

TravelGo: A Cloud-Based Instant Travel Reservation Platform

Project Overview:

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

Use Case Scenarios:

Scenario 1: Instant Travel Booking Experience

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

Scenario 2: Instant Email Alerts

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

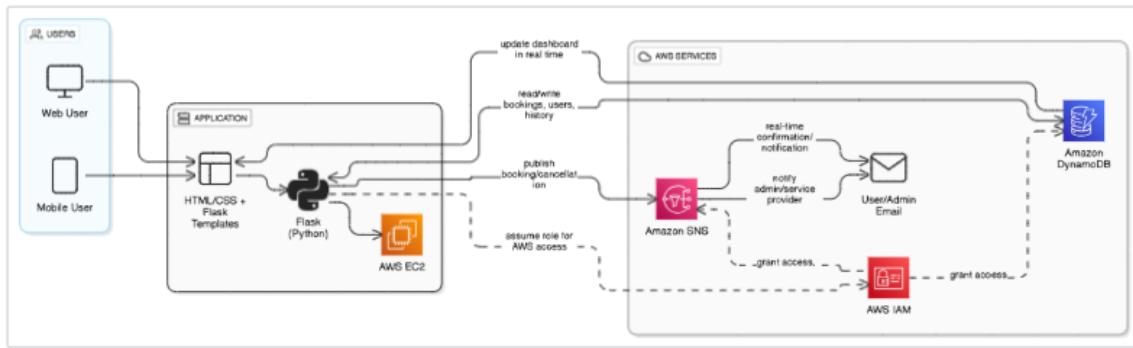
Scenario 3: Easy Booking Access and Control

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

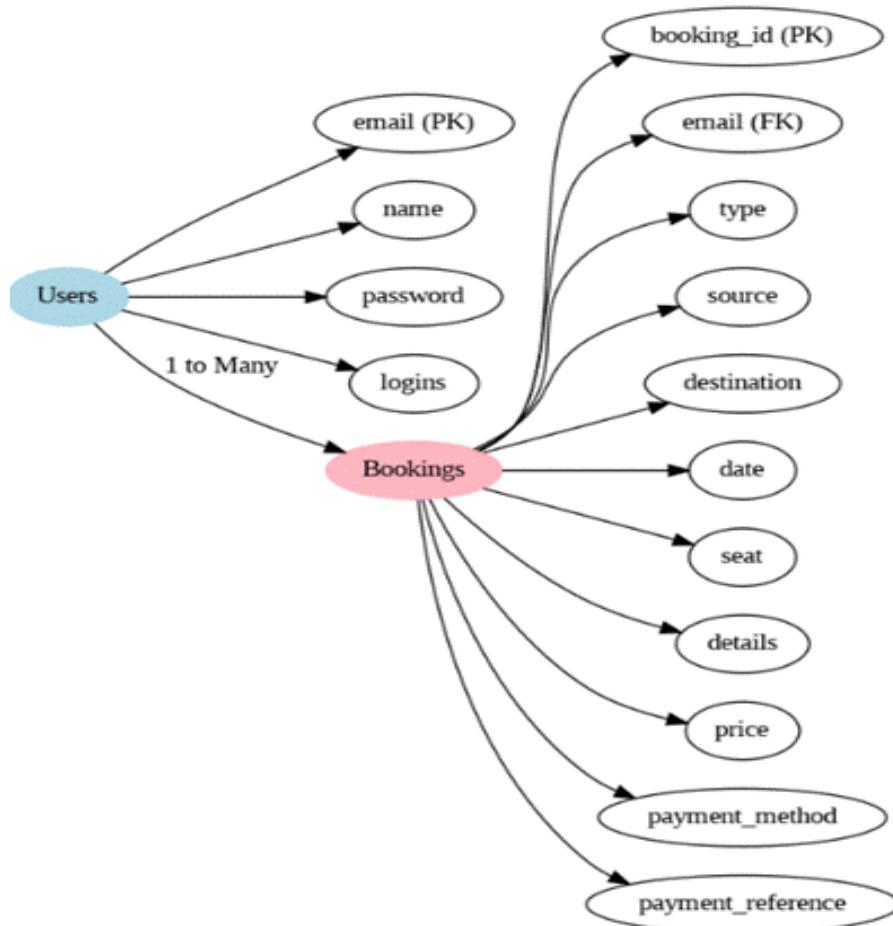
AWS Architecture Diagram

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

AWS Architecture



Entity Relationship Diagram



Requirements:

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

Development Steps:

1. AWS Account Setup and Login

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

2. DynamoDB Database Initialization

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

3. SNS Notification Configuration

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

4. Backend Development using Flask

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

5. IAM Role Configuration

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

6. EC2 Instance Launch

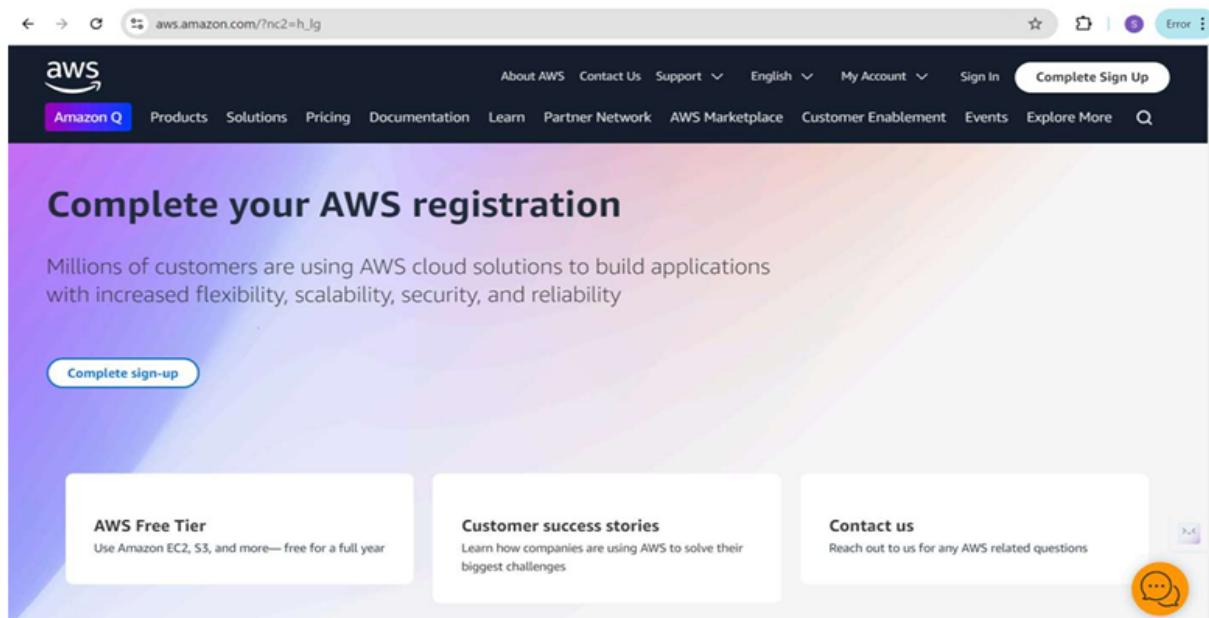
- Activity 6.1: Start an EC2 instance to host the backend.
- Activity 6.2: Set up security rules to allow HTTP and SSH traffic.

7. Flask Application Deployment

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

8. Testing and Launch:

1. AWS Account Setup and Login



2. Log in to the AWS Management Console

A screenshot of the AWS Sign In page. The top left features the AWS logo. Below it is a 'Sign in' section with two radio button options: 'Root user' (selected) and 'IAM user'. The 'Root user' option is described as 'Account owner that performs tasks requiring unrestricted access'. The 'IAM user' option is described as 'User within an account that performs daily tasks'. Below these is a 'Root user email address' input field containing 'username@example.com'. A 'Next' button is at the bottom of this section. To the right, there's a large purple banner with white text: 'AI Use Case Explorer', 'Discover AI use cases, customer success stories, and expert-curated implementation plans', and a 'Explore now >' button. At the very bottom of the page, there's a small note about cookie consent.

After logging into the AWS Console:

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

3. IAM Roles Creation

The screenshot shows the AWS IAM Dashboard. On the left, a sidebar lists 'Access management' (User groups, Users, Roles, Policies, Identity providers, Account settings, Root access management) and 'Access reports' (Resource analysis, Unused access, Analyzer settings, Credential report). The main area displays 'IAM resources' with a red box highlighting an 'Access denied' message: 'You don't have permission to iam:GetAccountSummary. To request access, copy the following text and send it to your AWS administrator.' Below this is a 'What's new' section and a 'Tools' section with a 'Diagnose with Amazon Q' button. On the right, there's an 'AWS Account' section with another 'Access denied' message related to 'iam>ListAccountAliases'. The bottom status bar shows 'CloudShell Feedback' and system icons.

The screenshot shows the 'Create role' wizard. The current step is 'Use case', which allows selecting a service to perform actions on behalf of. A dropdown menu shows 'EC2' selected. Below it, a list of use cases is provided, each with a description and a radio button. The 'EC2' option is selected. Other options include 'EC2 Role for AWS Systems Manager', 'EC2 Spot Fleet Role', 'EC2 - Spot Fleet Auto Scaling', 'EC2 - Spot Fleet Tagging', 'EC2 - Spot Instances', 'EC2 - Spot Fleet', and 'EC2 - Scheduled Instances'. At the bottom right are 'Cancel' and 'Next' buttons. The status bar at the bottom includes 'CloudShell Feedback', system icons, and the date '02-07-2025'.

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

The screenshot shows the AWS IAM 'Create role' wizard at Step 2: Add permissions. The user is searching for EC2 policies. The 'AmazonEC2FullAccess' policy is selected. The interface includes a sidebar with steps: Step 1 (Select trusted entity), Step 2 (Add permissions, which is active), and Step 3 (Name, review, and create). The main area shows a list of EC2 policies with filtering and pagination.

