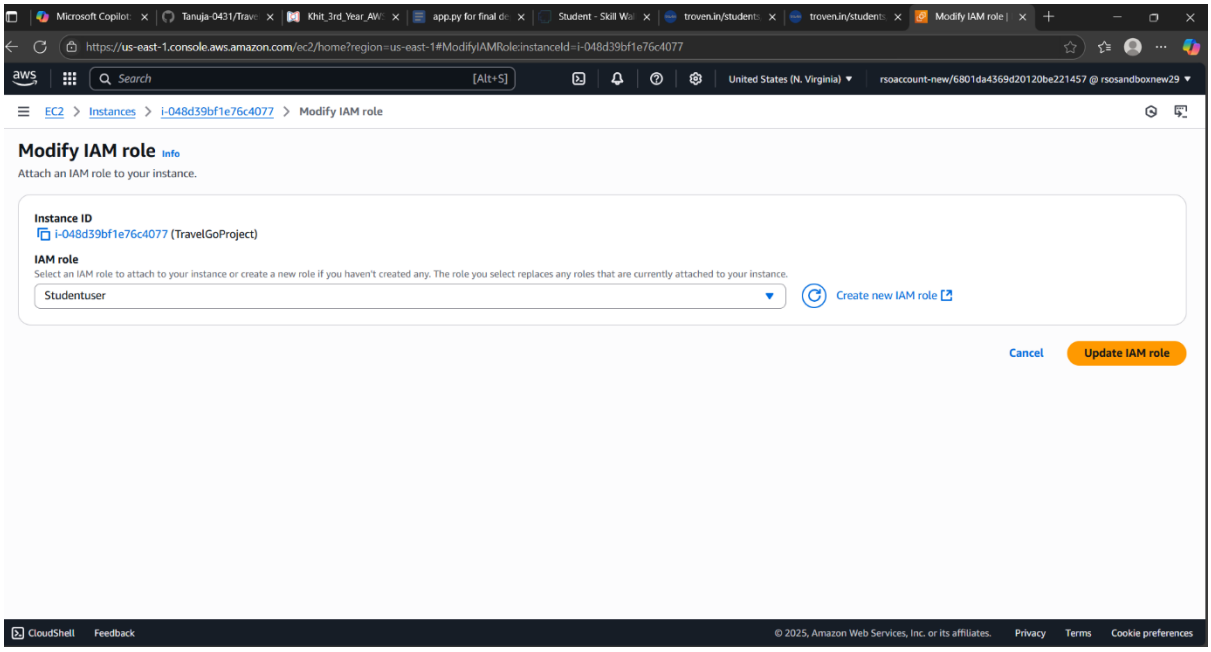
To ensure the server remains both accessible and manageable, the security group must allow specific types of network traffic. HTTP access (port 80) enables users to open the website in their browser, while SSH access (port 22) allows secure remote login for deployment and server maintenance. These rules, configured within AWS security groups, help maintain a balance between usability and system security.

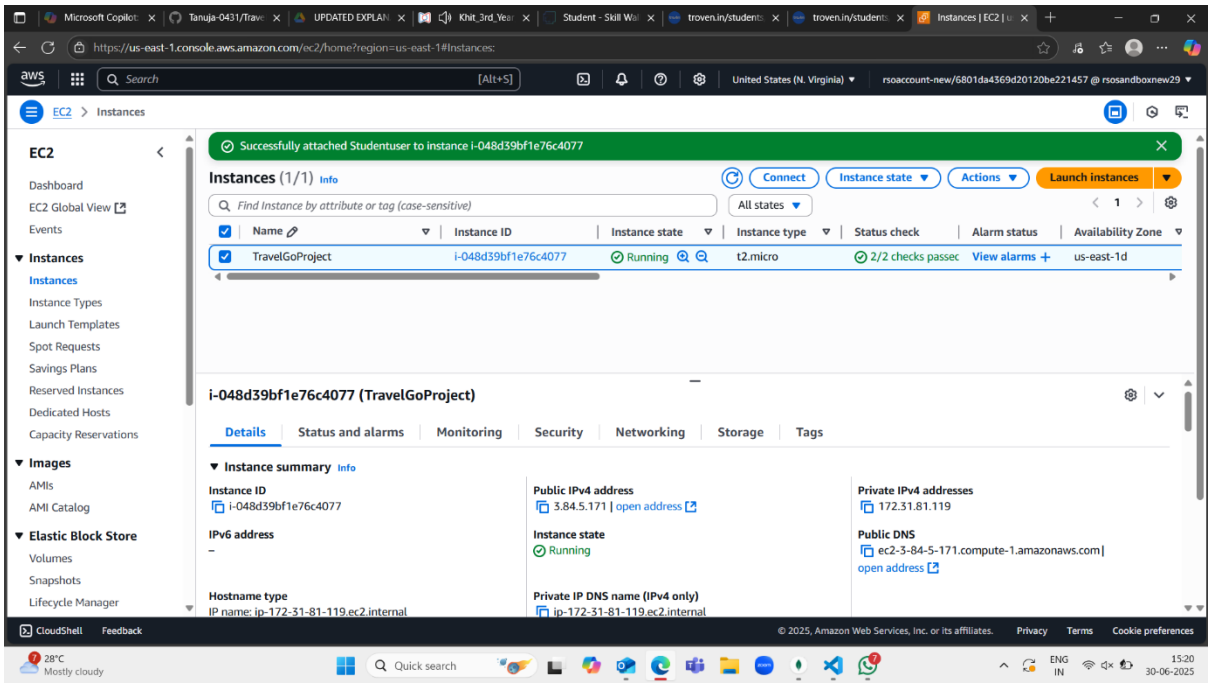




- To connect to EC2 using EC2 Instance Connect, start by ensuring that an IAM role is attached to your EC2 instance. You can do this by selecting your instance, clicking on Actions, then

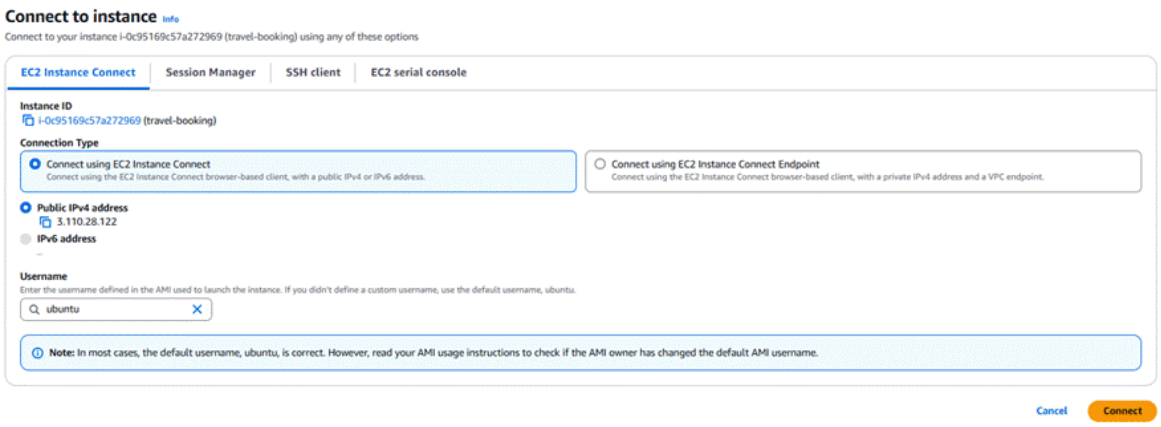
navigating to Security and selecting Modify IAM Role to attach the appropriate role. After the IAM role is connected, navigate to the EC2 section in the AWS Management Console. Select the EC2 instance you wish to connect to. At the top of the EC2 Dashboard, click the Connect button. From the connection 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.

