Travel Go: A Cloud-Powered Real-Time Travel Booking Platform

Project Description:

TravelGo simplifies travel planning by providing a single platform to book buses, trains, flights, and hotels.
As demand grows for real-time and hassle-free travel services, TravelGo meets this need using cloud tools.
It uses Flask for backend growth, AWS EC2 for hosting, DynamoDB for data storage, and AWS SNS to send instant Alerts.

Users can easily register, log in, search, and book transport or stay.
The system also includes features like booking history and dynamic seat selection.
These tools ensure a smooth, fast, and ready for every traveler.

.

Case 1: Illustrates Real-Time Booking for Users.

Using TravelGo, users log in, choose travel type —bus, train, flight, or hotel—and enter their booking choice.
The Flask backend processes these inputs and retrieves matching options from DynamoDB.

AWS EC2 handles traffic and delivers fast response times once the user confirms a booking.Thanks to its real-time design, the platform allows users to book travel without delays or long page loads—even during peak seasons.

Case 2: shows how TravelGo sends email alerts after a

booking.

Once the user confirms a booking, TravelGo uses AWS SNS to send the booking details by email.
For example, when a user books a flight, Flask handles the transaction while AWS SNS sends a email to the user and notifies the service provider if needed.
DynamoDB securely stores all booking data.

This smooth and fast communication helps build user trust and enhances the overall experience.

Case 3: explains how users book easily and access their records without delays.

Users can log in at any time to view or manage their previous bookings.
For example, a user may open the dashboard to check a hotel booking made a week earlier.
Flask retrieves this data from DynamoDB, ensuring instant access.

TravelGo's cloud-based design provides more handiness, while its simple interface makes it easy to cancel bookings or plan new trips.
With AWS EC2 handling the hosting, the platform runs smoothly—even when many users are online at once.

AWS setup

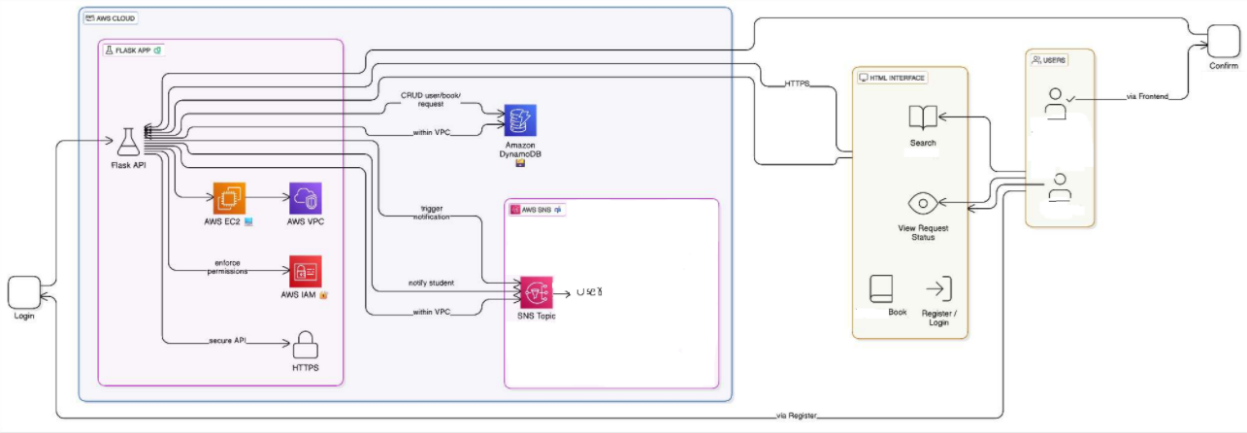
TravelGo uses Amazon Web Services (AWS) to build a fast and real-time travel booking platform..

- **AWS EC2** (Elastic Compute Cloud) hosts the Flask-based backend coating. It ensures high handiness and fast response times, even under heavy user load.
- **Amazon DynamoDB** serves as the primary database for storing user

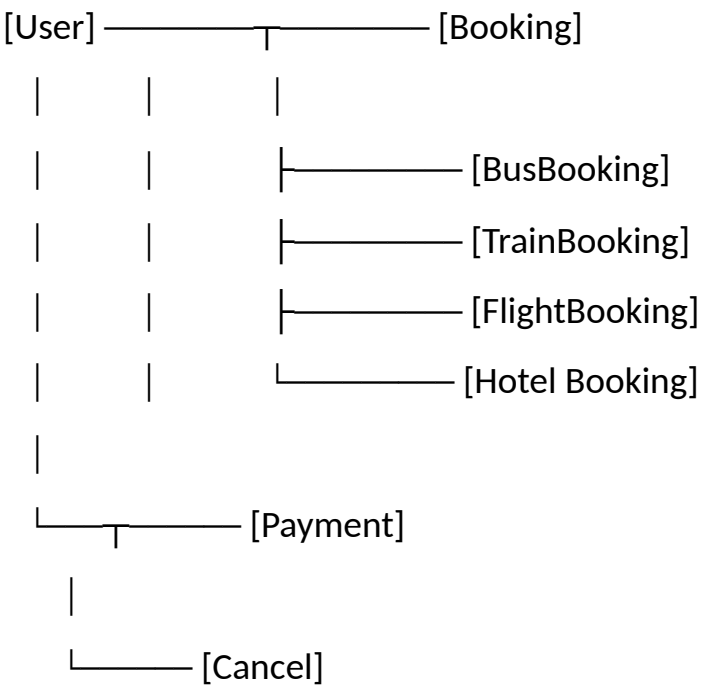
data, booking history, and travel records. Its low-latency performance supports instant data retrieval.

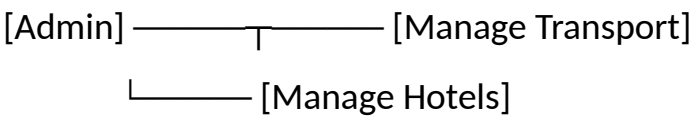
- **AWS SNS** enables real-time messages. *The system sends instant email alerts to users after they confirm a booking and can also notify service providers*

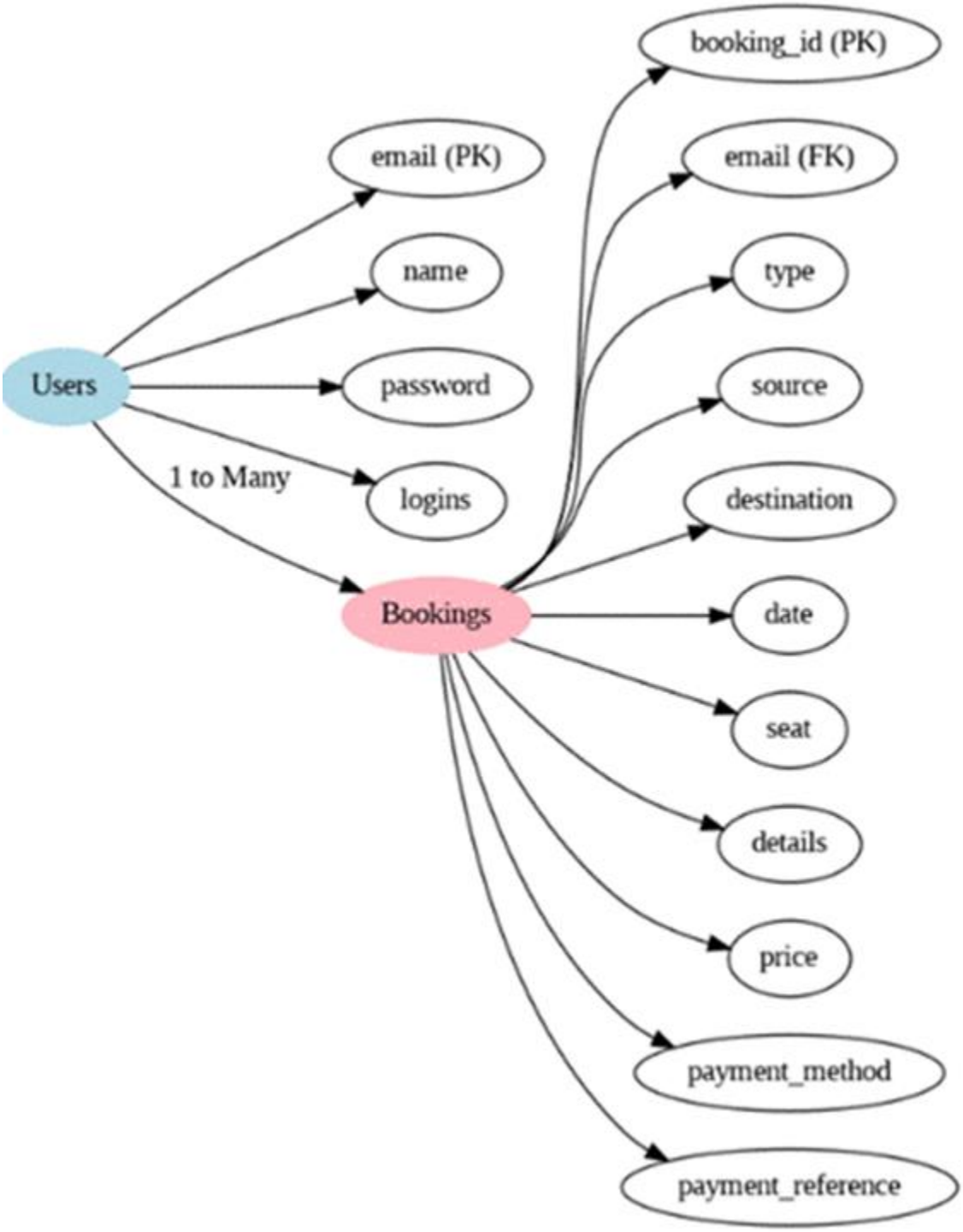
, IAM roles securely connect EC2 instances to other AWS services without hard coding credentials.



(ER)Diagram




[Admin]
[Manage Transport]
[Manage Hotels]



Pre-requisites:

1. Set up an AWS Account Setup: AWS Account Setup

2. see IAM: IAM Overview

3.Set up an Amazon EC2 Basics: EC2

4. Get DynamoDB Basics: DynamoDB

5. Set up SNS: SNS support

6. Use Git: Git Document

The Work Flow Of TravelGo Project

1. Start With AWS Account Setup and Login

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

action 1.2: Log in to the AWS Console

2. DynamoDB Database Creation and Setup

action 2.1: Create a DynamoDB Table.

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

3. SNS Setup

action 3.1: Create SNS topics for book request notice.

action 3.2: Subscribe users and library staff to SNS email.

4. Setup Backend and App

action 4.1: Develop the Backend Using Flask.

action 4.2: Integrate AWS Services Using boto3.

5. Setup IAM Role

action 5.1: Create IAM Role

action 5.2: Attach Policies

6. Setup EC2 Instance

action 6.1: Launch an EC2 instance to host the Flask.

action 6.2: Configure protection groups for HTTP, and SSH access.