The screenshot shows the AWS IAM 'Create role' wizard at Step 3: Name, review, and create. The role details are filled out: Role name is 'StudentUser' and Description is 'Allows EC2 instances to call AWS services on your behalf.' The Step 1: Select trusted entities section shows a JSON trust policy:

```

1- {
2-   "Version": "2012-10-17",
3-   "Statement": [
4-     {
5-       "Effect": "Allow",
6-       "Action": [
7-         "sts:AssumeRole"

```

4. DynamoDB Database Creation and Setup

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

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

Search results for 'dyn'

Services

- DynamoDB
- Amazon DocumentDB
- CloudFront
- Athena

Features

- Settings
- Clusters

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.

The bookings table was created successfully.

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.

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.

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.

Table settings

CloudShell Feedback © 2025, Amazon Web Services, Inc. or its affiliates. Privacy Terms Cookie preferences
30°C Mostly cloudy Search

Repeat the same procedure to create additional tables:

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

The screenshot shows the 'Create table' step in the AWS DynamoDB console. In the 'Table details' section, the table name is set to 'bookings'. The 'Partition key' is defined as 'user_email' of type 'String'. An optional 'Sort key - optional' field contains 'booking_date'. Under 'Table settings', the 'Default settings' option is selected. The browser status bar at the bottom indicates it's 13:42 on 02-07-2025.

The screenshot shows the 'Create table' step in the AWS DynamoDB console. In the 'Table details' section, the table name is set to 'trains'. The 'Partition key' is defined as 'train_number' of type 'String'. An optional 'Sort key - optional' field contains 'date'. Under 'Table settings', the 'Default settings' option is selected. The browser status bar at the bottom indicates it's 13:45 on 02-07-2025.

The screenshot shows the AWS DynamoDB console. On the left, there's a navigation sidebar with links like Dashboard, Tables, Explore items, PartiQL editor, Backups, Exports to S3, Imports from S3, Integrations, Reserved capacity, and Settings. Below that is a section for DAX with Clusters, Subnet groups, Parameter groups, and Events. The main area is titled "Tables (3) Info" and lists three tables:

Name	Status	Partition key	Sort key	Indexes	Replication Regions	Deletion protection	Favorite	Read
bookings	Active	user_email (\$)	booking_date (\$)	0	0	Off	☆	On-di
trains	Active	train_number (\$)	date (\$)	0	0	Off	☆	On-di
travelgo_users	Active	email (\$)	-	0	0	Off	☆	On-di

At the bottom of the page, there are links for CloudShell, Feedback, © 2025, Amazon Web Services, Inc. or its affiliates., Privacy, Terms, and Cookie preferences.

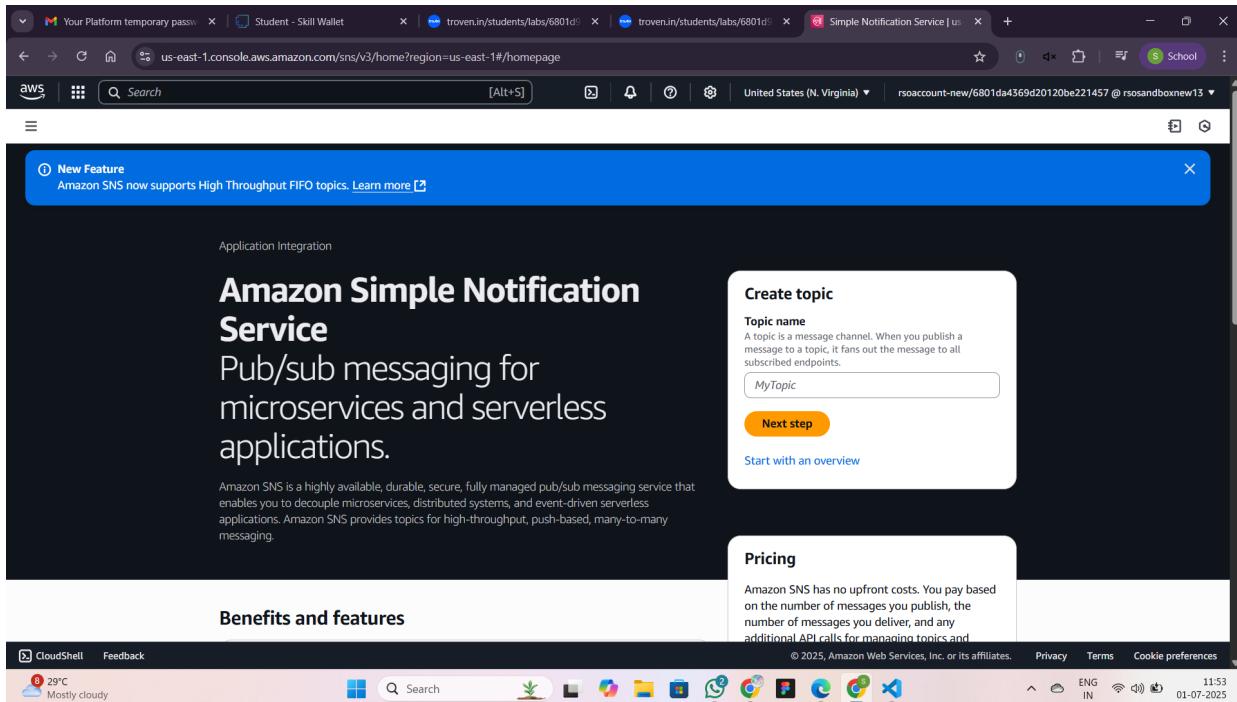
5. SNS Notification Setup

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

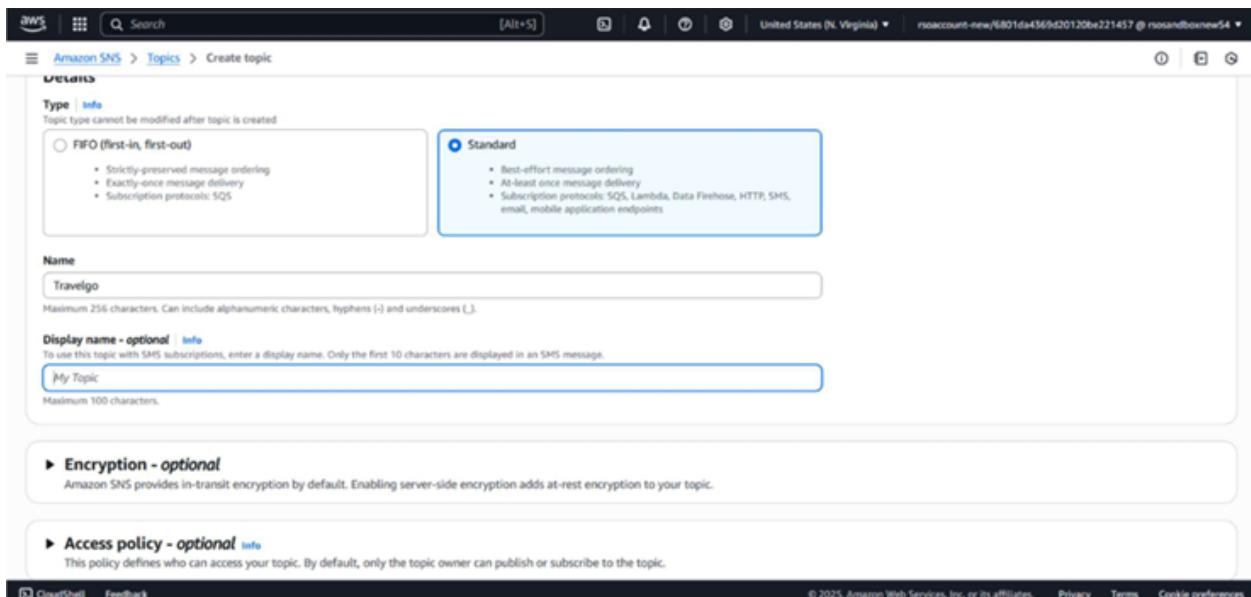
The screenshot shows the AWS search results for the query "sns". On the left, there's a sidebar with links for Services, Features, Resources (New), Documentation, Knowledge articles, Marketplace, Blog posts, Events, and Tutorials. The main search results are displayed under the heading "Search results for 'sns'".

The first result is "Simple Notification Service" with a star icon, described as "SNS managed message topics for Pub/Sub". Below it are other services: "Route 53 Resolver" (Resolve DNS queries in your Amazon VPC and on-premises network), "Route 53" (Scalable DNS and Domain Name Registration), and "AWS End User Messaging" (Engage your customers across multiple communication channels).

Below the search results, there are sections for "Features" with "Events" (ElastiCache feature) and "SMS" (AWS End User Messaging feature), and "Hosted zones" (Route 53 feature).



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



2. Click on create topic

To begin configuring notifications, navigate to the SNS dashboard and click on "Create Topic." Choose the topic type (Standard or FIFO) based on your requirements. Provide a meaningful name for the topic that reflects its purpose (e.g., booking-alerts). This topic will serve as the communication channel for sending notifications to subscribed users.

The screenshot shows the 'Create topic' configuration page for an AWS SNS topic. It includes sections for optional policies:

- Access policy - optional** Info
This policy defines who can access your topic. By default, only the topic owner can publish or subscribe to the topic.
- Data protection policy - optional** Info
This policy defines which sensitive data to monitor and to prevent from being exchanged via your topic.
- Delivery policy (HTTP/S) - optional** Info
The policy defines how Amazon SNS retries failed deliveries to HTTP/S endpoints. To modify the default settings, expand this section.
- Delivery status logging - optional** Info
These settings configure the logging of message delivery status to CloudWatch Logs.
- Tags - optional**
A tag is a metadata label that you can assign to an Amazon SNS topic. Each tag consists of a key and an optional value. You can use tags to search and filter your topics and track your costs. [Learn more](#)
- Active tracing - optional** Info
Use AWS X-Ray active tracing for this topic to view its traces and service map in Amazon CloudWatch. Additional costs apply.

At the bottom right are 'Cancel' and 'Create topic' buttons.

