

TRAVELGO – A Cloud-Based Travel Booking System

Project Description:

Travel Go is a unified travel booking solution built on Flask and hosted on AWS EC2. It centralizes buses, trains, flights, and hotel reservations into a single, user-friendly interface. User data and bookings are reliably stored in DynamoDB, and users receive real-time email notifications via AWS SNS whenever a booking is confirmed or cancelled. With dynamic seat selection, hotel filtering, responsive design, and interactive dashboards, Travel Go delivers a seamless travel experience end-to-end.

Scenario 1: Hassle-Free Multi-Mode Booking:

User: A professional planning a weekend trip from Hyderabad to Bangalore.

Steps:

1. Logs in and selects transport type (e.g., flight).
2. Searches options and views timings, prices, available seats.
3. Chooses preferred option (e.g., morning flight + seat 6A).
4. Proceeds to payment and sees booking confirmation instantly.

Scenario 2: Real-Time Booking Confirmation via AWS SNS:

User: A student booking a hotel room in Chennai for a research trip.

Steps:

1. Selects hotel, room type (budget/luxury), dates.
2. Completes booking; data is stored in DynamoDB.
3. Flask backend triggers AWS SNS.
4. Student receives email with booking summary.
5. Optional: Notification sent to supervisor and hotel admin.

Scenario 3: Personal Dashboard for Travel History:

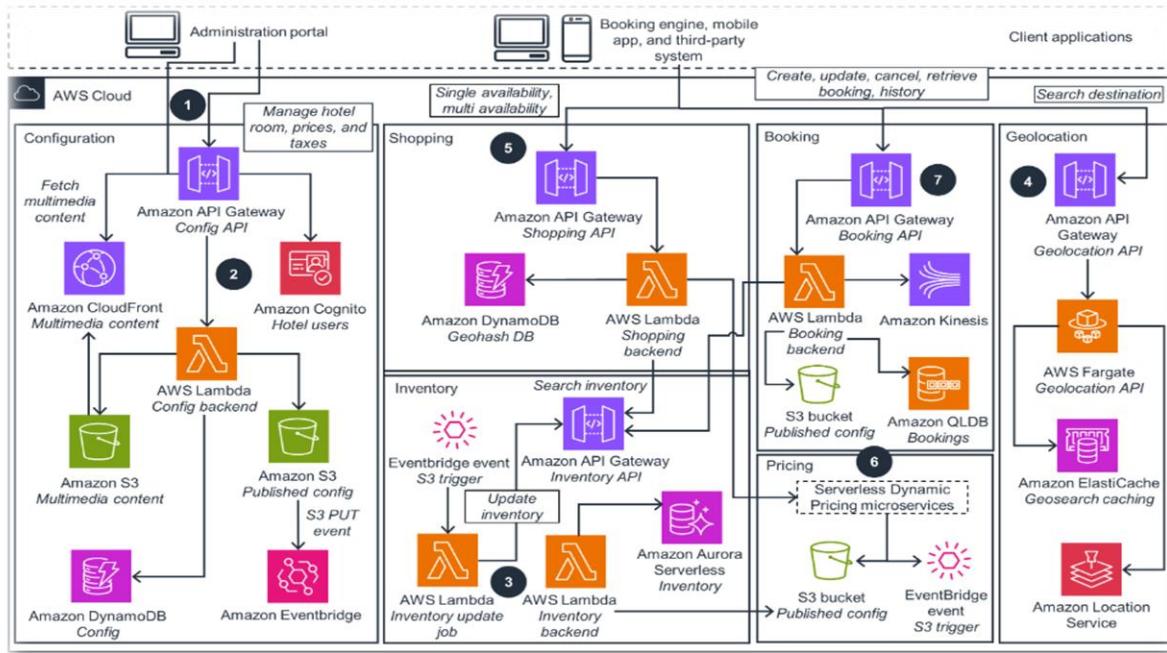
User: A daily commuter who frequently books buses and trains for work travel.

Steps:

1. Logs in and visits Travel Go dashboard.

- Flask fetches trips from DynamoDB, categorizes by transportation mode.

ARCHITECTURE



Travel Go – Pre-requisites:

1. AWS Account Setup:

- Create a standard AWS account to access services like EC2, DynamoDB, and SNS.
- Ensure permissions are configured for development usage.

2. Understanding IAM (Identity and Access Management)

- Learn IAM basics: users, groups, roles, and policies for permission control.
 - Apply the principle of least privilege to secure access.
- geeksforgeeks.org + 5datacamp.com + 5datacamp.com + 5youtube.com + 12docs.aws.amazon.com + 12abiabi0707.medium.com + 12

3. Amazon EC2 Basics

- Familiarize with EC2 instance creation, security groups, and SSH access.
- Understand VPC concepts and how to deploy applications within a secure network.

4. DynamoDB Fundamentals

- Grasp core NoSQL concepts: tables, items, primary keys.

5. AWS SNS (Simple Notification Service)

Learn how to publish messages and configure email notifications via SNS topics.

Project Work Flow:

1. AWS Account Setup and Login:

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

Activity 1.2: Log in to the AWS Management Console.

2. DynamoDB Database Creation and Setup

Activity 2.1: Create a DynamoDB Table.

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

3. SNS Notification Setup

Activity 3.1: Create SNS topics for book request notifications.

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

4. Backend Development and Application Setup

Activity 4.1: Develop the Backend Using Flask.

Activity 4.2: Integrate AWS Services Using boto3.

5. IAM Role Setup

Activity 5.1: Create IAM Role

Activity 5.2: Attach Policies

6. EC2 Instance Setup

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

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

7. Deployment on EC2

Activity 7.1: Upload Flask Files

Activity 7.2: Run the Flask App

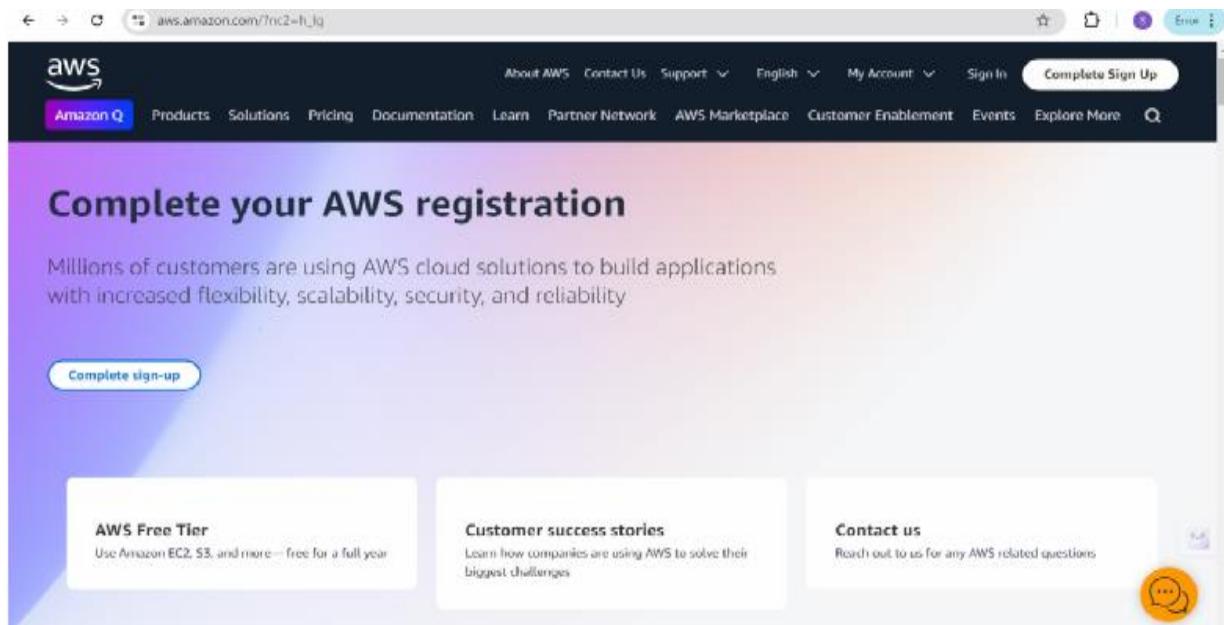
8. Testing and Deployment

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

Milestone 1: AWS Account Setup and Login

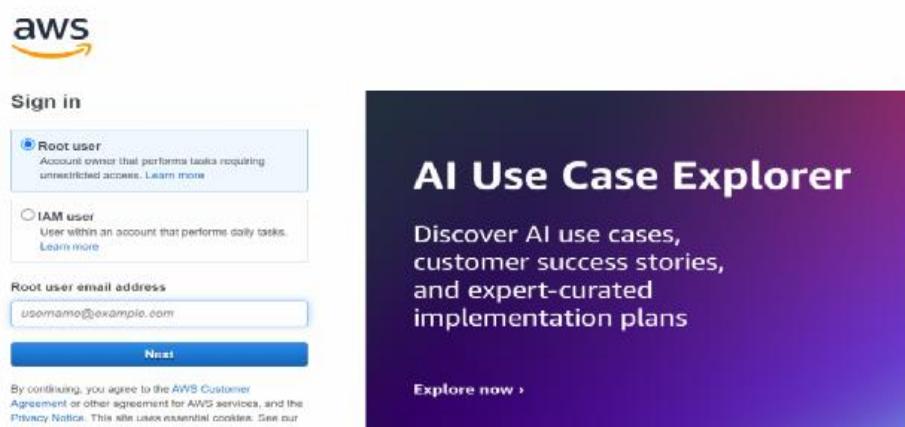
➤ Activity 1.1: Set up an AWS account if not already done.

- Sign up for an AWS account and configure billing settings.