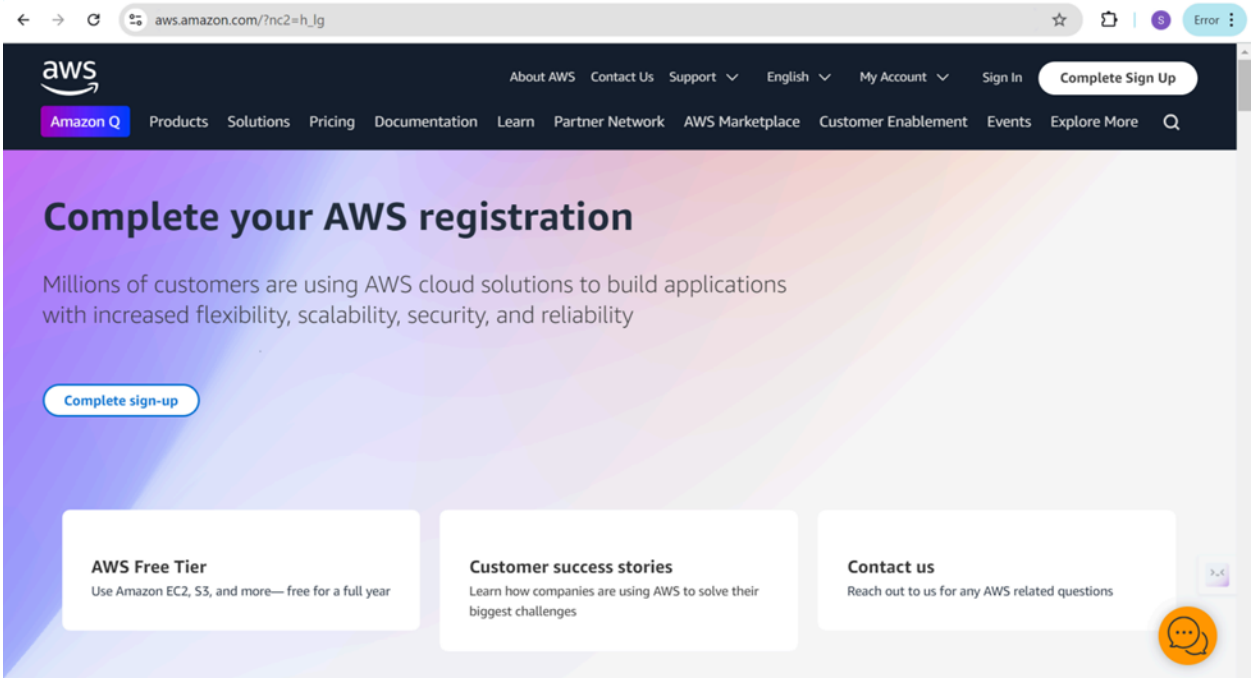
7. Now Deployment on EC2

action 7.1: Upload Flask Files

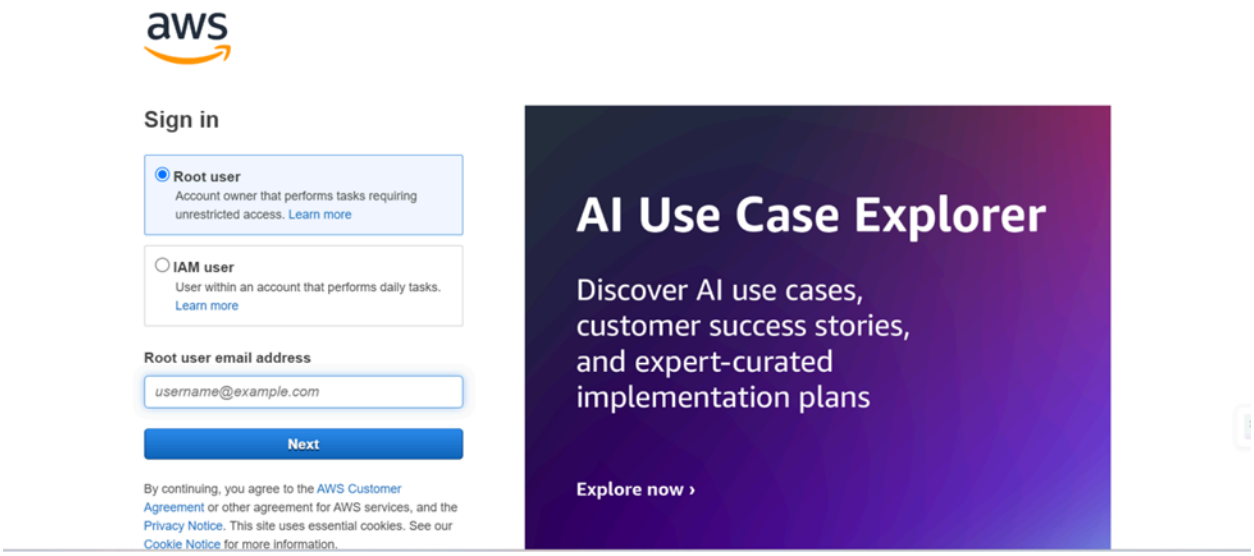
action 7.2: Run the Flask App

8. Let us Testing and Deploy

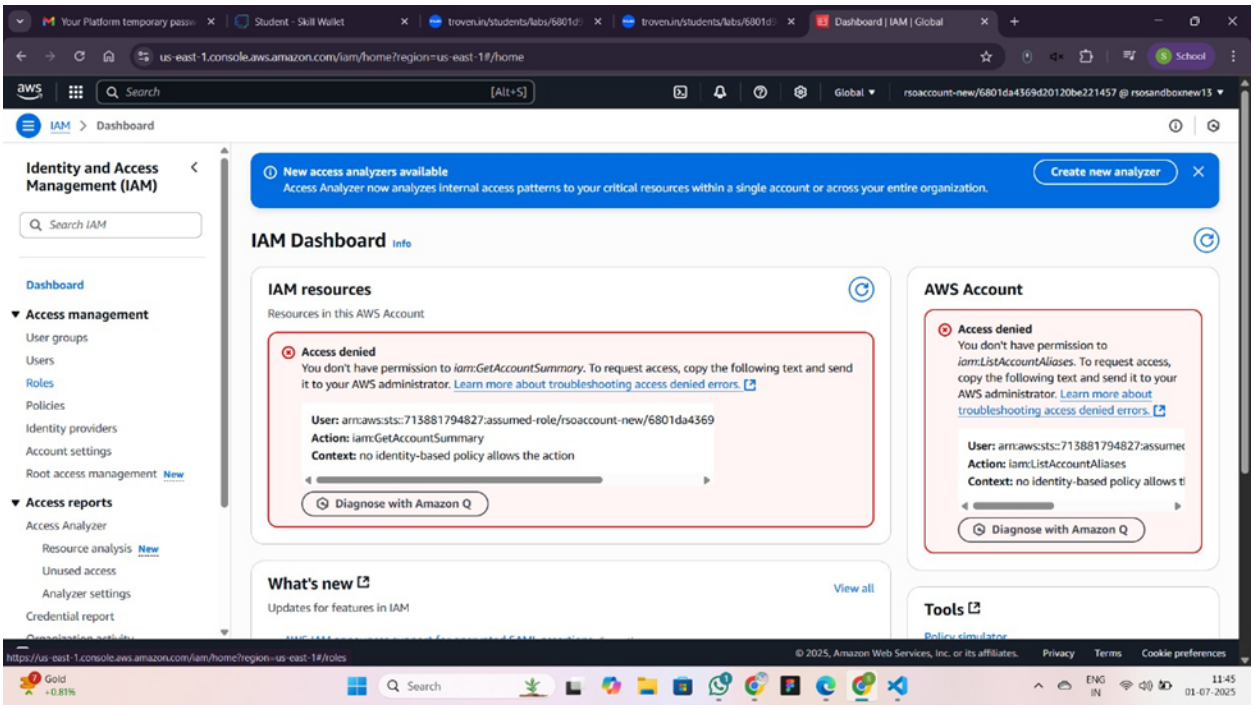
1: Setup AWS Account and Login

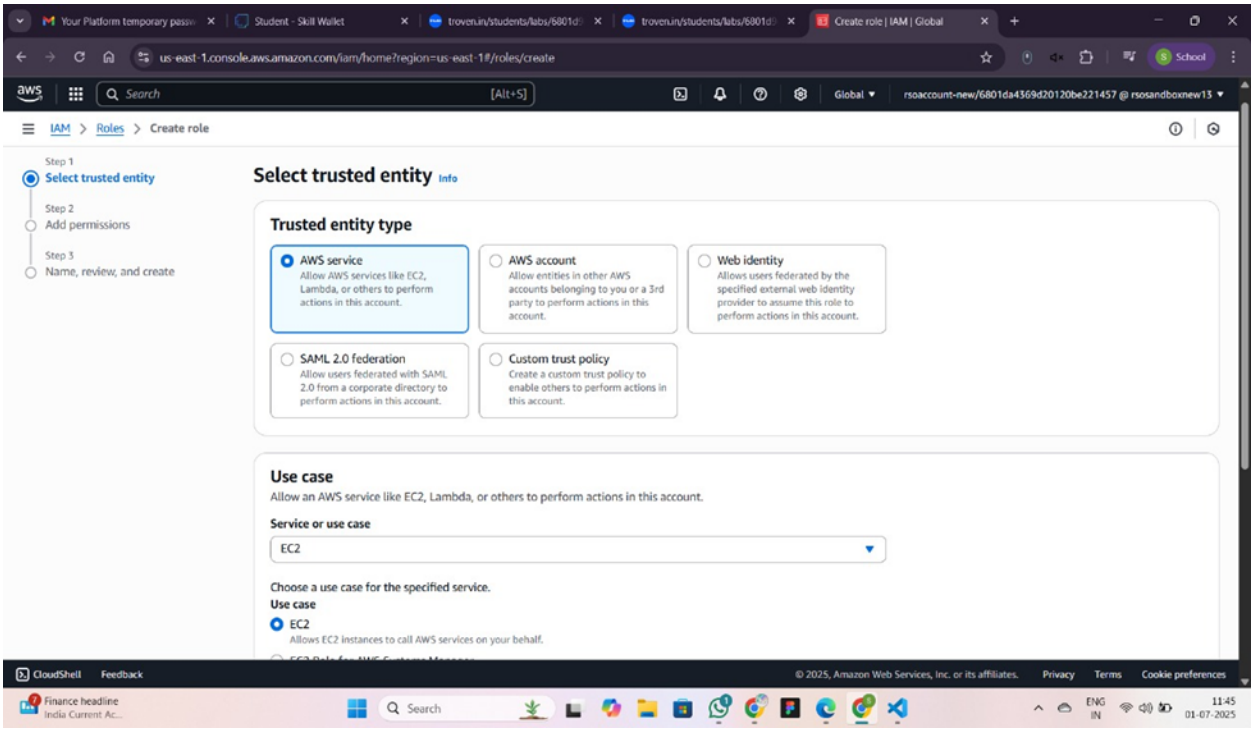


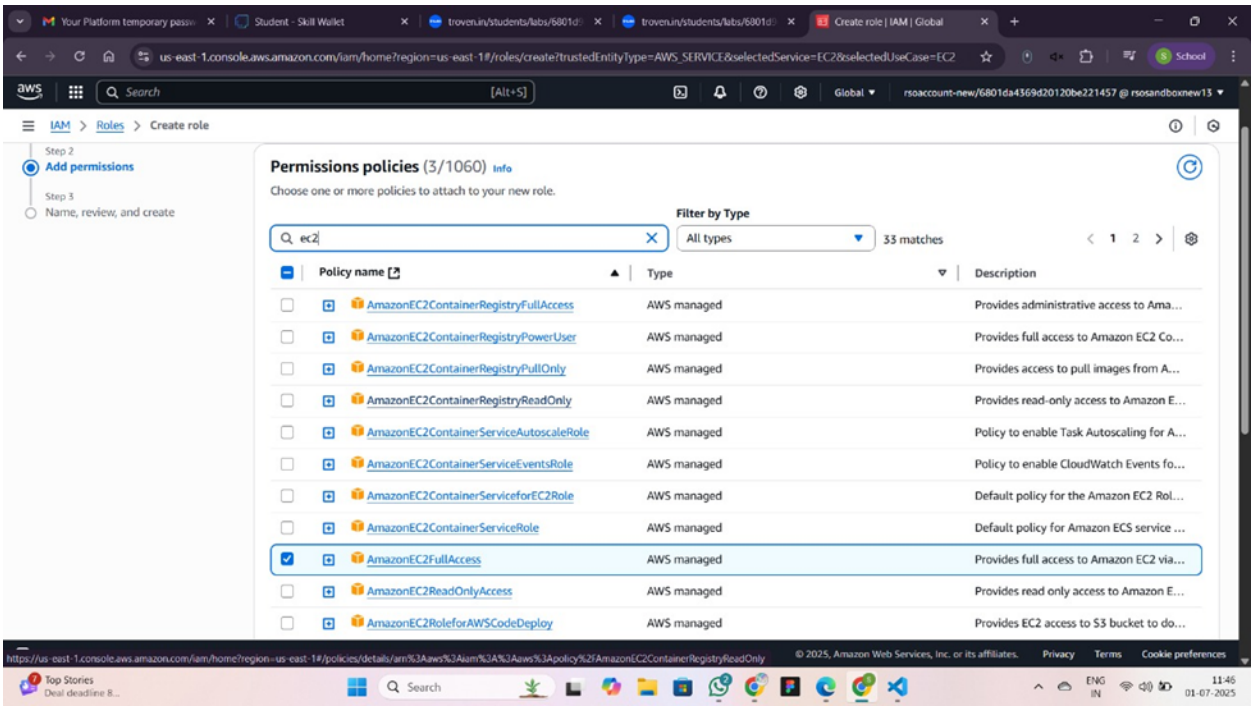
2: Next Log in to the AWS Console

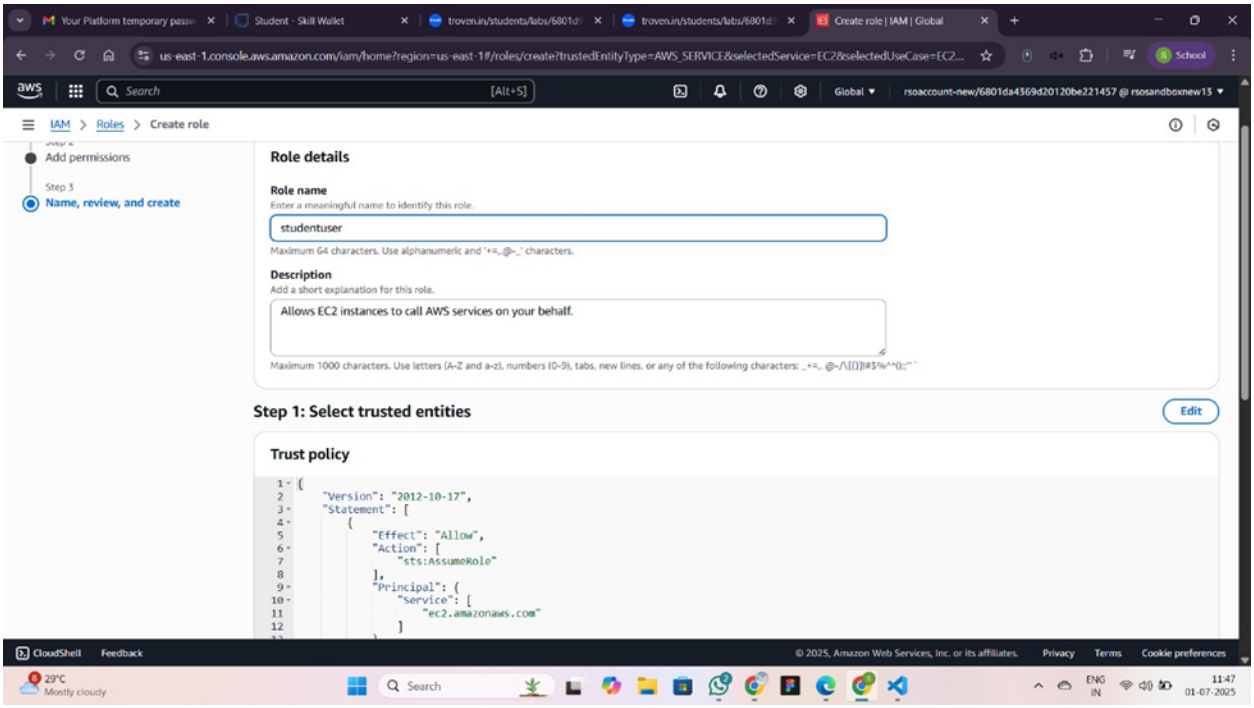


3. Create IAM Roles


Dashboard | IAM | Global
us-east-1.console.aws.amazon.com/iam/home?region=us-east-1#/home
IAM > Dashboard
Identity and Access Management (IAM)
Search IAM
Dashboard
Access management
User groups
Users
Roles
Policies
Identity providers
Account settings
Root access management New
Access reports
Access Analyzer
Resource analysis New
Unused access
Analyzer settings
Credential report
New access analyzers available
Access Analyzer now analyzes internal access patterns to your critical resources within a single account or across your entire organization.
Create new analyzer
IAM Dashboard Info
IAM resources
Resources in this AWS Account
Access denied
You don't have permission to iam:GetAccountSummary. To request access, copy the following text and send it to your AWS administrator. Learn more about troubleshooting access denied errors.
User: arn:aws:sts::713881794827:assumed-role/rsoaccount-new/6801da4369
Action: iam:GetAccountSummary
Context: no identity-based policy allows the action
Diagnose with Amazon Q
AWS Account
Access denied
You don't have permission to iam:ListAccountAliases. To request access, copy the following text and send it to your AWS administrator. Learn more about troubleshooting access denied errors.
User: arn:aws:sts::713881794827:assume
Action: iam:ListAccountAliases
Context: no identity-based policy allows t
Diagnose with Amazon Q
What's new
Updates for features in IAM
View all
Tools
© 2025, Amazon Web Services, Inc. or its affiliates.
Privacy
Terms
Cookie preferences


Create role | IAM | Global
us-east-1.console.aws.amazon.com/iam/home?region=us-east-1#/roles/create
IAM > Roles > Create role
Step 1
Select trusted entity
Step 2
Add permissions
Step 3
Name, review, and create
Select trusted entity Info
Trusted entity type
AWS service
Allow AWS services like EC2, Lambda, or others to perform actions in this account.
AWS account
Allow entities in other AWS accounts belonging to you or a 3rd party to perform actions in this account.
Web identity
Allows users federated by the specified external web identity provider to assume this role to perform actions in this account.
SAML 2.0 federation
Allow users federated with SAML 2.0 from a corporate directory to perform actions in this account.
Custom trust policy
Create a custom trust policy to enable others to perform actions in this account.
Use case
Allow an AWS service like EC2, Lambda, or others to perform actions in this account.
Service or use case
EC2
Choose a use case for the specified service.
Use case
EC2
Allows EC2 instances to call AWS services on your behalf.
CloudShell
Feedback
© 2025, Amazon Web Services, Inc. or its affiliates.
Privacy
Terms
Cookie preferences





4. DynamoDB Database Creation and Setup

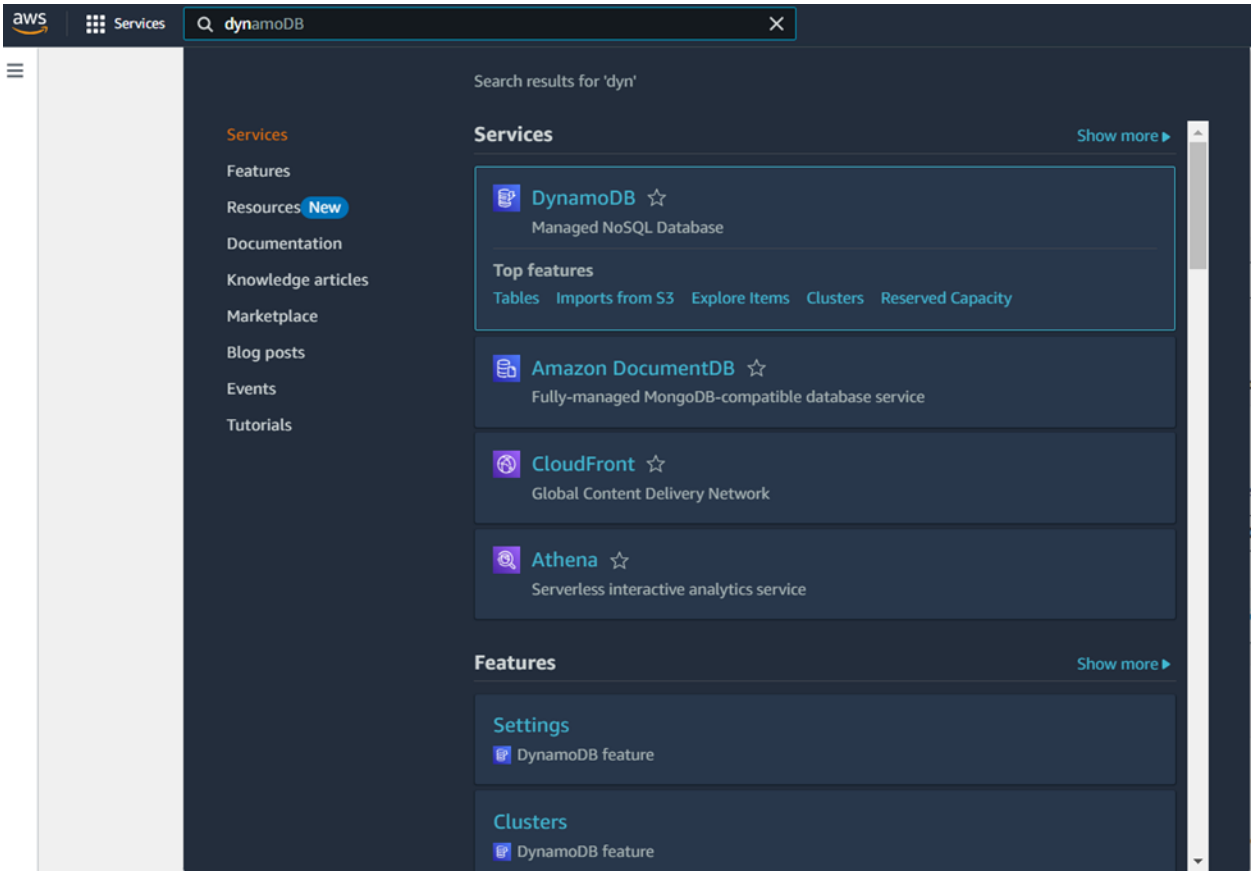
- In the AWS Console, navigate to DynamoDB and click on create tables