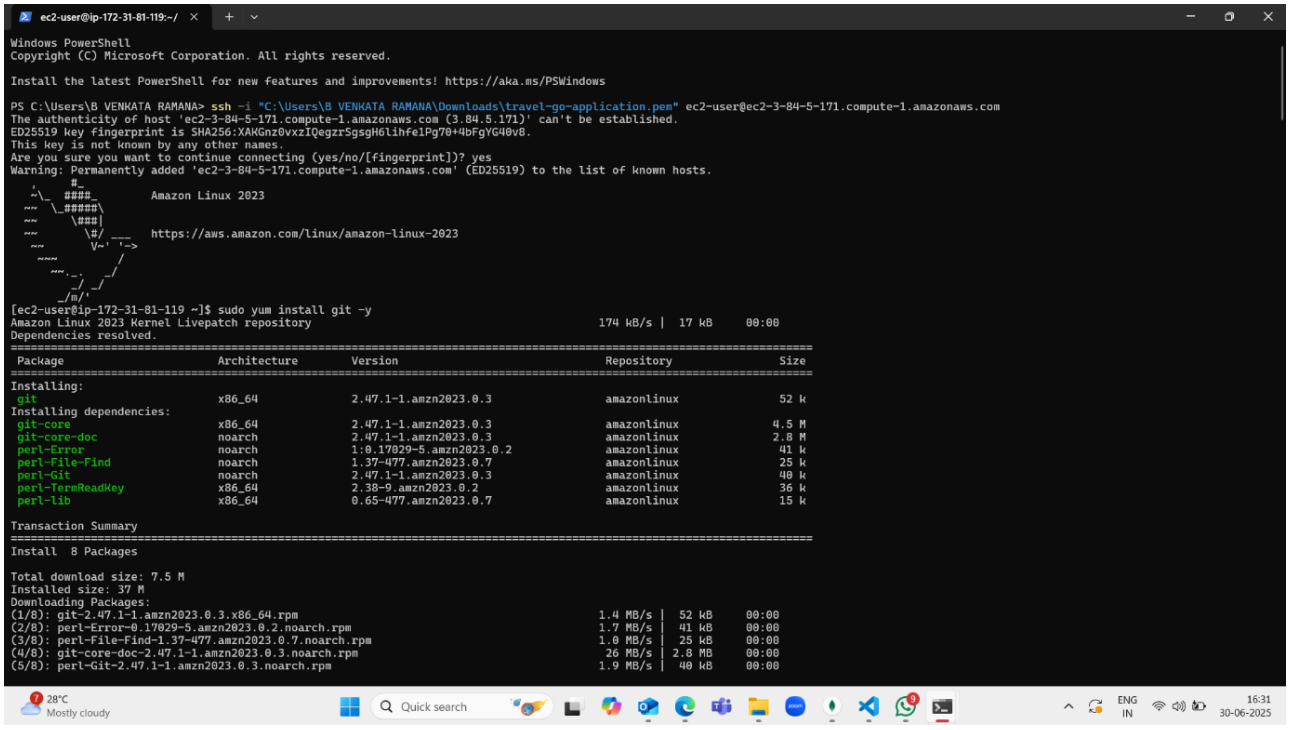




- Now connect the EC2 with the files

SMARTBRIDGE
Let's Bridge the Gap





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/Tanuja-0431/TravelGo.git"
- **This will download your project to the EC2 instance.**
- To navigate to the project directory, run the following command:

  cd <your-folder> i.e.; Travelgo

- 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:us-east-1:586794467071:Travelgo:90e78453-f4b5-485a-85bd-dc379e38a683**