➤ Activity 1.2: Log in to the AWS Management Console

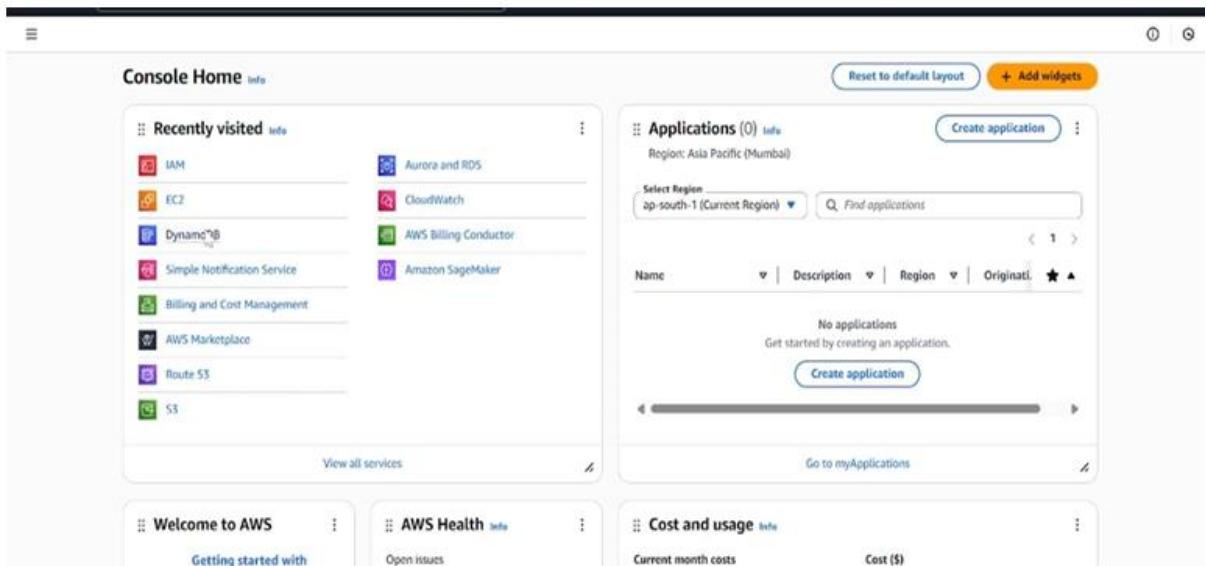
- After setting up your account, log in to the AWS Management Console.



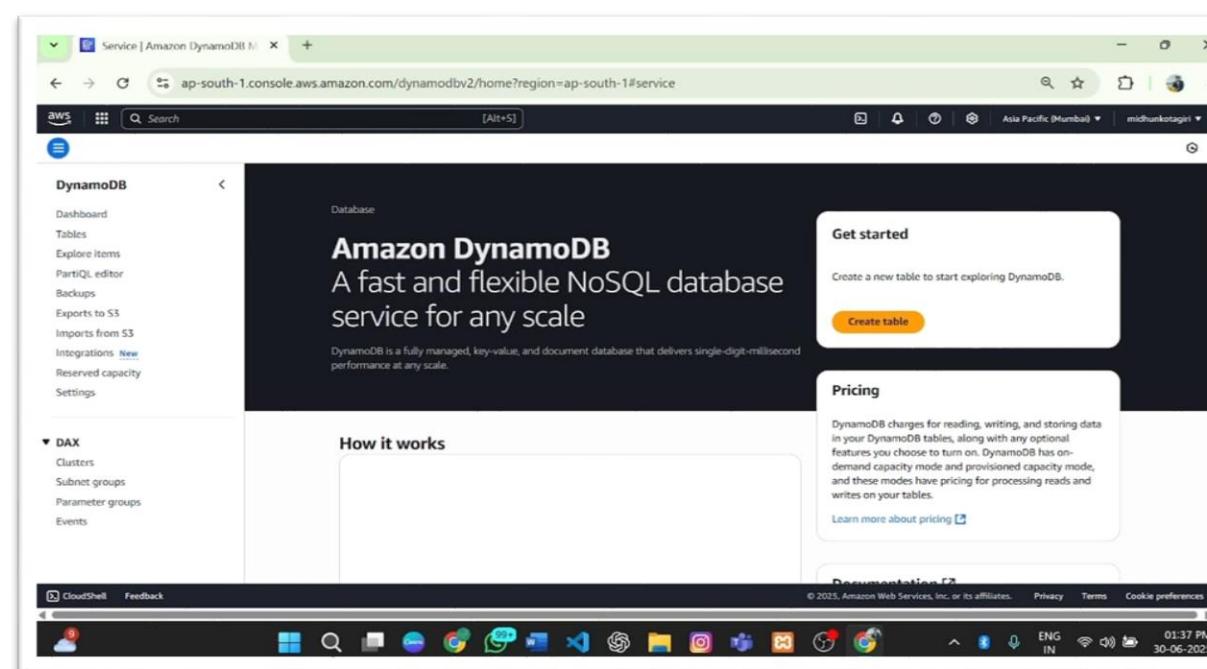
Milestone 2: DynamoDB Database Creation and Setup

➤ Activity 2.1: Navigate to the DynamoDB

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



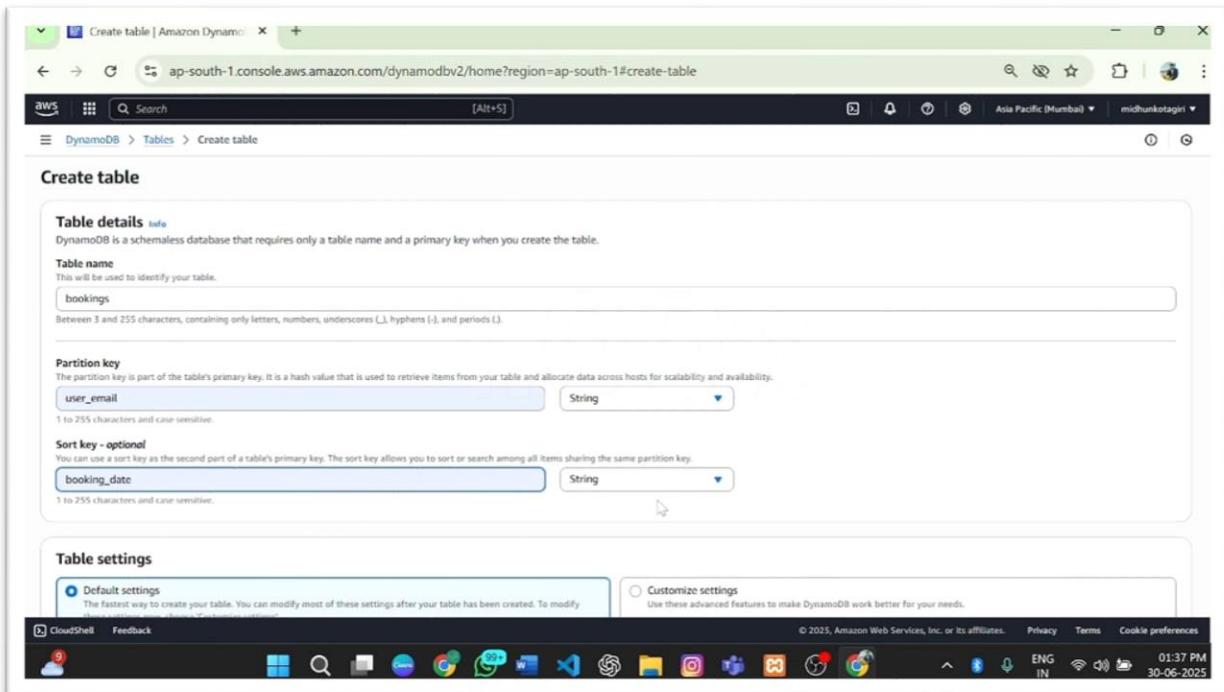
The screenshot shows the AWS Console Home page. On the left, there's a sidebar with 'Recently visited' services: IAM, Aurora and RDS, EC2, CloudWatch, AWS Billing Conductor, Amazon SageMaker, Simple Notification Service, Billing and Cost Management, AWS Marketplace, Route 53, and S3. Below this is a 'View all services' link. To the right, there's a 'Applications' section with a table header for Name, Description, Region, and Originati. It says 'No applications' and has a 'Create application' button. At the bottom of the main area are links for 'Welcome to AWS', 'AWS Health', and 'Cost and usage'. The status bar at the bottom shows the URL as 'ap-south-1.console.aws.amazon.com/dynamodbv2/home?region=ap-south-1#service', the region as 'Asia Pacific (Mumbai)', and the user as 'midhunkotagi@...'.



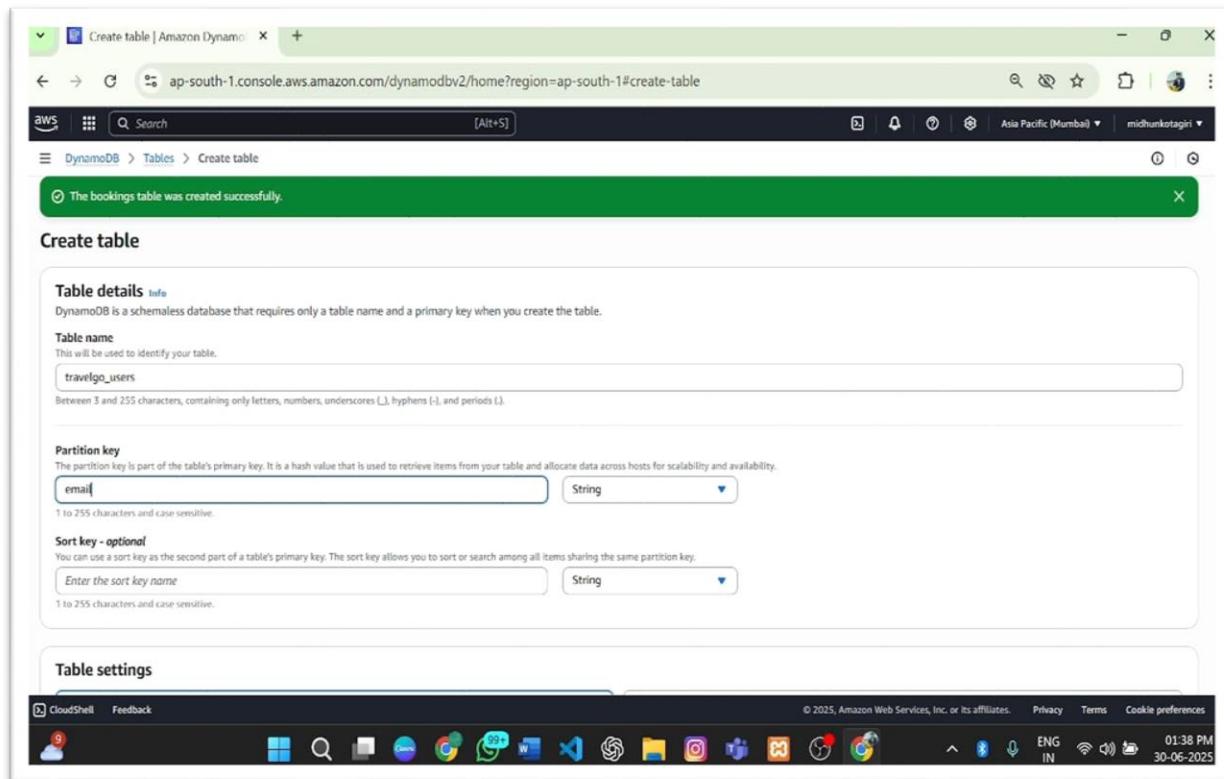
The screenshot shows the Amazon DynamoDB service homepage. The left sidebar has navigation links: Dashboard, Tables, Explore items, PartiQL editor, Backups, Exports to S3, Imports from S3, Integrations, Reserved capacity, and Settings. A 'DAX' section is also present. The main content area features the heading 'Amazon DynamoDB: A fast and flexible NoSQL database service for any scale'. It includes a 'Get started' button and a 'Pricing' section. The bottom of the page includes a 'Documentation' link, copyright information (© 2025, Amazon Web Services, Inc. or its affiliates.), and a footer with various icons and the date '30-06-2025'.