3. Configure the SNS topic and note down the **Topic ARN**.

4. Click on create subscription.

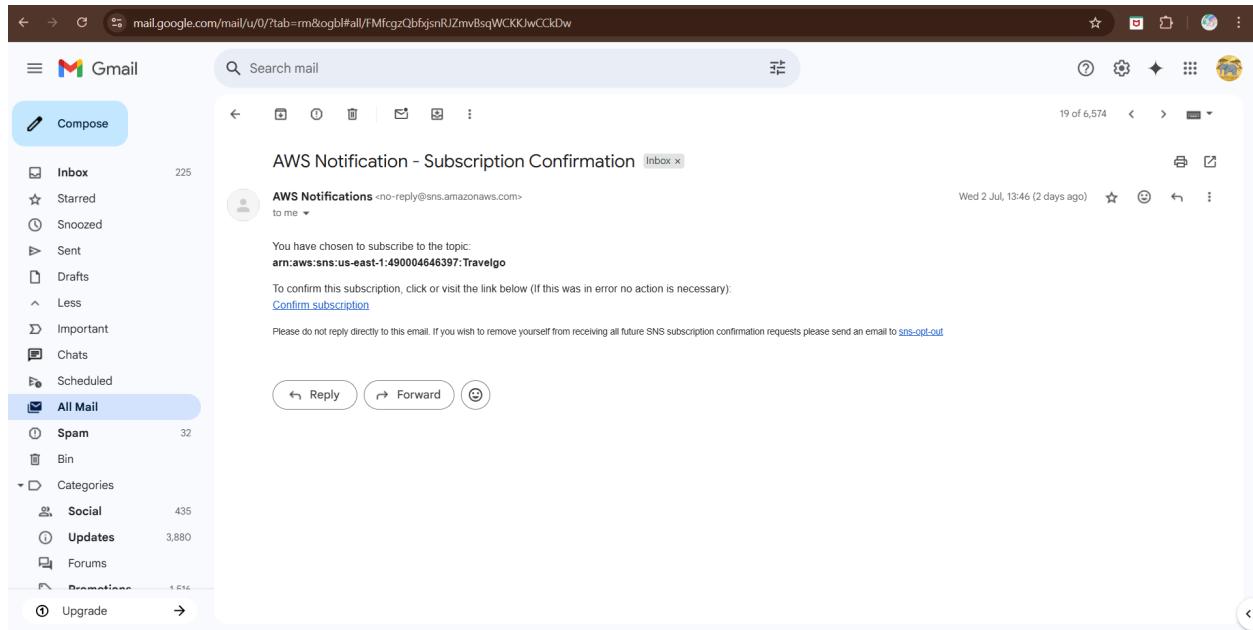
The screenshot shows the 'Create subscription' configuration page for an AWS SNS topic. It includes sections for optional policies:

- Subscription filter policy - optional** Info
This policy filters the messages that a subscriber receives.
- Redrive policy (dead-letter queue) - optional** Info
Send undeliverable messages to a dead-letter queue.

At the bottom right are 'Cancel' and 'Create subscription' buttons.

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

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



Backend Configuration and Coding

```

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

```

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

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

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

```

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

```

SNS connection:

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

```

22 SNS_TOPIC_ARN = 'arn:aws:sns:us-east-1:490004646397:Travelgo:372db6a7-7131-4571-8397-28b62c9e08a8'
23
24 # Function to send SNS notifications
25 # This function is duplicated in the original code, removing the duplicate.
26 def send_sns_notification(subject, message):
27     try:
28         sns_client.publish(
29             TopicArn=SNS_TOPIC_ARN,
30             Subject=subject,
31             Message=message
32         )
33     except Exception as e:
34         print(f"SNS Error: Could not send notification - {e}")
35         # Optionally, flash an error message to the user or log it more robustly.

```

Routes:

Route Overview of TravelGo:

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

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

```

38 @app.route('/')
39 def index():
40     return render_template('index.html')
41
42 @app.route('/register', methods=['GET', 'POST'])
43 def register():
44     if request.method == 'POST':
45         email = request.form['email']
46         password = request.form['password']
47
48         # Check if user already exists
49         # This uses get_item on the primary key 'email', so no GSI needed.
50         existing = users_table.get_item(Key={'email': email})
51         if 'Item' in existing:
52             flash('Email already exists!', 'error')
53             return render_template('register.html')
54
55         # Hash password and store user
56         hashed_password = generate_password_hash(password)
57         users_table.put_item(Item={'email': email, 'password': hashed_password})
58         flash('Registration successful! Please log in.', 'success')
59         return redirect(url_for('login'))
60     return render_template('register.html')

```

```
62     @app.route('/login', methods=['GET', 'POST'])
63     def login():
64         if request.method == 'POST':
65             email = request.form['email']
66             password = request.form['password']
67
68             # Retrieve user by email (primary key)
69             user = users_table.get_item(Key={'email': email})
70
71             # Authenticate user
72             if 'Item' in user and check_password_hash(user['Item']['password'], password):
73                 session['email'] = email
74                 flash('Logged in successfully!', 'success')
75                 return redirect(url_for('dashboard'))
76             else:
77                 flash('Invalid email or password!', 'error')
78                 return render_template('login.html')
79
80     return render_template('login.html')
```

```
81     @app.route('/logout')
82     def logout():
83         session.pop('email', None)
84         flash('You have been logged out.', 'info')
85         return redirect(url_for('index'))
86
87     @app.route('/dashboard')
88     def dashboard():
89         if 'email' not in session:
90             return redirect(url_for('login'))
91         user_email = session['email']
92
93         # Query bookings for the logged-in user using the primary key 'user_email'
94         # No GSI is needed here as 'user_email' is likely the partition key for the bookings_table.
95         response = bookings_table.query(
96             KeyConditionExpression=Key('user_email').eq(user_email),
97             ScanIndexForward=False # Get most recent bookings first
98         )
99         bookings = response.get('Items', [])
100
101        # Convert Decimal types from DynamoDB to float for display if necessary
102        for booking in bookings:
103            if 'total_price' in booking:
104                try:
105                    booking['total_price'] = float(booking['total_price'])
106                except (TypeError, ValueError):
107                    booking['total_price'] = 0.0 # Default value if conversion fails
108
109        return render_template('dashboard.html', username=user_email, bookings=bookings)
```

```
110     @app.route('/train')
111     def train():
112         if 'email' not in session:
113             return redirect(url_for('login'))
114
115     return render_template('train.html')
```

```

116 @app.route('/confirm_train_details')
117 def confirm_train_details():
118     if 'email' not in session:
119         return redirect(url_for('login'))
120
121     booking_details = {
122         'name': request.args.get('name'),
123         'train_number': request.args.get('trainNumber'),
124         'source': request.args.get('source'),
125         'destination': request.args.get('destination'),
126         'departure_time': request.args.get('departureTime'),
127         'arrival_time': request.args.get('arrivalTime'),
128         'price_per_person': Decimal(request.args.get('price')),
129         'travel_date': request.args.get('date'),
130         'num_persons': int(request.args.get('persons')),
131         'item_id': request.args.get('trainId'), # This is the train ID
132         'booking_type': 'train',
133         'user_email': session['email'],
134         'total_price': Decimal(request.args.get('price')) * int(request.args.get('persons'))
135     }
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492 @app.route('/cancel_booking', methods=['POST'])
493 def cancel_booking():
494     if 'email' not in session:
495         return redirect(url_for('login'))
496
497     booking_id = request.form.get('booking_id')
498     user_email = session['email']
499     booking_date = request.form.get('booking_date') # This is crucial as it's the sort key
500
501     if not booking_id or not booking_date:
502         flash("Error: Booking ID or Booking Date is missing for cancellation.", 'error')
503         return redirect(url_for('dashboard'))
504
505     try:
506         # Delete item using the primary key (user_email and booking_date)
507         # This does not use GSI, so it remains unchanged.
508         bookings_table.delete_item(
509             Key={'user_email': user_email, 'booking_date': booking_date}
510         )
511         flash(f"Booking {booking_id} cancelled successfully!", 'success')
512     except Exception as e:
513         flash(f"Failed to cancel booking {booking_id}: {str(e)}", 'error')
514
515     return redirect(url_for('dashboard'))

```

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

Deployment code:

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

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

6. EC2 Instance Setup:

Harini-devx09 Update app.py		
static	Initial commit: TravelGo project uploaded	3 days ago
templates	Initial commit: TravelGo project uploaded	3 days ago
venv	Initial commit: TravelGo project uploaded	3 days ago
venv_new	Initial commit: TravelGo project uploaded	3 days ago
README.md	First commit	3 days ago
app.py	Update app.py	yesterday

Launch an EC2 instance to host the Flask application.

The screenshot shows the AWS search interface with the query 'EC2' entered in the search bar. The results are categorized into 'Services' and 'Features'.

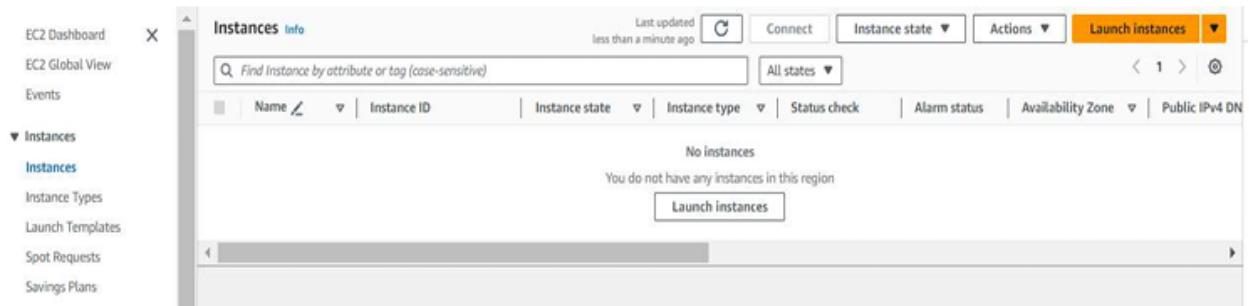
Services (9)

- EC2: Virtual Servers in the Cloud
- EC2 Image Builder: A managed service to automate build, customize and deploy OS images
- AWS Compute Optimizer: Recommend optimal AWS Compute resources for your workloads
- AWS Firewall Manager: Central management of firewall rules