- 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
```

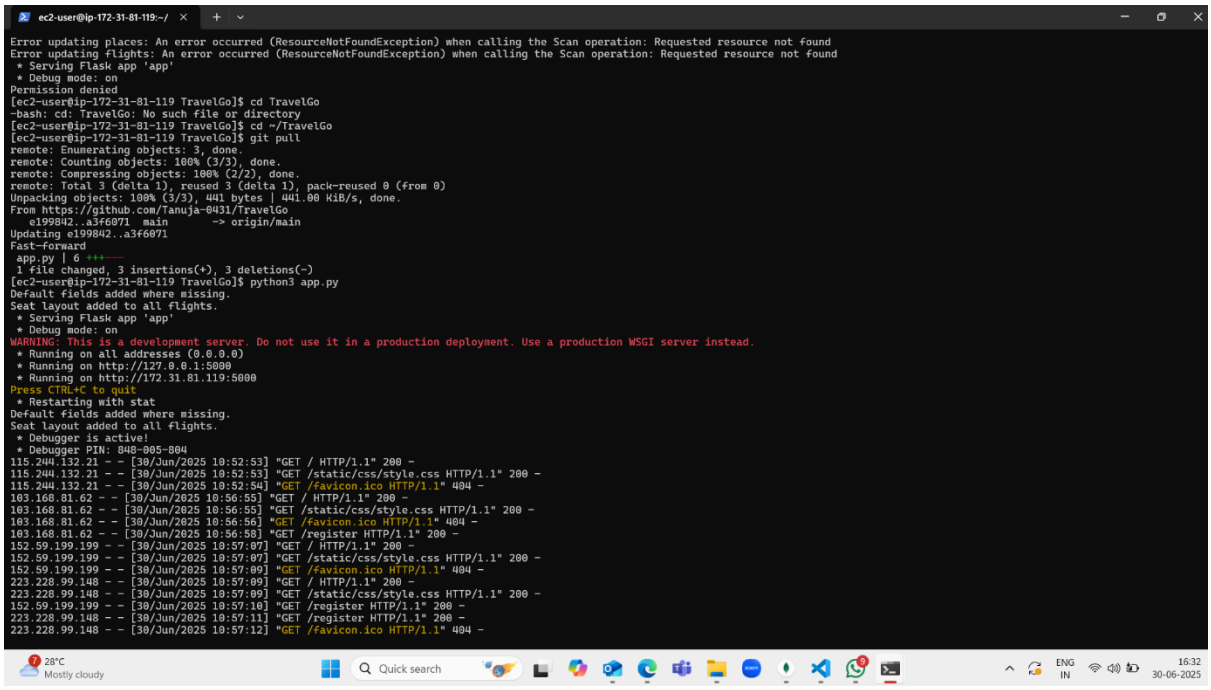
- Launch the Flask app:

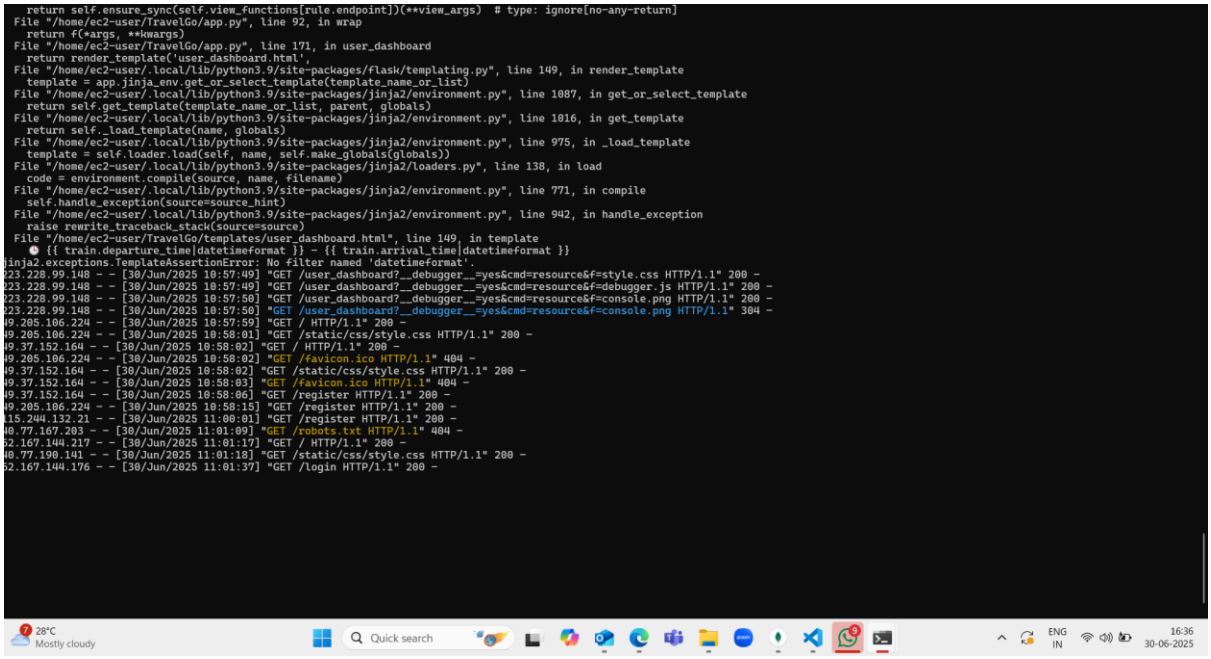
```
sudo -E venv/bin/python3 app.py
```

Verify the Flask app is running:

http://your-ec2-public-ip i.e.; http://3.84.5.171:5000/

- Run the Flask app on the EC2 instance





- Access the website through:

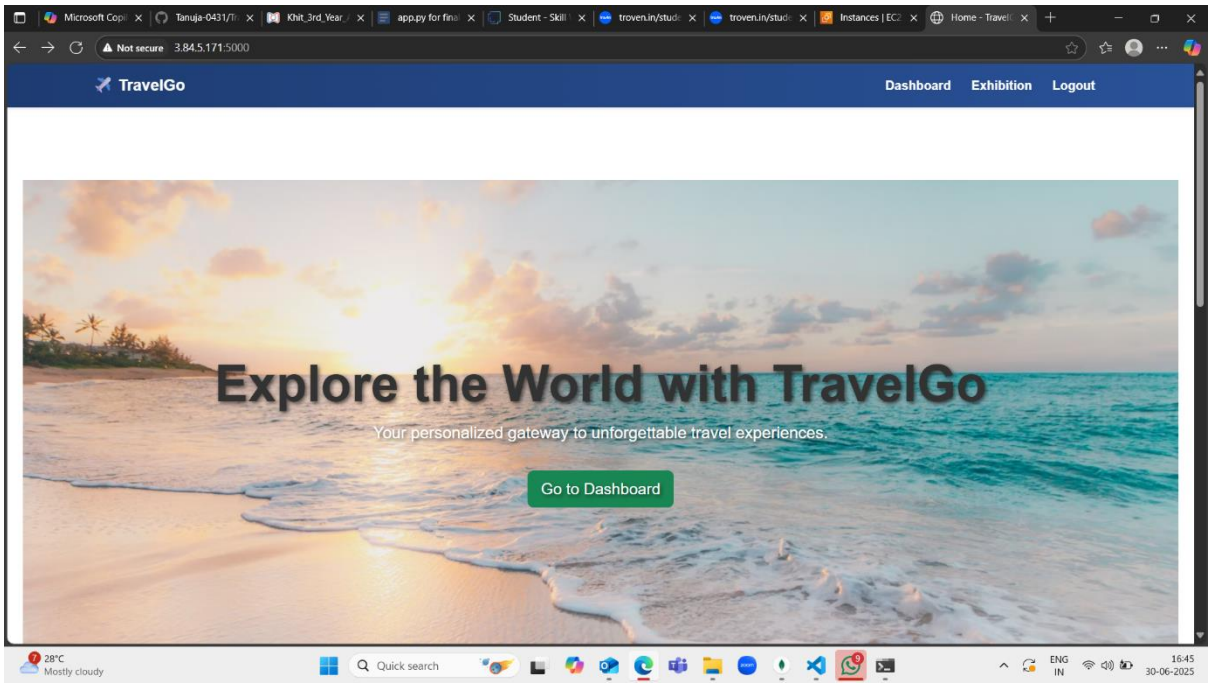
PublicIPs: http://3.84.5.171:5000/

Milestone 8 : Testing and Deployment

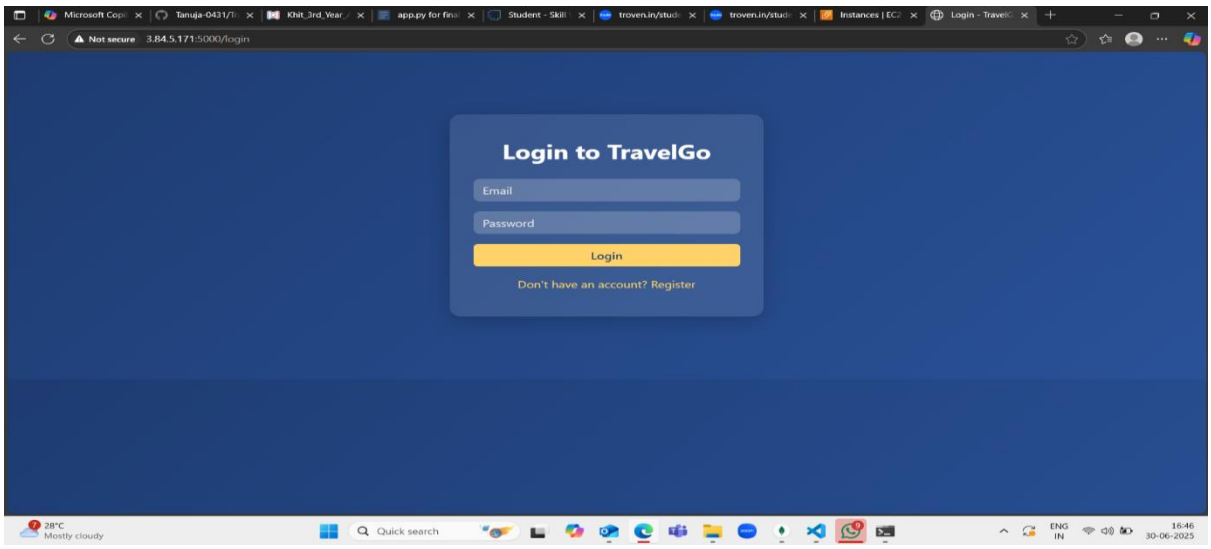