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

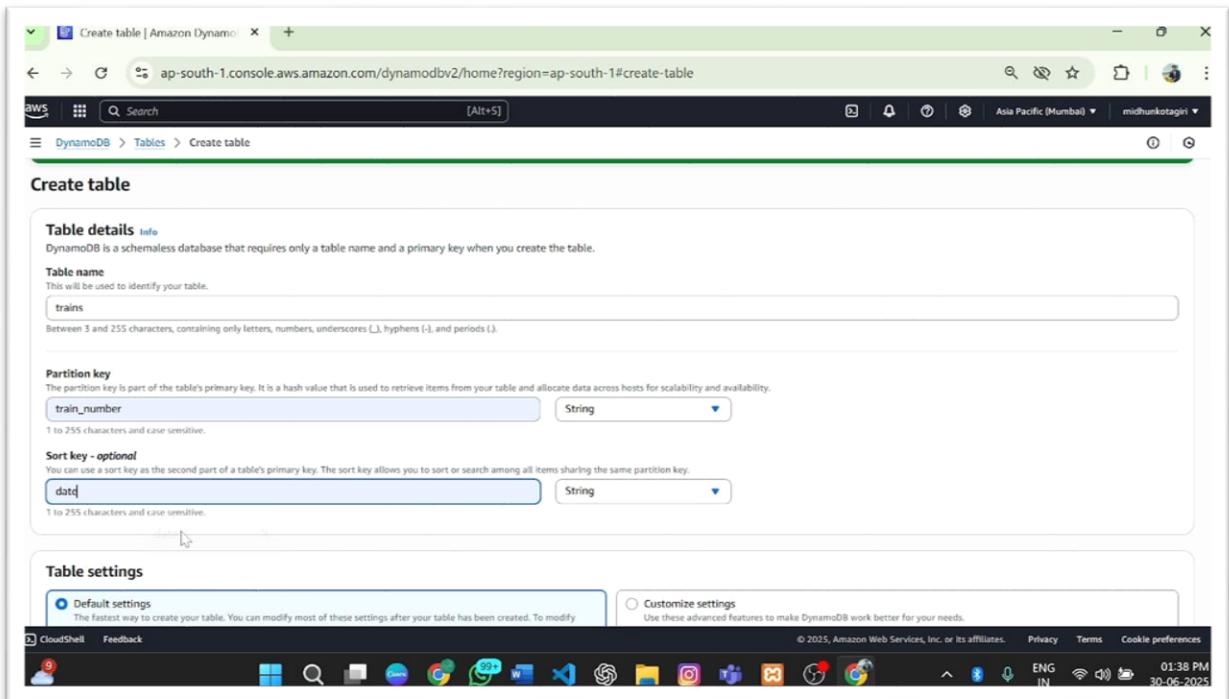
- Create Users table with partition key "Email" with type String and click on create tables.



- Follow the same steps to create a travel go users table with Email as the primary key.



- Follow the same steps to create a trains table with train number as the partition key.



Tags

Tags are pairs of keys and optional values, that you can assign to AWS resources. You can use tags to control access to your resources or track your AWS spending.

No tags are associated with the resource.

[Add new tag](#)

You can add 50 more tags.

[Cancel](#) [Create table](#)

Milestone 3: SNS Notification Setup

- **Activity 3.1: Create SNS topics for sending email notifications to users and library staff.**
 - In the AWS Console, search for SNS and navigate to the SNS Dashboard.

The screenshot shows the AWS DynamoDB Services page on the left and a table editor interface on the right.

DynamoDB Services:

- Simple Notification Service (SNS)
- Route 53 Resolver
- Route 53

Features:

- Events
 - ElastiCache feature
- SMS
 - AWS End User Messaging feature
- Hosted zones
 - Route 53 feature

Were these results helpful?

Yes No

Table Editor (Right):

Region	Deletion protection	Favorite	Read capacity
US East (N. Virginia)	Off	Star	On-demand
US West (Oregon)	Off	Star	On-demand
EU (Ireland)	Off	Star	On-demand
EU (Paris)	Off	Star	On-demand
AP South (Mumbai)	Off	Star	On-demand
AP East (Chennai)	Off	Star	On-demand
AP North (Tokyo)	Off	Star	On-demand
AP Southeast (Singapore)	Off	Star	On-demand
AP Central (Sydney)	Off	Star	On-demand
MEA (Bahrain)	Off	Star	On-demand
CA (Montreal)	Off	Star	On-demand
SA (Sao Paulo)	Off	Star	On-demand

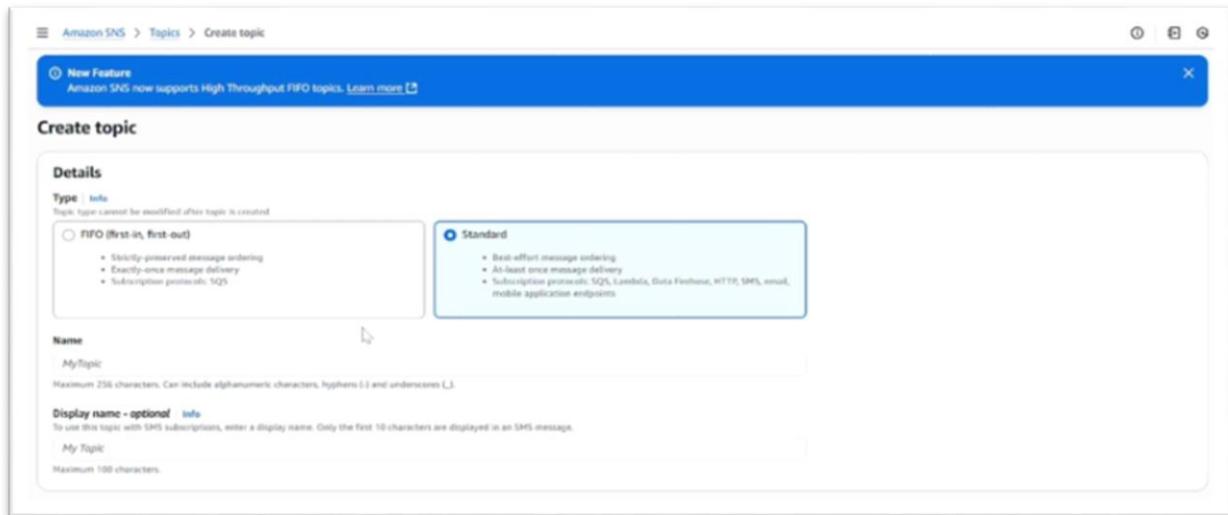
The screenshot shows the Amazon SNS console. On the left, a sidebar menu includes 'Amazon SNS' (selected), 'Dashboard', 'Topics', 'Subscriptions', 'Mobile', 'Push notifications', and 'Text messaging (SMS)'. The main content area has a dark header 'Application Integration' and a large central heading: 'Amazon Simple Notification Service' followed by 'Pub/sub messaging for microservices and serverless applications.' Below this, a paragraph describes Amazon SNS as a highly available, durable, secure, fully managed pub/sub messaging service. In the top right corner, there's a blue circular icon with an 'i' and the text 'New Feature: Amazon SNS now supports in-place message archiving and replay for FIFO topics. [Learn more](#)'. To the right of the main content, a white sidebar box contains 'Create topic' and a 'Topic name' input field with 'MyTopic'. A yellow 'Next step' button is below it, and a link 'Start with an overview' is at the bottom. At the very bottom right, a 'Pricing' link is visible.