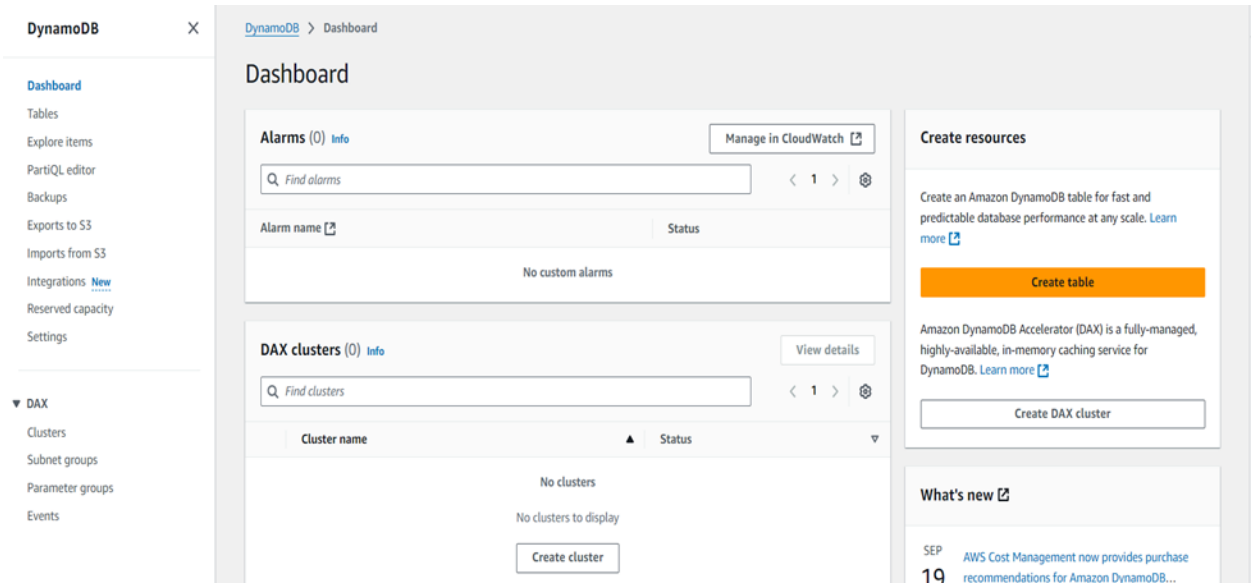
To manage user data and bookings, TravelGo uses **Amazon DynamoDB**, a

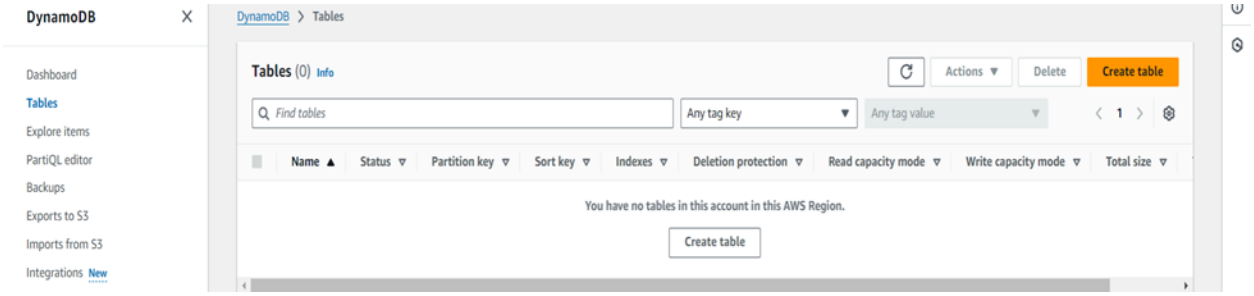
fast and scalable NoSQL database service offered by AWS.

To begin, you open the AWS Console and navigate to the DynamoDB section. From there, you can create a new table by giving it a name—such as "Users" or "Bookings"—and defining a primary key. The primary key could be something like a user ID or booking ID, which uniquely places each record. You can also add a sort key if you want to organize the data further, like by date.

Next, you choose the content mode.



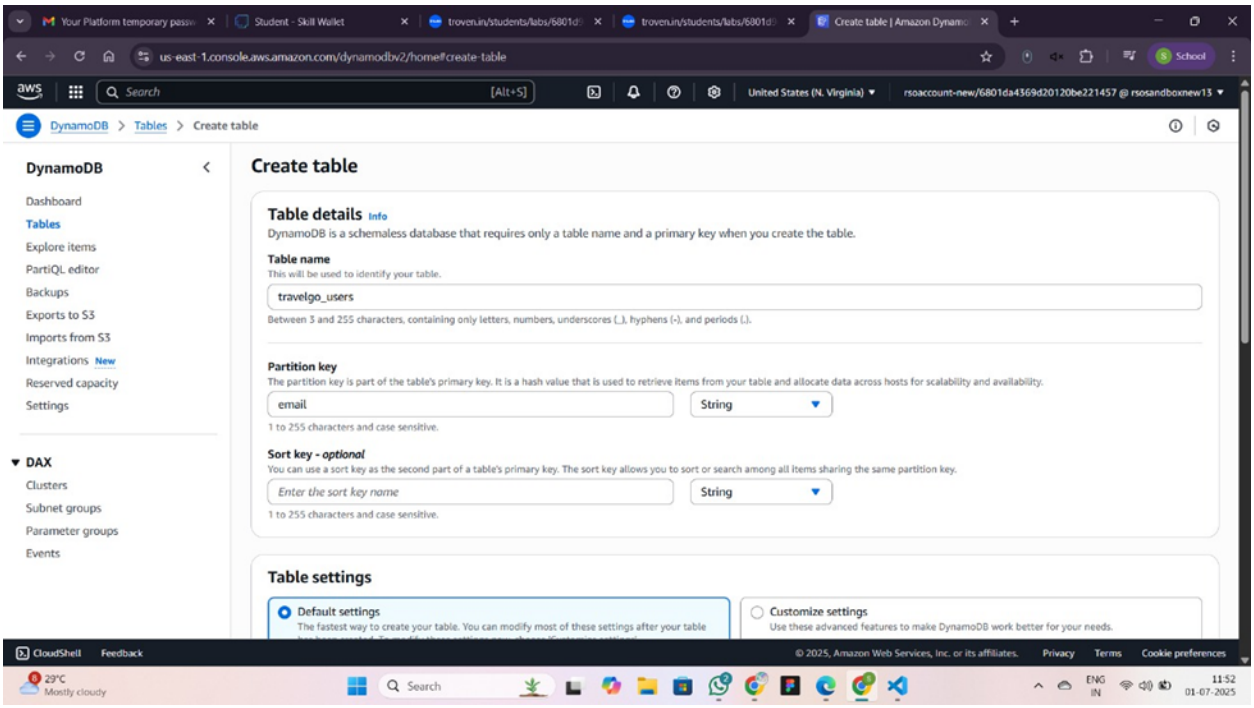




Create a DynamoDB table for storing details and book

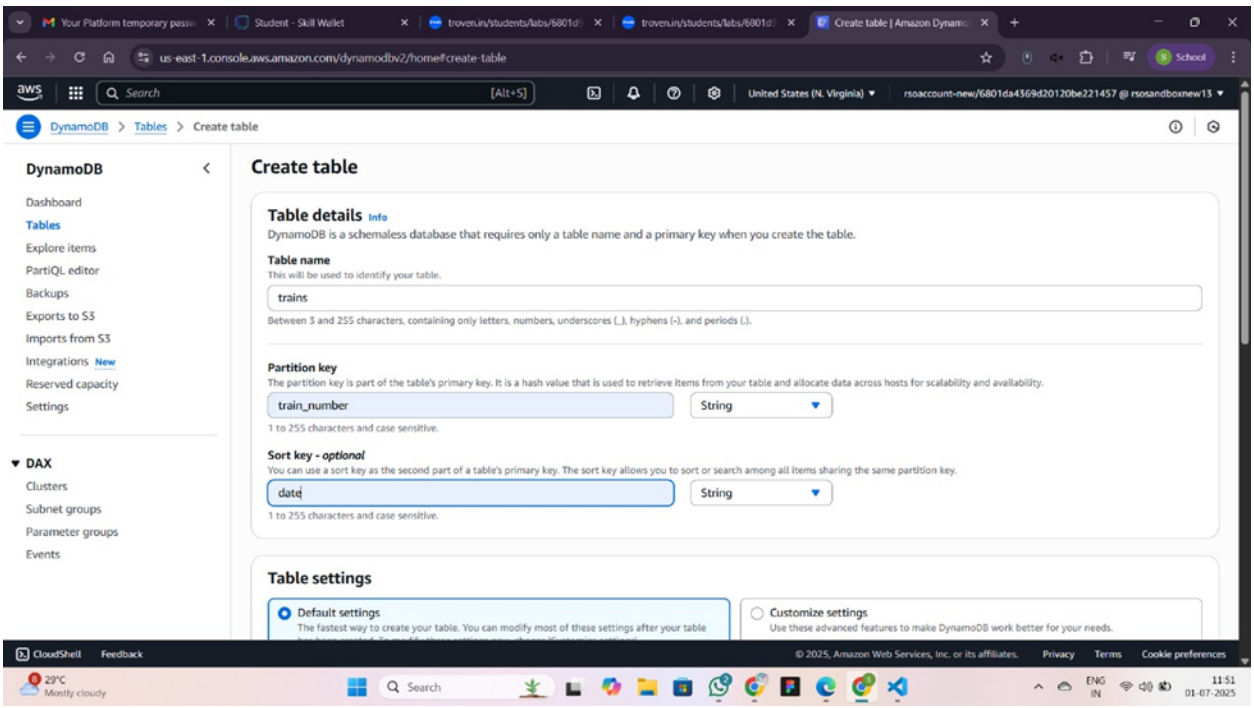
Requests

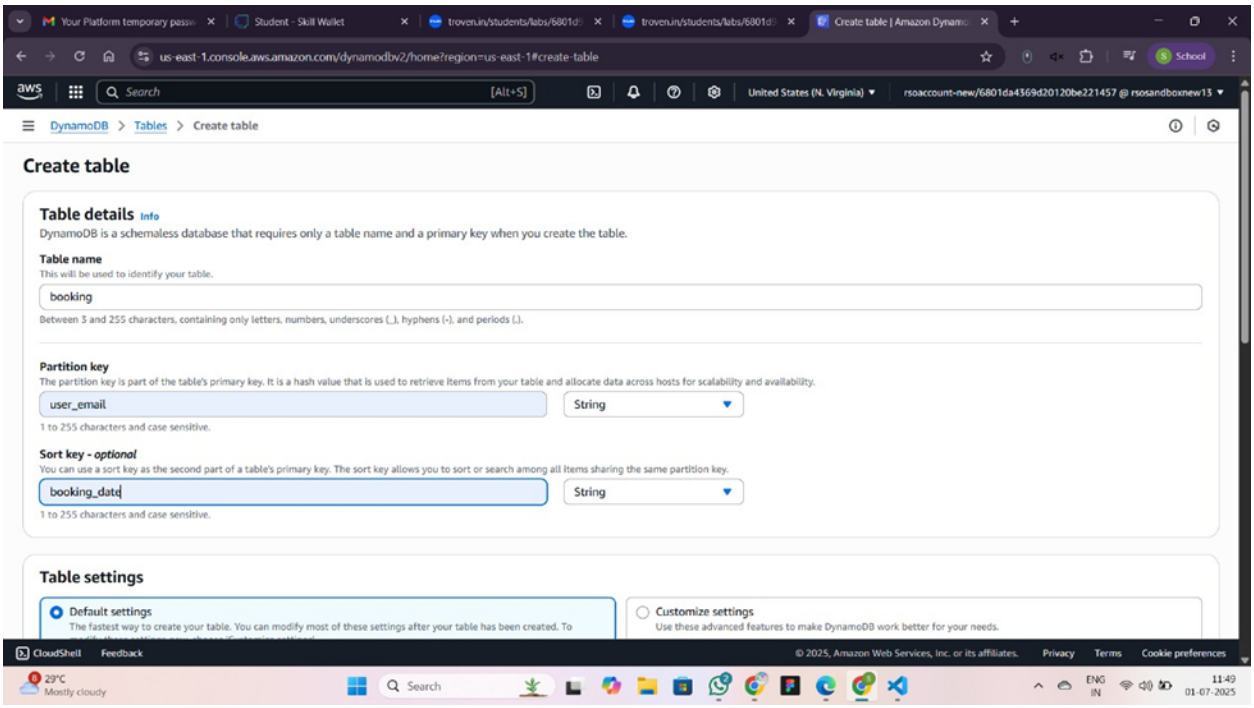
1. Create a Users table with partition key “Email” with type String and click on create tables.

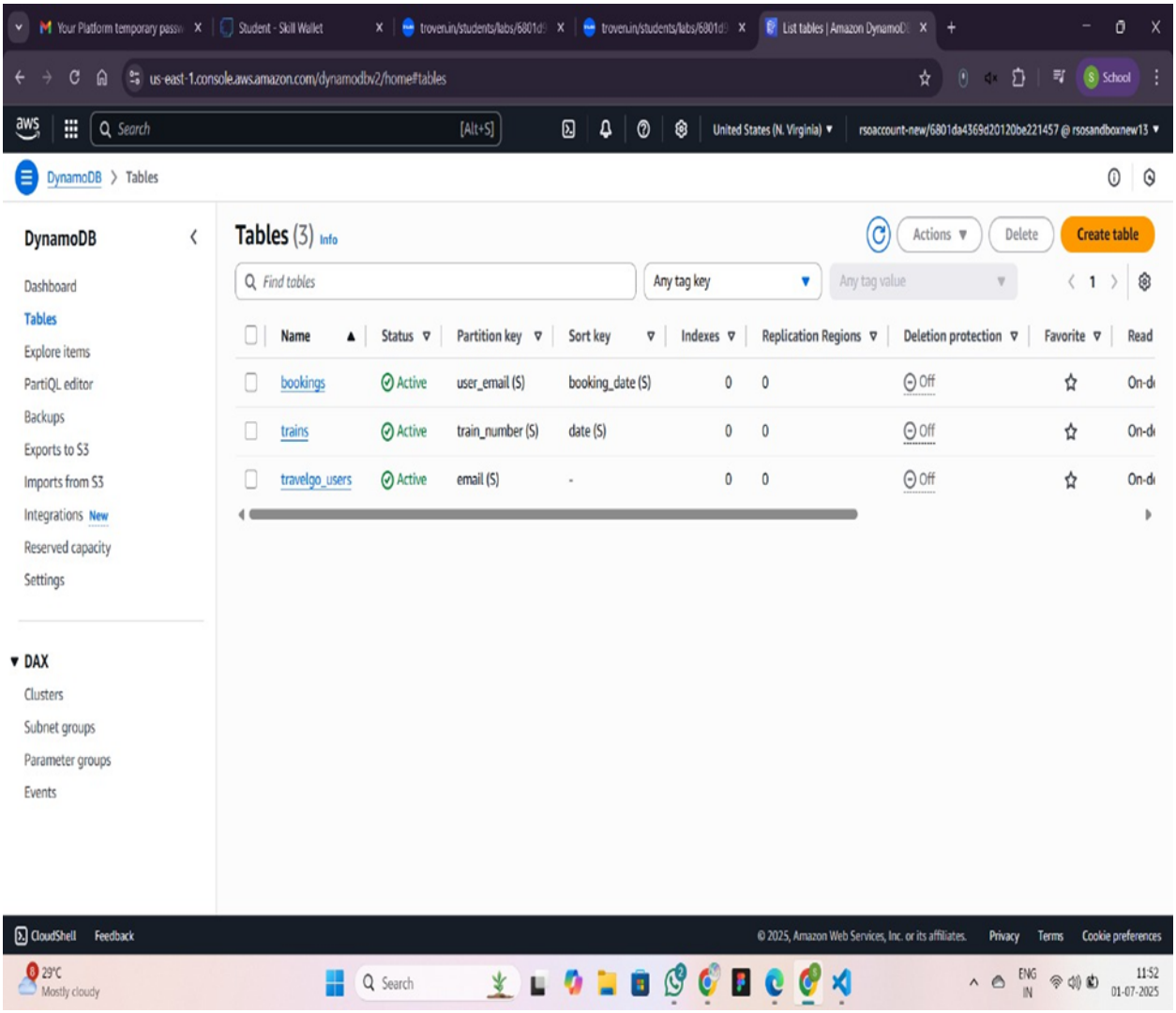


Follow the same steps to create a train table with train number as the partition key.

Create a Bookings table with the partition key set as the user's email and the sort key as the booking date.