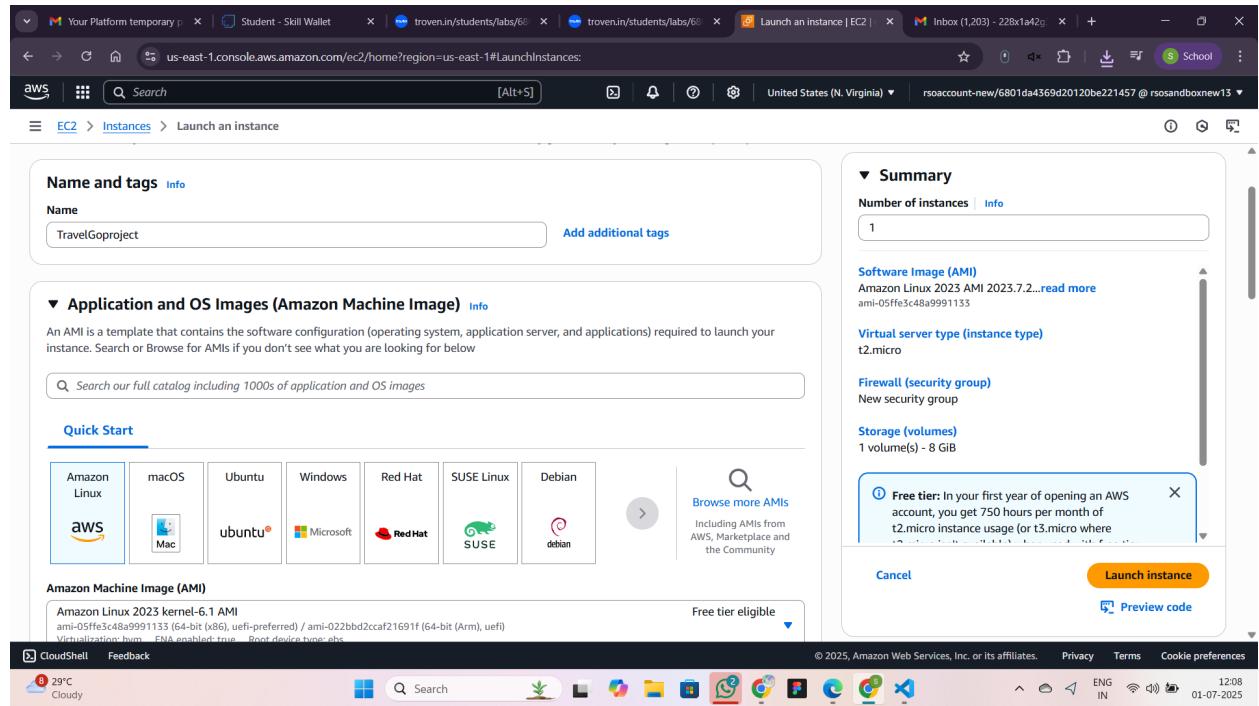
Features (40)

- Export snapshots to EC2: Lightsail feature
- Dashboard: EC2 feature

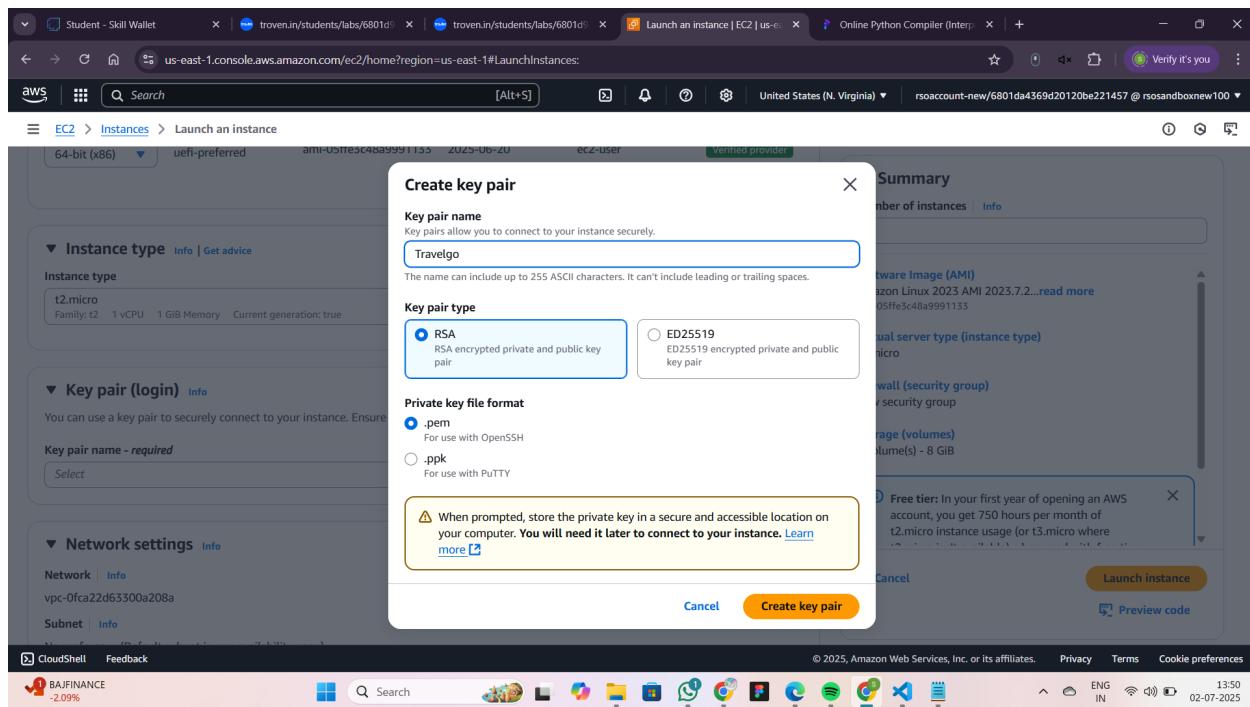
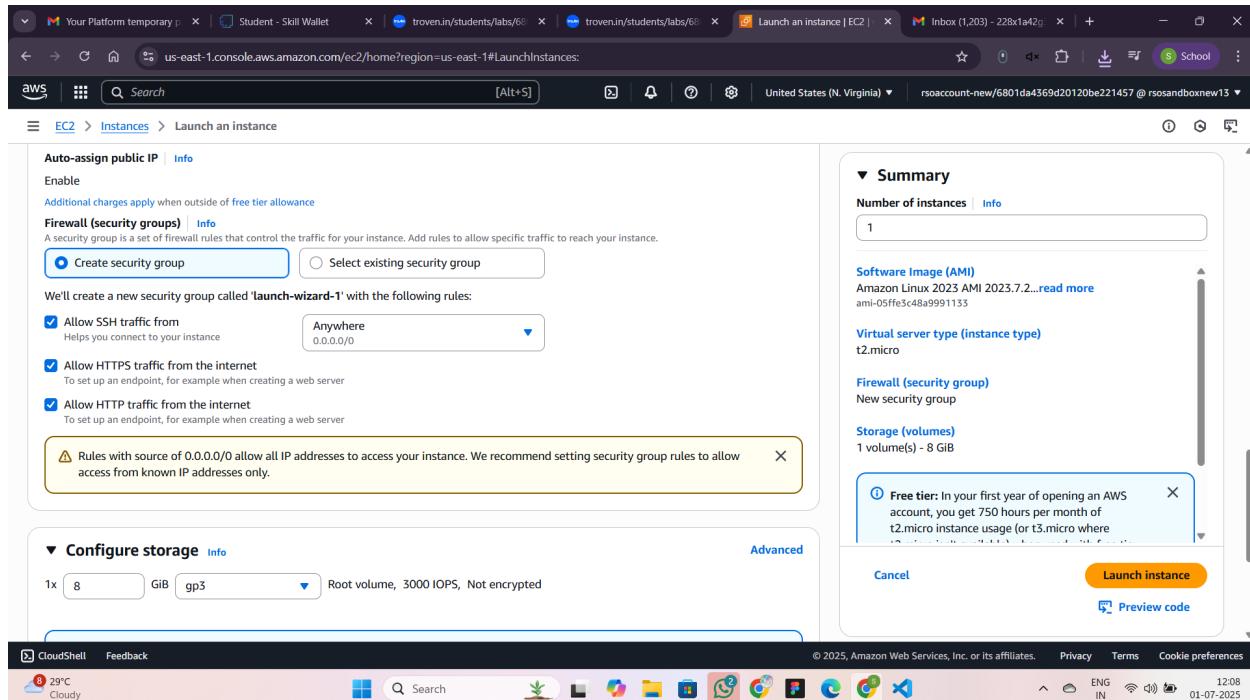
1. Click on launch Instance:



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



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



Wait for the confirmation message like "Successfully initiated launch of instance i-0e7f9bfc28481d812," indicating the instance is being provisioned. During this process, create a key pair named Travelgo (RSA type) with .pem file format. Save this private key securely, as it will be needed for future SSH access.

The screenshot shows the AWS EC2 Instances Launch an instance page. A green success message box at the top right says "Success: Successfully initiated launch of instance (i-0e7f9bfc28481d812)". Below it, a "Launch log" section is collapsed. Under "Next Steps", there are four cards: "Create billing and free tier usage alerts", "Connect to your instance", "Connect an RDS database", and "Create EBS snapshot policy". Each card has a "Create [service] alerts" button. At the bottom, tabs for "Manage detailed monitoring", "Create Load Balancer", "Create AWS budget", and "Manage CloudWatch alarms" are visible.

Edit Inbound Rules:

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

The screenshot shows the AWS Security Groups Edit inbound rules page. It lists existing rules for security group sg-067a674658fc162b. A new rule is being added with the following details:

Security group rule ID	Type	Protocol	Port range	Source	Description - optional
sgr-09d54498e961354af	HTTPS	TCP	443	Custom	0.0.0.0/0
sgr-089fcf1743166b570	SSH	TCP	22	Custom	0.0.0.0/0
sgr-0acec6809fe0352e2	HTTP	TCP	80	Custom	0.0.0.0/0
-	Custom TCP	TCP	5000	Anywh...	0.0.0.0/0

A note at the bottom says: "⚠️ 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." At the bottom right are "Cancel", "Preview changes", and "Save rules" buttons.