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

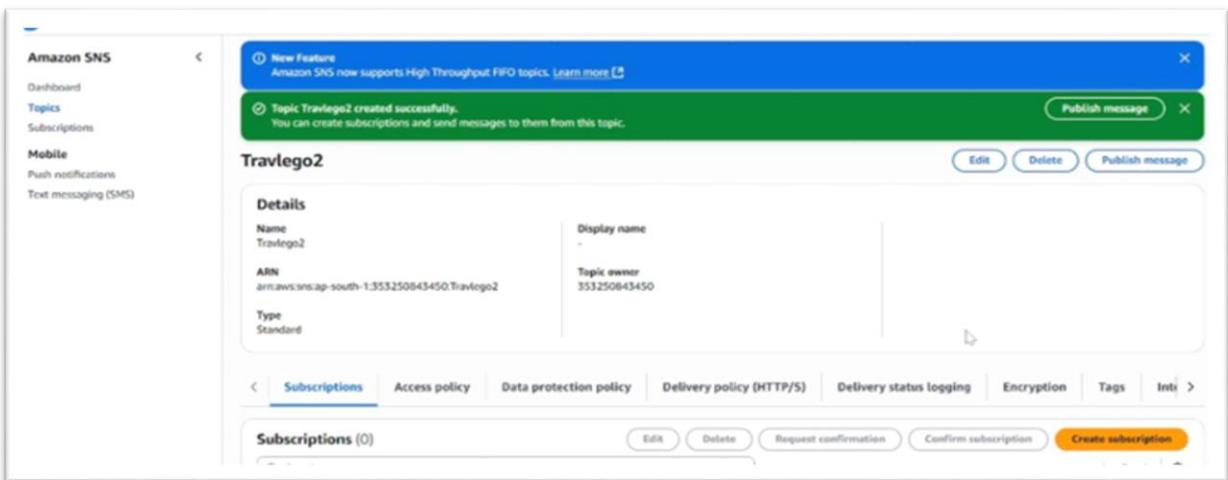
The screenshot shows the Amazon SNS Topics page. On the left, there's a navigation sidebar with 'Amazon SNS' at the top, followed by 'Dashboard', 'Topics', 'Subscriptions', and 'Mobile' which has 'Push notifications' under it. A blue banner at the top right says 'New Feature: Amazon SNS now supports High Throughput FIFO topics. Learn more' with a link icon. The main area has a header 'Topics (3)' with a search bar below it. There are three topic entries: 'Topic 1', 'Topic 2', and 'Topic 3'. Each entry has 'Edit', 'Delete', 'Publish message', and 'Create topic' buttons. Below the topics is a table with columns 'Name', 'Type', and 'ARN'. The 'Name' column is sorted in ascending order, indicated by an upward arrow.

Name	Type	ARN
Topic 1		
Topic 2		
Topic 3		

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



- o Configure the SNS topic and note down the Topic ARN.



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

Create subscription

Details

Topic ARN: arn:aws:sns:ap-south-1:353250843450:Travlego2

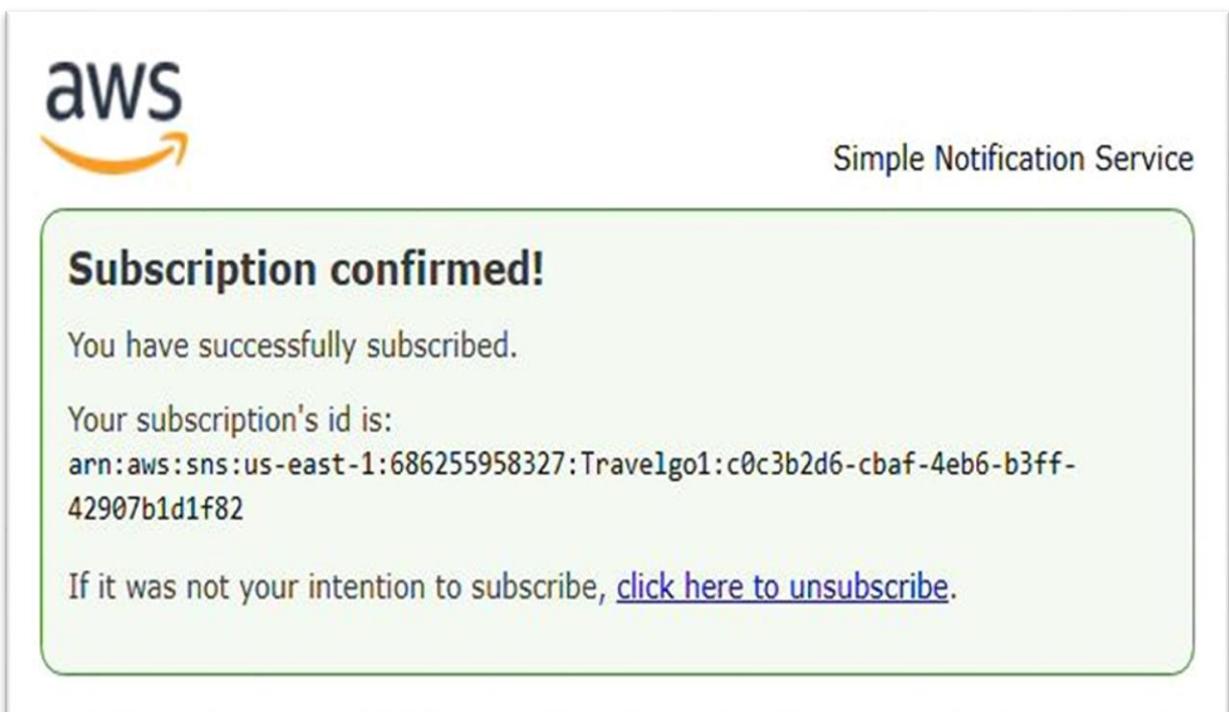
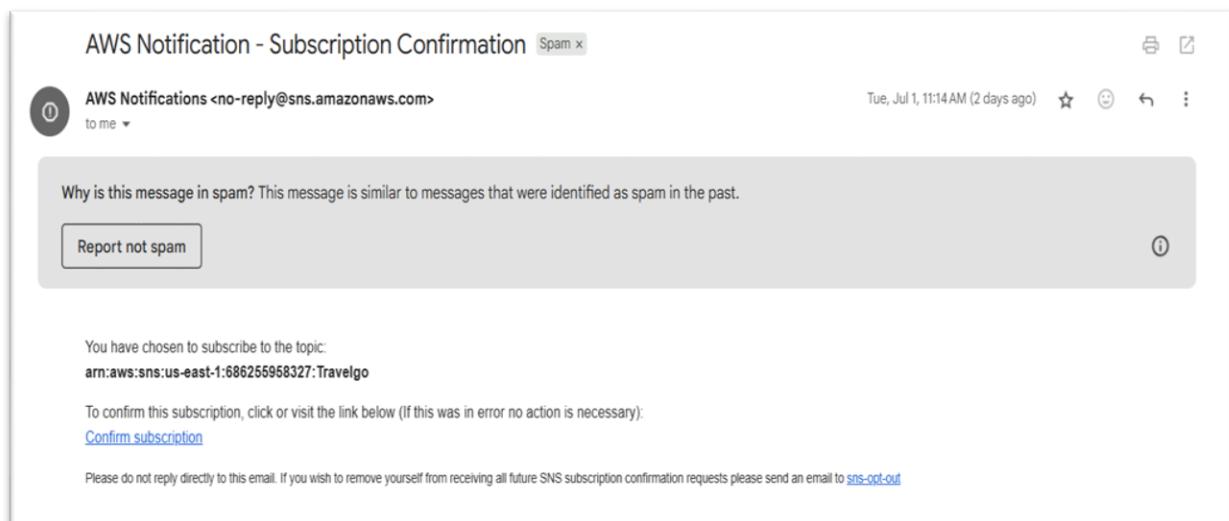
Protocol: Email

Endpoint: test@example.com

After your subscription is created, you must confirm it. [Info](#)

Subscription filter policy - optional [Info](#)
This policy filters the messages that a subscriber receives.

Redrive policy (dead-letter queue) - optional [Info](#)

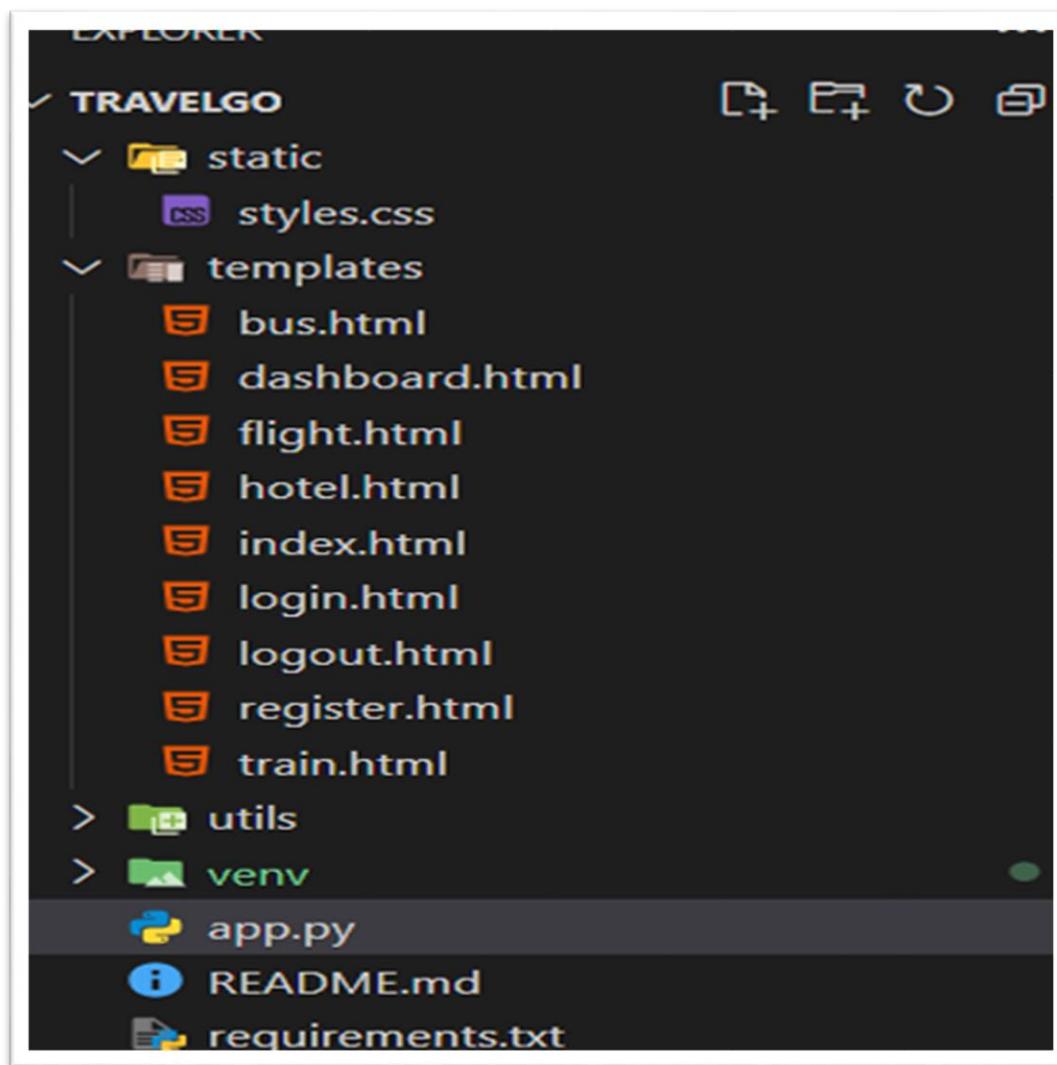


- Successfully done with the SNS mail subscription and setup, now store the ARN link.