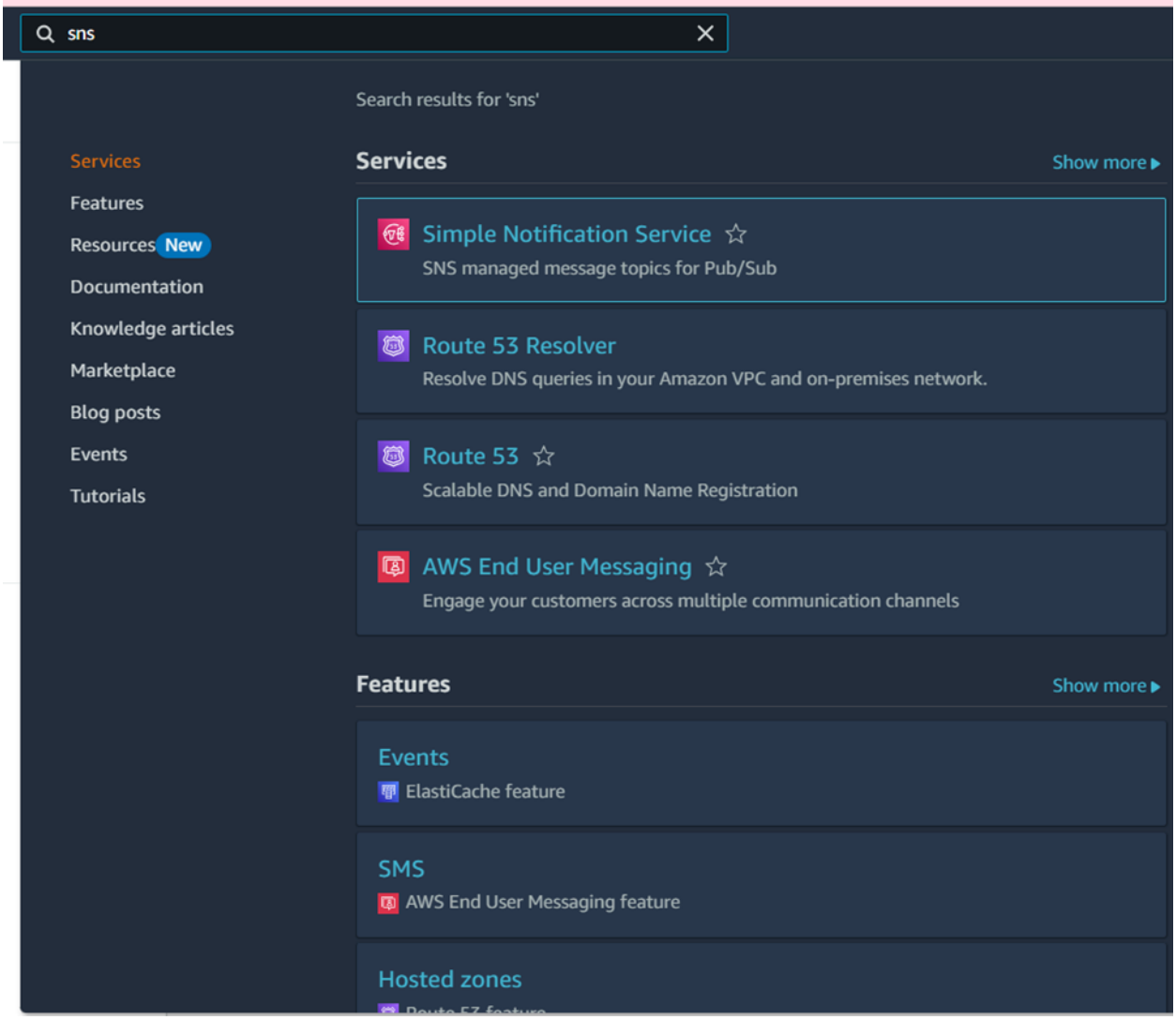

us-east-1.console.aws.amazon.com/dynamodbv2/home?region=us-east-1#create-table
DynamoDB > Tables > Create table
Create table
Table details Info
DynamoDB is a schemaless database that requires only a table name and a primary key when you create the table.
Table name
This will be used to identify your table.
booking
Between 3 and 255 characters, containing only letters, numbers, underscores (_), hyphens (-), and periods (.).
Partition key
The partition key is part of the table's primary key. It is a hash value that is used to retrieve items from your table and allocate data across hosts for scalability and availability.
user_email
String
1 to 255 characters and case sensitive.
Sort key - optional
You can use a sort key as the second part of a table's primary key. The sort key allows you to sort or search among all items sharing the same partition key.
booking_date
String
1 to 255 characters and case sensitive.
Table settings
Default settings
The fastest way to create your table. You can modify most of these settings after your table has been created. To
Customize settings
Use these advanced features to make DynamoDB work better for your needs.
CloudShell
Feedback
© 2025, Amazon Web Services, Inc. or its affiliates.
Privacy
Terms
Cookie preferences
29°C
Mostly cloudy
Search
ENG
IN
11:49
01-07-2025

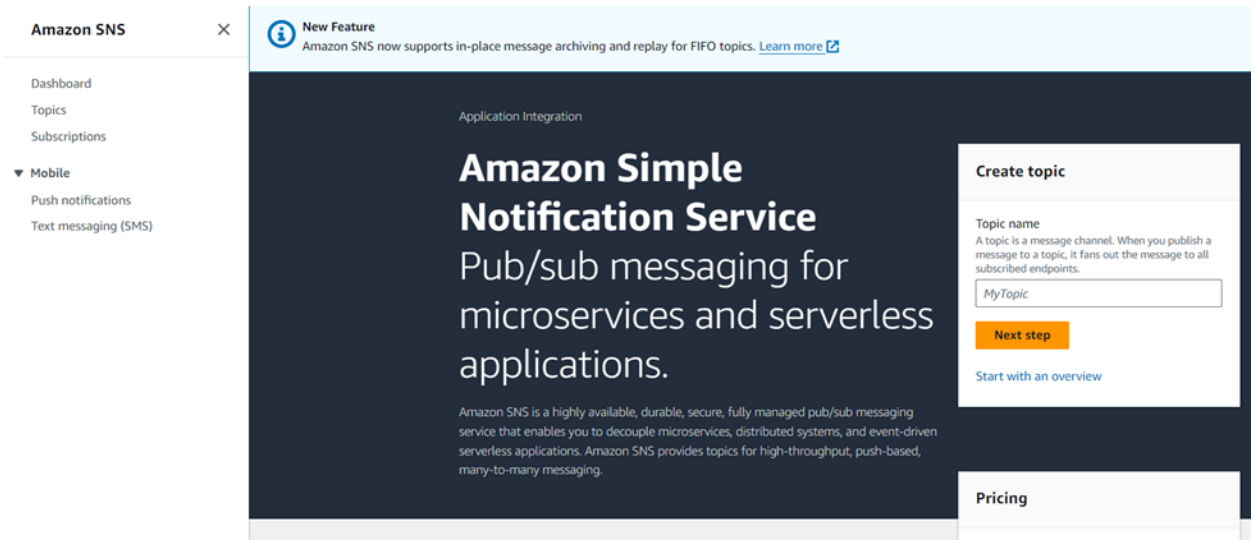


SNS Notified Setup

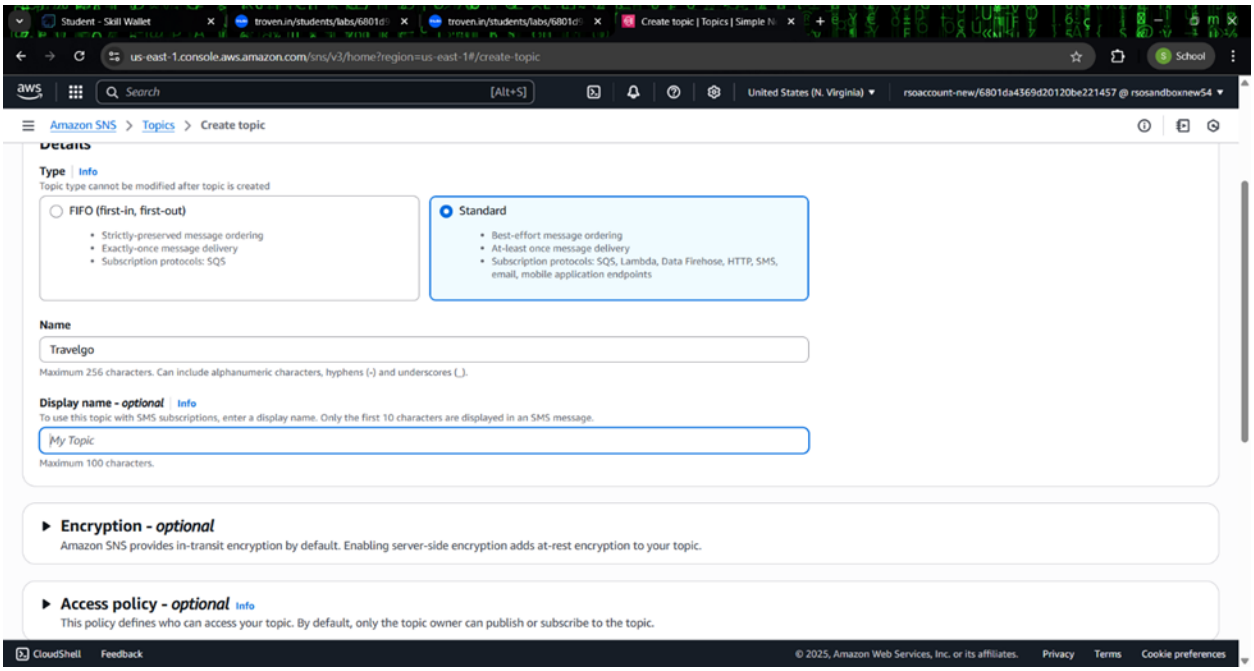
1. Create SNS topics for sending email to users.

2. In the AWS Console, search for “SNS” and go to the SNS dashboard.

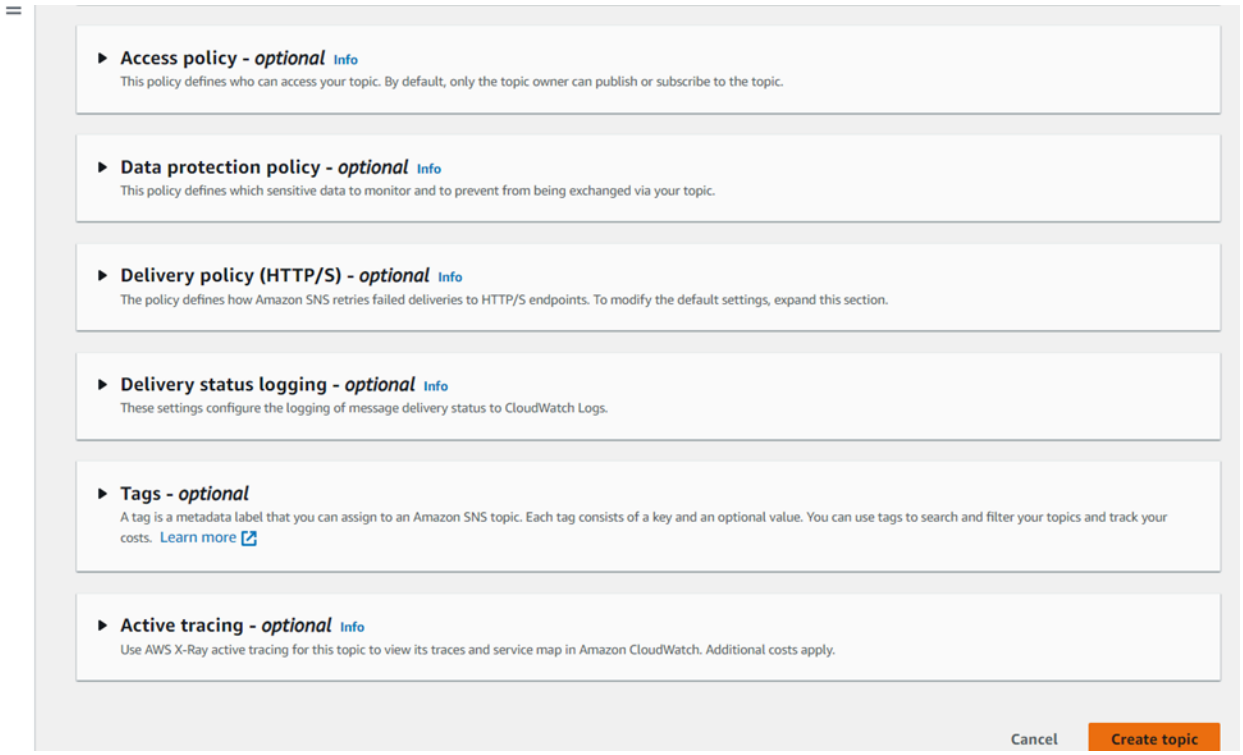

sns
Search results for 'sns'
Services
Features
Resources New
Documentation
Knowledge articles
Marketplace
Blog posts
Events
Tutorials
Services
Show more
Simple Notification Service
SNS managed message topics for Pub/Sub
Route 53 Resolver
Resolve DNS queries in your Amazon VPC and on-premises network.
Route 53
Scalable DNS and Domain Name Registration
AWS End User Messaging
Engage your customers across multiple communication channels
Features
Show more
Events
ElastiCache feature
SMS
AWS End User Messaging feature
Hosted zones


Amazon SNS
Dashboard
Topics
Subscriptions
Mobile
Push notifications
Text messaging (SMS)
New Feature
Amazon SNS now supports in-place message archiving and replay for FIFO topics. Learn more
Application Integration
Amazon Simple Notification Service
Pub/sub messaging for microservices and serverless applications.
Amazon SNS is a highly available, durable, secure, fully managed pub/sub messaging service that enables you to decouple microservices, distributed systems, and event-driven serverless applications. Amazon SNS provides topics for high-throughput, push-based, many-to-many messaging.
Create topic
Topic name
A topic is a message channel. When you publish a message to a topic, it fans out the message to all subscribed endpoints.
MyTopic
Next step
Start with an overview
Pricing

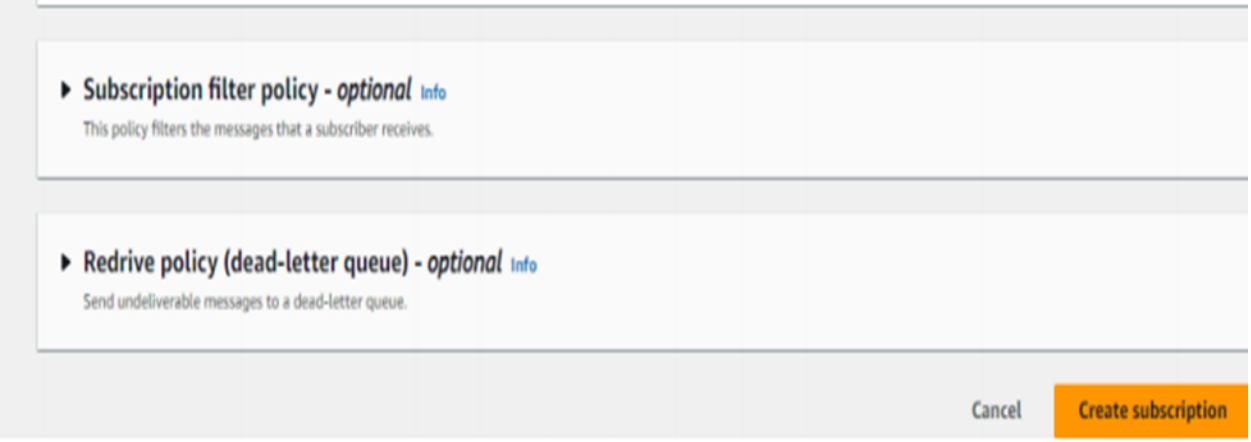
1. Click on **Create Topic** and choose a name for the topic