Inbound security group rules successfully modified on security group (sg-067a674658fcb162b | launch-wizard-1)

sg-067a674658fcb162b - launch-wizard-1

Details

Security group name launch-wizard-1	Security group ID sg-067a674658fcb162b	Description launch-wizard-1 created 2025-07-02T08:18:42.546Z	VPC ID vpc-0fcfa22d63300a208a
Owner 490004646397	Inbound rules count 4 Permission entries	Outbound rules count 1 Permission entry	

Inbound rules | Outbound rules | Sharing - new | VPC associations - new | Tags

Inbound rules (4)

Name	Security group rule ID	IP version	Type	Protocol	Port range
-	sgr-09d54498e961354af	IPv4	HTTPS	TCP	443
-	sgr-089fcf174316b570	IPv4	SSH	TCP	22
-	sgr-0b7df6b8dbd8ee50c	IPv4	Custom TCP	TCP	5000
-	sgr-0b7df6b8dbd8ee50c	IPv4	HTTP	TCP	80

Modify IAM ROLE

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

Instances (1/1) Info

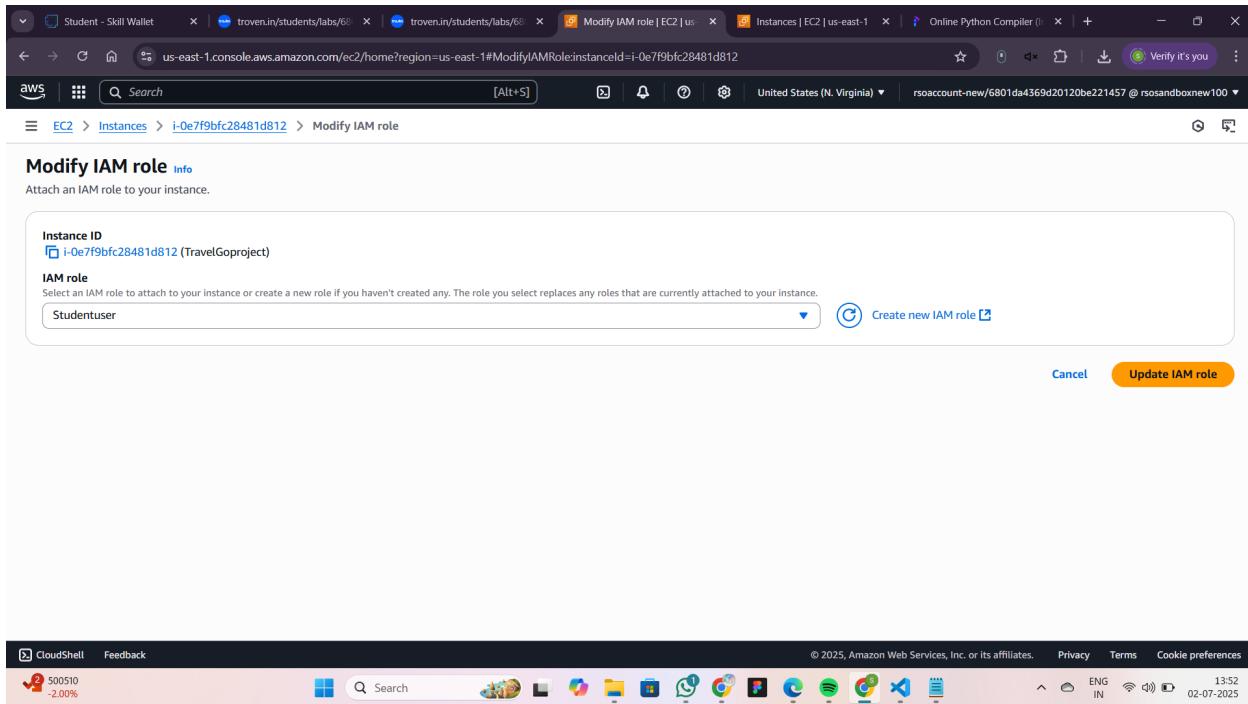
i-0e7f9bfc28481d812 (TravelGoproject)

Actions

- Instance diagnostics
- Instance settings
- Networking
- Security** (highlighted)
- Image and templates
- Monitor and troubleshoot

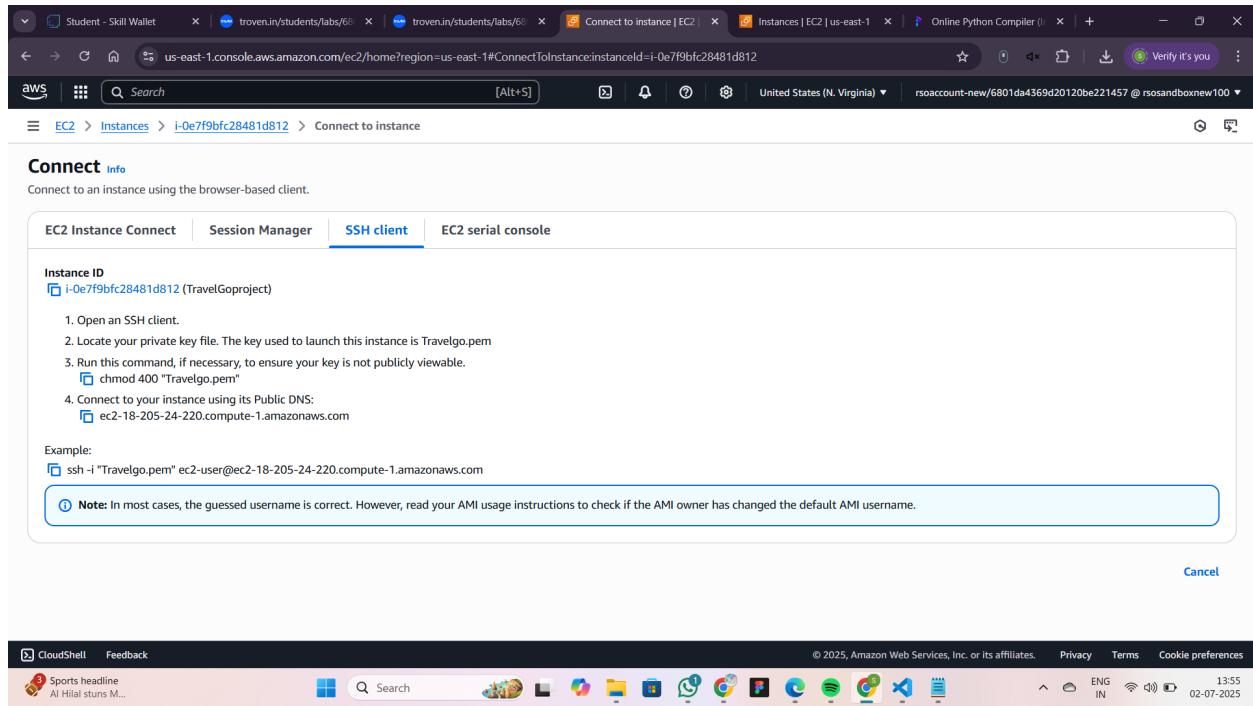
Modify IAM role

Inbound rules



The screenshot shows the 'Instances' page in the AWS EC2 console. A success message at the top states 'Successfully attached Studentuser to instance i-0e7f9bfc28481d812'. The main table lists one instance: 'TravelGoproject' (i-0e7f9bfc28481d812), which is 'Running' and has an 't2.micro' instance type. The details page for this instance shows its configuration, including its public and private IP addresses, instance state (Running), and DNS information (ec2-18-205-24-220.compute-1.amazonaws.com).

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



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

```

Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Install the latest PowerShell for new features and improvements! https://aka.ms/PSWindows

PS C:\Users\harin> ssh -i "C:\Users\harin\Downloads\Travelgo.pem" ec2-user@ec2-34-203-236-31.compute-1.amazonaws.com
'#
~\_\####_          Amazon Linux 2023
~~ \####\
~~ \###|
~~ \#/ ___ https://aws.amazon.com/linux/amazon-linux-2023
~~ V~' '-->
~~ ._. / \
~~ /_/
~/m/'

Last login: Tue Jul 1 07:49:37 2025 from 223.196.198.62
[ec2-user@ip-172-31-92-82 ~]$ sudo yum install git -y
Last metadata expiration check: 2:07:34 ago on Tue Jul 1 06:47:08 2025.
Package git-2.47.1-1.amzn2023.0.3.x86_64 is already installed.
Dependencies resolved.
Nothing to do.
Complete!
[ec2-user@ip-172-31-92-82 ~]$ git clone https://github.com/Harini-devx09/TravelGo.git
fatal: destination path 'TravelGo' already exists and is not an empty directory.
[ec2-user@ip-172-31-92-82 ~]$ cd TravelGo

```

- sudo yum install git -y
- git clone your_repository_url
- cd your_project_directory

```
[ec2-user@ip-172-31-82-161 TravelGo]$ sudo yum install python3 -y
Last metadata expiration check: 0:17:09 ago on Wed Jul  2 08:33:38 2025.
Package python3-3.9.23-1.amzn2023.0.1.x86_64 is already installed.
Dependencies resolved.
Nothing to do.
Complete!
[ec2-user@ip-172-31-82-161 TravelGo]$ sudo yum install python3-pip -y
Last metadata expiration check: 0:17:18 ago on Wed Jul  2 08:33:38 2025.
Dependencies resolved.
=====
  Package           Architecture      Version       Repository   Size
=====
Installing:
  python3-pip        noarch          21.3.1-2.amzn2023.0.11    amazonlinux  1.8 M
Installing weak dependencies:
  libxcrypt-compat   x86_64          4.4.33-7.amzn2023            amazonlinux  92 k
=====
Transaction Summary
=====
Install 2 Packages

Total download size: 1.9 M
Installed size: 11 M
Downloading Packages:
(1/2): libxcrypt-compat-4.4.33-7.amzn2023.x86_64.rpm           2.0 MB/s |  92 kB     00:00
(2/2): python3-pip-21.3.1-2.amzn2023.0.11.noarch.rpm           27 MB/s | 1.8 MB     00:00
=====
Total                                         19 MB/s | 1.9 MB     00:00

Running transaction check
Transaction check succeeded.
Running transaction test
Transaction test succeeded.
Running transaction
  Preparing                           :               1/1
  Installing  libxcrypt-compat-4.4.33-7.amzn2023.x86_64           1/2
  Installing  python3-pip-21.3.1-2.amzn2023.0.11.noarch           2/2
  Running scriptlet: python3-pip-21.3.1-2.amzn2023.0.11.noarch           2/2
  Verifying   libxcrypt-compat-4.4.33-7.amzn2023.x86_64           1/2
  Verifying   python3-pip-21.3.1-2.amzn2023.0.11.noarch           2/2
=====
Installed:
```