Testing and deployment involve verifying that your application works as expected before making it publicly accessible. Start by testing locally or on a staging environment to catch bugs and ensure functionality. Once tested, deploy the application to an EC2 instance, configure necessary services, and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

Functional Testing to verify the Project

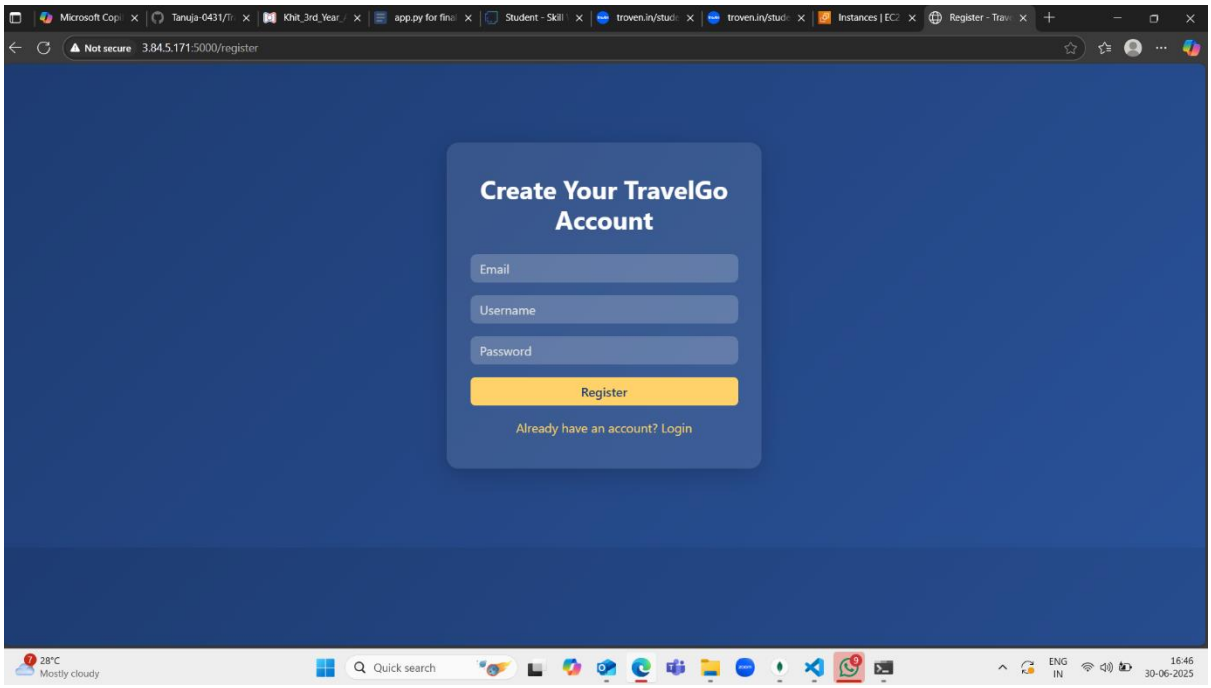
Home Page:



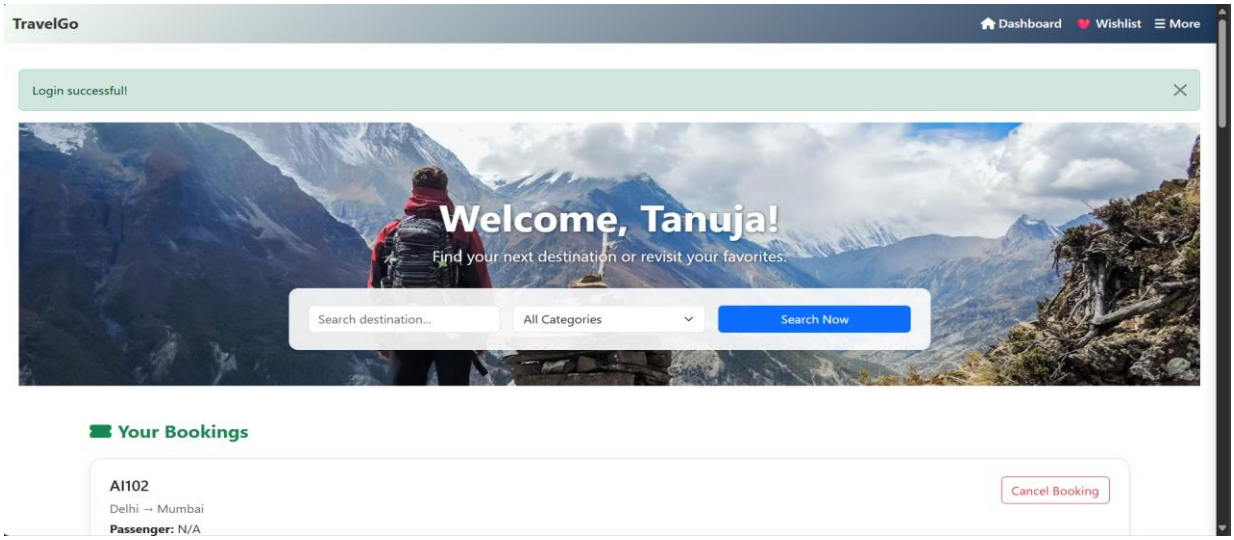