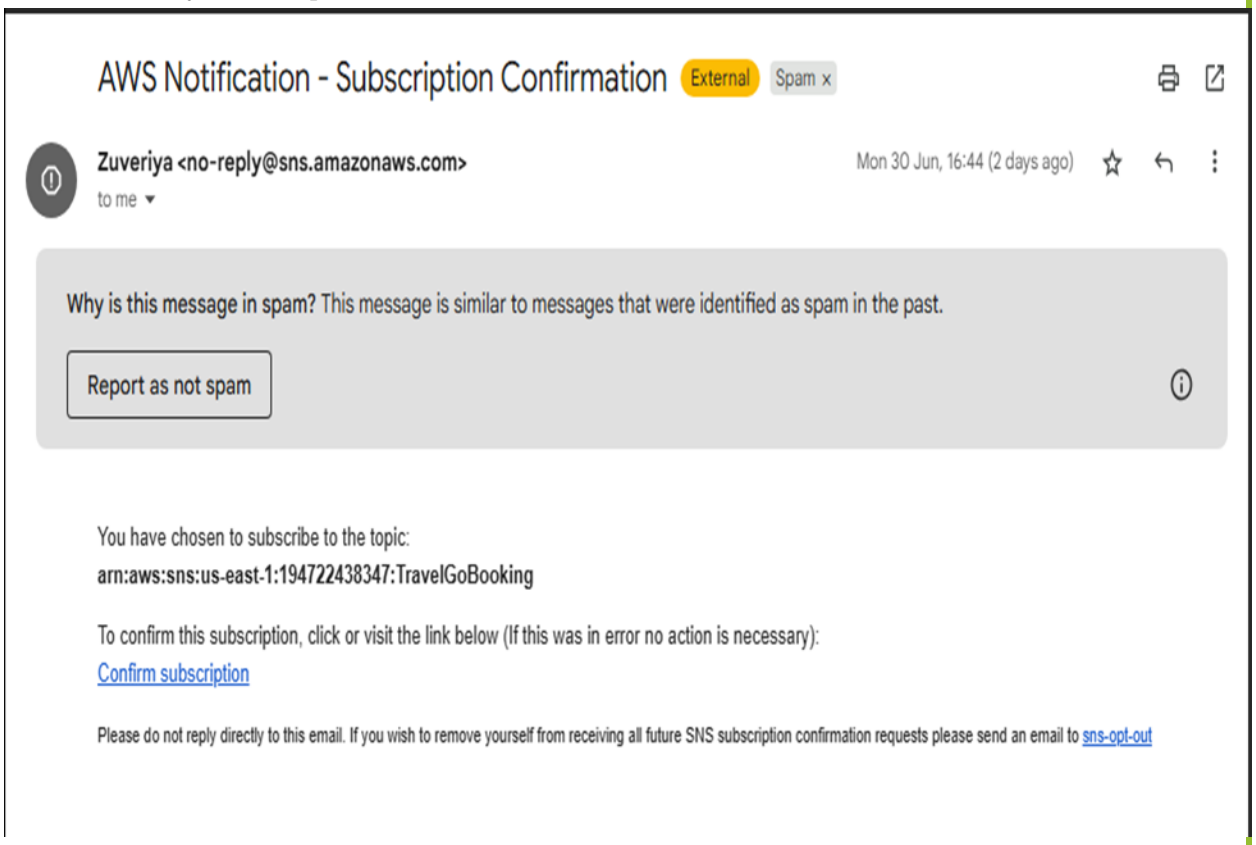
2. Click on create topic



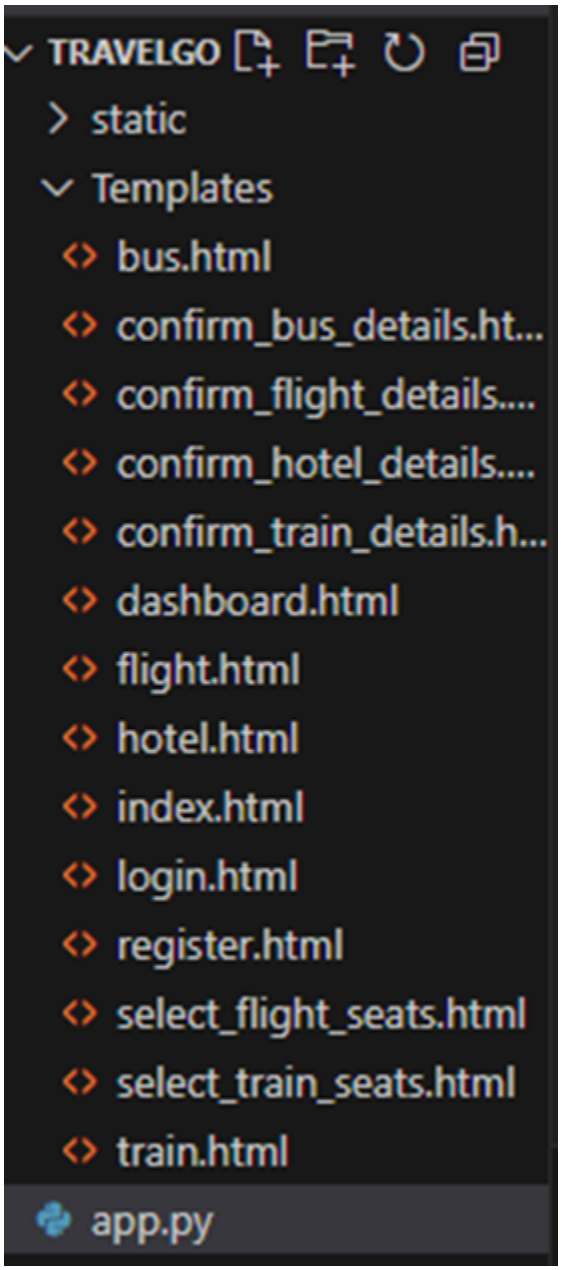
3. Configure the SNS topic and note down the **Topic ARN**.
4. Click on create subscription



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



Setup Backend Develop and App



Code Description:

```
from flask import Flask, render_template, request, redirect, url_for, session, jsonify
from pymongo import MongoClient
from werkzeug.security import generate_password_hash, check_password_hash
import json
from datetime import datetime
from bson.objectid import ObjectId # Still needed for other potential ObjectIds from DB
from bson.errors import InvalidId # Import this for specific error handling
```

 Flask: format the web app.

 render template: Renders HTML templates using Jinja2.

 request: Handles incoming data from forms or API calls.

 redirect and url_for: Used for redirect between pages (e.g., login → dashboard).

 session: Stores user-specific data across routes (e.g., user login session).
 personify: Converts Python data into JSON format (commonly used in API responses).

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

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

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

SNS connection:

```
# Function to send SNS notifications
# This function is duplicated in the original code, removing the duplicate.
def send_sns_notification(subject, message):
    try:
        sns_client.publish(
            TopicArn=SNS_TOPIC_ARN,
            Subject=subject,
            Message=message
        )
    except Exception as e:
        print(f"SNS Error: Could not send notification - {e}")
        # Optionally, flash an error message to the user or log it more robustly.
```

Routes:

- The `app.py` file in the TravelGo project manages all the backend routes for user interactions and booking functionality. It starts with **authentication routes** such as `/signup`, `/login`, and `/logout` to handle user registration, login, and logout functionalities.Once authenticated, users are redirected to the **dashboard** via the `/dashboard` route, which acts as a central hub for navigating through different booking options. The root route `/` usually leads to the landing or home page of the platform.The `/bus`, `/train`, `/flight`, and `/hotel` routes render respective booking pages where users can input their travel details.
- Each booking type has a check route like `/confirm_bus_details`, `/confirm_train_details`, etc., to validate and preview booking data.
- After check, final routes such as `/final_confirm_bus_booking` or `/final_confirm_flight_booking` handle storing the booking details in the database.

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

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

        # Check if user already exists
        # This uses get_item on the primary key 'email', so no GSI needed.
        existing = users_table.get_item(Key={'email': email})
        if 'Item' in existing:
            flash('Email already exists!', 'error')
            return render_template('register.html')

        # Hash password and store user
        hashed_password = generate_password_hash(password)
        users_table.put_item(Item={'email': email, 'password': hashed_password})
        flash('Registration successful! Please log in.', 'success')
        return redirect(url_for('login'))
    return render_template('register.html')
```

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

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

        # Authenticate user
        if 'Item' in user and check_password_hash(user['Item']['password'], password):
            session['email'] = email
            flash('Logged in successfully!', 'success')
            return redirect(url_for('dashboard'))
        else:
            flash('Invalid email or password!', 'error')
            return render_template('login.html')
    return render_template('login.html')

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

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

    # Query bookings for the logged-in user using the primary key 'user_email'
    # No GSI is needed here as 'user_email' is likely the partition key for the bookings_table.
    response = bookings_table.query(
        KeyConditionExpression=Key('user_email').eq(user_email),
        ScanIndexForward=False # Get most recent bookings first
    )
    bookings = response.get('Items', [])

    # Convert Decimal types from DynamoDB to float for display if necessary
    for booking in bookings:
        if 'total_price' in booking:
            try:
                booking['total_price'] = float(booking['total_price'])
            except (TypeError, ValueError):
                booking['total_price'] = 0.0 # Default value if conversion fails
    return render_template('dashboard.html', username=user_email, bookings=bookings)
```

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

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

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

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

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

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

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

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

Deployment Code:

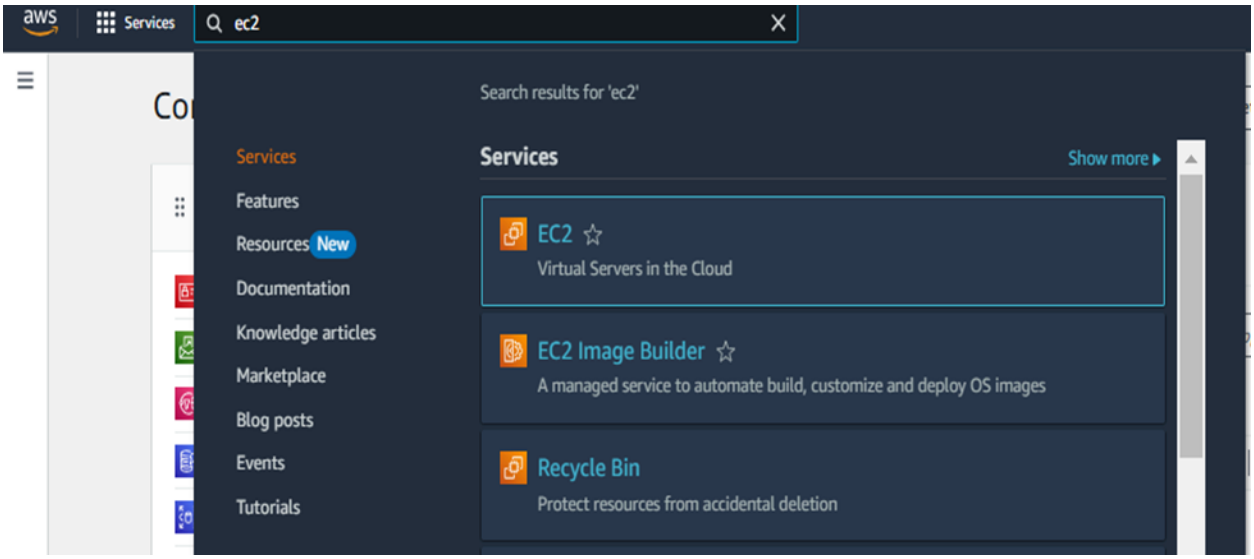
```python
if __name__ == '__main__':
    # IMPORTANT: In a production environment, disable debug mode and specify a production-ready host.
    app.run(debug=True, host='0.0.0.0')
```

start the Flask server to listen on all network port (0.0.0.0) at port 5000 with debug mode enabled for testing

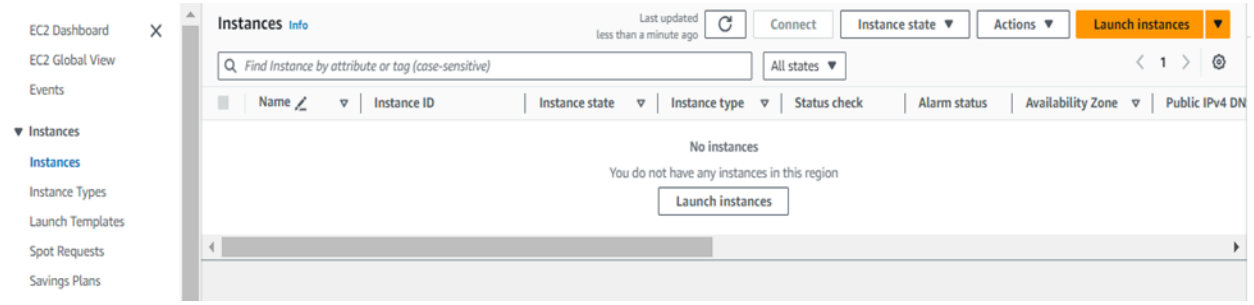
Setup EC2 Instance:

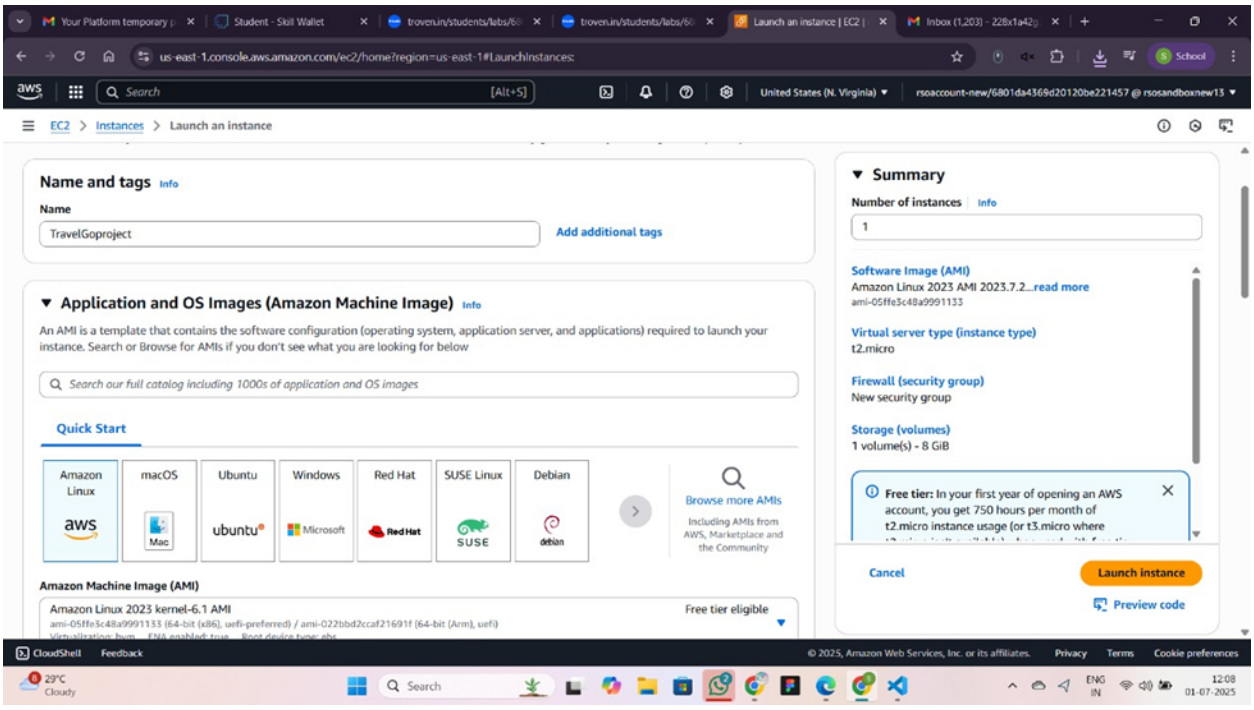
static	Initial commit
templates	Update statistics.html
app.py	Update app.py

Launch an EC2 instance to host the Flask app.