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

```
libxcrypt-compat-4.4.33-7.amzn2023.x86_64           python3-pip-21.3.1-2.amzn2023.0.11.noarch
=====
Complete!
[ec2-user@ip-172-31-82-161 TravelGo]$ pip install flask
Defaulting to user installation because normal site-packages is not writeable
Collecting flask
  Downloading flask-3.1.1-py3-none-any.whl (103 kB)
    |██████████| 103 kB 14.3 MB/s
Collecting jinja2>=2.1.2
  Downloading jinja2-3.1.6-py3-none-any.whl (134 kB)
    |██████████| 134 kB 21.9 MB/s
Collecting werkzeug>=3.1.0
  Downloading werkzeug-3.1.3-py3-none-any.whl (224 kB)
    |██████████| 224 kB 53.0 MB/s
Collecting blinker>=1.9.0
  Downloading blinker-1.9.0-py3-none-any.whl (8.5 kB)
Collecting markupsafe>=2.1.1
  Downloading MarkupSafe-3.0.2-cp39-cp39-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (20 kB)
Collecting click>=8.1.3
  Downloading click-8.1.8-py3-none-any.whl (98 kB)
    |██████████| 98 kB 12.4 MB/s
Collecting itsdangerous>=2.2.0
  Downloading itsdangerous-2.2.0-py3-none-any.whl (16 kB)
Collecting importlib-metadata>=3.6.0
  Downloading importlib_metadata-3.6.0-py3-none-any.whl (27 kB)
Collecting zipp>=3.2.0
  Downloading zipp-3.23.0-py3-none-any.whl (10 kB)
Installing collected packages: zipp, markupsafe, werkzeug, jinja2, itsdangerous, importlib-metadata, click, blinker, flask
Successfully installed blinker-1.9.0 click-8.1.8 flask-3.1.1 importlib-metadata-3.6.0 itsdangerous-2.2.0 jinja2-3.1.6 markupsafe-3.0.2 werkzeug-3.1.3 zipp-3.23.0
[ec2-user@ip-172-31-82-161 TravelGo]$ pip install boto3
Defaulting to user installation because normal site-packages is not writeable
Collecting boto3
  Downloading boto3-1.39.1-py3-none-any.whl (139 kB)
    |██████████| 139 kB 15.6 MB/s
Collecting botocore<1.40.0,>=1.39.1
  Downloading botocore-1.39.1-py3-none-any.whl (13.8 kB)
    |██████████| 13.8 kB 43.2 MB/s
Requirement already satisfied: jmespath<2.0.0,>=0.7.1 in /usr/lib/python3.9/site-packages (from boto3) (0.10.0)
Collecting s3transfer<0.14.0,>=0.13.0
  Downloading s3transfer-0.13.0-py3-none-any.whl (85 kB)
```

```

  Downloading s3transfer-0.13.0-py3-none-any.whl (85 kB)
|██████████| 85 kB 8.7 MB/s
Requirement already satisfied: python-dateutil<3.0.0,>=2.1 in /usr/lib/python3.9/site-packages (from botocore<1.40.0,>=1.39.1->boto3) (2.8.1)
Requirement already satisfied: urllib3<1.27,>=1.25.4 in /usr/lib/python3.9/site-packages (from botocore<1.40.0,>=1.39.1->boto3) (1.25.10)
Requirement already satisfied: six>=1.5 in /usr/lib/python3.9/site-packages (from botocore<1.40.0,>=1.39.1->boto3) (1.15.0)
Installing collected packages: botocore, s3transfer, boto3
Successfully installed boto3-1.39.1 botocore-1.39.1 s3transfer-0.13.0
[ec2-user@ip-172-31-82-161 TravelGo]$ python3 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 all addresses (0.0.0.0)
 * Running on http://127.0.0.1:5000
 * Running on http://172.31.82.161:5000
Press CTRL+C to quit
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 913-119-841

```

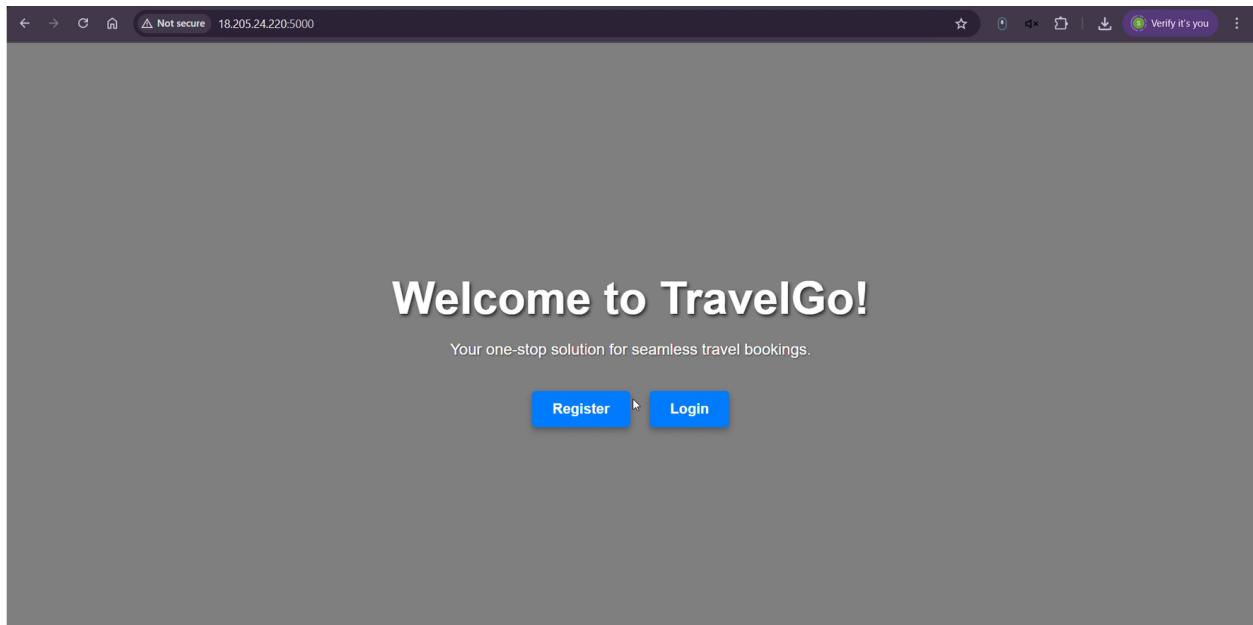
- pip install flask
- pip install boto3
- python3 app.py

Deployment Steps Summary:

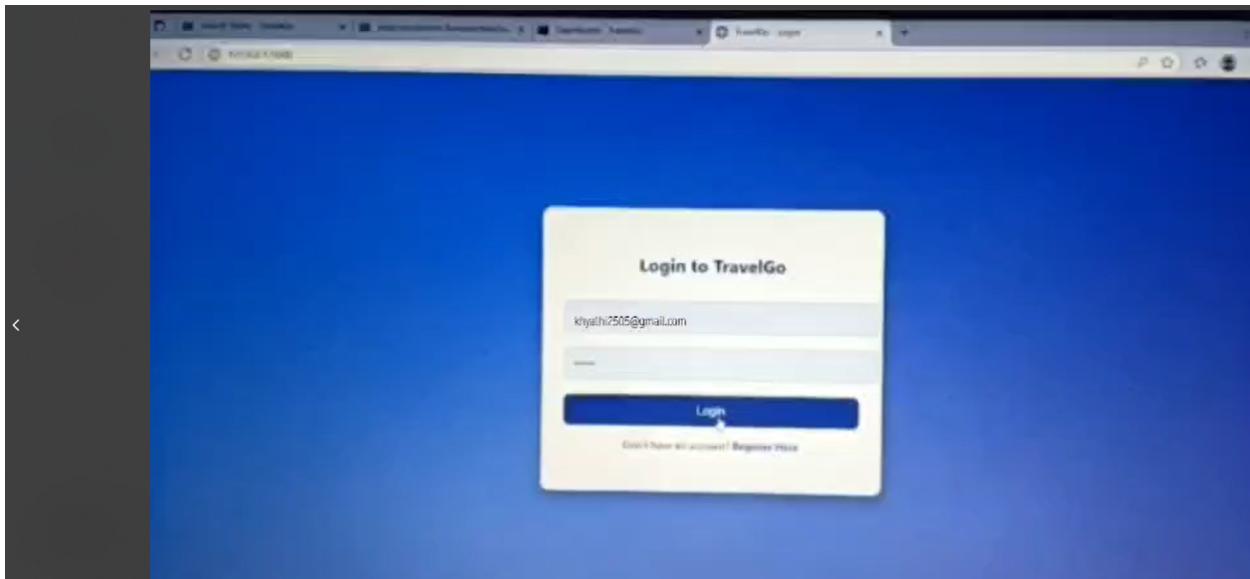
- sudo yum install git -y
- git clone your_repository_url
- cd your_project_directory
- sudo yum install python3 -y
- sudo yum install python3-pip -y
- pip install flask
- pip install boto3
- python3 app.py

User Interface Screens:

- Homepage:



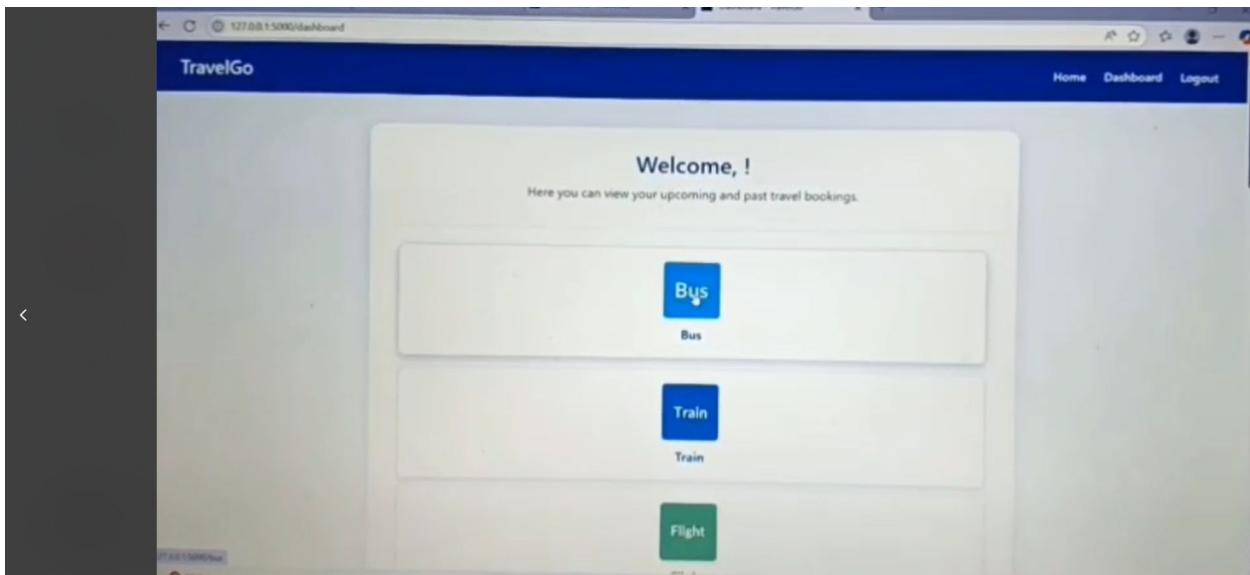
- Register and Login Portals:



- Dashboard View:

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

Lets have a look at my dashboard page!!!