Milestone 4: Backend Development and Application Setup

➤ Activity 4.1: Develop the backend using Flask

- File Explorer Structure



Description of the code :

- Flask App Initialization

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

Description:

Import essential libraries including Flask utilities for routing, template rendering, session handling, and request processing; Boto3 and DynamoDB conditions for performing AWS

DynamoDB operations; security module for password hashing and verification; Datetime for handling timestamps; Decimal for precise number representation (required by DynamoDB); and UUID for generating unique booking and user identifiers.

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

Description:

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

Dynamodb Setup:

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

Description:

Define references to DynamoDB tables used in the application.

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

SNS Connection

```
SNS_TOPIC_ARN = 'arn:aws:sns:us-east-1:713881794827:travelgo:302dcf42-b7b

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

Description:

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

Routes for Web Pages

➤ Home Route:

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

Description:

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

Register Route:

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

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

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

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

        flash('Registration successful! Please log in.', 'success')
        return redirect(url_for('login'))
    return render_template('register.html')
```

Description:

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

login Route:

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

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

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

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

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

            flash('Invalid email or password. Please try again.', 'error')
            return render_template('login.html')

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

Description:

Handles the /login route for user authentication via GET and POST methods. On POST, it retrieves the user from travelgo_users by email and verifies the password using

check_password_hash(). If valid, it stores user details in the session and updates the login count in DynamoDB. On failure, it flashes an error message and re-renders the login page.

Dashboard Route:

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

    user_email = session['email']

    # Query bookings for the logged-in user
    try:
        response = bookings_table.query(
            KeyConditionExpression=Key('user_email').eq(user_email),
            ScanIndexForward=False # Get most recent bookings first
        )
        bookings = response.get('Items', [])

        # Convert Decimal types from DynamoDB to float for display
        for booking in bookings:
            for key, value in booking.items():
                if isinstance(value, Decimal):
                    booking[key] = float(value)