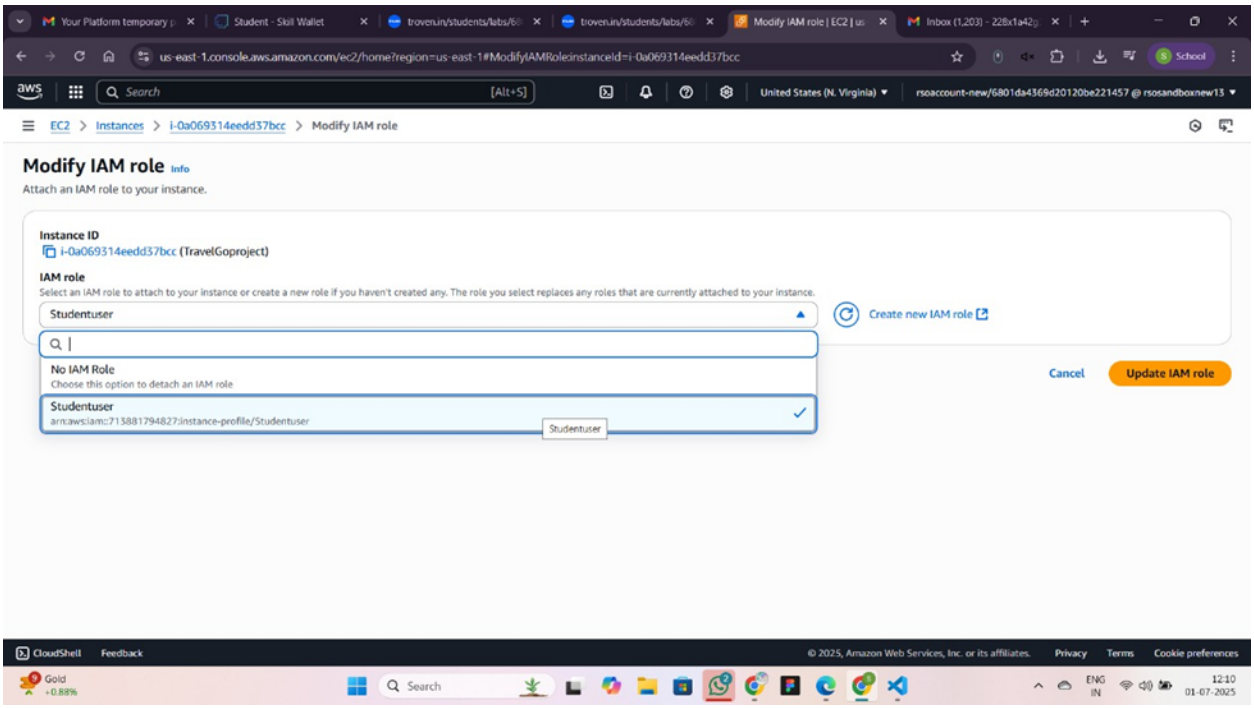


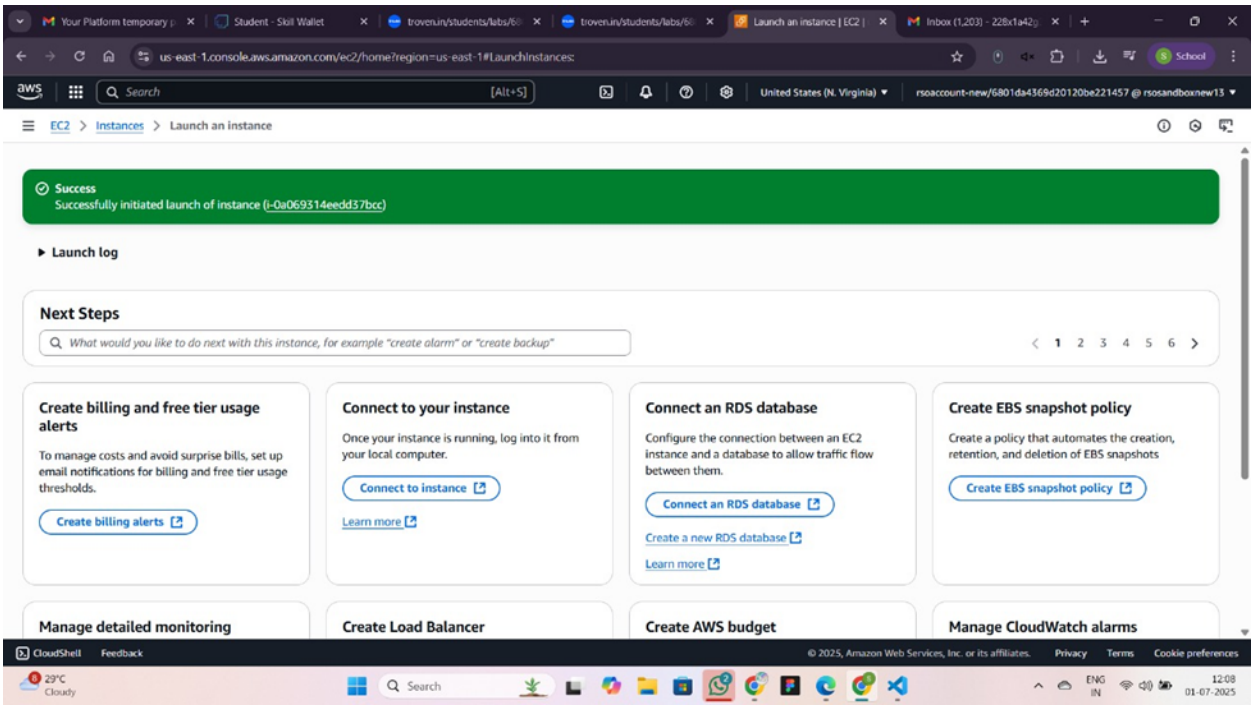
1. Click on launch Instance:





Modify IAM Roles”





Create Key Pair:

Instance type Info | Get advice

Instance type

t2.micro — Free tier eligible

Family: t2 1 vCPU 1 GiB Memory Current generation: true
On-Demand Linux base pricing: 0.0124 USD per Hour
On-Demand Windows base pricing: 0.017 USD per Hour
On-Demand RHEL base pricing: 0.0268 USD per Hour
On-Demand SUSE base pricing: 0.0124 USD per Hour

All generations

Compare instance types

Additional costs apply for AMIs with pre-installed software

Key pair (login) Info

You can use a key pair to securely connect to your instance. Ensure that you have access to the selected key pair before you launch the instance.

Key pair name - *required*

Select

Create new key pair

Key pair type

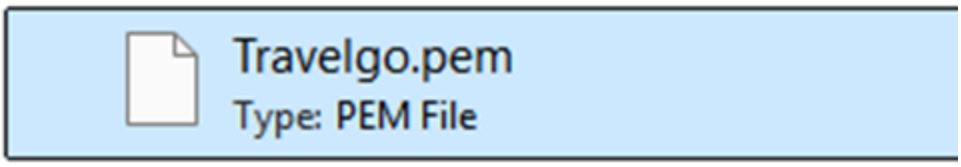
- RSA — RSA encrypted private and public key pair
- ED25519 — ED25519 encrypted private and public key pair

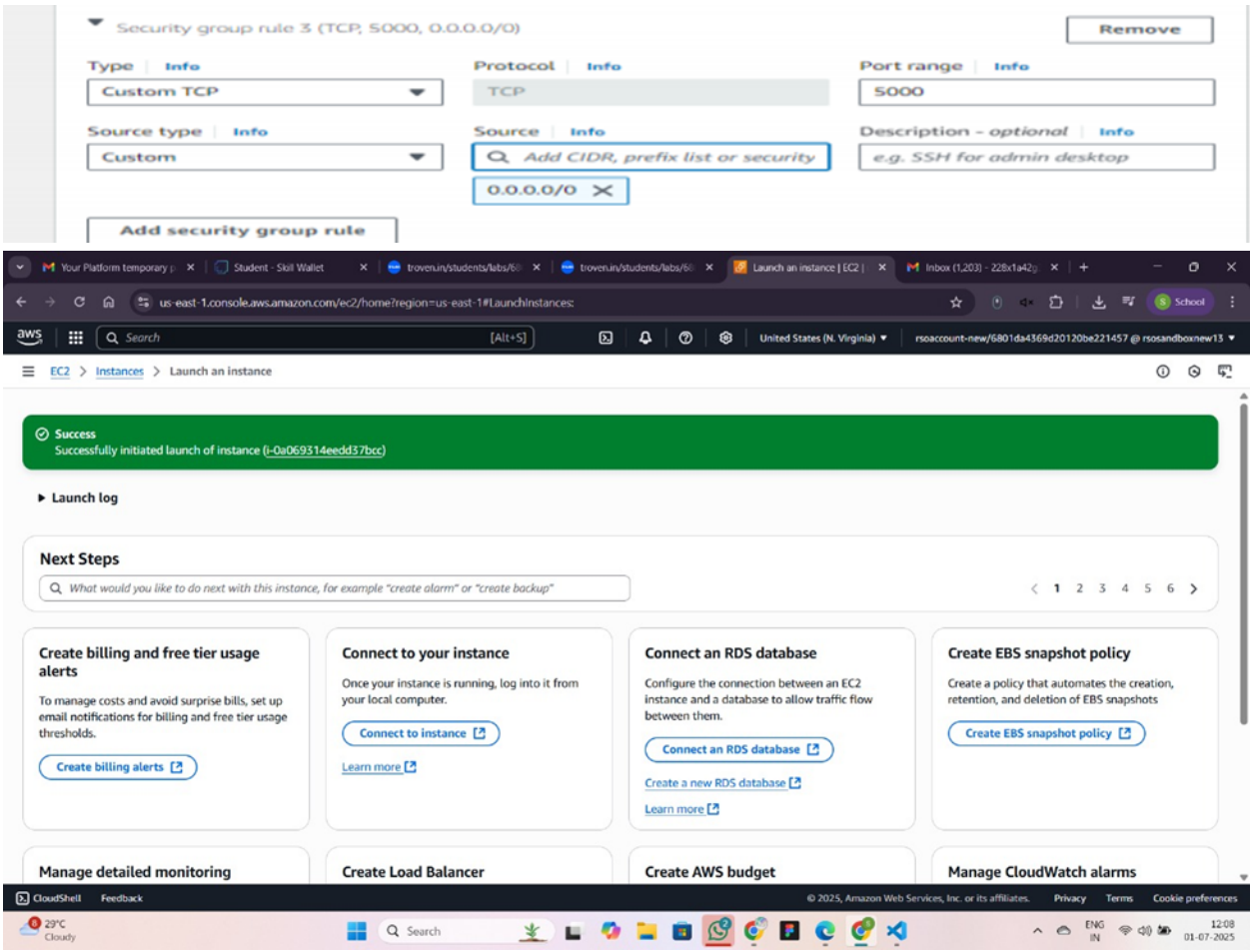
Private key file format

- .pem — For use with OpenSSH
- .ppk — For use with PuTTY

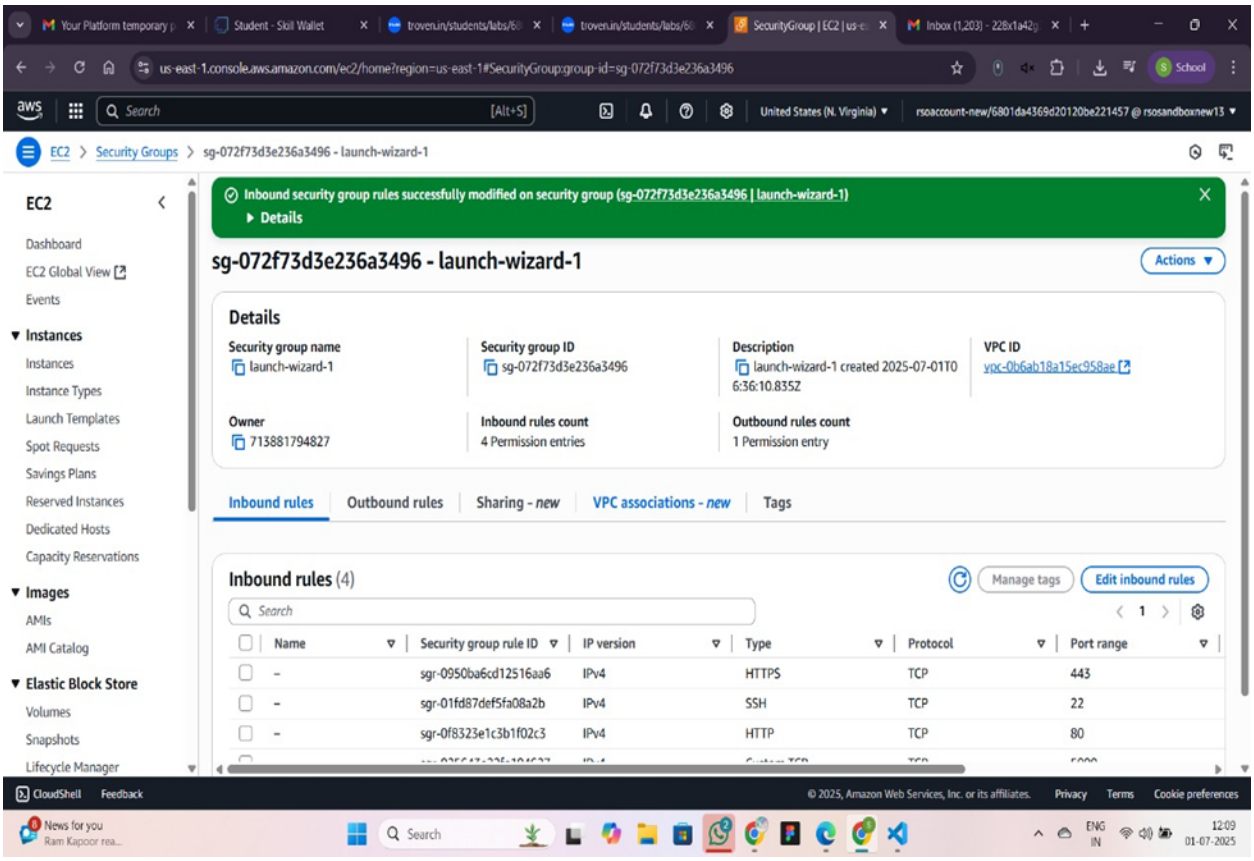
⚠ When prompted, store the private key in a secure and accessible location on your computer. **You will need it later to connect to your instance.** Learn more

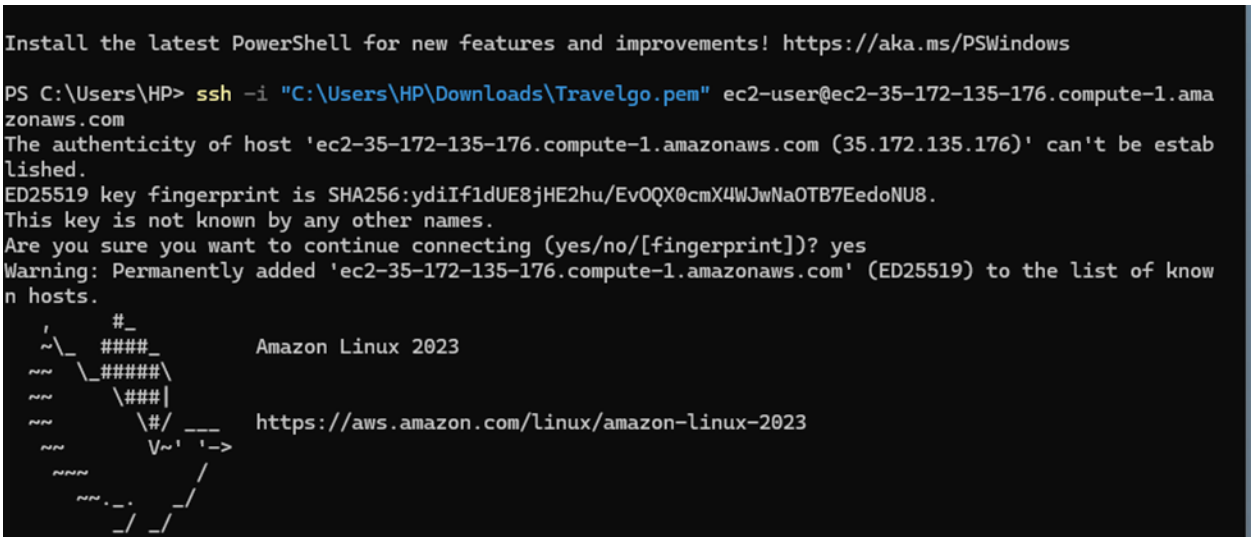
Cancel | Create key pair





Start by attaching an IAM role to your EC2 instance to connect using EC2 Instance Connect. You can do this by selecting your instance, clicking on **Actions**, then navigate to **surety** and selecting **Modify IAM Role** to attach the suitable role.c. Select the **EC2 instance** you wish to connect to. In the top of the **EC2 Dashboard**, click the **Connect** button. From the menu, 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





First, install and set up the AWS CLI with your credentials to connect to an AWS EC2 instance using PowerShell.The EC2 instance must have a public IP address, port 22 open in its protection group (for Linux), and the correct IAM role attached if you're using EC2 Instance Connect.

Next, open PowerShell and use the SSH command with your `.pem` key file

to connect. Before that, ensure the.`pem` file has the correct permissions to avoid access errors.

If you are using EC2 Instance Connect, you can connect directly through the AWS Console or by sending your SSH public key using a CLI command.

This process allows you to securely access your EC2 instance from your local machine using PowerShell.

```
[ec2-user@ip-172-31-90-127 ~]$ sudo yum install git -y
Amazon Linux 2023 Kernel Livepatch repository                                    154 kB/s |  1
7 kB     00:00
Dependencies resolved.
================================================================================================
 Package                    Architecture       Version                        Repository
            Size
================================================================================================
Installing:
 git                        x86_64             2.47.1-1.amzn2023.0.3          amazonlinux
            52 k
Installing dependencies:
 git-core                   x86_64             2.47.1-1.amzn2023.0.3          amazonlinux
           4.5 M
 git-core-doc               noarch             2.47.1-1.amzn2023.0.3          amazonlinux
           2.8 M
 perl-Error                 noarch             1:0.17029-5.amzn2023.0.2       amazonlinux
            41 k
 perl-File-Find             noarch             1.37-477.amzn2023.0.7          amazonlinux
            25 k
 perl-Git                   noarch             2.47.1-1.amzn2023.0.3          amazonlinux
            40 k
 perl-TermReadKey           x86_64             2.38-9.amzn2023.0.2            amazonlinux
            36 k
```

Once you've connected to your EC2 instance using PowerShell, the next step is to install the required Python libraries. These include Flask for building the web app and PyMongo for connecting to MongoDB. You can install them by running the suitable `pip` commands.