Login Page:

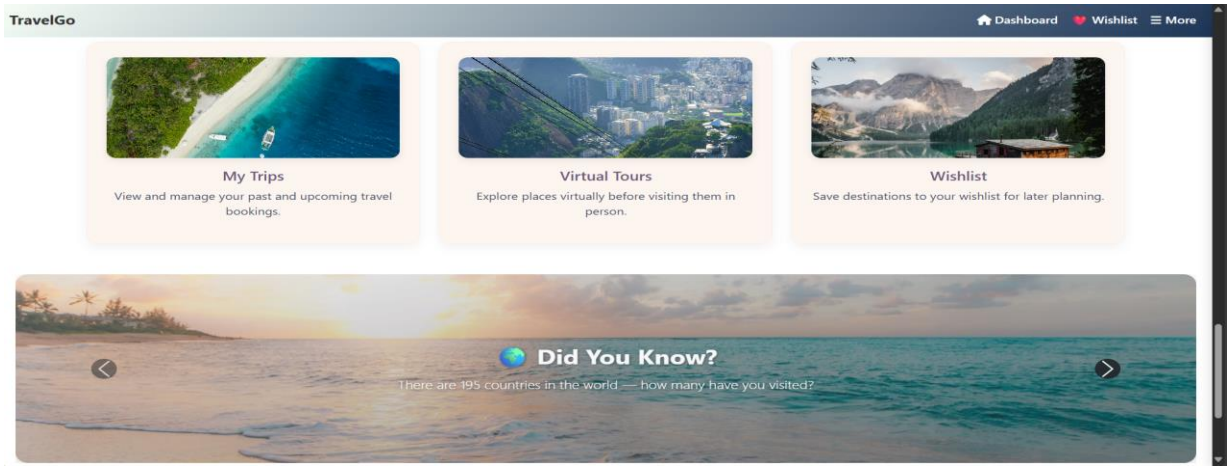


Register Page:

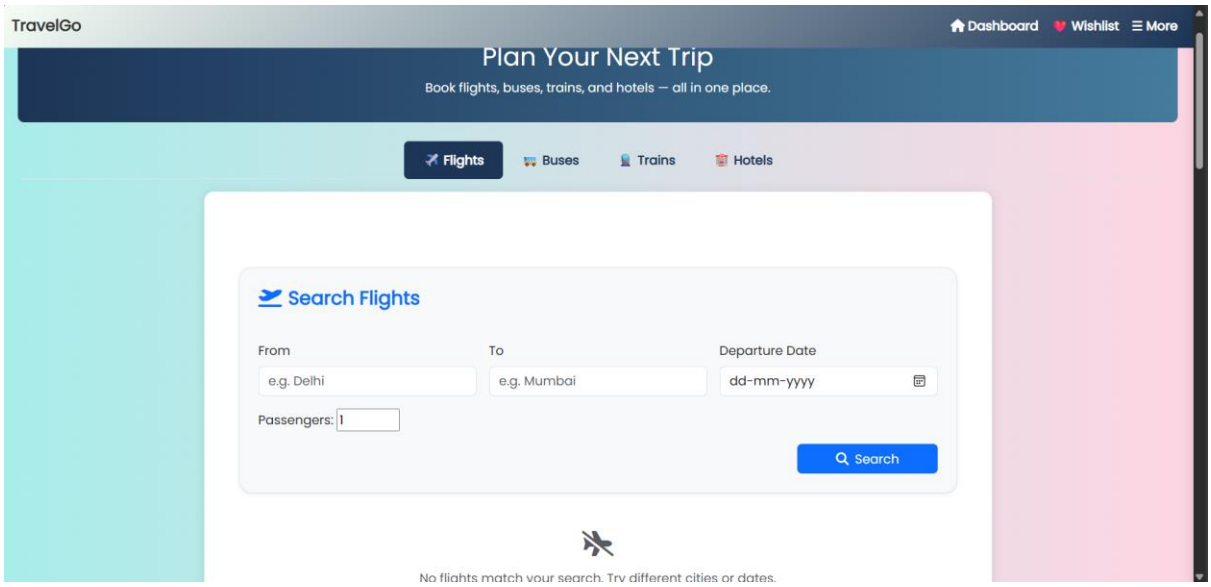


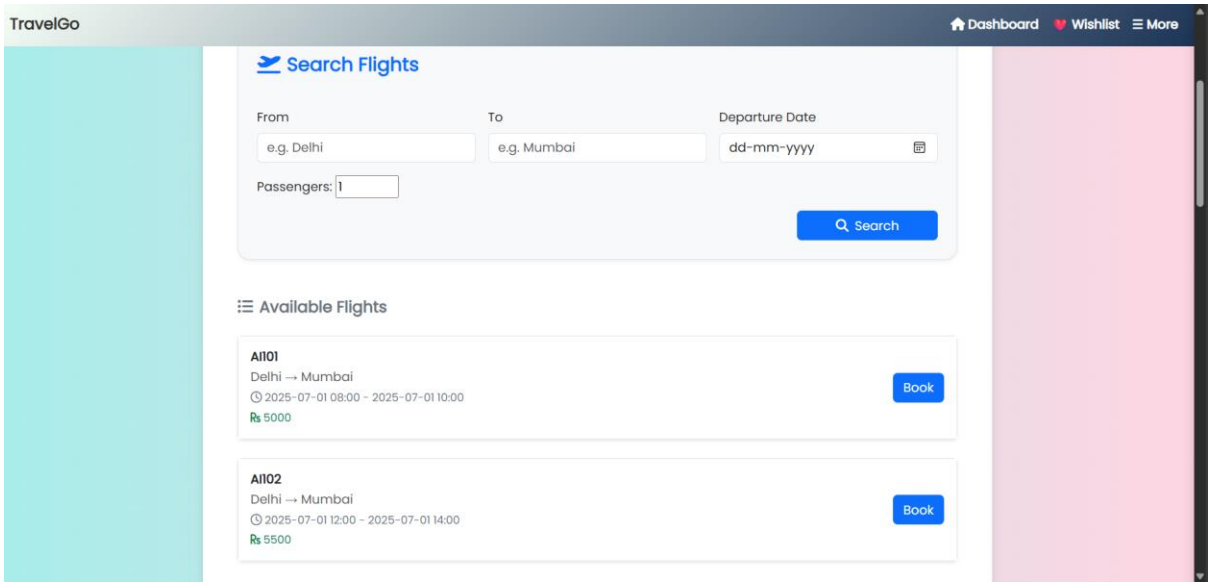
Dashboard page:

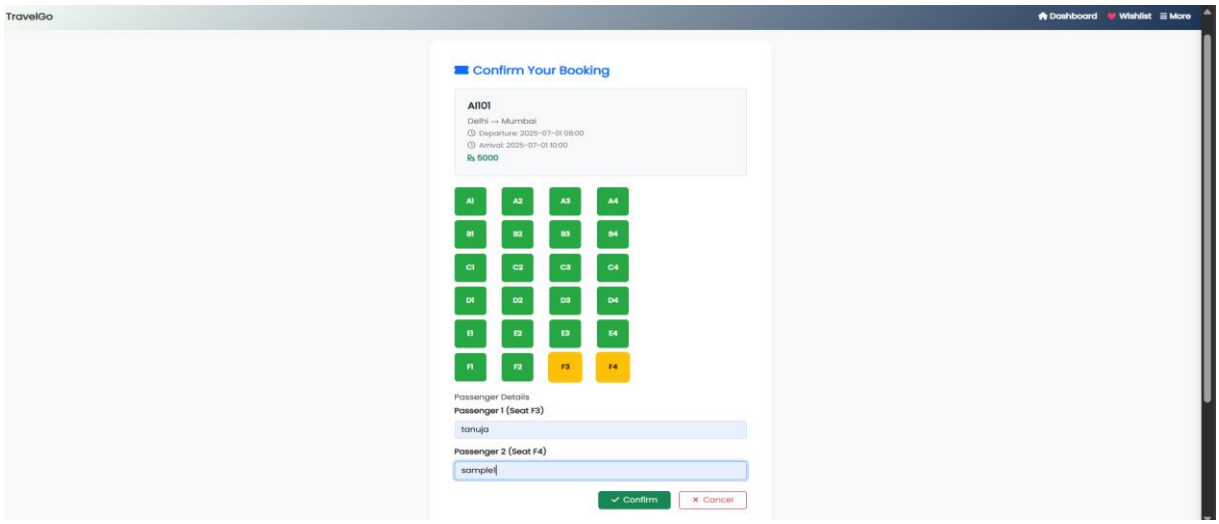




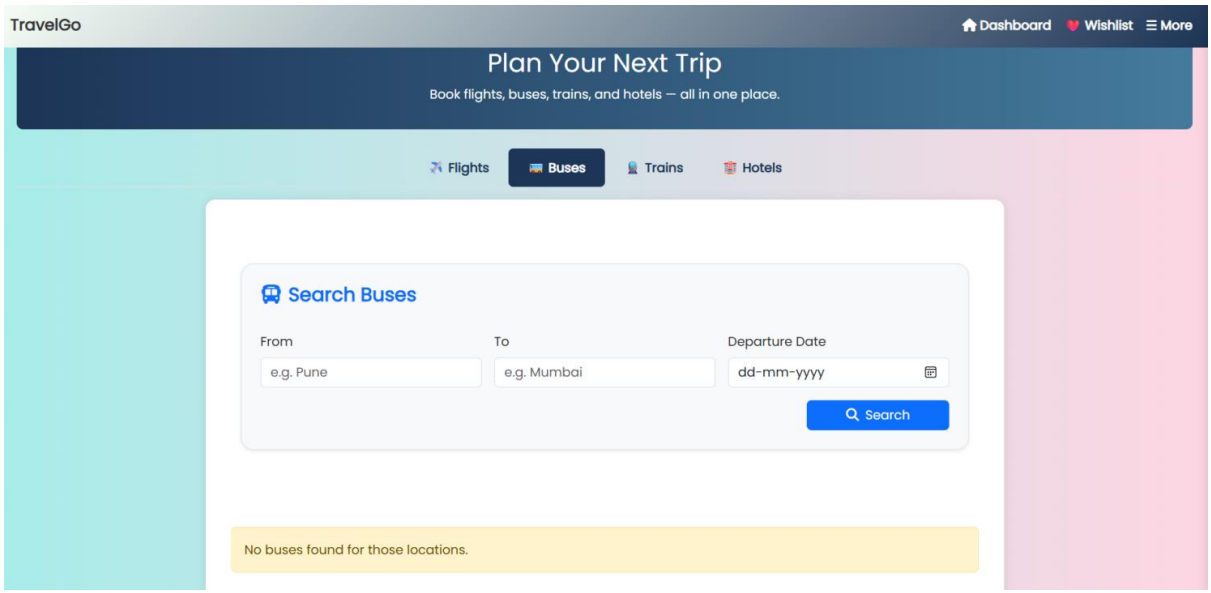
Bookings page:

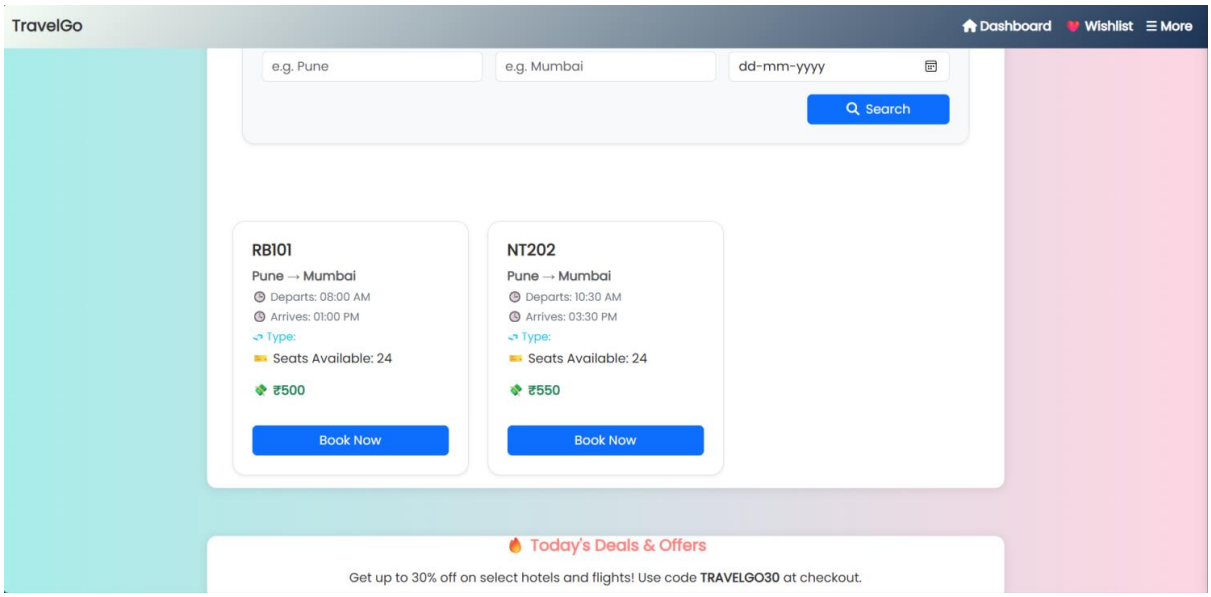


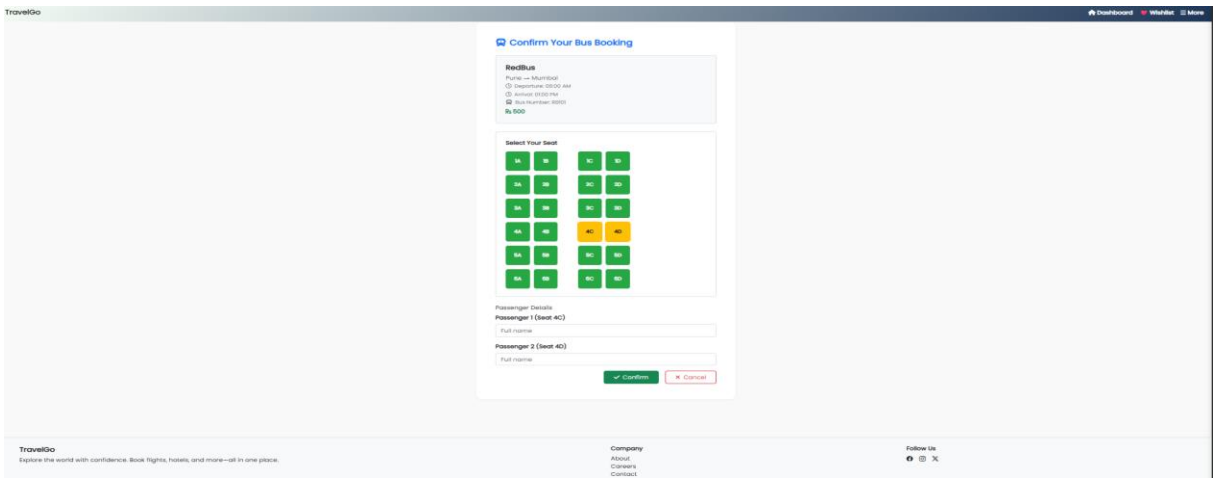




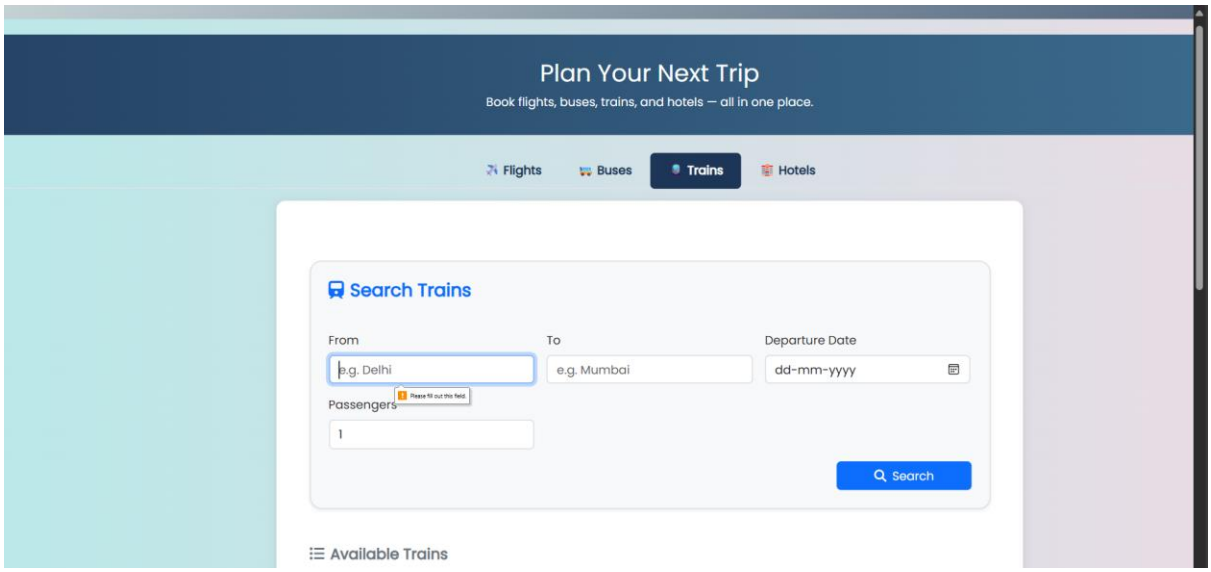
Bus bookings page:

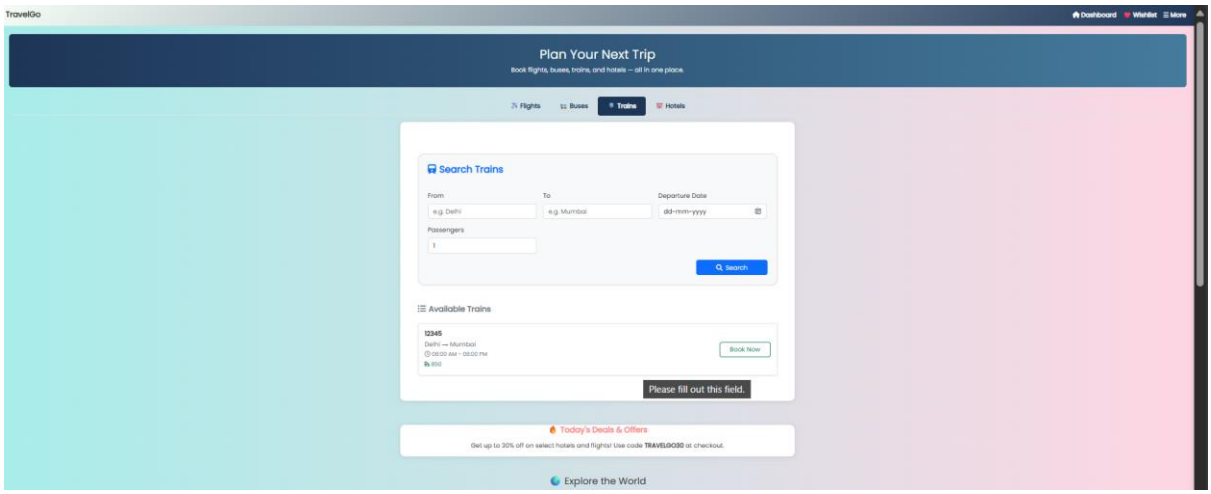


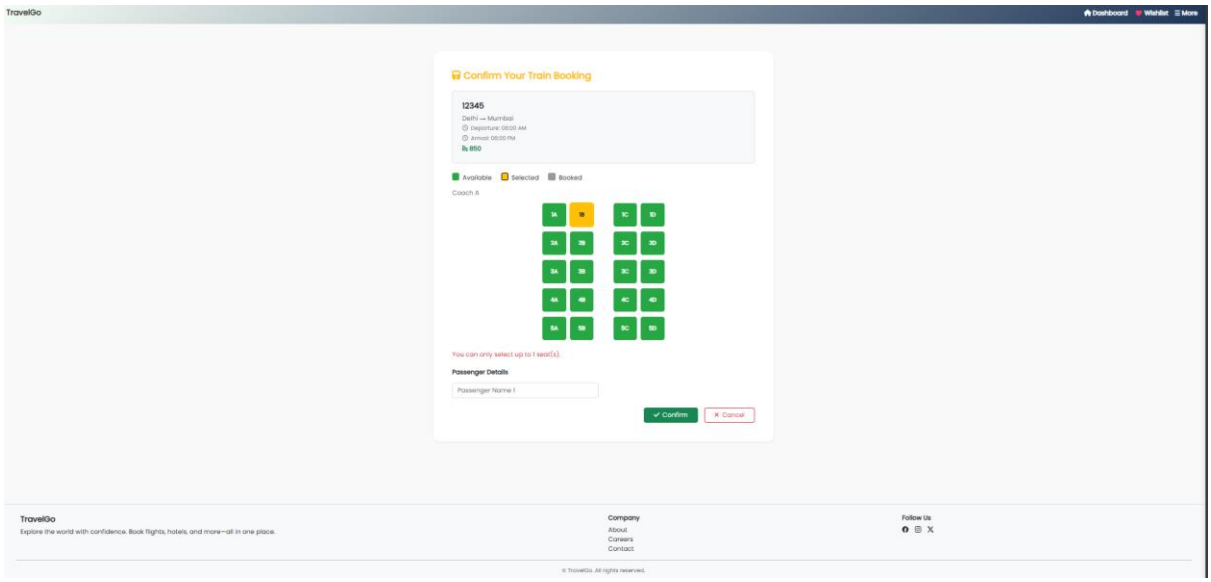