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

    return render_template('dashboard.html', username=user_email, bookings=bookings)
```

Description:

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

Booking Route:

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

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

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

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

Description:

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

Deployment Code:

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

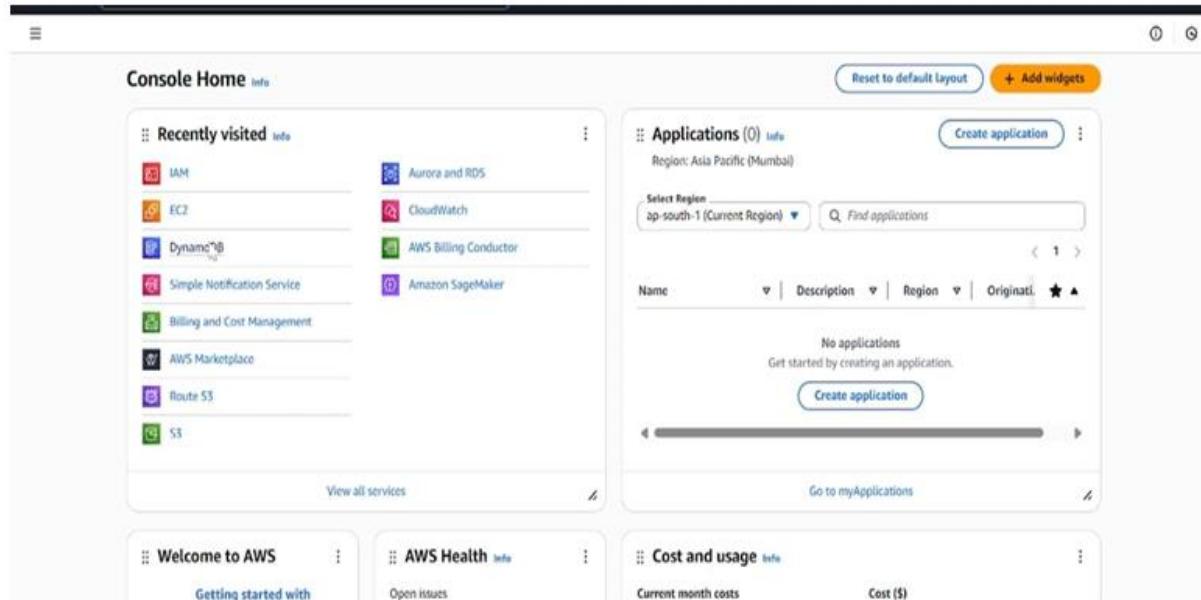
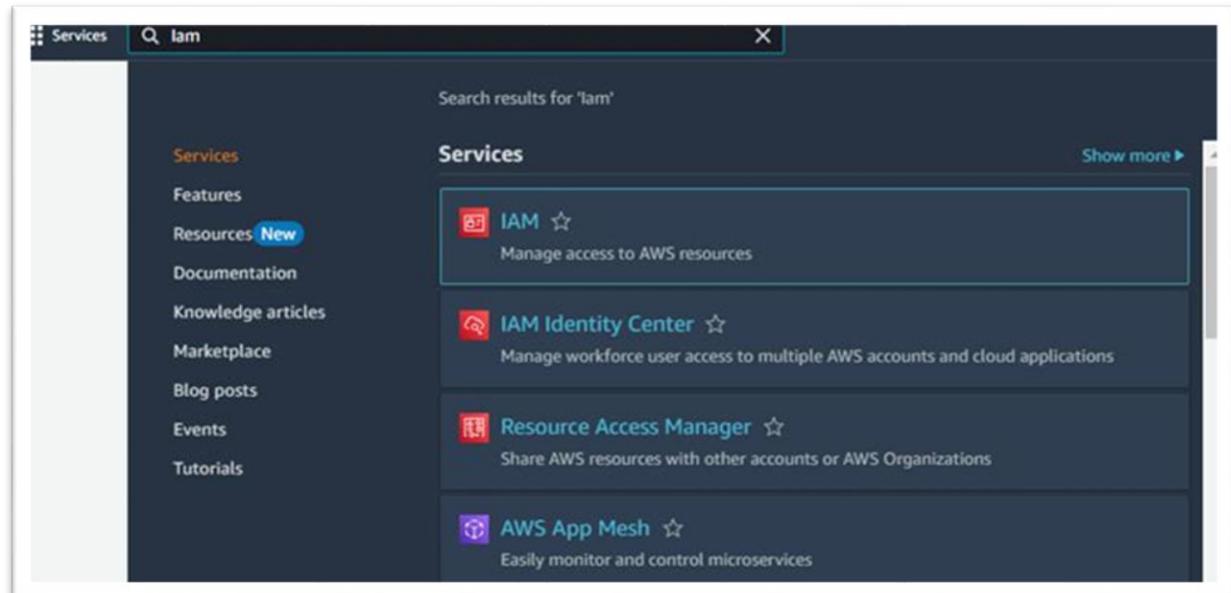
Description:

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

Milestone 5: IAM Role Setup

➤ Activity 5.1: Create IAM Role.

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



The screenshot shows the AWS IAM Dashboard. On the left, there's a navigation sidebar with options like 'User groups', 'Users', 'Roles' (which is selected), 'Policies', 'Identity providers', 'Account settings', and 'Root access management'. Below that is another section for 'Access reports' with 'Access Analyzer', 'Resource analysis', 'Unused access', and 'Analyzer settings'. The main area has a blue banner at the top stating 'New access analyzers available' and 'Create new analyzer'. It includes sections for 'Security recommendations' (with two items: 'Root user has MFA' and 'Root user has no active access keys') and 'AWS Account' (with details like 'Account ID: 353250843450', 'Account Alias: Create', and a 'Sign-in URL for IAM users in this account: https://353250843450signin.aws.amazon.com/console').

The screenshot shows the 'Create role' wizard. The user is on the 'Choose a service or use case' step. A dropdown menu is open, showing 'Commonly used services' with 'EC2' selected and 'Lambda' as the next option. Other services listed include Amazon Aurora DSQL, Amazon EMR Serverless, Amazon OpenSearch Service, Amazon Q Business, Amazon Grafana, Amplify, and API Gateway. A note says 'With SAML 2.0 from [redacted] you can perform actions in [redacted] is account.' At the bottom right are 'Cancel' and 'Next' buttons.

Activity 5.2: Attach Policies.

Attach the following policies to the role:

- **AmazonDynamoDBFullAccess**: Allows EC2 to perform read/write operations on DynamoDB.
- **AmazonDynamoDBFullAccess**: Allows EC2 to perform read/write operations on DynamoDB.
- **AmazonEC2FullAccess**: Provides full permissions to manage all EC2 resources, including instances, volumes, and networking.

IAM > Roles > Create role

Policy name	Type	Description
AmazonEC2ContainerRegistryFullAccess	AWS managed	Provides adminis
AmazonEC2ContainerRegistryPowerUser	AWS managed	Provides full acce
AmazonEC2ContainerRegistryPullOnly	AWS managed	Provides access to
AmazonEC2ContainerRegistryReadOnly	AWS managed	Provides read-on
AmazonEC2ContainerServiceAutoscaleRole	AWS managed	Policy to enable t
AmazonEC2ContainerServiceEventsRole	AWS managed	Policy to enable e
AmazonEC2ContainerServiceforEC2Role	AWS managed	Default policy for
AmazonEC2ContainerServiceRole	AWS managed	Default policy for
<input checked="" type="checkbox"/> AmazonEC2FullAccess	AWS managed	Provides full acce
AmazonEC2ReadOnlyAccess	AWS managed	Provides read onl

IAM > Roles > Create role

Step 1 Select trusted entity

Step 2 Add permissions

Step 3 Name, review, and create

Add permissions Info

Permissions policies (3/1057) Info

Choose one or more policies to attach to your new role.

Filter by Type

Q: SNS All types 5 matches

Policy name	Type	Description
<input checked="" type="checkbox"/> AmazonSNSFullAccess	AWS managed	Provides full acce
AmazonSNSReadOnlyAccess	AWS managed	Provides read onl
AmazonSNSRole	AWS managed	Default policy for
AWSElasticBeanstalkRoleSNS	AWS managed	(Elastic Beanstalk
AWSToTDeviceDefenderPublishFindingsToSNSMitigatio...	AWS managed	Provides message

IAM > Roles > Create role

Step 1 Select trusted entity

Step 2 Add permissions

Step 3 Name, review, and create

Add permissions Info

Permissions policies (2/1057) Info

Choose one or more policies to attach to your new role.

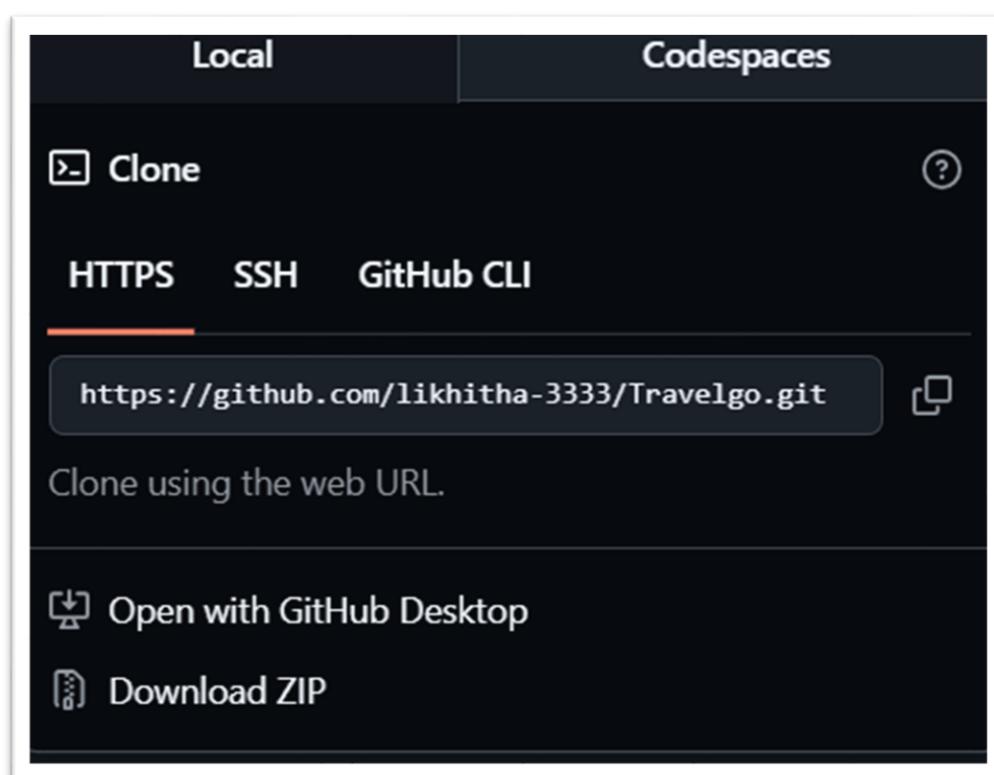
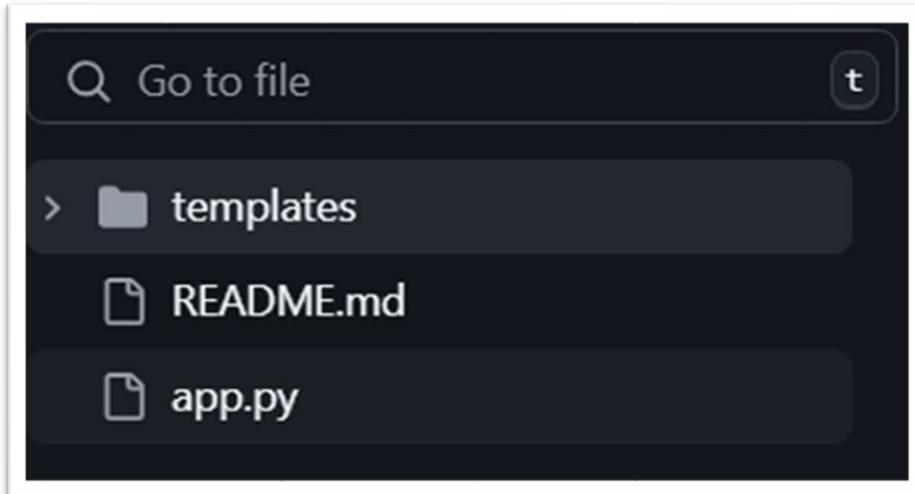
Filter by Type

Q: Dynamo All types 6 matches

Policy name	Type	Description
<input checked="" type="checkbox"/> AmazonDynamoDBFullAccess	AWS managed	Provides full acce
AmazonDynamoDBFullAccess_v2	AWS managed	Provides full acce