- **Booking Tabs: Bus, Train, Flight, Hotel**

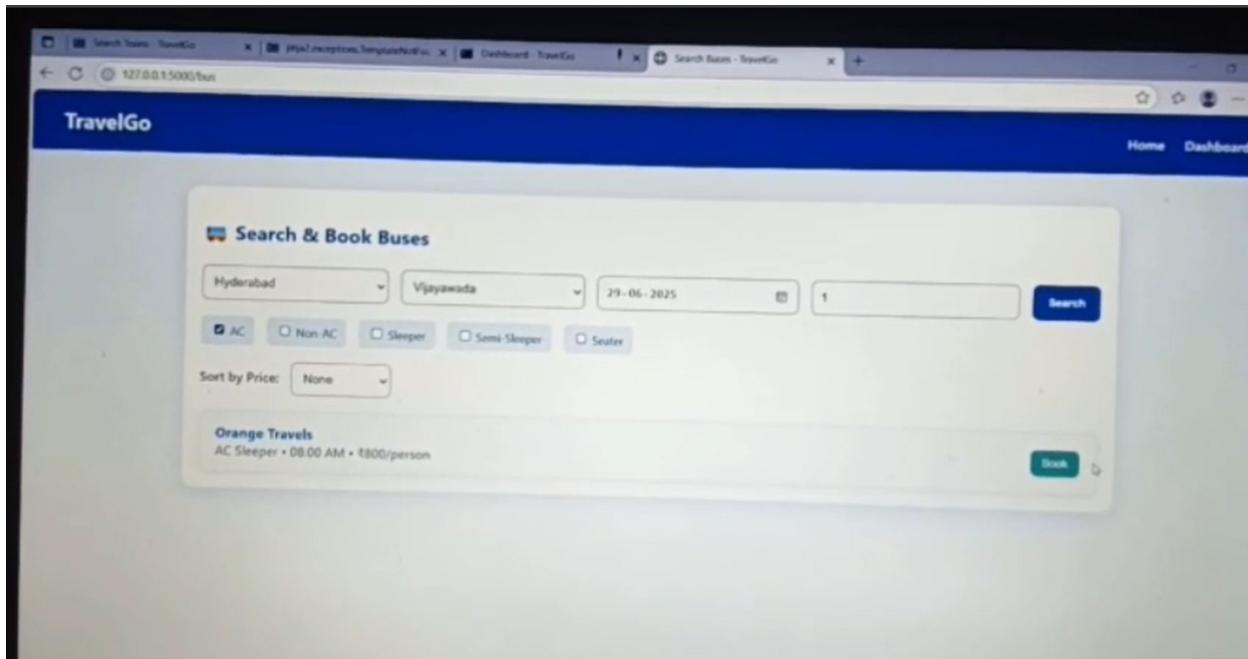
For demonstration purposes, a seat selection section has also been included in the bus booking module. This allows users to choose their preferred seats during the booking process, enhancing the overall user experience.

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

- **Bus booking:**

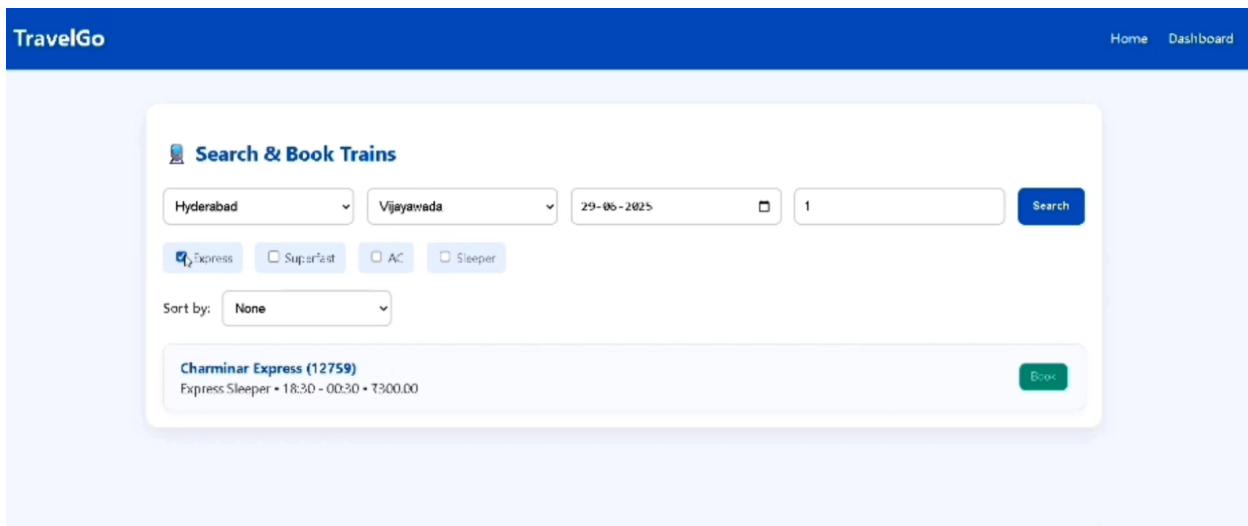
The screenshot shows the 'Search & Book Buses' interface. The 'From' dropdown is set to 'Hyderabad'. Below it, a list of cities is displayed: Hyderabad, Vijayawada, Guntur, Bengaluru, and Chennai. To the right of the dropdown are three checkboxes: 'Sleeper', 'Semi-Sleeper', and 'Seater'. A 'Search' button is located at the bottom right of the search bar area.

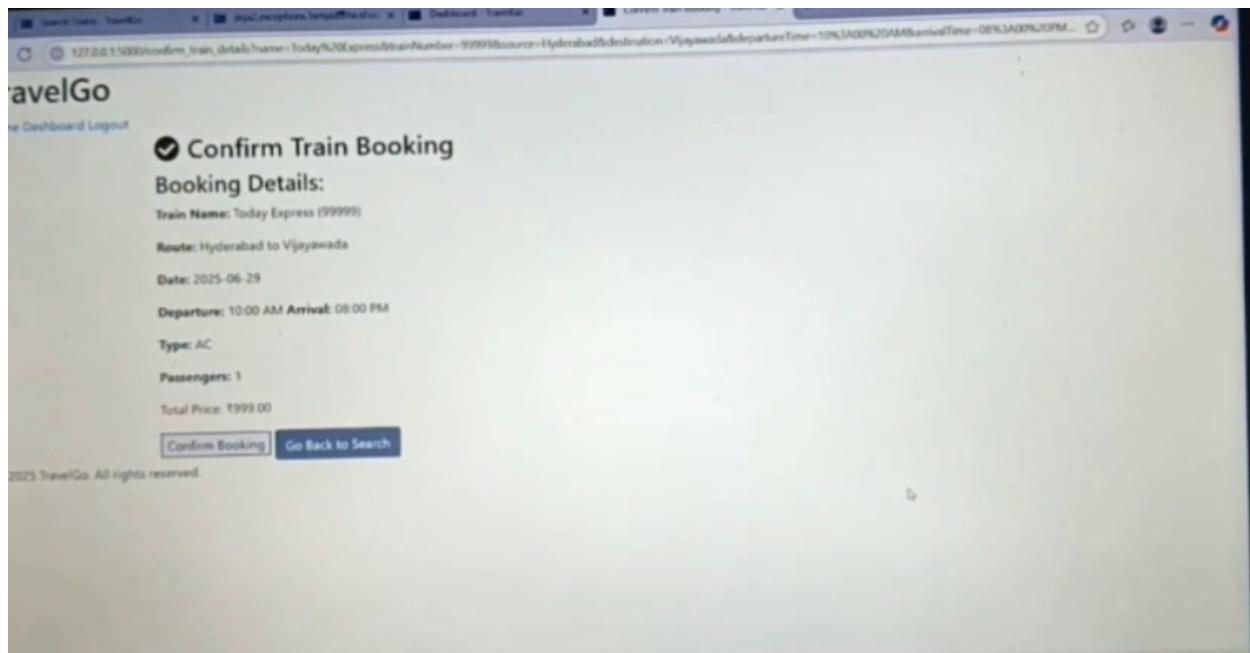
The screenshot shows a seat selection grid for a bus. The grid is organized into 5 rows (A, B, C, D, E) and 6 columns. The seats are numbered from 1 to 6. Seats 4 and 5 in row E are highlighted in blue, while others are grey. The word 'Driver' is at the top left, and 'Door' is at the top right. A 'Proceed to Confirm Booking' button is at the bottom.