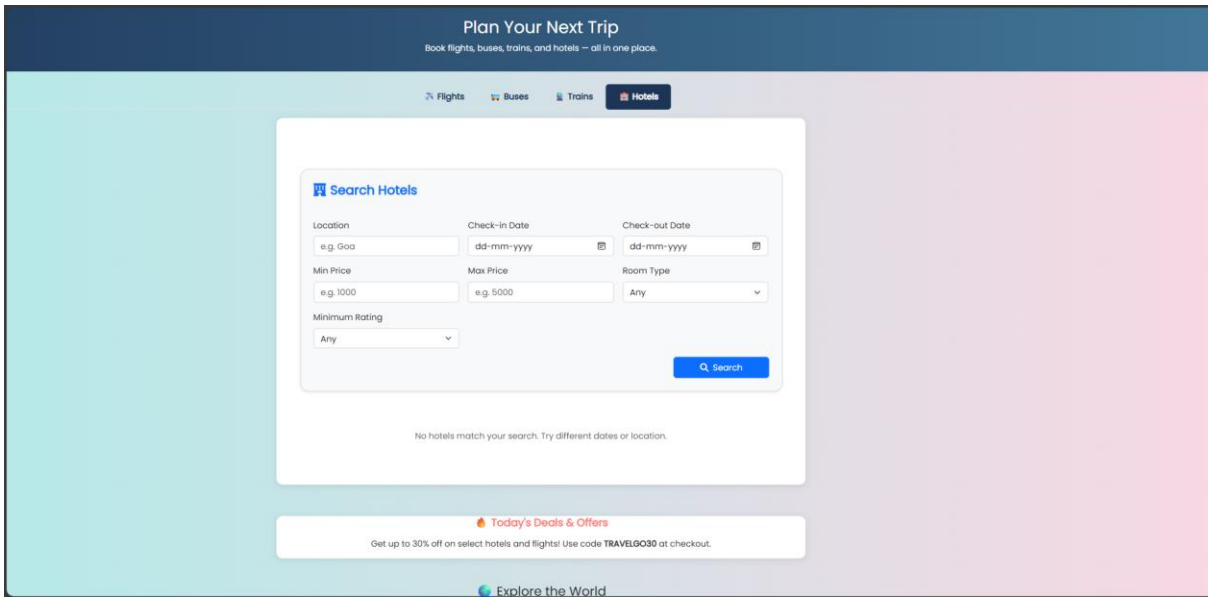
Train bookings page:

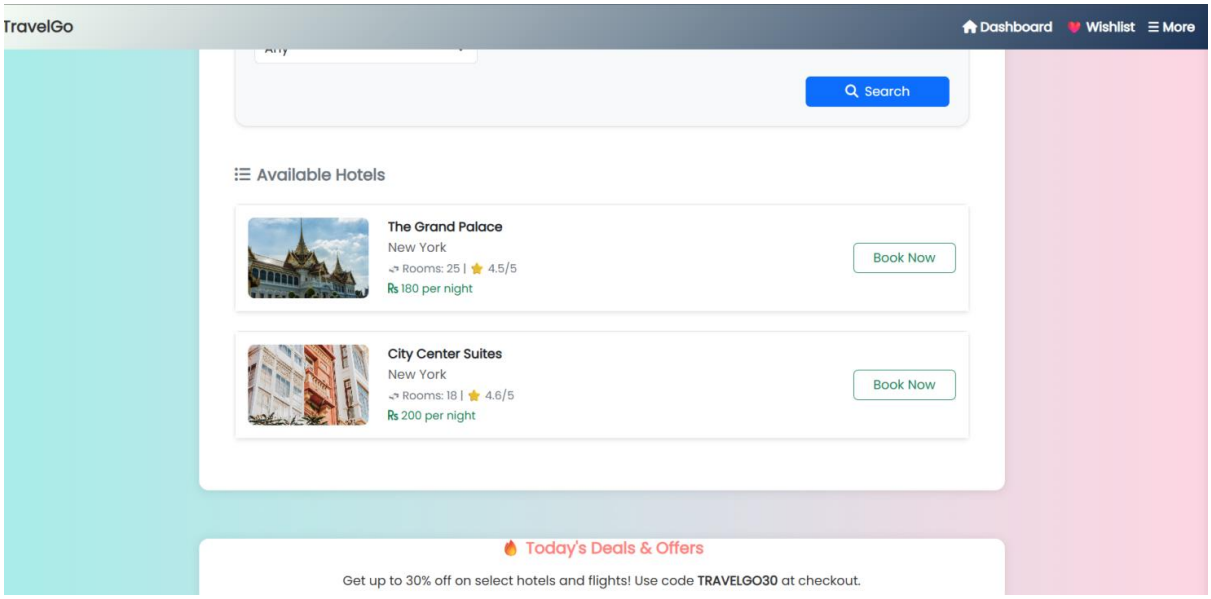


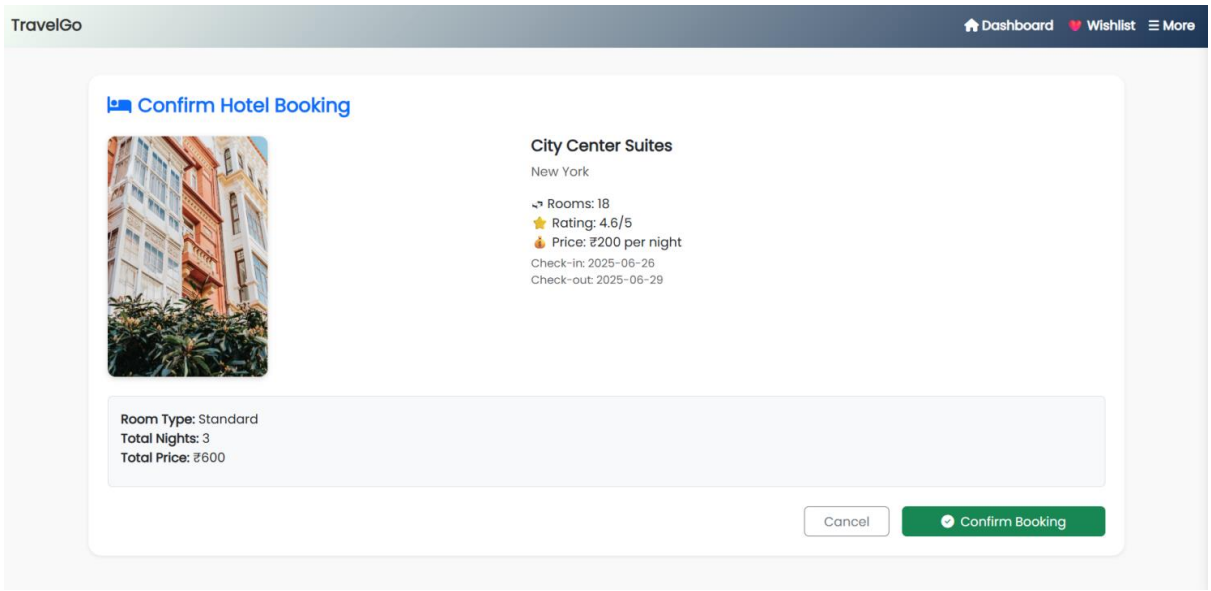




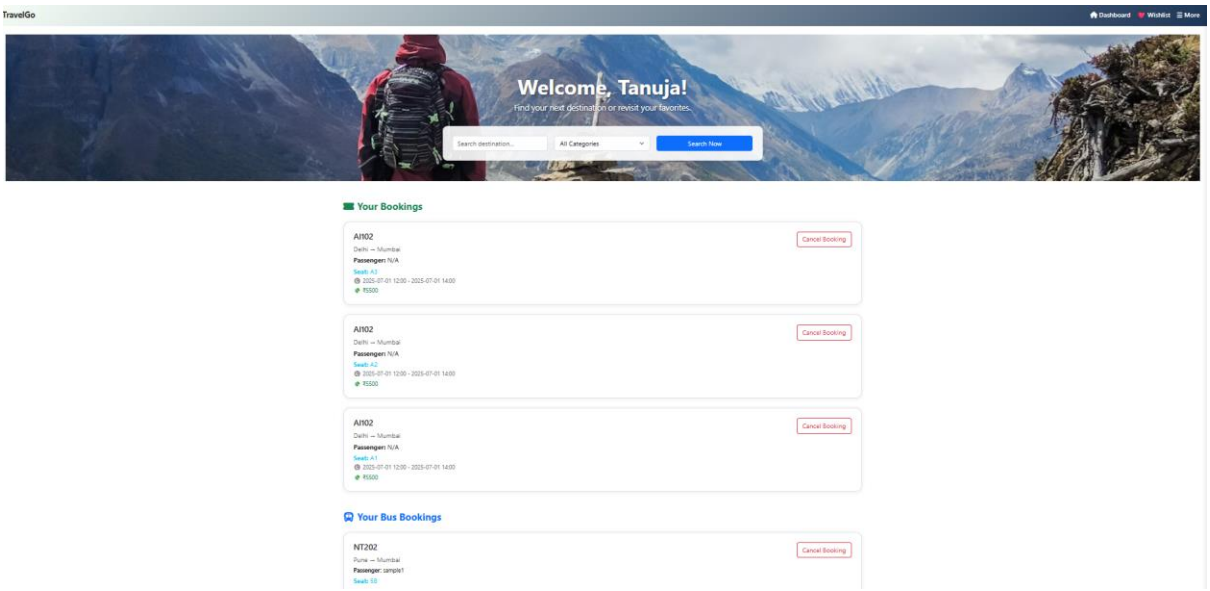
Hotels booking page:

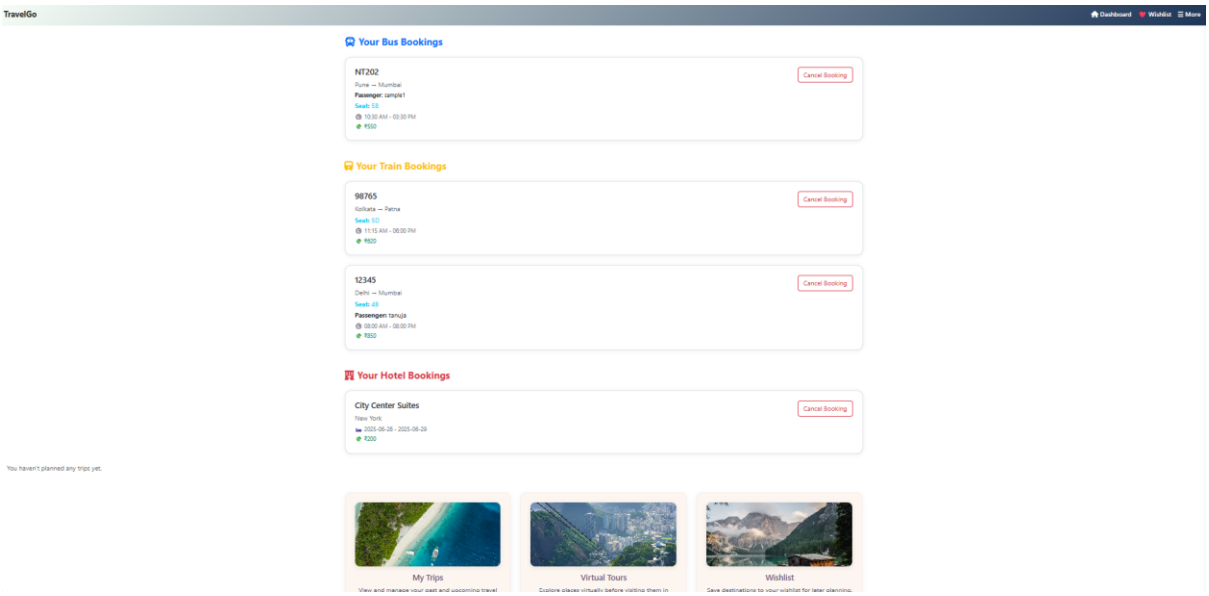






Recent bookings:





Conclusion

The TravelGo Website has been successfully developed and deployed using a scalable and cloud-native architecture. Leveraging AWS services such as EC2 for hosting, DynamoDB for real-time data management, and SNS for instant booking and cancellation notifications, the platform provides a seamless travel booking experience for users. TravelGo enables registered users to search and book buses, trains, flights, and hotels in a centralized, intuitive interface, eliminating the complexities of navigating multiple travel services.

The cloud infrastructure ensures high availability and smooth performance even during peak usage, while the Flask backend ensures efficient handling of user authentication, dynamic booking flows, and data transactions. Real-time notification integration via AWS SNS allows users to receive booking confirmations and cancellations immediately via email, improving communication and user engagement.

In summary, the TravelGo Website offers a modern, reliable, and user-friendly solution for managing travel and accommodation needs. It highlights the potential of cloud-based platforms in building unified travel systems, simplifying operations, and enhancing the overall user experience.