AmazonDynamoDBFullAccesswithDataPipeline	AWS managed	This policy is on a
AmazonDynamoDBReadOnlyAccess	AWS managed	Provides read onl
AWSLambdaDynamoDBExecutionRole	AWS managed	Provides list and

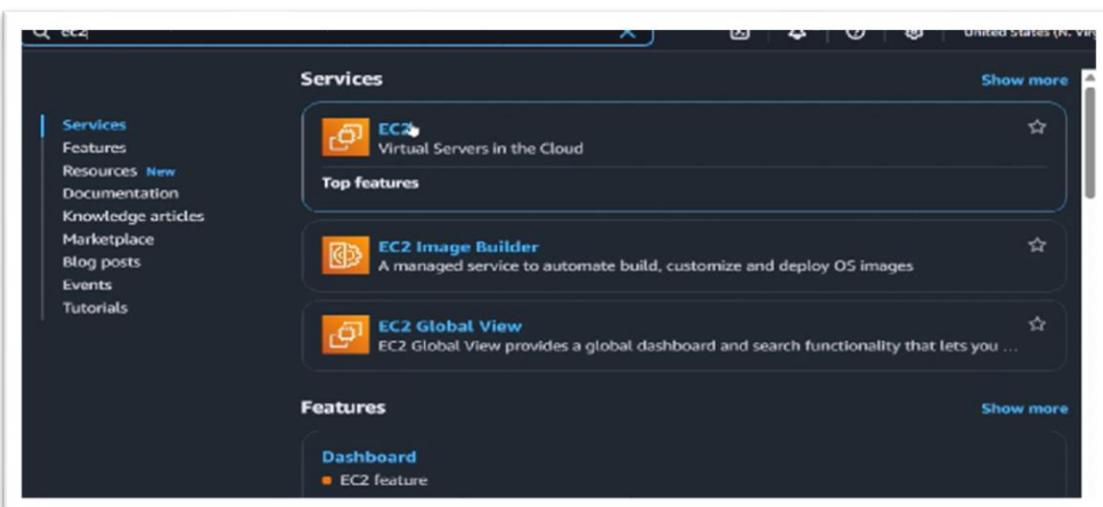
Milestone 6: EC2 Instance Setup

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

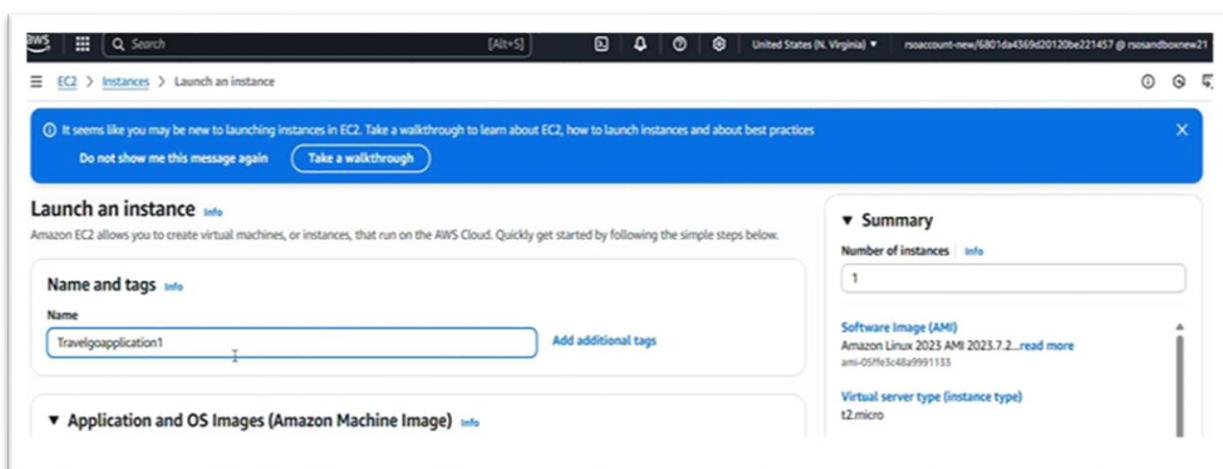


Activity 6.1: Launch an EC2 instance to host the Flask application.

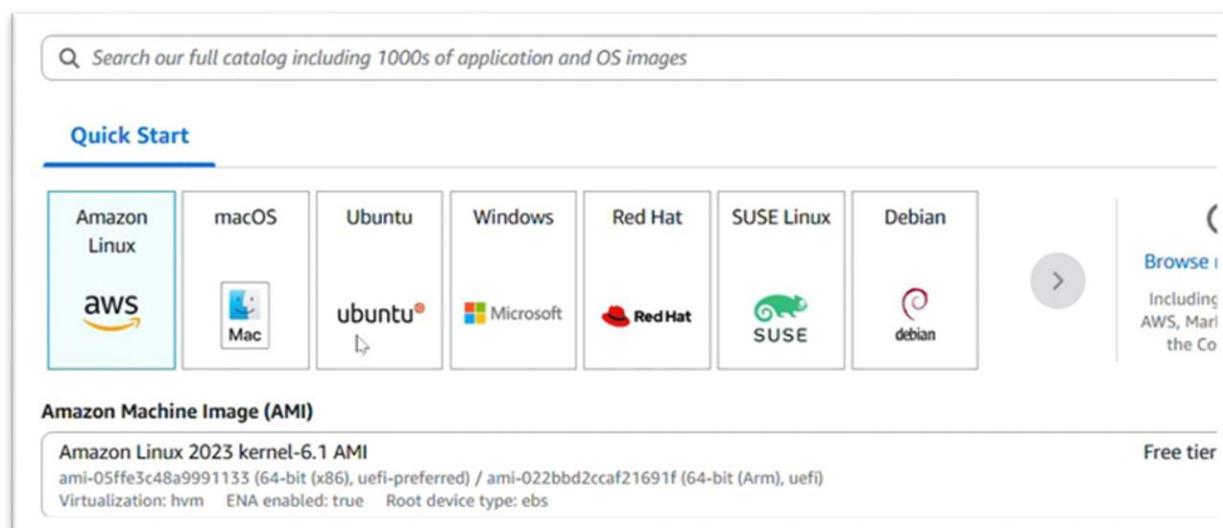
- Launch EC2 Instance ◦ In the AWS Console, navigate to EC2 and launch a new instance.



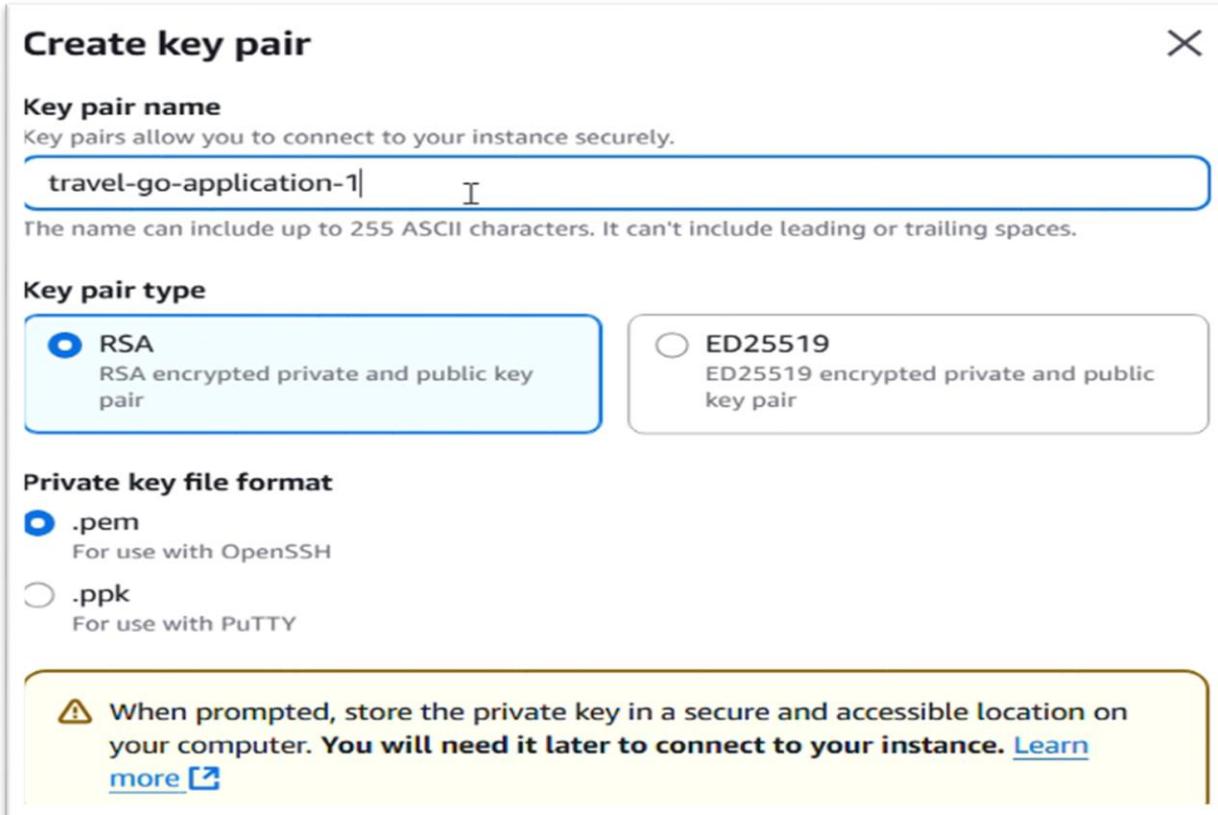
- Click on Launch instance to launch EC2 instance.



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



- Create and download the key pair for Server access.



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

Edit inbound rules:

dit inbound rules [Info](#)

bound rules control the incoming traffic that's allowed to reach the instance.

Inbound rules Info	Type Info	Protocol Info	Port range Info	Source Info	Description - optional Info
sgr-04f74a082f34acd4e	Custom TCP	TCP	5000	Custom	<input type="text"/> 0.0.0.0/0 Delete
sgr-0e91ec1a5ea31f68	HTTPS	TCP	443	Custom	<input type="text"/> 0.0.0.0/0 Delete
sgr-0a2fdcbe6ce079d51	SSH	TCP	22	Custom	<input type="text"/> 0.0.0.0/0 Delete

[Add rule](#)

⚠️ Rules with source of 0.0.0.0/0 or ::/0 allow all IP addresses to access your instance. We recommend setting security group rules to allow access from known IP addresses only.

[Cancel](#) [Preview changes](#) [Save rules](#)

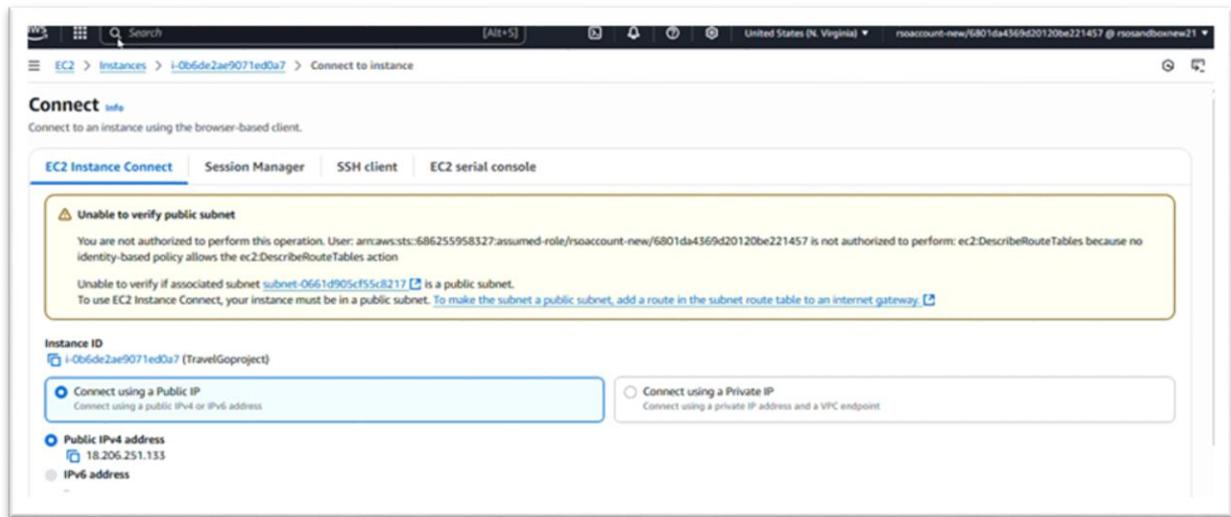
- To connect to EC2 using EC2 Instance Connect, start by ensuring that an IAM role is attached to your EC2 instance. You can do this by selecting your instance, clicking on Actions, then navigating to Security and selecting Modify IAM Role to attach the appropriate role. After the IAM role is connected, navigate to the EC2 section in the AWS Management Console. Select the EC2 instance you wish to connect to. At the top of the EC2 Dashboard,

click the Connect button. From the connection methods presented, choose EC2 Instance Connect. Finally, click Connect again, and a new browser-based terminal will open, allowing you to access your EC2 instance directly from your browser.

The screenshot shows the AWS EC2 Instances page. On the left, there's a sidebar with navigation links for EC2, Dashboard, EC2 Global View, Events, Instances (selected), Instance Types, Launch Templates, Spot Requests, Savings Plans, Reserved Instances, Dedicated Hosts, Capacity Reservations, Images, AMIs, and AMI Catalog. The main content area has a search bar and a table titled 'Instances (1/1)'. The table shows one instance: 'TravelGoproject' (Instance ID: i-0b6de2ae9071ed0a7, Status: Running, Type: t2.micro, Public IP: ec2-18-20...). Below the table, a detailed view for 'i-0b6de2ae9071ed0a7 (TravelGoproject)' is shown, including Security details (IAM Role: -, Owner ID: 606255958327, Security groups: sg-074f829b17da4071a), Launch time (Tue Jul 01 2025 11:17:47 GMT+0530 (India Standard Time)), and a note about the instance being created by a launch wizard.

The screenshot shows the AWS EC2 Security Groups page. The sidebar includes links for EC2, Dashboard, EC2 Global View, Events, Instances, Instance Types, Launch Templates, Spot Requests, Savings Plans, Reserved Instances, Dedicated Hosts, Capacity Reservations, Images, AMIs, and AMI Catalog. The main content shows the 'sg-074f829b17da4071a - launch-wizard-1' security group. It displays details like Security group name: 'launch-wizard-1', Security group ID: 'sg-074f829b17da4071a', Description: 'launch-wizard-1 created 2025-07-01T05:45:44.696Z', Owner: '606255958327', Inbound rules count: '3 Permission entries', and Outbound rules count: '1 Permission entry'. Below this, the 'Inbound rules' tab is selected, showing three entries: a Custom TCP rule (Port 5000) allowing traffic from 0.0.0.0/0, an HTTPS rule (Port 443) allowing traffic from 0.0.0.0/0, and an SSH rule (Port 22) allowing traffic from 0.0.0.0/0. There are tabs for Outbound rules, Sharing - new, VPC associations - new, and Tags. At the bottom, there's a section for modifying IAM role, where 'studentuser' is selected, and buttons for 'Create new IAM role' and 'Update IAM role'.

- Now connect the EC2 with the files



Milestone 7: Deployment on EC2

Activity 7.1: Install Software on the EC2 Instance

Install Python3, Flask, and Git:

On Amazon Linux 2:

```
Sudo yum install git -y
```

```
sudo yum install python3 -y
```

```
sudo yum install python3-pip -y
```

Pip install flask

Pip install boto3

Activity 7.2: Clone Your Flask Project from GitHub

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

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

Note: change your-github-username and your-repository-name with your credentials here:

'git clone <https://github.com/AlekhyaPenubakula/InstantLibrary.git>'

- This will download your project to the EC2 instance.

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

```
cd InstantLibrary
```

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

```
Run the Flask Application
```

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

Verify the Flask app is running:

<http://your-ec2-public-ip>

- o Run the Flask app on the EC2 instance

```
File  C:\travelgo\app.py , line 2, in <module>
      import boto3
ModuleNotFoundError: No module named 'boto3'
PS C:\travelgo> & C:/Users/likhita/AppData/Local/Programs/Python/Python313/python.exe c:/travelgo/app.py
*   Serving Flask app 'app'
*   Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead
*   Running on http://127.0.0.1:5000
Press CTRL+C to quit
*   Restarting with stat
*   Debugger is active!
*   Debugger PIN: 446-964-444
127.0.0.1 - - [03/Jul/2025 09:39:12] "GET / HTTP/1.1" 200 -
```

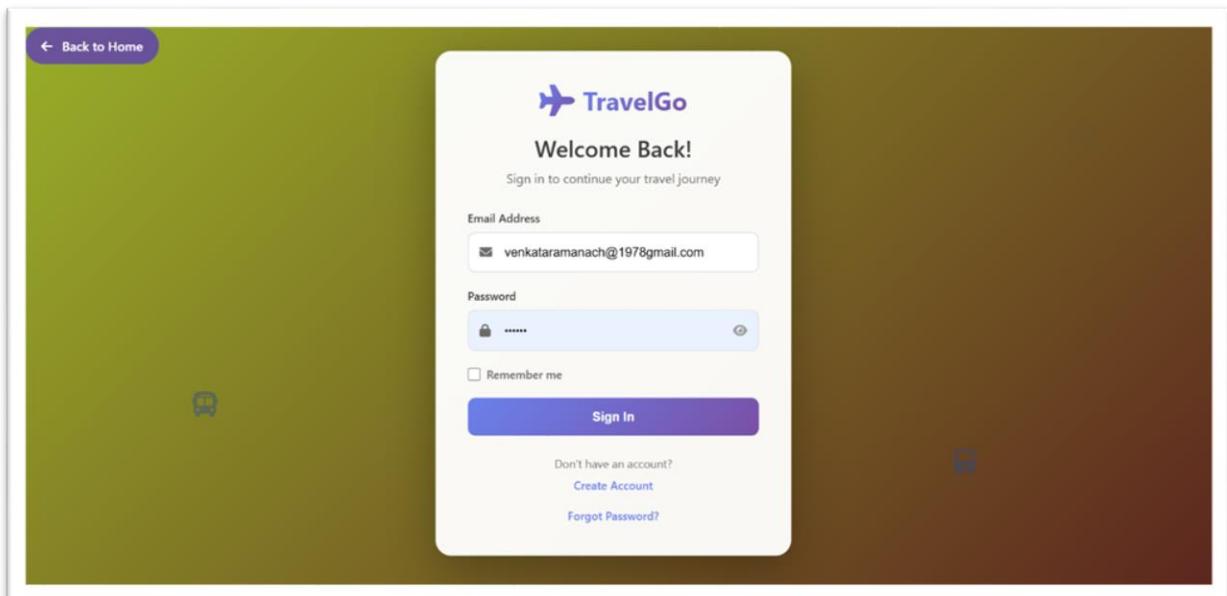
Access the website through:

PublicIPs: <http://18.206.251.133:5000/>

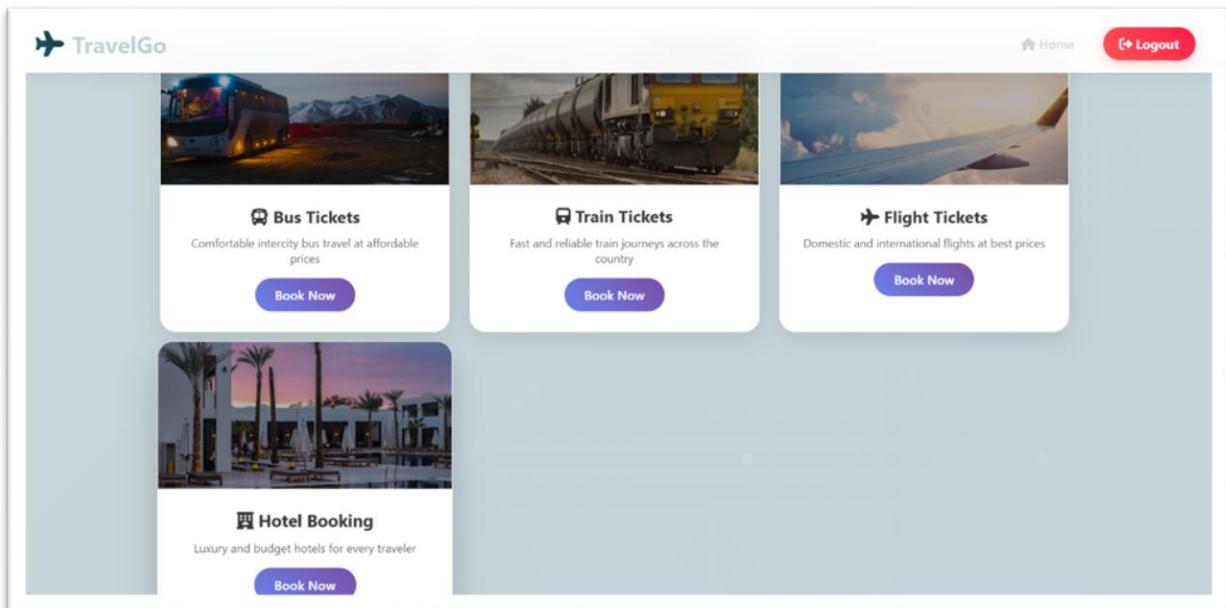
Milestone 8: Testing and Deployment

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

Login page:



Dashboard page:



Bus booking page:



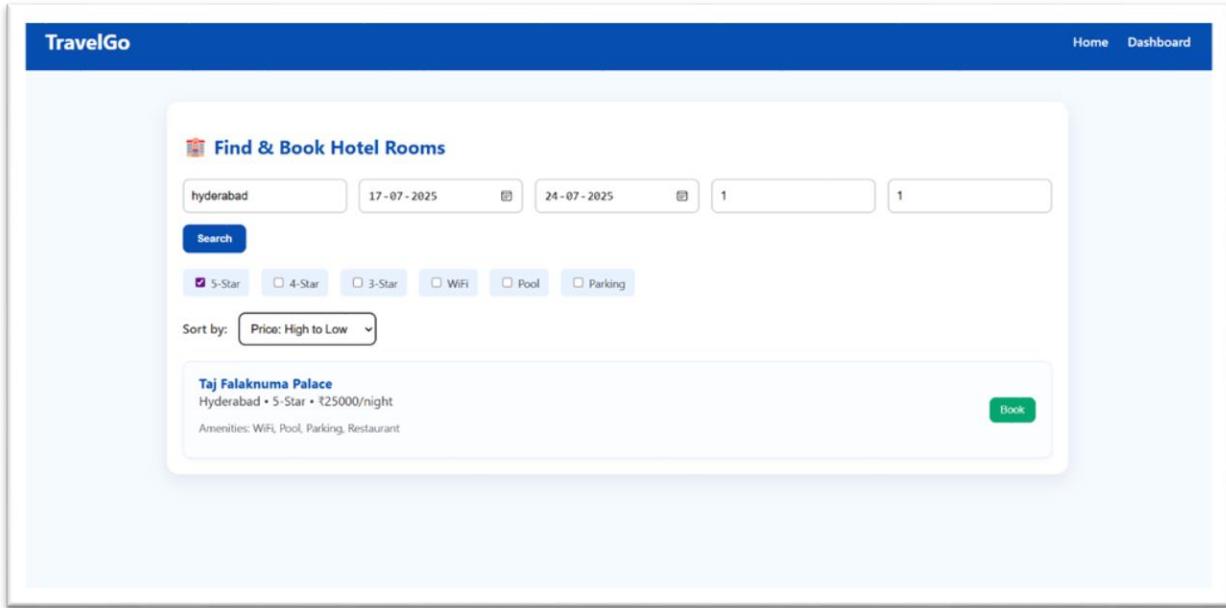
Train booking page:

The screenshot shows a "Train Booking" page with a teal header bar. The header includes a logo, the title "Train Booking", a "Dashboard" button, and a "Logout" button. Below the header, there's a search form with fields for "From Station" (Delhi), "To Station" (Mumbai), and "Travel Date" (18-06-2025). A green "Search Trains" button is below the form. The main content area displays a message "Found 3 trains" and a list of results. The first result is for the "Rajdhani Express" (12301) from New Delhi (NDL) to Howrah (HWH). It shows travel time as 17h 15m and status as "ON TIME". The booking class options (1A, 2A, 3A) are listed, along with a "Book Now" button.

Flight booking page:

The screenshot shows a "Flight Booking" page. At the top, there are two radio buttons: "Round Trip" (selected) and "One Way". Below that are fields for "From" (Delhi), "To" (Mumbai), "Departure Date" (30-06-2025), and "Return Date" (12-07-2025). There are dropdown menus for "Passengers" (set to 1) and "Class" (set to Economy). A blue "Search Flights" button is located below these fields. The main content area shows a flight result for "Air India (AI101)" from Delhi to Mumbai. It details the departure at 08:30 and arrival at 11:00, with a duration of 2h 30m. The price is listed as ₹4,500, and a "Book Now" button is provided.

Hotel booking page:



Conclusion:

TravelGo brings together cutting-edge cloud infrastructure and a sleek, intuitive user interface to redefine travel bookings. Leveraging Flask, AWS EC2, DynamoDB, and SNS, the platform delivers real-time performance, secure data management, and instantaneous notifications. Users can seamlessly book buses, trains, flights, and hotels all in one place, with smart dashboards and filtering tools enhancing the experience. Scalable and dependable, TravelGo meets the demands of contemporary travel. It showcases how full-stack architecture combined with cloud services can solve real-world challenges by boosting convenience, reliability, and technical excellence.