• Train booking:

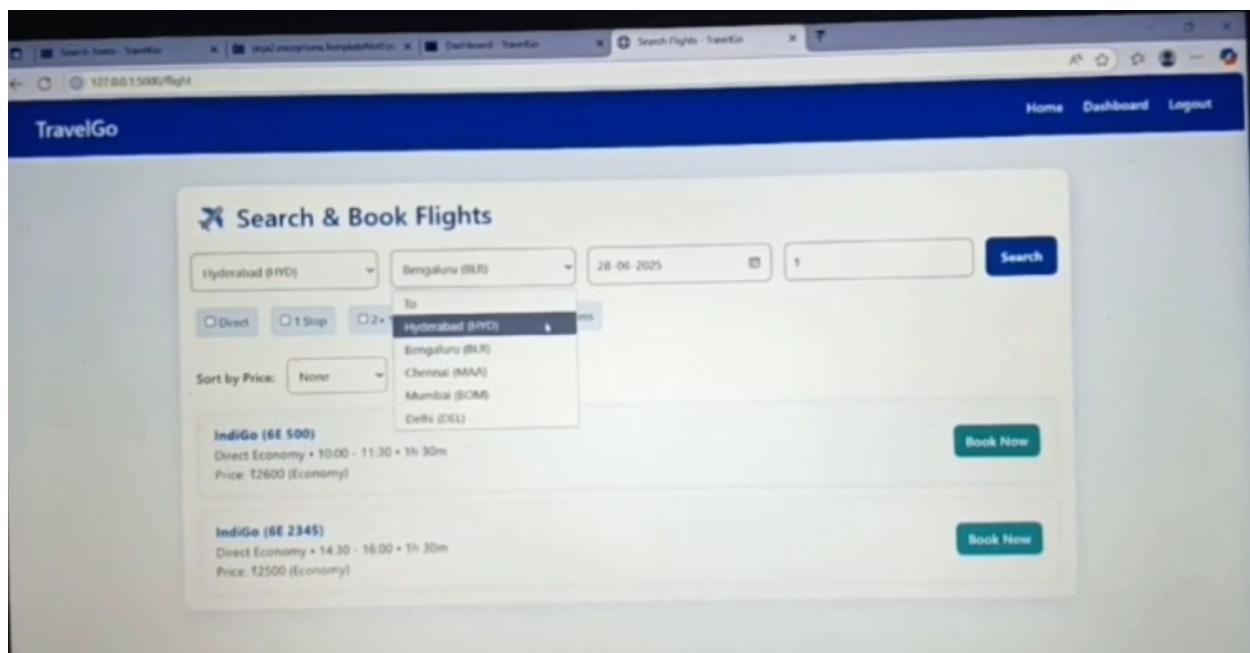
The train booking module enables users to search and reserve train tickets by selecting routes, dates, and times. It fetches real-time data and displays available options tailored to user input. Once a booking is made, the system confirms the reservation and stores the details in the database. Bookings are reflected instantly on the user dashboard for easy tracking. This module ensures a seamless and efficient booking experience for train travelers.





• Flight booking:

→ The flight booking section enables users to search and reserve flights quickly based on their preferred source, destination, and date. Upon entering the details, the system fetches available flight options and displays them with timing, pricing, and route information. The interface is clean and responsive, allowing users to confirm bookings with just a few clicks. Once booked, flight details are stored securely and reflected in the dashboard. This module ensures a smooth and efficient booking experience tailored for air travel.



TravelGo

Home Dashboard

Search & Book Flights

Origin: Hyderabad (HYD) | Destination: [dropdown] | Date: 29-06-2025 | Passengers: 1

Indigo Vistara Air India Direct Stop

Sort by: None

Please select Origin, Destination, Date, and valid Number of Passengers to search.

Confirm Your Flight Booking

Airline:	Indigo
Flight Number:	6E 234
Route:	Hyderabad -> Mumbai
Departure Time:	08:00
Arrival Time:	09:30
Travel Date:	2025-06-29
Number of Passengers:	1
Price per Person:	₹3500.0
Total Price:	₹3500.0

• Hotel booking:

The hotel booking section allows users to search and reserve accommodations based on their destination and travel dates. The interface displays available hotels with details such as location, price, and amenities. Users can easily compare options and proceed with booking in just a few clicks. Once confirmed, the booking details appear on the dashboard for easy tracking. The process is designed to be seamless, intuitive, and efficient for all types of travelers.

Find & Book Hotel Rooms

 5-Star 4-Star 3-Star WiFi Pool ParkingSort by: **Taj Falaknuma Palace**

Hyderabad • 5-Star • ₹25000.00/night

Amenities: WiFi, Pool, Parking, Restaurant

Total for 1 nights, 1 rooms: ₹25000.00

The screenshot shows the TravelGo website interface for booking a perfect stay. At the top, there is a header bar with the TravelGo logo, a 'Not secure' warning, and navigation links for Home, Dashboard, and Hotels. Below the header is a search form with fields for destination ('Hyderabad'), check-in date ('03-07-2025'), check-out date ('07-07-2025'), number of guests ('1'), and number of rooms ('1'). A large blue 'Search' button is located to the right of the search fields. Below the search form are filtering options for star ratings (5-Star checked, 4-Star, 3-Star), amenities (WiFi, Pool, Parking), and a sorting dropdown set to 'None'. A section for 'Taj Falaknuma Palace' is displayed, showing it is a 5-star hotel in Hyderabad at ₹25000/night, with amenities including WiFi, Pool, Parking, and Restaurant. A green 'Book' button is visible on the right.

Confirm Your Hotel Booking

Hotel Name: Taj Falaknuma Palace
 Location: Hyderabad
 Check-in Date: 2025-06-29
 Check-out Date: 2025-06-30
 Number of Rooms: 1
 Number of Guests: 1
 Price per Night: ₹25000.0

Total Price: ₹25000.0

◇ At the bottom of the dashboard, users can view a summary of all their bookings, including travel details, dates, and status. A dedicated "Cancel" button is provided next to each entry, allowing users to easily cancel any upcoming reservations if needed.

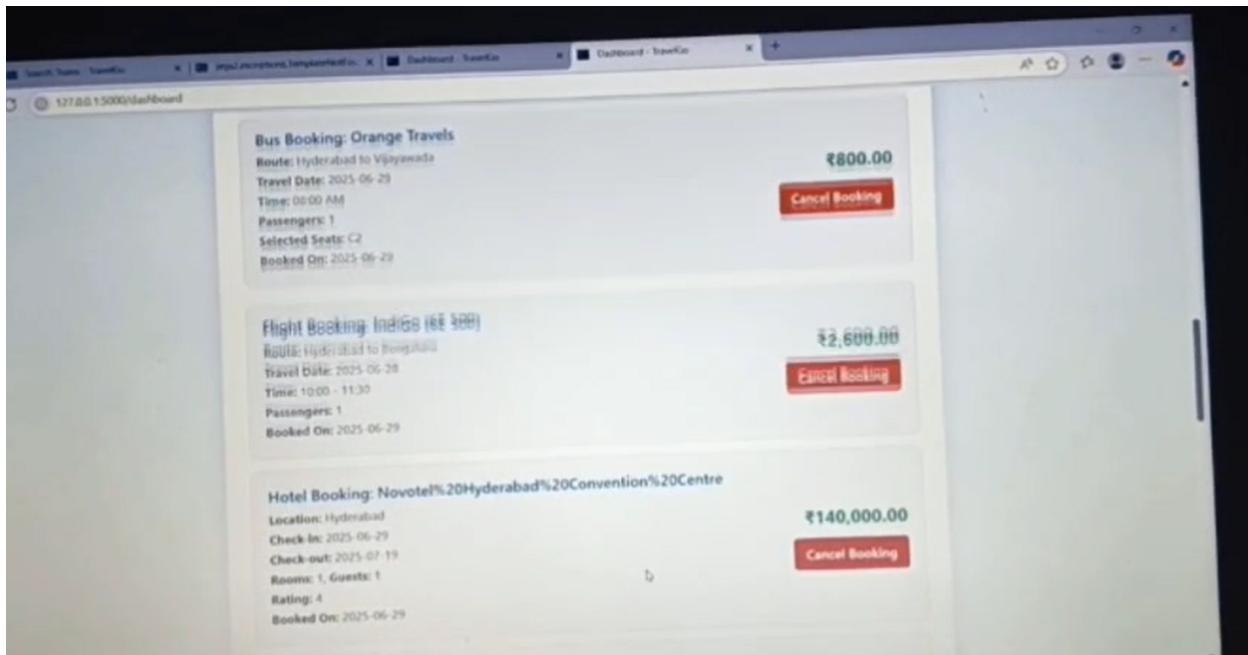
This section provides a centralized view for managing bookings efficiently. It ensures transparency by displaying every confirmed ticket in an organized format. The cancel feature updates the backend in real time, removing the booking from the list and database. This functionality enhances user control and flexibility while planning or modifying travel. It also improves overall convenience by reducing the need to revisit individual booking sections.

All Bookings:

YOUR BOOKINGS	
Hotel Booking: Novotel%20Hyderabad%20Convention%20Centre	₹98,000.00
Location: Hyderabad	Cancel Booking
Check-in: 2025-06-29	
Check-out: 2025-07-13	
Rooms: 1 Guest(s)	
Rating: 4.5	
Booked On: 2025-06-26	
Flight Booking: IndiGo (6E 234)	₹3,500.00
Route: Hyderabad to Mumbai	
Travel Date: 2025-07-01	
Time: 06:00 AM - 09:30 AM	
Passenger(s): 1	Cancel Booking

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

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



Final Conclusion – Elevating Travel with TravelGo:

The TravelGo platform has been thoughtfully designed and successfully deployed using a cloud-native, scalable architecture that meets the dynamic needs of modern travelers. By integrating key AWS services – including EC2 for robust hosting, DynamoDB for real-time and flexible data management, and SNS for instant email notifications – the platform delivers a smooth, all-in-one travel booking experience.

Users can seamlessly register, log in, and book buses, trains, flights, and hotels through a unified and intuitive interface, eliminating the hassle of navigating multiple apps or sites. The backend, powered by Flask, efficiently manages user sessions, dynamic data transactions, and booking flows – ensuring performance and responsiveness at every step.

With real-time email alerts triggered by AWS SNS, users stay informed about their bookings and cancellations the moment they occur. This instant feedback loop not only boosts trust but also enhances overall usability.

In essence, TravelGo stands as a smart, reliable, and user-friendly travel assistant. It demonstrates how cloud technology can unify complex travel services into one cohesive platform – streamlining operations and elevating the end-user experience.

 **With TravelGo, travel planning is no longer a hassle—it's an experience.**

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