If your project includes a `requirements.txt` file, it's recommend to use that file to install all essential libraries in one go. This helps ensure that the EC2 surroundings have every needed to run your app smoothly.

```
Complete!
[ec2-user@ip-172-31-90-127 ~]$ git clone https://github.com/Zuveriya2527/TravelGo.git
Cloning into 'TravelGo'...
remote: Enumerating objects: 41, done.
remote: Counting objects: 100% (41/41), done.
remote: Compressing objects: 100% (33/33), done.
remote: Total 41 (delta 6), reused 34 (delta 3), pack-reused 0 (from 0)
Receiving objects: 100% (41/41), 2.46 MiB | 33.14 MiB/s, done.
Resolving deltas: 100% (6/6), done.
[ec2-user@ip-172-31-90-127 ~]$ cd TravelGo
[ec2-user@ip-172-31-90-127 TravelGo]$ sudo yum install python3 -y

sudo yum install python3-pip -y

pip install flask

pip install boto3
Last metadata expiration check: 0:03:23 ago on Mon Jun 30 11:29:21 2025.
Package python3-3.9.23-1.amzn2023.0.1.x86_64 is already installed.
Dependencies resolved.
Nothing to do.
Complete!
Last metadata expiration check: 0:03:23 ago on Mon Jun 30 11:29:21 2025.
Dependencies resolved.
================================================================================
 Package                 Architecture   Version                          Repository           Size
================================================================================
Installing:
```

```
Requirement already satisfied: python-dateutil<3.0.0,>=2.1 in /usr/lib/python3.9/site-packages (from
botocore<1.39.0,>=1.38.46->boto3) (2.8.1)
Requirement already satisfied: urllib3<1.27,>=1.25.4 in /usr/lib/python3.9/site-packages (from botoco
re<1.39.0,>=1.38.46->boto3) (1.25.10)
Requirement already satisfied: six>=1.5 in /usr/lib/python3.9/site-packages (from python-dateutil<3.0
.0,>=2.1->botocore<1.39.0,>=1.38.46->boto3) (1.15.0)
Installing collected packages: botocore, s3transfer, boto3
Successfully installed boto3-1.38.46 botocore-1.38.46 s3transfer-0.13.0
[ec2-user@ip-172-31-90-127 TravelGo]$ python3 app.py
python3: can't open file '/home/ec2-user/TravelGo/app.py': [Errno 2] No such file or directory
[ec2-user@ip-172-31-90-127 TravelGo]$ ls -la
total 4
drwxr-xr-x. 4 ec2-user ec2-user  49 Jun 30 11:31 .
drwx------. 6 ec2-user ec2-user 118 Jun 30 11:32 ..
drwxr-xr-x. 8 ec2-user ec2-user 163 Jun 30 11:31 .git
-rw-r--r--. 1 ec2-user ec2-user  26 Jun 30 11:31 README.md
drwxr-xr-x. 4 ec2-user ec2-user  86 Jun 30 11:31 Travel
[ec2-user@ip-172-31-90-127 TravelGo]$ cd Travel
[ec2-user@ip-172-31-90-127 Travel]$ ls -la
total 48
drwxr-xr-x. 4 ec2-user ec2-user    86 Jun 30 11:31 .
drwxr-xr-x. 4 ec2-user ec2-user    49 Jun 30 11:31 ..
-rw-r--r--. 1 ec2-user ec2-user    46 Jun 30 11:31 .gitignore
-rw-r--r--. 1 ec2-user ec2-user   158 Jun 30 11:31 README.md
-rw-r--r--. 1 ec2-user ec2-user 22920 Jun 30 11:31 app.py
drwxr-xr-x. 4 ec2-user ec2-user    47 Jun 30 11:31 static
drwxr-xr-x. 2 ec2-user ec2-user 16384 Jun 30 11:31 templates
```

```
Last login: Mon Jun 30 11:28:37 2025 from 103.174.110.191
[ec2-user@ip-172-31-90-127 ~]$ cd TravelGo/Travel
[ec2-user@ip-172-31-90-127 Travel]$ source venv/bin/activate
-bash: venv/bin/activate: No such file or directory
[ec2-user@ip-172-31-90-127 Travel]$ sudo yum update -y
sudo yum install python3 -y
Last metadata expiration check: 0:21:27 ago on Mon Jun 30 11:29:21 2025.
Dependencies resolved.
Nothing to do.
Complete!
Last metadata expiration check: 0:21:28 ago on Mon Jun 30 11:29:21 2025.
Package python3-3.9.23-1.amzn2023.0.1.x86_64 is already installed.
Dependencies resolved.
Nothing to do.
Complete!
[ec2-user@ip-172-31-90-127 Travel]$ python3 -m venv venv
[ec2-user@ip-172-31-90-127 Travel]$ source venv/bin/activate
(venv) [ec2-user@ip-172-31-90-127 Travel]$ pip install flask boto3 pymongo
Collecting flask
  Using cached flask-3.1.1-py3-none-any.whl (103 kB)
Collecting boto3
  Using cached boto3-1.38.46-py3-none-any.whl (139 kB)
Collecting pymongo
  Downloading pymongo-4.13.2-cp39-cp39-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (938 kB)
     |                                | 938 kB 15.1 MB/s
Collecting markupsafe>=2.1.1
  Using cached MarkupSafe-3.0.2-cp39-cp39-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (20 kB)
Collecting jinja2>=3.1.2
```

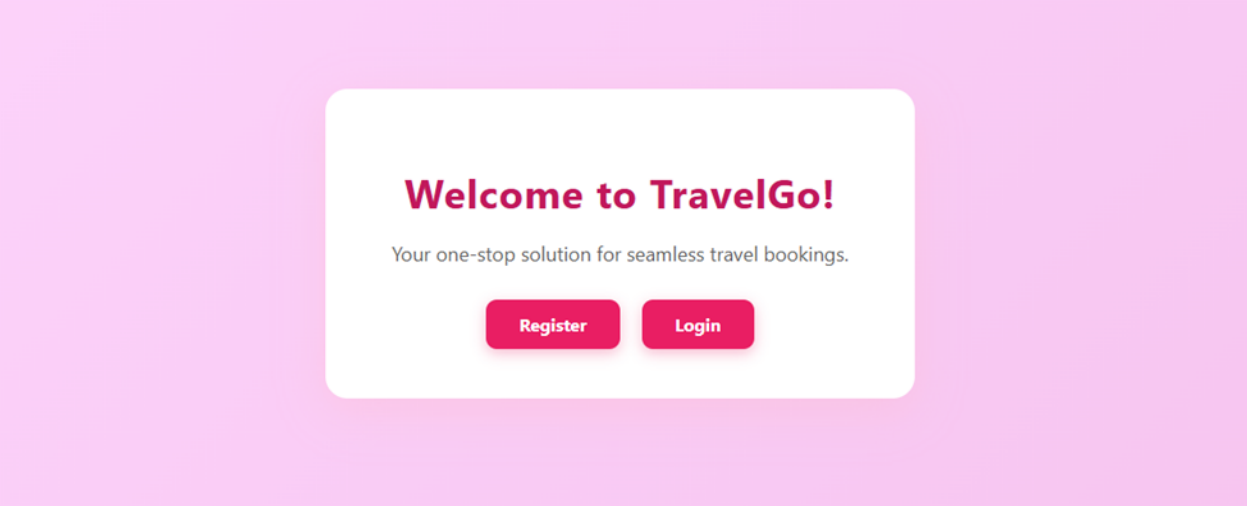
```
(venv) [ec2-user@ip-172-31-90-127 Travel]$ python app.py
 * Serving Flask app 'app'
 * Debug mode: on
Address already in use
Port 5000 is in use by another program. Either identify and stop that program, or start the server wi
th a different port.
(venv) [ec2-user@ip-172-31-90-127 Travel]$ sudo lsof -i :5000
COMMAND   PID     USER   FD   TYPE DEVICE SIZE/OFF NODE NAME
python3 27007 ec2-user    6u  IPv4  44850      0t0  TCP *:commplex-main (LISTEN)
python3 27008 ec2-user    6u  IPv4  44850      0t0  TCP *:commplex-main (LISTEN)
python3 27008 ec2-user    7u  IPv4  44850      0t0  TCP *:commplex-main (LISTEN)
(venv) [ec2-user@ip-172-31-90-127 Travel]$ sudo kill -9 27007 27008
(venv) [ec2-user@ip-172-31-90-127 Travel]$ python 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 WSG
I server instead.
 * Running on all addresses (0.0.0.0)
 * Running on http://127.0.0.1:5000
 * Running on http://172.31.90.127:5000
Press CTRL+C to quit
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 844-840-745
Client_loop: send disconnect: Connection reset
```

Checking Testing and Deployment

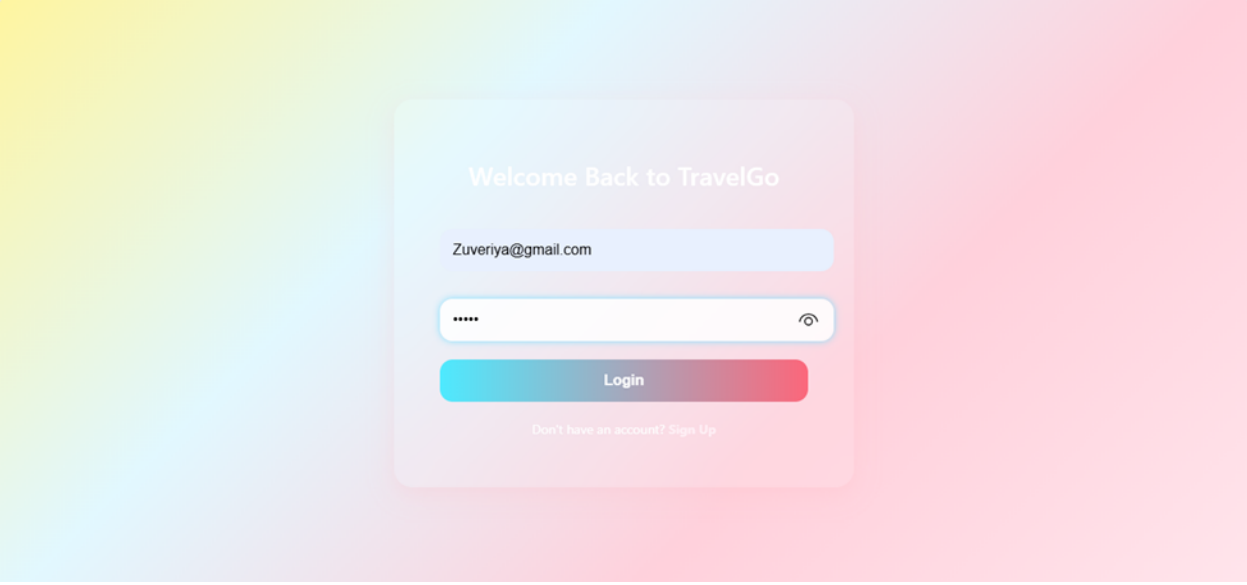
Home Page:

The **Home Page** of TravelGo serves as the entry point for users, providing a simple and intuitive interface to explore travel booking

options. It encloses the platform's core features—allowing users to book buses, trains, flights, and hotels—all from one place. The page includes piloting to log in or sign up, along with quick links to useable booking services. Designed for ease of use, the home page highlights TravelGo's goal of making travel planning fast.



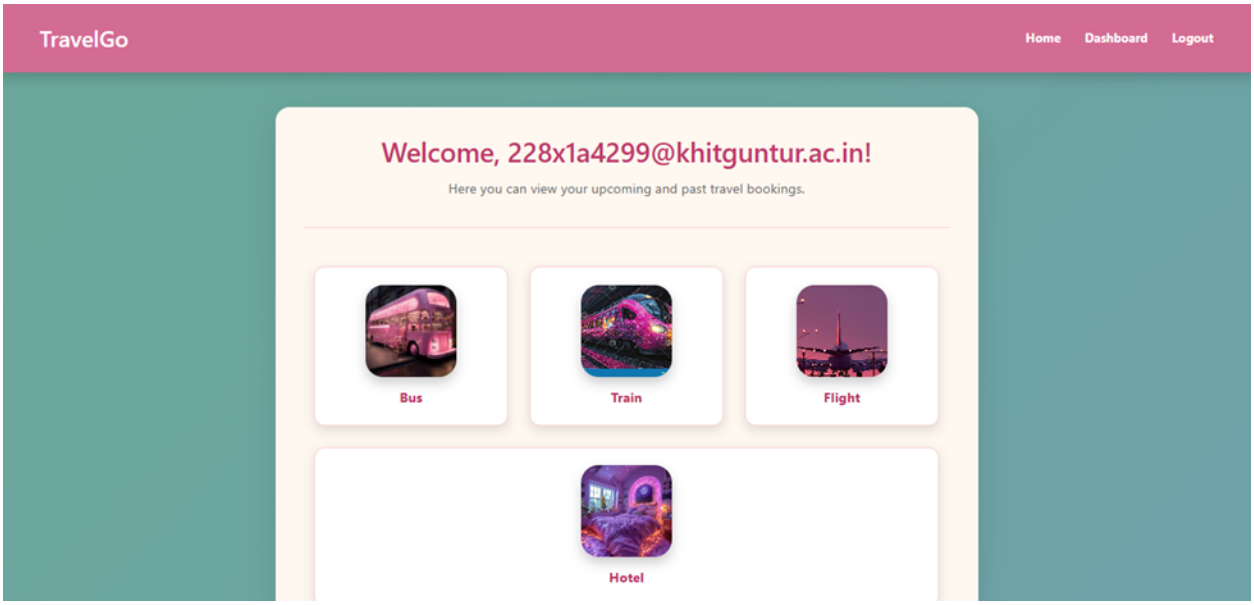
Login Page:



Dashboard:

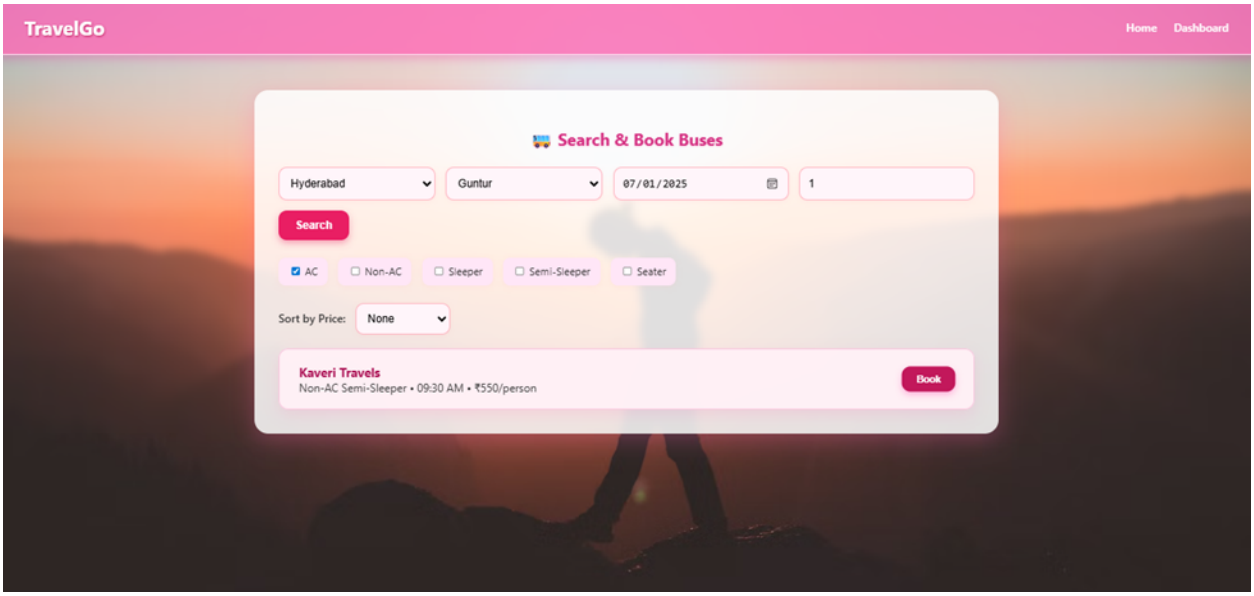
The **Dashboard Page** provides users with a personal interface to manage their travel action. After login in, users can choose to book buses, trains, flights, or hotels from the dashboard. It displays recent bookings, upcoming

trips, and account details in a clean, organized layout. The dashboard serves as a central hub, allowing users to track bookings, novice cancel, and receive updates—all in one convenient location.



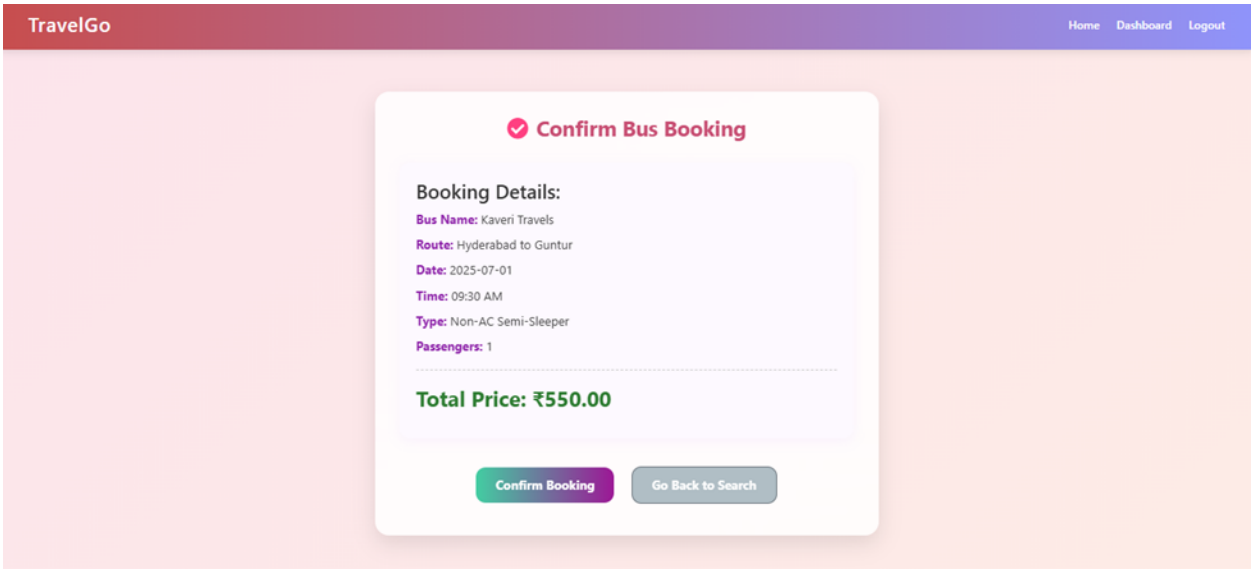
Bookings:

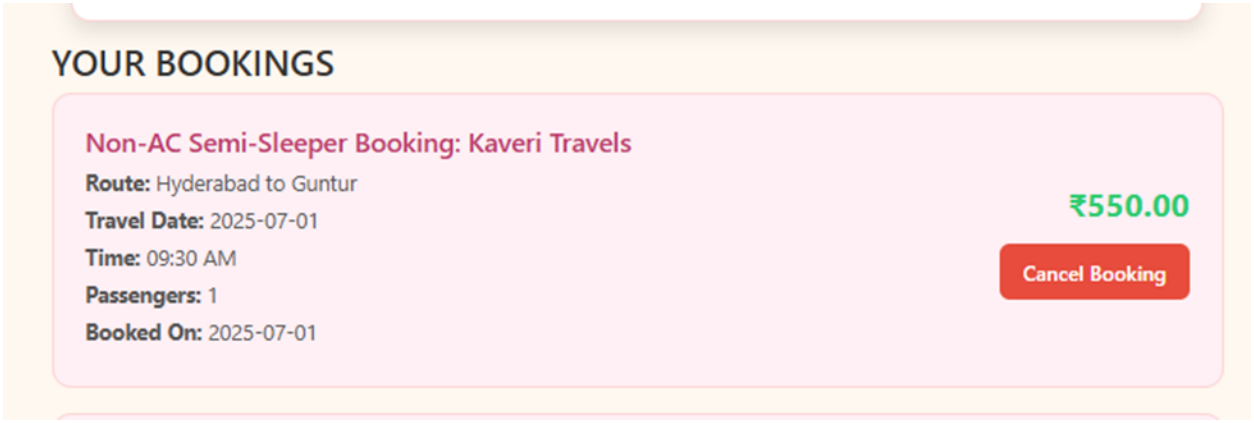
- Bus



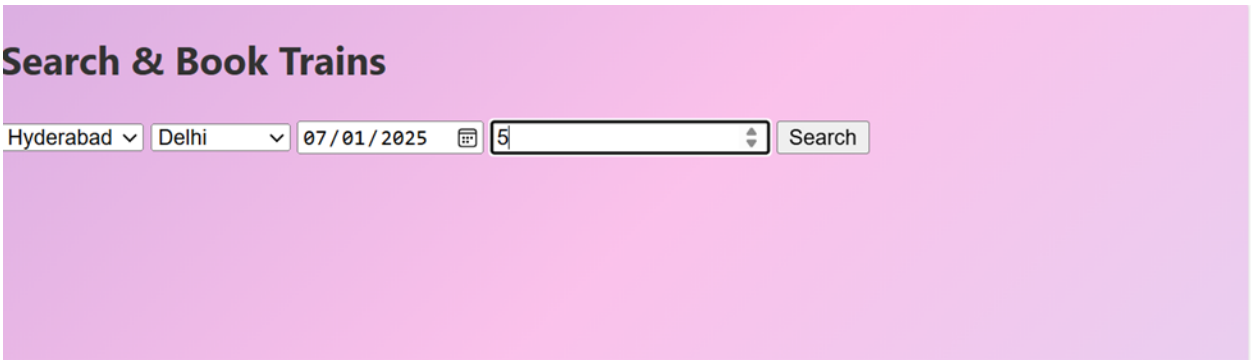
The **Bus Booking Page** allows users to search and book bus tickets by entering details such as the source, goal, journey date, and number of passengers. After users submit the form, the system displays available bus options, and users can select their preferred

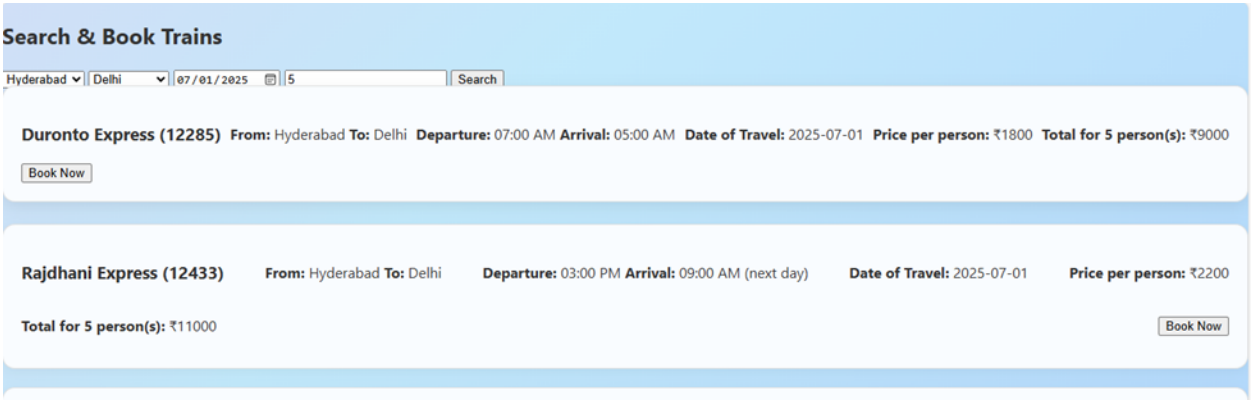
service After selection, users proceed to confirm seat numbers and fare details. The page ensures a smooth and intuitive booking flow, helping users reserve bus tickets and efficient within the TravelGo platform.

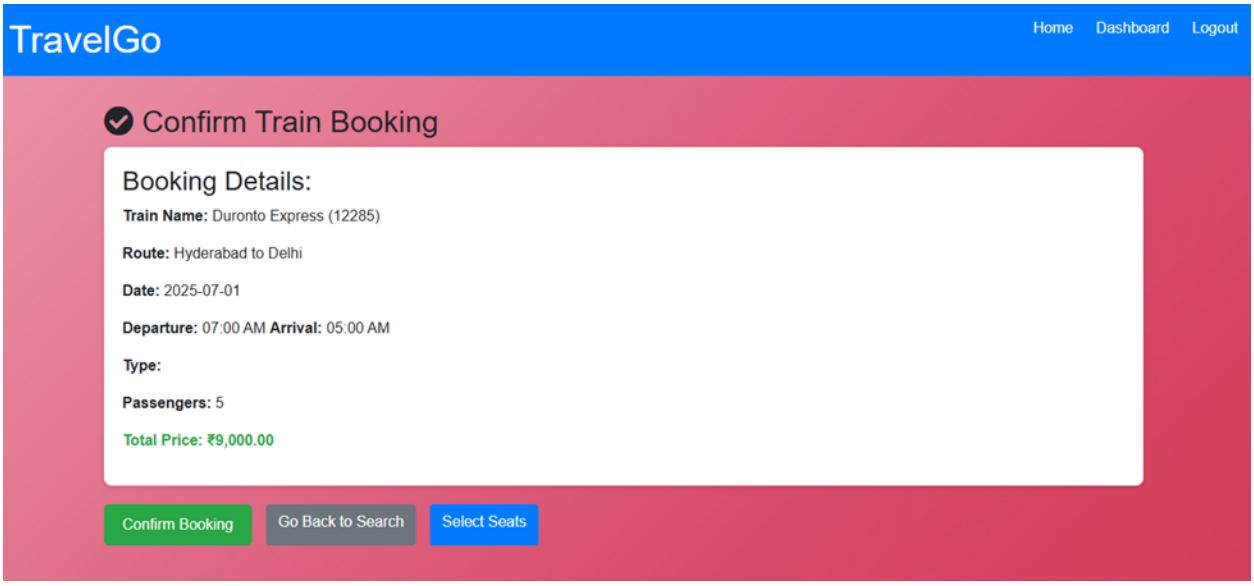




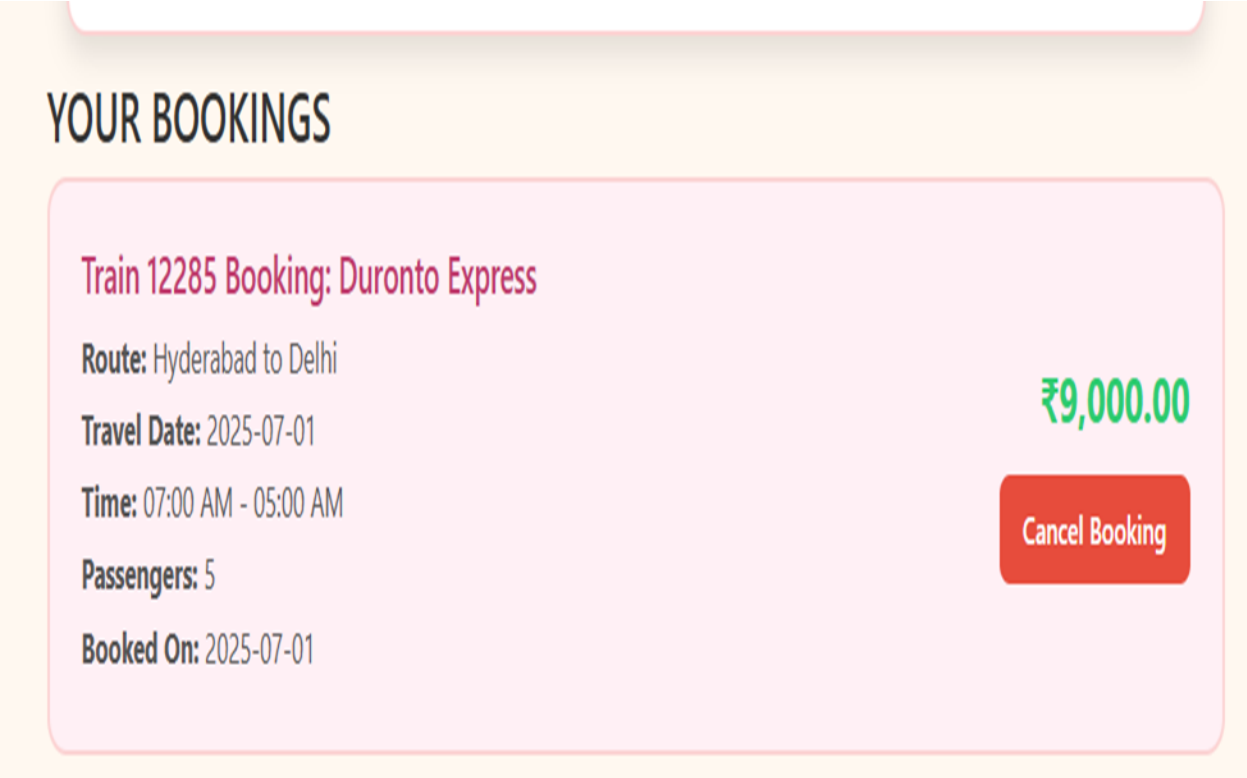
- Trains



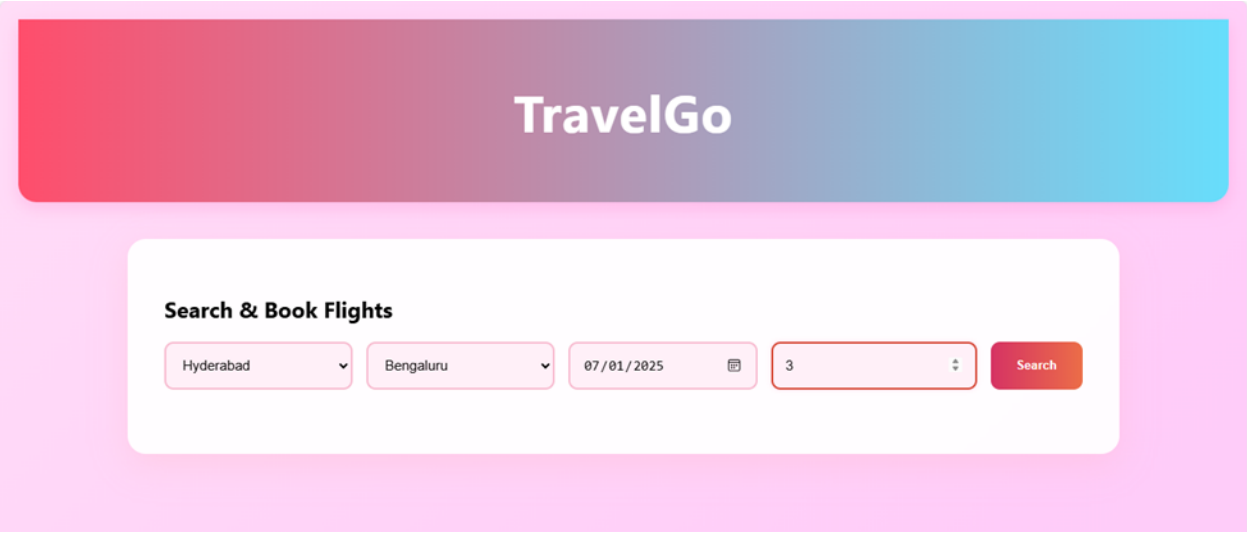




The **Train Booking Page** enables users to book train tickets by entering essential travel details such as source, goal, date of journey, and number of passengers. Based on the input, the system displays useable trains with options to view timing, class, and fare. Users can select a preferred train, choose seats, and proceed to confirm the booking.



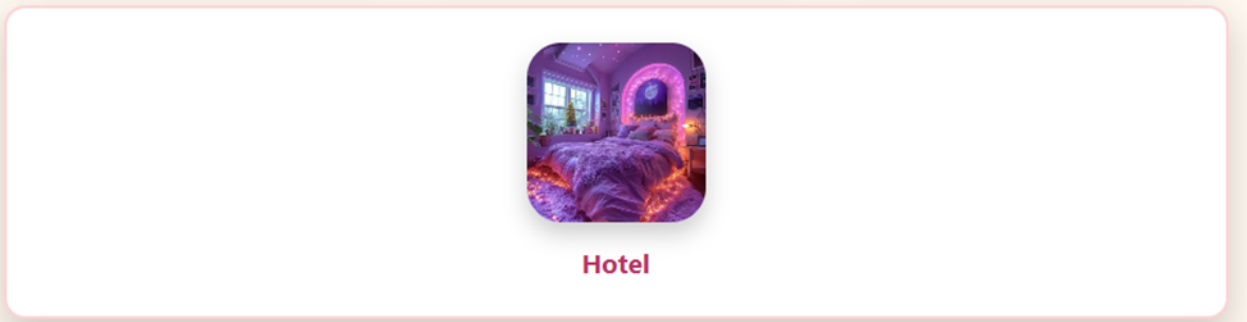
- Flight



The **Flight Booking Page** enables users to search and book flights by entering the departure and arrival locations, travel date, and number of passengers.After users submit the search, the system shows a list of available flights with details like airline name, departure and arrival times,

duration, and fare. Users can select a flight, choose seats if there, and proceed to confirm the booking. This page ensures a smooth and efficient flight booking undergo within the TravelGo platform.

- Hotel



Booking details:

The **Hotel Booking Page** allows users to search for hotels by entering the location, check-in and check-out dates, number of guests, and room druthers. Based on the input, the platform displays a list of useable hotels

with details like room types, prices, and comforts. Users can select a hotel, view room options, and proceed with booking check. The page offers a seamless undergo for reserving accommodations, making it easy for users to plan and manage their stays through TravelGo.

□Conclusion of Travel Go Project

The **Travel Go** project delivers a robust, cloud-powered travel booking platform that simplifies the process of booking **buses, trains, flights, and hotels** for users in real time. By using **Flask** for backend growth, **AWS EC2** for authentic deployment, **DynamoDB** for scalable data storage, and **AWS SNS** for instant notices, the system offers a seamless and responsive user undergo.

From user hallmark to booking manage and real-time alerts, Travel Go addresses the critical challenges of modern travel planning with and abstractness. The platform's ensures scalable, protection, and performance—making it a practical solution for travelers. This project demonstrates how cloud applied science can be in effect used to build smart, user-centric apps that operate smoothly under real-world conditions

228x1a4299@khitguntur.ac.in
7/2/2025